

# Doubly Efficient Private Information Retrieval and Fully Homomorphic RAM Computation from Ring LWE

Wei-Kai Lin  
Northeastern University

Ethan Mook  
Northeastern University

Daniel Wichs\*  
Northeastern University  
and NTT Research

December 8, 2022

## Abstract

A (*single server*) *private information retrieval* (PIR) allows a client to read data from a public database held on a remote server, without revealing to the server which locations she is reading. In a *doubly efficient PIR* (DEPIR), the database is first preprocessed, but the server can subsequently answer any client’s query in time that is sub-linear in the database size. Prior work gave a plausible candidate for a *public-key* variant of DEPIR, where a trusted party is needed to securely preprocess the database and generate a corresponding public key for the clients; security relied on a new non-standard code-based assumption and a heuristic use of ideal obfuscation. In this work we construct the stronger *unkeyed* notion of DEPIR, where the preprocessing is a deterministic procedure that the server can execute on its own. Moreover, we prove security under just the standard *ring learning-with-errors* (RingLWE) assumption. For a database of size  $N$  and any constant  $\varepsilon > 0$ , the preprocessing run-time and size is  $O(N^{1+\varepsilon})$ , while the run-time and communication-complexity of each PIR query is  $\text{poly log}(N)$ . We also show how to update the preprocessed database in time  $O(N^\varepsilon)$ . Our approach is to first construct a standard PIR where the server’s computation consists of evaluating a multivariate polynomial; we then convert it to a DEPIR by preprocessing the polynomial to allow for fast evaluation, using the techniques of Kedlaya and Umans (STOC ’08).

Building on top of our DEPIR, we construct general *fully homomorphic encryption for random-access machines* (RAM-FHE), which allows a server to homomorphically evaluate an arbitrary RAM program  $P$  over a client’s encrypted input  $x$  and the server’s preprocessed plaintext input  $y$  to derive an encryption of the output  $P(x, y)$  in time that scales with the RAM run-time of the computation rather than its circuit size. Prior work only gave a heuristic candidate construction of a restricted notion of RAM-FHE. In this work, we construct RAM-FHE under the RingLWE assumption with circular security. For a RAM program  $P$  with worst-case run-time  $T$ , the homomorphic evaluation runs in time  $T^{1+\varepsilon} \cdot \text{poly log}(|x| + |y|)$ .

---

\*Research supported by NSF grant CNS-1750795, CNS-2055510 and the Alfred P. Sloan Research Fellowship.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Our Results	4
1.2	Our Techniques: DEPIR	5
1.3	Our Techniques: RAM-FHE	9
1.4	Other Related Work	11
<b>2</b>	<b>Preliminaries</b>	<b>12</b>
2.1	Multi-variate Polynomial Evaluation and Interpolation	13
2.2	Ring LWE	14
<b>3</b>	<b>Algebraic Somewhat Homomorphic Encryption (ASHE)</b>	<b>15</b>
3.1	ASHE from RingLWE	16
<b>4</b>	<b>DEPIR</b>	<b>18</b>
4.1	Definition	18
4.2	Construction	20
<b>5</b>	<b>Updatable DEPIR</b>	<b>24</b>
5.1	Multi-Round Updatable DEPIR	26
5.2	Round-Optimal Updatable DEPIR	30
<b>6</b>	<b>ASHE-FHE</b>	<b>32</b>
6.1	Leveled FHE from RingLWE	34
6.2	ASHE-FHE Construction	36
<b>7</b>	<b>RAM-FHE</b>	<b>38</b>
7.1	Definition of the RAM Model and RAM-FHE	38
7.2	RAM-FHE with Random Access to $y$	41
7.3	RAM-FHE with Random Access to $x$	44
7.4	RAM-FHE with Random Access to $z$	46
<b>8</b>	<b>Extensions and Variants of RAM-FHE</b>	<b>51</b>
8.1	Alternative Efficiency Tradeoffs	51
8.2	Leveled RAM-FHE without Circular Security	53
8.3	Updatable RAM-FHE	55
8.4	Multi-Hop / Multi-Input RAM-FHE	55
<b>A</b>	<b>Fast Polynomial Evaluation with Preprocessing</b>	<b>65</b>
A.1	Polynomials Over $\mathbb{Z}_q$	66
A.2	Polynomials Over Extension Rings	69
<b>B</b>	<b>Multivariate Polynomial Interpolation</b>	<b>73</b>

<b>C</b>	<b>Constructions from Other Assumptions</b>	<b>74</b>
C.1	DEPIR from Approximate GCD . . . . .	74
C.2	DEPIR from Module LWE . . . . .	75

# 1 Introduction

Can we design an encrypted version of Google’s search engine that would enable users to search the Internet privately, without revealing their queries to Google? *Fully homomorphic encryption (FHE)* gives an unsatisfactory solution to this problem, where Google would need to run a huge computation over the entire content of the Internet to answer each encrypted search query. Is there a way for Google to preprocess the Internet content into a special data structure that would allow it to answer any future encrypted search query efficiently by only accessing a small number of locations in the data structure? We give the first solution to this problem as a special case of our work. We begin by describing the primitives that we construct and what was previously known.

**PIR and its limitations.** A (*single-server*) *private information retrieval (PIR)* scheme [CGKS95,KO00] is a protocol between a server who holds a database  $DB \in \{0, 1\}^N$  and a client who wishes to learn the  $i$ ’th location  $DB[i]$  of the database without revealing the index  $i \in [N]$  to the server. For example, a client may want to retrieve a movie from Netflix without revealing to Netflix which one. A trivial solution is for the server to simply send the entire database  $DB$  to the client. The goal of PIR is to solve this problem with much lower *communication complexity*, which should be sub-linear in the database size  $N$ , and ideally just  $\text{poly log } N$ .<sup>1</sup>

PIR is one of the most fundamental cryptographic primitives and has been studied extensively over the last 25+ years. It has found a huge number of applications as well as connections to many other problems in cryptography. Moreover, we can view PIR as a very basic form of *fully homomorphic encryption (FHE)* for functions whose domain size  $N$  is polynomial, by taking the database to be the truth table of the function.

However, PIR comes with a major limitation. While it provides low *communication complexity*, it inherently requires the server to read the entire database  $DB$  during each protocol execution and therefore the *computational complexity* on the server side is at least linear in  $N$ . For example, using PIR, Netflix would need to perform a huge computation over its entire movie database to serve a single movie to a client. Indeed, if the server did not access some database location during the protocol execution, then it would learn that the client’s index is not equal to that location, which would violate privacy. While inherent, this limitation significantly restricts PIR’s usefulness.

**DEPIR.** *Doubly efficient PIR (DEPIR)* [BIM00,CHR17,BIPW17] gets around the above limitation and simultaneously provides low communication and server computation. To do so, it modifies the original setting by allowing the database  $DB \in \{0, 1\}^N$  to be *preprocessed* into some static data structure  $\widetilde{DB}$  that is stored on the server. This one-time preprocessing is allowed to run in time that is linear, or even slightly super-linear, in the database size  $N$ . Subsequently, any client can run a PIR protocol with the server to learn the value  $DB[i]$  without revealing the index  $i$ , where both the communication and the server/client computation during the protocol are sub-linear in the database size  $N$ , and ideally just  $\text{poly log } N$ . In particular, this means that the server only reads a sub-linear number of locations in the data structure  $\widetilde{DB}$  during each PIR protocol execution.

One can define 3 flavors of DEPIR, from strongest to weakest, depending on the nature of the preprocessing: *unkeyed, public-key and secret-key DEPIR*. Unkeyed DEPIR is the simplest and strongest notion, and will be the default notion in this paper. In unkeyed DEPIR, the preprocessing

---

<sup>1</sup>Throughout the introduction, we omit fixed polynomial factors in the security parameter in efficiency expressions.

that maps the database  $DB$  to the data structure  $\widetilde{DB}$  is a deterministic function that the server executes on its own, and subsequently any client can run a PIR protocol with the server. Prior works also considered relaxations to public-key and secret-key DEPIR, both of which require a trusted party with secret random coins to setup the system by generating a key that is given to the client(s) and creating the preprocessed database  $\widetilde{DB}$  that is given on the server. In a public-key DEPIR, security holds even if the key is known to the server, meaning that it can be made public and anybody can subsequently run a PIR protocol with the server using this key. In a secret-key DEPIR, security only holds if key is kept secret from the server, meaning that the key can only be given to some designated group of trusted clients who can run PIR protocol executions with the server, but if any one of the clients is ever compromised then all security is lost.<sup>2</sup>

We note that there are other thematically related but fundamentally different methods for accessing data privately with sub-linear communication and computation, including oblivious RAM (ORAM) and PIR with client-side preprocessing. Each of these notions fails to satisfy some of the crucial goals of DEPIR. We defer a detailed comparison to Section 1.4.

**Prior work on DEPIR.** Beimel, Ishai and Malkin [BIM00] proposed the notion of (unkeyed) DEPIR, and asked whether it is possible in the single server setting. Two seminal prior works of Canetti, Holmgren and Richelson [CHR17] and Boyle, Ishai, Pass and Wootters [BIPW17] gave the first candidate constructions of keyed DEPIR. They showed how to construct the weakest flavor of secret-key DEPIR under a new non-standard computational hardness assumption based on secretly permuted Reed-Muller codes. This assumption was further studied by [BHW19, BHMW21], but we don't currently know how to relate it to any standard hardness assumptions previously used in cryptography. The work of Boyle et al. [BIPW17] also showed how to upgrade secret-key DEPIR to public-key DEPIR via a heuristic use of ideal program obfuscation. Unfortunately, we do not know how to prove the security of this transformation under *any* concrete assumption, such as indistinguishability obfuscation. Moreover, in the resulting public-key DEPIR, a trusted party is crucially needed to create the obfuscated program in the public-key, and therefore the same ideas do not yield even a heuristic candidate for unkeyed DEPIR.<sup>3</sup> In summary, it was left as an open problem to instantiate even the weakest form of secret-key DEPIR under any standard hardness assumption studied previously, to instantiate public-key DEPIR under any concrete assumption without using ideal obfuscation, or to give any plausibly secure candidate for unkeyed DEPIR.

**FHE and its limitations.** PIR can be thought of as a special case of *fully homomorphic encryption (FHE)* [RAD78, Gen09, BV11b, BV11a] for functions with a polynomial-size domain. In a general FHE scheme, a client can encrypt some input  $x$  and the server can homomorphically evaluate any circuit  $C$  over the encrypted input and derive an encryption of the output  $C(x)$ , without learning anything about  $x$ . The run-time of the homomorphic evaluation scales with the circuit size  $O(|C|)$ , while encryption time only scales with the input size  $O(|x|)$  and decryption time scales with the output size  $O(|C(x)|)$ . Unfortunately, FHE suffers from the same underlying efficiency limitation

---

<sup>2</sup>Unkeyed DEPIR implies standard PIR, but neither secret-key nor public-key DEPIR appear to do so since they require trusted setup. While secret-key DEPIR already captures a difficult technical challenge, it is unsuitable for most of the target applications of DEPIR, where we want to make a database accessible to large group of untrusted clients.

<sup>3</sup> The techniques in [BIPW17] may conceivably be adapted to achieve a stronger variant of public-key DEPIR than explicitly claimed, where a trusted third party is only needed to generate the public key (AKA a structured common reference string), but the server can then deterministically preprocess the database  $DB$  on its own using the public key.

as PIR. For example, the homomorphic evaluation of an Internet search over an encrypted query would run in time that is linear in the entire size of the Internet! In general, FHE requires us to express the computation as a circuit. Many computations, such as an Internet search, are very efficient in the standard *random-access machine* (RAM) model of computation, but become hugely inefficient when translated into circuits. In particular, computation in the circuit model is always at least linear in the data size, while in the RAM model, many interesting computations (e.g., binary search) run in sub-linear (e.g., logarithmic) time, leading to huge (e.g., exponential) gaps in efficiency between them. Moreover, even if the RAM computation already runs in at least linear time, translating it into a circuit can still incur an additional quadratic overhead [CR72, PF79].

**RAM-FHE.** Analogously to how DEPIR gets around the limitations of PIR, we consider a general notion of *RAM-FHE* [HHWW19] that gets around the limitations of FHE and allows us to directly evaluate RAM programs over encrypted data. In a RAM-FHE, the server holds some plaintext input  $y$  that it deterministically preprocesses into a static data structure  $\tilde{y}$ . The client chooses encryption/decryption keys and encrypts some input  $x$ , sending the resulting ciphertext  $ct_x$  to the server. Let  $P$  be some RAM program with worst-case run-time  $T$  that operates over the inputs  $x, y$  stored in random-access memory, and has access to unlimited amounts of additional read/write random-access memory. Then the server can homomorphically evaluate  $P$  by operating over  $ct_x, \tilde{y}$  to derive an encryption of the output  $P(x, y)$ , where the run-time of the homomorphic evaluation should not be much larger than  $T$ , and in particular, can be sub-linear in  $|x|$  and  $|y|$ . The above formulation is quite flexible, and captures several important special cases:

1. *Encrypted queries over a public database.* We can think of  $y$  as some large public database held by the server and  $x$  as the client's private query over the database. For example, this captures the setting of encrypted Google search, where  $y$  contains Google's Internet index and  $x$  is the client's search term. Note that the database  $y$  is preprocessed once and for all, independently of the encryption/decryption key, and the resulting data structure  $\tilde{y}$  can then be used to perform many homomorphic computations with different clients who send encrypted inputs  $ct_x$  under different encryption keys.
2. *Encrypted database.* We can also think of  $x$  as being a large encrypted database that the client outsources to the server. Later the server can homomorphically execute any computation  $P(x, y)$  with some input  $y$  of its choice and derive an encrypted output for the client.
3. *No database.* RAM-FHE also captures the setting where  $x$  is some short encrypted input,  $y$  is empty, and  $P(x)$  is a RAM program that uses random-access read/write memory during the execution. In this case, we avoid the quadratic overhead of converting  $P$  into a circuit.

We also consider *multi-input and multi-hop* variants of RAM-FHE, that allow us to perform computations over many encrypted inputs  $x_i$  that were encrypted separately under the same key, and many plaintext inputs  $y_i$  that were preprocessed separately, and also to use the outputs of prior RAM-FHE evaluations as encrypted inputs for future RAM-FHE evaluations.

We note that an alternative to RAM-FHE would be for the client to run the program  $P$  over her input  $x$  locally and use DEPIR whenever the program wants to read from  $y$  stored on the server. The advantage of RAM-FHE is: (1) using just DEPIR, the round complexity is linear in the program's run-time  $T$ , while RAM-FHE gives the optimal two rounds of interaction, (2) using only DEPIR, the client's run-time is linear in  $T$ , while using RAM-FHE it is sub-linear.

**Prior Work on RAM-FHE.** The work of Hamlin, Holmgren, Weiss and Wichs [HHWW19] introduced RAM-FHE, but only defined a more restricted form, which allows the server to homomorphically evaluate a RAM programs  $P$  over an encrypted input  $x$ , but does not allow for an additional preprocessed plaintext input  $y$  from the server. In particular, it does not capture encrypted queries over a public database (e.g., encrypted Internet search). They gave a candidate construction using generic secret-key DEPIR *and* a heuristic use of ideal obfuscation.<sup>4</sup> The use of ideal obfuscation in [HHWW19] goes far beyond what is needed to go from secret-key to public-key DEPIR, and there is no known techniques to construct RAM-FHE generically from public-key or even unkeyed DEPIR, without an *additional* heuristic use of ideal obfuscation. In summary, it remained an open problem to give any plausibly secure candidate for our notion of RAM-FHE, or to construct the restricted notion of RAM-FHE from [HHWW19] under any concrete assumption without ideal obfuscation, let alone under standard assumptions.

## 1.1 Our Results

**DEPIR.** In this work, we construct the strongest notion of *unkeyed DEPIR* under the standard *ring learning with errors (RingLWE)* [LPR10] assumption with quasi-polynomial approximation factors. This is a well-studied hardness assumption, and is implied by the hardness of finding short vectors in ideal lattices in the worst case. Furthermore, it is the basis of the new NIST standard for public-key encryption [NIS].<sup>5</sup> For any constant  $\varepsilon > 0$ , we get a scheme where, for a database  $\text{DB} \in \{0, 1\}^N$ , the preprocessing run-time and the size of the preprocessed data structure  $\widetilde{\text{DB}}$  is  $O(N^{1+\varepsilon})$ , while the client and server run-times as well as the communication-complexity of each PIR query are just  $\text{poly log}(N)$ . Alternatively, we can also get smaller preprocessing run-time/size of just  $N \cdot 2^{\tilde{O}(\sqrt{\log N})} = N^{1+o(1)}$  at the cost of having slightly larger PIR query run-time/communication of  $2^{\tilde{O}(\sqrt{\log N})} = N^{o(1)}$  and also requiring a slightly stronger form of RingLWE with “sub-sub-exponential” approximation factors. Just like in [CHR17, BIPW17], our PIR protocol consists of just two rounds, where the client’s query fully determines some set of locations  $I$  in the data structure  $\widetilde{\text{DB}}$ , and the server’s response only depends on the bits  $\widetilde{\text{DB}}[I]$  in those locations.

**Updatable DEPIR.** We also construct an *updatable DEPIR* that allows the server to update individual bits of the database  $\text{DB}$  and correspondingly update the preprocessed data structure  $\widetilde{\text{DB}}$  in sub-linear time. In particular, for any constant  $\varepsilon > 0$ , we can achieve update time  $O(N^\varepsilon)$  while preserving preprocessing time/size  $O(N^{1+\varepsilon})$  and PIR query time/communication  $\text{poly log}(N)$ . Alternatively, we can achieve smaller update time  $N^{o(1)}$  and preprocessing time/size  $N^{1+o(1)}$ , at the cost of increasing PIR query time/communication to  $N^{o(1)}$ .

**RAM-FHE.** We show how to use the techniques behind our DEPIR construction to get general RAM-FHE scheme under the RingLWE assumption with quasi-polynomial approximation factors, along with a circular-security assumption. This is the same circular-security assumption that

---

<sup>4</sup>The restricted notion of RAM-FHE from [HHWW19] is not known to even imply public-key DEPIR, whereas our notion implies unkeyed DEPIR as a special case. It appears that the techniques of [HHWW19] may conceivably achieve a stronger notion of RAM-FHE than claimed – one that would be similar to ours, with the caveat that a trusted party is needed to generate a structured common reference string, analogously to footnote 3.

<sup>5</sup>Alternatively, we also mention how to construct unkeyed DEPIR under the approximate-GCD assumption [vGHV10], the NTRU assumption [HPS98, LTV12], or the module LWE assumption [BGV12, LS15] with constant rank.

is needed to get standard (unleveled) FHE from RingLWE [BV11b, BGV12]. For any constant  $\varepsilon > 0$  we get a scheme where, for any bound  $N$ , key generation takes time  $\text{poly log}(N)$ , the encryption of the input  $x \in \{0, 1\}^{\leq N}$  takes time  $O(|x|^{1+\varepsilon})\text{poly log}(N)$ , the preprocessing of  $y \in \{0, 1\}^{\leq N}$  takes time  $O(|y|^{1+\varepsilon})\text{poly log}(N)$ , and the homomorphic evaluation of a RAM program  $P(x, y)$  having worst-case run-time  $T \leq N$  takes time  $T^{1+\varepsilon}\text{poly log}(N)$ . The size of the evaluated ciphertext and its decryption time are  $|P(x, y)|\text{poly log}(N)$ . Alternatively, we can replace all  $\varepsilon$ 's in the exponents by  $o(1)$ , at the cost of replacing all the  $\text{poly log}(N)$  factors by  $N^{o(1)}$  and requiring a slightly stronger form of RingLWE with “sub-sub-exponential” approximation factors.<sup>6</sup> We also achieve a more limited form of *leveled* RAM-FHE without a circular security assumption. Such a scheme requires fixing an upper bound on the run-time  $T$  of the supported programs already during key generation; the run-time of key generation and the size of the public key (which the client needs to send to the server to enable homomorphic evaluation) become  $T \cdot \text{poly log}(N)$ , but all other efficiency measures are preserved. Lastly, our schemes allows for multi-input and multi-hop homomorphic evaluation.

## 1.2 Our Techniques: DEPIR

Our construction of DEPIR departs significantly from the prior works of [CHR17, BIPW17]. The prior works start with a multi-server DEPIR based on Reed-Muller codes. It has the right efficiency, but is completely insecure as a single-server scheme, when one server plays the role of all others. They then propose to patch security by randomly permuting the locations inside the data structure to get a plausibly secure secret-key scheme, which can be upgraded to a public-key scheme via a heuristic use of ideal obfuscation.

We instead start by constructing a basic (single-server) PIR without preprocessing that is secure under standard assumptions, but does not yet have the efficiency properties of DEPIR. We then show how to preprocess the server computation to get the right efficiency, while security remains unaffected. The basic PIR has special structure, where the client’s query consists of some small number of *ring* elements  $\alpha_1, \dots, \alpha_m \in R$ , and the server responds by evaluating a low degree multivariate polynomial  $f_{\text{DB}}(\alpha_1, \dots, \alpha_m)$  over them, where  $f_{\text{DB}}$  depends on the database DB. We construct such a PIR using a special type of *somewhat homomorphic encryption* (SHE) that we call “*algebraic SHE*” (ASHE). In an ASHE, ciphertexts are elements of a ring  $R$ , and to homomorphically evaluate a low-degree multivariate polynomial  $f$  over the plaintexts, one just evaluates the same polynomial  $f$  over the ciphertexts. Using ASHE, we get a PIR where the client encrypts each digit (in some base  $d$ ) of the index  $i \in [N]$  and sends the resulting ciphertext ring elements  $\alpha_1, \dots, \alpha_m$  to the server. The server homomorphically evaluates the function  $f_{\text{DB}}(i) = \text{DB}[i]$ , which can be expressed as a multivariate polynomial in the digits of  $i$ . To homomorphically evaluate  $f_{\text{DB}}$  over the encrypted digits of  $i$ , the server simply evaluates the same polynomial  $f_{\text{DB}}$  over the ciphertext ring elements and sends the response  $\beta = f_{\text{DB}}(\alpha_1, \dots, \alpha_m)$ . We then use preprocessing to get DEPIR efficiency by relying on the beautiful work of Kedlaya and Umans [KU08, KU11], which shows how to preprocess low-degree multivariate polynomials  $f$  over a ring  $R$  into a data structure that can be used to evaluate the polynomial on any future input extremely efficiently, in time sub-linear in the description length of the polynomial. Magically, by combining these ideas and carefully setting the parameters, we get our result!

We fill in the details in the rest of the technical overview. We first review the result of [KU08,

<sup>6</sup>See Theorem 7.7 for a more fine-grained statement and Section 8.1 for alternative parameter trade-offs.

[KU11] on preprocessing polynomials for fast evaluation in Section 1.2.1. We then discuss how to construct the basic PIR using ASHE in Section 1.2.2. Finally, we discuss how to instantiate ASHE from *RingLWE* by adapting the SHE of [BV11b] in Section 1.2.3.

### 1.2.1 Preprocessing Polynomials

Kedlaya and Umans [KU11] show how to preprocess polynomials  $f(X_1, \dots, X_m)$  over a ring  $R$  to create a data structure that allows us to later evaluate the polynomial on any given input  $(\alpha_1, \dots, \alpha_m) \in R^m$  extremely efficiently, in time that is sublinear in the description length of the polynomial. The result works over many types of rings  $R$ , including  $R = \mathbb{Z}_q$  for arbitrary  $q$ , as well as polynomial rings  $R = \mathbb{Z}_q[Y]/(E(Y))$  or even  $R = \mathbb{Z}_q[Y, Z]/(E(Y), E'(Z))$  for monic polynomials  $E, E'$  (and more). Let  $r = |R|$  be the cardinality of the ring, and assume  $f(X_1, \dots, X_m)$  has individual degree  $< d$  in each variable. The description of such a polynomial consists of  $N = d^m$  coefficients and hence evaluating it naively would take at least  $\Omega(N \log r)$  time. The work of [KU11] shows how to preprocess such a polynomial in time  $N \cdot O(m(\log m + \log d + \log \log r))^m \cdot \text{poly}(d, m, \log r)$  into a data structure of at most the above size, such that, on any future input  $(\alpha_1, \dots, \alpha_m) \in R^m$ , we can use the data structure to evaluate  $f(\alpha_1, \dots, \alpha_m)$  in just  $\text{poly}(d, m, \log r)$  time.<sup>7</sup> This gives significant savings, at least in some select parameter ranges. We will specifically be interested in the case where the polynomial  $f$  has individual degree  $< d = \log^c N$  for a constant  $c$ , the number of variables is  $m = \log_d N = \log N / (c \log \log N)$ , and the cardinality of the ring is  $r = 2^{\text{poly}(\lambda, \log N)}$  for security parameter  $\lambda \leq N$ . In this case, for any  $\varepsilon > 0$ , by choosing a sufficiently large  $c$ , we get a preprocessing time/size of  $O(N^{1+\varepsilon})\text{poly}(\lambda)$  and evaluation time  $\text{poly}(\lambda, \log N)$ .

**Technical Sketch.** We include a sketch of the technique of [KU11] for the parameter regime discussed above. Start with the special case of  $R = \mathbb{Z}_q$ . We reinterpret the polynomial  $f(X_1, \dots, X_m)$  over  $\mathbb{Z}_q$  as a polynomial over the integers  $\mathbb{Z}$ , where the coefficients and the inputs are taken from the set  $\{0, \dots, q-1\}$ . The maximal value that this polynomial can take over the integers is  $\leq M := d^m(q-1)^{dm} = 2^{\text{poly}(\lambda, \log N)}$ . Let  $p_1, \dots, p_\ell$  be the set of all primes  $p_i \leq 16 \log M$ , which ensures that  $\prod_{i=1}^\ell p_i \geq M$ . To evaluate the polynomial  $f$  over the integers, it suffices to evaluate it modulo each of the primes  $p_i$  separately and then reconstruct the answer over the integers using the *Chinese Remainder Theorem (CRT)*. So we reduced the problem of evaluating the polynomial  $f$  modulo  $q = 2^{\text{poly}(\lambda, \log N)}$  to that of evaluating it modulo  $p_i$  for a small set of  $\ell = O(\log M) = \text{poly}(\lambda, \log N)$  much smaller primes  $p_i = O(\log M) = \text{poly}(\lambda, \log N)$ .

As a first attempt towards preprocessing the polynomial, consider constructing  $\ell$  tables, where the  $i$ 'th table simply stores all the possible evaluations of the polynomial  $f$  on all  $p_i^m$  possible inputs  $(\alpha_1, \dots, \alpha_m)$  modulo  $p_i$ . This allows for extremely fast evaluation on any input  $(\alpha_1, \dots, \alpha_m) \in \mathbb{Z}_q^m$  just by looking up one entry in each table and then using CRT. Unfortunately, the size of the tables is  $O(\log M)^m = \text{poly}(\lambda, \log N)^{O(\log N / \log \log N)}$ , which is slightly super-polynomial in  $\lambda$ . Instead, we first use the above idea recursively to further reduce the problem of evaluating the polynomial  $f$  modulo each of the primes  $p_i$  to that of evaluating it modulo another list of even smaller primes  $p'_j$  of size  $p'_j = \text{poly}(\log \lambda, \log N) = \text{poly} \log N$ . Amazingly, now the primes are

<sup>7</sup>This is not the most general or optimized form of [KU11], but is the right one for our work. While all the techniques are described in [KU11], the above theorem/parameters are not written explicitly. Therefore, we give a stand-alone proof in Appendix A. Recent work [BGKM22, BGG<sup>+</sup>22] improves on [KU11], but does not seem to benefit our setting.

small enough that we can construct tables with all possible evaluations of the polynomial on all  $(p'_j)^m = \text{poly}(N)$  inputs efficiently! By a more careful analysis, we get the parameters claimed.

The result extends to rings  $R = \mathbb{Z}_q[Y]/(E(Y))$ , by reducing the problem over such rings to that over  $\mathbb{Z}_r$  for some  $r \gg q$  that depends on  $|R|$ . Instead of evaluating  $f(\alpha_1, \dots, \alpha_m)$  over  $R$ , we evaluate it over  $\mathbb{Z}_r$  by taking all the coefficient/input ring elements and substituting  $Y = M$  for some sufficiently large integer  $M \in \mathbb{Z}$  and doing all the computation modulo  $r$  for some sufficiently large  $r \gg M$ , such that there is no wrap-around. The output is an integer whose base- $M$  digits correspond to the coefficients of  $Y$  in the correct evaluation of  $f(\alpha_1, \dots, \alpha_m)$  over  $R$ . We can further extend the result to  $R = \mathbb{Z}_q[Y, Z]/(E(Y), E'(Z))$  (and beyond) analogously.

### 1.2.2 PIR from ASHE

We construct a basic PIR scheme (without preprocessing) where the client query consists of a small number of ring elements, and the server response is computed by evaluating some low-degree multivariate polynomial  $f_{\text{DB}}$  over them. Once we have such a scheme with the right parameters, we can immediately convert it to a DEPIR by applying the preprocessing of [KU11] on the polynomial  $f_{\text{DB}}$ . We construct such basic PIR by using an *algebraic somewhat-homomorphic encryption (ASHE)* scheme, which is a special type of SHE that we define in our work as follows.

**ASHE.** In an ASHE, we can set the plaintext space to be a prime field  $\mathbb{F}_d$  for any prime  $d$ , and the ciphertext space is some ring  $R$ , such that there is a natural way to “lift”  $\mathbb{F}_d$  into a subset of  $R$ . For any polynomial  $f(X_1, \dots, X_m)$  over  $\mathbb{F}_d$  of some a-priori bounded total degree  $< D$ , if we’re given ciphertexts  $\text{ct}_1, \dots, \text{ct}_m \in R$  encrypting plaintext values  $\mu_1, \dots, \mu_m \in \mathbb{F}_d$  respectively, one can homomorphically evaluate  $f$  by simply evaluating a “lifted” version of it over the ciphertexts, resulting in  $\text{ct}^* = f(\text{ct}_1, \dots, \text{ct}_m)$  such that  $\text{ct}^* \in R$  is an encryption of  $f(\mu_1, \dots, \mu_m)$ . The encryption/decryption time and the bit-size of ring elements can be  $\text{poly}(\lambda, d, D)$ , growing polynomially with the security parameter  $\lambda$ , the size of the plaintext space  $\mathbb{F}_d$ , and the total degree  $D$ .

**PIR from ASHE.** To construct PIR from ASHE, the client takes the index  $i \in [N]$  written in base- $d$  as  $i = (i_1, \dots, i_m)$  for  $m = \log_d N$  and encrypts each digit  $i_j \in \mathbb{F}_d$  separately using an ASHE with plaintext space  $\mathbb{F}_d$  to get ciphertexts  $\text{ct}_1, \dots, \text{ct}_m \in R$ .<sup>8</sup> There is a bijection between the  $N = d^m$  coefficients of a polynomial  $f(X_1, \dots, X_m)$  of individual degree  $< d$  in each variable and its evaluation on all  $N = d^m$  possible inputs in  $\mathbb{F}_d^m$ . So there is a unique polynomial  $f_{\text{DB}}(X_1, \dots, X_m)$  such that  $f_{\text{DB}}(i_1, \dots, i_m) = \text{DB}[i]$  for all  $i = (i_1, \dots, i_m) \in [N]$ . Furthermore, we can compute the coefficients of  $f_{\text{DB}}$  given  $\text{DB}$  in quasi-linear time. To give a PIR response, the server needs to homomorphically evaluate the polynomial  $f_{\text{DB}}$  over the encrypted data. In an ASHE scheme, the server does so by simply evaluating the same polynomial (lifted to  $R$ ) over the ciphertexts to compute  $\text{ct}^* = f_{\text{DB}}(\text{ct}_1, \dots, \text{ct}_m)$  in the ring  $R$ . We need to select the parameters carefully to be able use the result of [KU11] to preprocess the polynomial  $f_{\text{DB}}$  to allow for fast online evaluation. Perhaps the most natural first attempt would be to use the binary base with  $d = 2, m = \log N$ . Unfortunately, this parameter choice will not work since the expression for the run-time/size of the preprocessing in [KU11] has a factor of  $m^m = \log N^{\log N} = N^{\log \log N}$  which is super-polynomial! Instead, we use  $d = \log^c N$  for some constant  $c$  and  $m = \log_d N = \log N / (c \log \log N)$ . This matches the parameter regime of [KU11] that we already discussed previously, and yields the claimed efficiency.

<sup>8</sup>Let’s assume that the database size  $N = d^m$  is a power of  $d$ , else we can pad it to make it so.

### 1.2.3 Constructing ASHE

We now turn to the problem of constructing ASHE. “Modern” FHE schemes based on [GSW13] (often called 3rd and 4th generation) rely on non-algebraic operations (e.g., bit-decomposition) at every step, and therefore don’t appear to give us ASHE. Instead, we go back to some of the “older” (1st and 2nd generation) FHE/SHE schemes [vGHV10, BV11b, BGV12, LTV12], which contain ASHE under the hood. In fact, the scheme of [vGHV10] based on approximate GCD directly gives us an extremely simple ASHE over the integers, or equivalently, over  $\mathbb{Z}_q$  for a sufficiently large  $q$ . Similarly, the scheme of [LTV12] based on NTRU directly gives an ASHE over a polynomial ring  $\mathbb{Z}_q[Z]/(Z^n + 1)$ . As our main scheme, we show that (with a little more work), we can get ASHE from just RingLWE via the FHE/SHE of Brakerski and Vaikuntanathan [BV11b].<sup>9</sup> Concretely, we take the most basic symmetric-key SHE of [BV11b] and lightly modify it to handle a plaintext space  $\mathbb{F}_d$  for some prime  $d$ . The scheme works over a ring  $R = \mathbb{Z}_q[Z]/(Z^n + 1)$  where  $q \gg d$  is relatively prime to  $d$ . Let  $\chi$  be an error distribution that samples “small” ring elements.

- The secret key is a random ring element  $s \leftarrow R$ .
- To encrypt  $\mu \in \mathbb{F}_d$ , choose a random  $a \leftarrow R$ , an “error”  $e \leftarrow \chi$  and output  $(a, b = a \cdot s + d \cdot e + \mu)$ .

We can think of a ciphertext  $(a, b)$  as defining a formal linear polynomial  $p_{a,b}(Y) = b - a \cdot Y$  over  $R$ , such that if we evaluate the polynomial on the secret key  $s$ , the result  $p_{a,b}(s) \in R$  is small relative to  $q$  and its constant term is equal to  $\mu$  modulo  $d$ . This allows us to decrypt. Furthermore, by adding and multiplying such ciphertext polynomials, we can add and multiply the corresponding encrypted messages. The degree of the ciphertext as a polynomial over  $R[Y]$  and the magnitude of the error grow with the number of multiplications. To homomorphically evaluate polynomials of total degree  $< D$ , then we need to set  $q = 2^{\text{poly}(D, \log d)}$  to handle the error growth.

At first sight, this does not quite give us an ASHE scheme. For one, the input ciphertexts consist of 2 ring elements  $(a, b) \in R^2$  rather than a single ring element as desired, and output ciphertexts consist of up to  $D$  ring elements. However, we can turn this into an ASHE scheme. Instead of thinking of the ciphertexts as consisting of tuples of elements in  $R$ , we can think of them as elements of the larger ring  $R[Y]/(Y^D + 1) \cong \mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1)$ . Note that modding out by  $(Y^D + 1)$  is only needed to formally make this into a finite ring, but it does not have any affect on the homomorphic evaluation when restricted to polynomials of degree  $< D$ .

In summary, by plugging the above ASHE from RingLWE into our construction of PIR from ASHE, we get a PIR from RingLWE, where the server needs to evaluate the polynomial  $f_{\text{DB}}$  over the ring  $\mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1)$ . By preprocessing  $f_{\text{DB}}$ , we then get DEPIR from RingLWE.

### 1.2.4 Updatable DEPIR

We also construct an *updatable DEPIR* scheme that allows the server to update individual bits of DB and correspondingly update the preprocessed data structure  $\widetilde{\text{DB}}$  in sub-linear time. The construction uses a basic (non-updatable) DEPIR scheme with sufficiently good parameters generically, and is inspired by ideas from [HOWW19, HHWW19], which are in turn inspired by hierarchical ORAM [GO96]. In a nutshell, our updatable DEPIR data structure  $\widetilde{\text{DB}}$  consists of a hierarchy of  $L = \log N$  levels of exponentially increasing size. Each level  $\ell \in \{0, \dots, L\}$  contains a database  $\text{DB}_\ell$

<sup>9</sup>See Appendix C for the alternative construction from approximate GCD or module LWE with a constant rank, which generalizes RingLWE.

consisting of  $2^\ell$  location/value pairs  $(i, b)$  sorted according to the location  $i \in [N]$ , and a preprocessed data structure  $\widetilde{\text{DB}}_\ell$  for  $\text{DB}_\ell$  under the basic DEPIR scheme. Initially, all the data is contained inside the  $N = 2^L$  pairs  $(i, \text{DB}[i])$  in the largest level  $L$ , and the remaining levels are empty. To update  $\text{DB}[i] := b$ , the server first puts the pair  $(i, b)$  in a temporary buffer and, after every  $2^\ell$  updates, the server will take all the pairs  $(i, b)$  contained in the first  $\ell$  levels  $\text{DB}_0, \dots, \text{DB}_\ell$  and in the temporary buffer, and sort them according to location  $i$  into the database  $\text{DB}_\ell$  at level  $\ell$ , taking only the freshest copy from the smallest level if there are conflicting location/value pairs  $(i, b)$  with the same  $i$  at different levels. The server then applies the basic DEPIR-preprocessing to this database  $\text{DB}_\ell$  to create  $\widetilde{\text{DB}}_\ell$ , and empties all the data from levels  $0, \dots, \ell - 1$ . The DEPIR-preprocessed data-structure  $\widetilde{\text{DB}}_\ell$  at every level  $\ell$  allows the client to privately execute arbitrary RAM computation over the data in  $\text{DB}_\ell$  by making a sequence of basic DEPIR queries to the server. Therefore, to query for a location  $i$  in the updatable DEPIR scheme, the client makes a sequence of queries to the basic DEPIR to perform a binary search for the location  $i$  in every level  $\ell \in [L]$  and takes the freshest such tuple  $(i, b)$  found in the smallest level. This gives us a construction where the amortized cost of the updates is as claimed.<sup>10</sup> We can de-amortize this using the same techniques as used by [OS97] in the context of hierarchical ORAM.

A downside of the above updatable DEPIR scheme is that the PIR protocol now requires  $O(\log N)$  rounds of interaction for the client to run binary search by making DEPIR queries. However, we can reduce this to the optimal 2 rounds by using RAM-FHE, discussed next. In particular, RAM-FHE allows the client to send an encrypted index  $i$  to the server, and the server can homomorphically perform binary search over the data  $\text{DB}_\ell$  in each level  $\ell$ , by using random access to the data structure  $\widetilde{\text{DB}}_\ell$  to efficiently derive the encrypted output, without additional interaction.

### 1.3 Our Techniques: RAM-FHE

We extend the ideas behind our DEPIR construction to get a general RAM-FHE scheme that allows us to homomorphically evaluate an arbitrary RAM program  $P(x, y)$  over an encrypted input  $x$ , and a preprocessed plaintext input  $y$ . We will think of  $x, y$  as stored in *read-only* random-access memory, but we also allow for additional *read/write memory*, denoted by  $z$ , that can be used during the execution. Our construction proceeds in stages. We first start with a simpler case, where the program only has random-access to  $y$ , while  $x, z$  can only be accessed in some fixed pre-determined order. This already suffices for many interesting scenarios (e.g., encrypted Internet search) where the encrypted input  $x$  and read/write memory  $z$  are small, and only the plaintext input  $y$  (e.g., the Internet) is large. It also suffices to construct round-optimal updatable DEPIR discussed above. We then augment this solution to allow random-access to  $x$  and finally also to  $z$ .

**Random access to  $y$ .** Our main observation is that we can take the RingLWE-based homomorphic encryption scheme of [BV11b, BGV12] and simultaneously think of it as an FHE scheme or an ASHE scheme depending on need. In particular, we can perform arbitrary computations over encrypted data using general FHE evaluation, by giving up on the algebraic structure of ASHE, or we can also evaluate low-degree multivariate polynomials over the encrypted data using ASHE evaluation, which simply evaluates the polynomial over the ciphertexts. Moreover, we can seamlessly go back and forth: we can take ciphertexts outputted by general FHE evaluation and think

---

<sup>10</sup>Note that the cost of an update depends on the preprocessing time of the underlying basic DEPIR, and therefore we crucially rely on the fact that our preprocessing has low overhead.

of them as ASHE ciphertexts, and we can also take ciphertexts outputted by the ASHE evaluation and translate them back into FHE ciphertexts. We can achieve this using a combination of the re-linearization, modulus-switching and bootstrapping ideas from [BV11b, BGV12]. For bootstrapping, we need an encrypted key cycle, which requires circular security.

Using such hybrid *ASHE-FHE* scheme, we can construct a RAM-FHE with random access to  $y$  as follows. The server preprocesses the long input  $y$  into a data structure  $\tilde{y}$  the same way as in our DEPIR. The client encrypts the input  $x$  using the FHE. To evaluate the program  $P$ , the server starts running the program under FHE. Whenever the program wants to read some location  $i$  of  $y$ , the server derives an FHE encryption of the location  $i$ . This is the same as an ASHE encryption of  $i$ , which is a good query for the index  $i$  in our DEPIR scheme. Therefore, the server simply uses the DEPIR scheme to answer the query by only looking up a few locations inside the data structure  $\tilde{y}$ . This results in a PIR response, which is an ASHE encryption of the memory location  $y[i]$ . The server translates this back into an FHE encryption of  $y[i]$ , which allows it to continue evaluating the program under FHE.

**Random access to  $x$ .** We can extend the above idea to also allow random-access to the encrypted input  $x$ . Instead of directly encrypting  $x$  via the FHE, the client first chooses a key  $k$  for a pseudo-random function (PRF)  $F_k$  and one-time pads  $x$  with the PRF outputs to get  $\bar{x} = (x[i] \oplus F_k(i))_i$ . It then applies the DEPIR preprocessing on  $\bar{x}$  to get a data structure  $\tilde{x}$ . Finally it encrypts the PRF key  $k$  under an FHE scheme to get  $\text{ct}_k$  and sends  $\text{ct}_x = (\tilde{x}, \text{ct}_k)$  to the server as an encryption of  $x$ . The server evaluates the program  $P$  under FHE as before. We already saw how to handle random-access to  $y$ . We can handle random-access to  $x$  similarly. Whenever the program wants to access some location  $i$  of  $x$ , the server derives an FHE encryption of  $i$ , interprets it as a DEPIR query, and uses the data structure  $\tilde{x}$  to answer it by only reading a few locations. This results in an ASHE encryption of  $\bar{x}[i] = x[i] \oplus F_k(i)$ , which can be converted to an FHE encryption. The server then use the FHE ciphertext  $\text{ct}_k$  to remove  $F_k(i)$  under the FHE and get an FHE encryption of  $x[i]$ , which allows it to continue the computation.

**Random-access to read/write memory.** Finally, we show how to allow read/write random access to some memory  $z$ , which initially starts out empty. To do so, we show that the server can maintain some dynamic data structure  $\tilde{z}$  that corresponds to the memory  $z$ . Given an FHE encryption of  $(i, b)$ , there is a way for the server to efficiently perform a “write” operation that updates  $\tilde{z}$  and corresponds to setting  $z[i] := b$ . Given an FHE encryption of  $i$ , there is a way for the server to efficiently perform a “read” operation that outputs an FHE encryption of  $z[i]$  by reading a small number of locations of  $\tilde{z}$ . Essentially, our implementation of  $\tilde{z}$  is the same as updatable DEPIR, but all the pairs  $(i, b)$  stored in  $\text{DB}_\ell$  at each level  $\ell$  are now encrypted under FHE. During a write operation, the server performs all the sorting/merging of levels under FHE. The only subtlety is that, during a “read” operation, the server is running DEPIR queries on top of data that is already FHE encrypted, and therefore an output of the DEPIR query is a “double-FHE encryption” of the data. Here we can rely on the fact that we have an encrypted key cycle, which is anyway needed for FHE, to strip off one layer of FHE encryption and go from double-FHE encryptions to standard FHE encryptions.

## 1.4 Other Related Work

We compare DEPIR to thematically related primitives that allow clients to efficiently access data on a remote server. We focus on the target use-case where a server holds a large public database DB that it wants to make available to a large group of clients who want to privately read the data without revealing the locations to the server.

**ORAM.** *Oblivious RAM* (ORAM) [GO96] allows a client to preprocess some data DB into a dynamic data structure stored on the server, while the client only keeps a short secret key. The client can read (and write) to the database privately, and the data structure on the server is updated in each such operation. The communication and server/client computation for each access is only  $\text{poly} \log(N)$ , and as of recently, we even have solutions that achieve  $O(\log N)$  efficiency [PPRY18, AKL<sup>+</sup>20], which is known to be optimal [LN18, KL21]. The main limitation of ORAM, compared to unkeyed and public-key DEPIR, is that the former only allows a single designated client with a secret key to access the data. If we wanted to use ORAM for our target use-case where many mutually distrustful clients want private access to a common database, we would need to set up a separate data structure that is as large as the entire database for every client in the system. Moreover, this solution would only achieve good amortized efficiency when the client eventually accesses a sufficiently large fraction of the database to amortize the linear cost of the client-specific setup. In contrast, using unkeyed or public-key DEPIR, the database is just preprocessed once and for all, and *any* client can privately access it later at a low cost without any client-specific setup.<sup>11</sup> The main advantage of ORAM over DEPIR is that we have practically optimized constructions of the former under minimal cryptographic assumptions.

**PIR with Client-Side Preprocessing.** Another related primitive is the recent works on PIR with client-side preprocessing [CHK22, ZLTS22] (see also [CK20]). In this setting, the server just stores the original database  $\text{DB} \in \{0, 1\}^N$ . However, for each new client that wants to use the scheme, the server must execute a client-specific preprocessing step in linear  $\tilde{O}(N)$  time that results in the client storing an  $\tilde{O}(\sqrt{N})$ -sized “hint”. Subsequently, the client can run a protocol to retrieve  $\text{DB}[i]$  without revealing the index  $i$  to the server, where the amortized server/client computation during the protocol is  $\tilde{O}(\sqrt{N})$ . The client’s hint is updated during each such protocol execution, requiring the client to be stateful. Unfortunately, it is known that either the hint size or the server work must be at least  $\sqrt{N}$  in any such protocol. Prior work constructed such schemes using fully homomorphic encryption (LWE), or even just linearly homomorphic encryption with somewhat worse parameters. The main limitations of PIR with client-side preprocessing compared to DEPIR are as follows: (1) it requires performing a separate  $\Omega(N)$ -time client-specific preprocessing for each new client before the client can make any queries; as with ORAM this only gives good amortized efficiency when the client eventually accesses a sufficiently large portion of the database to amortize this cost, (2) the clients need to be stateful and maintain a dynamic hint of size  $\Omega(\sqrt{N})$ , (3) the amortized server work per query needs to be at least  $\Omega(\sqrt{N})$ . In contrast, unkeyed DEPIR only requires a universal one-time preprocessing that can be reused for all the clients without any

---

<sup>11</sup>The comparison between ORAM and secret-key DEPIR is more subtle, and the only real advantage of the latter is that the data structure  $\tilde{\text{DB}}$  stored on the server is static rather than dynamic, meaning that the server’s state does not need to be updated after each operation. While this makes the problem of secret-key DEPIR technically challenging, its usefulness over ORAM as a stand-alone primitive is somewhat limited.

client-specific setup, state or keys. Furthermore, the server work per query can be just  $\text{poly} \log(N)$ . On the other hand, the main advantage of PIR with client-side preprocessing is that the server just stores the database DB in the clear, while in DEPIR it stores a larger data structure  $\widetilde{\text{DB}}$ .

**PANDA.** The work of [HOWW19] constructs another related primitive called *private anonymous data access (PANDA)*. It allows a trusted third party to preprocess some database  $\text{DB} \in \{0, 1\}^N$  into a dynamic data structure stored on the server and give individual secret keys to each client in some large group of users. After this setup, each of the clients can privately and anonymously access the database, without revealing which client is performing each access. Security holds as long as server colludes with at most  $t$  of the clients, for some collusion bound  $t$ . The run-time/size of the preprocessing, as well as the run-time of each protocol execution scale similarly in the data size  $N$  as our DEPIR, but both also scale linearly in the collusion bound  $t$  (but not in the total number of clients). The work of [HOWW19] constructed PANDA generically from FHE. Unkeyed (and public-key) DEPIR imply PANDA and therefore the former are strictly stronger primitives. The main limitations of PANDA compared to DEPIR are: (1) it requires a trusted setup, (2) only a designated group of clients with secret keys can access the data, (3) security only holds if the collusion size is bounded by  $t$ , (4) the efficiency scales linearly with  $t$ .

**Multi-Server Solutions.** The work of Beimel, Ishai and Malkin [BIM00] introduced the concept of (unkeyed) DEPIR and gave constructions in the multi-server setting with many non-colluding servers. They left it as an open problem to construct single-server DEPIR, which we resolve in our work.

**Garbled RAM.** We also mention related works on garbled RAM [LO13, GH<sup>L</sup>+14, GLO15] and succinct garbled RAM [GHRW14, BGL<sup>+</sup>15, CHJV15, CH16, CCHR16], which allow a client to outsource RAM computation, potentially over a long private input that the client previously preprocessed. All of these solutions used ORAM under the hood and suffer from the same limitations. In particular, only a designated client can outsource computations over her previously preprocessed data. If we wanted to use these solutions to allow private access to a public database, we would need the server to store a separately preprocessed version of the database DB for each client in the system. Furthermore, all currently known succinct solutions, where the client’s work is sub-linear in the run-time of the computation, rely on the “heavy machinery” of indistinguishability obfuscation.

**Functional Encryption for RAMs.** Two recent works of [JLL22, ACFQ22] study *functional encryption for RAMs (RAM-FE)*. The latter shows how to construct a strong form of RAM-FE, where the decryption time is sublinear in the input, using functional encryption for circuits and public-key DEPIR. Since unkeyed DEPIR is even stronger than public-key DEPIR, we can plug our result to get functional encryption for RAMs from functional encryption for circuits and RingLWE.

## 2 Preliminaries

Define  $\mathbb{N} = \{0, 1, 2, \dots\}$  to be the set of natural numbers,  $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$  to be the set of integers and  $\mathbb{R}$  to be the set of real numbers. For any integer  $n \geq 1$ , define  $[n] = \{1, \dots, n\}$ , and

$\llbracket n \rrbracket = \{0, 1, \dots, n-1\}$ . For an array  $A \in \{0, 1\}^n$ , we index the array from 0, and  $A[i]$  denotes the bit in position  $i \in \llbracket n \rrbracket$ . By default, all our logarithms are base 2 and  $\log n$  stands for  $\log_2 n$ . For any  $q \in \mathbb{N}$ , let  $\mathbb{Z}_q$  be the ring  $\mathbb{Z}/q\mathbb{Z}$ . For a prime  $p$ , let  $\mathbb{F}_p$  be the finite field of order  $p$ . A function  $\nu : \mathbb{N} \rightarrow \mathbb{N}$  is said to be negligible, denoted  $\nu(n) = \text{negl}(n)$ , if for every positive polynomial  $p(\cdot)$  and all sufficiently large  $n$  it holds that  $\nu(n) < 1/p(n)$ . We use the abbreviation PPT for probabilistic polynomial time. For a finite set  $S$ , we write  $a \leftarrow S$  to mean  $a$  is sampled uniformly randomly from  $S$ . For a randomized algorithm  $A$ , we let  $a \leftarrow A(\cdot)$  denote the process of running  $A(\cdot)$  and assigning the outcome to  $a$ ; when  $A$  is deterministic, we write  $a := A(\cdot)$  instead. We denote the security parameter by  $\lambda$ . For two distributions  $X, Y$  parameterized by  $\lambda$  we say that they are computationally indistinguishable, denoted by  $X \approx_c Y$  if for every PPT distinguisher  $D$  we have  $|\Pr[D(X) = 1] - \Pr[D(Y) = 1]| = \text{negl}(\lambda)$ .

## 2.1 Multi-variate Polynomial Evaluation and Interpolation

For any ring  $R$ , let  $R[X_1, \dots, X_m]$  denote the ring of polynomials with coefficients in  $R$  and symbolic variables  $X_1, \dots, X_m$ . For any polynomial  $f(X_1, \dots, X_m) \in R[X_1, \dots, X_m]$  and any  $\alpha = (\alpha_1, \dots, \alpha_m) \in R^m$ ,  $f(\alpha) \in R$  is the evaluation of  $f$  on  $\alpha$ .

**Polynomial Evaluation with Preprocessing.** We rely on a result of Kedlaya and Umans [KU11], which shows how to preprocess a multivariate polynomial  $f$  into a static data structure such that, for any input  $\alpha$  given later, we can use the data structure to evaluate  $f(\alpha)$  quickly, in time that is sublinear in the description-length of the polynomial. In this work, we will rely on the result for multivariate polynomials  $f$  over rings of the form  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for some  $q \in \mathbb{N}$  and arbitrary (non-constant) monic polynomials  $E_1, E_2$ . Note that this includes the rings  $\mathbb{Z}_q$  and  $\mathbb{Z}_q[Y]/(E_1(Y))$  as a special case.

**Theorem 2.1** (Preprocessing Polynomials [KU11]). *Let  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for some  $q \in \mathbb{N}$  and arbitrary monic polynomials  $E_1$  over  $Y$  and  $E_2$  over  $Z$  with degrees  $e_1, e_2 > 0$  respectively. Let  $f \in R[X_1, X_2, \dots, X_m]$  be a polynomial of individual degree  $< d$  in every variable. Then, there is some preprocessing algorithm that takes the coefficients of  $f$  as an input, runs in time*

$$S = d^m \cdot \text{poly}(m, d, \log |R|) \cdot O(m(\log m + \log d + \log \log |R|))^m$$

*and outputs a data structure of size at most  $S$ , and there is some evaluation algorithm with random access to the data structure, that is given an evaluation point  $\alpha \in R^m$  and computes  $f(\alpha)$  in time*

$$\text{poly}(d, m, \log |R|).$$

While the ideas behind the above theorem are discussed in [KU11], the concrete statement with clear parameters is not stated explicitly.<sup>12</sup> Therefore, for completeness, we give a stand-alone proof of the above result in Appendix A.

<sup>12</sup>The relevant material appears in [KU11, Section 4], which focuses on algorithms for multipoint evaluation of a polynomial  $f$ . Implicitly, all of these algorithms first preprocess the polynomial into a data structure as in our theorem and then use the data structure to evaluate the polynomial on all the points. The results there are stated for rings of the form  $R = \mathbb{Z}_q$  or  $R = \mathbb{Z}_q[Y]/(E(Y))$ , and it is informally noted that the results extend to other rings. The fact that these algorithms can be seen as preprocessing polynomials into a data structure that allows for fast evaluation is discussed, but is only stated explicitly in [KU11, Section 5] for the (harder) case of high-degree univariate polynomials, which then reduces to our case of low-degree multivariate polynomials.

**Polynomial Interpolation.** Let  $\mathbb{F}_d$  be a field of prime order  $d$  and let  $m \in \mathbb{N}$  be an integer. Given any  $d^m$  values  $y_{(x_1, \dots, x_m)} \in \mathbb{F}_d$  for all  $(x_1, \dots, x_m) \in \mathbb{F}_d^m$  there is a unique  $m$ -variate polynomial  $f(X_1, \dots, X_m) \in \mathbb{F}_d[X_1, \dots, X_m]$  having individual degree  $< d$  in each variable<sup>13</sup> such that  $f(x_1, \dots, x_m) = y_{(x_1, \dots, x_m)}$  for all  $(x_1, \dots, x_m) \in \mathbb{F}_d^m$ . This follows since there is a 1-to-1 mapping from the  $d^m$  coefficients of such polynomials to the  $d^m$  evaluations on all inputs. Furthermore, it is possible to “interpolate” the coefficients of the polynomial  $f(X_1, \dots, X_m)$  given the  $d^m$  evaluation in quasi-linear time. We were unable to find a proof of this presumably known result in the literature, and therefore we give a simple proof in Appendix B. The algorithm is recursive and uses single-variate polynomial interpolation in the base case.

**Lemma 2.2** (Multi-variate polynomial interpolation). *Let  $\mathbb{F}_d$  be a field of prime order  $d$ , and let  $m \in \mathbb{N}$  an integer. Let  $\{y_{(x_1, \dots, x_m)} \in \mathbb{F}_d\}_{(x_1, \dots, x_m) \in \mathbb{F}_d^m}$  be any set of  $d^m$  values. Then there is an algorithm that runs in quasi-linear time  $O(d^m \cdot m \cdot \text{poly} \log d)$  and recovers the coefficients of a polynomial  $f(X_1, \dots, X_m) \in \mathbb{F}_d[X_1, \dots, X_m]$  with individual degree  $< d$  in each variable such that  $f(x_1, \dots, x_m) = y_{(x_1, \dots, x_m)}$  for all  $(x_1, \dots, x_m) \in \mathbb{F}_d^m$ .*

## 2.2 Ring LWE

The “learning with errors over rings” problem (RingLWE) was introduced by Lyubashevsky, Peikert, and Regev [LPR10], following a long line of earlier work constructing cryptosystems using ideal lattices e.g., [HPS98, Mic02, PR06, LM06].

**Norm in a Ring.** Let  $R = \mathbb{Z}_q[Z]/(Z^n + 1)$ , and let  $a = \sum_{i=0}^{n-1} a_i Z^i \in R$ . We define the norm of  $a$  via  $\|a\|_\infty \stackrel{\text{def}}{=} \max |a_i|$ , where we identify  $a_i \in \mathbb{Z}_q$  with its integer representative in the range  $(-q/2, \dots, q/2]$ . A distribution  $\chi$  over  $R$  is  $\beta$ -bounded if  $\Pr[\|e\|_\infty \leq \beta : e \leftarrow \chi] = 1$ .

**Definition 2.3** (The RingLWE assumption [LPR10]). *Let  $n = n(\lambda) \in \mathbb{Z}$ , and  $q = q(\lambda) \in \mathbb{Z}$  be integers. Define the ring  $R = \mathbb{Z}_q[Z]/(Z^n + 1)$ , and let  $\chi = \chi(\lambda)$  denote an error distribution over the ring  $R$ . The RingLWE $_{n,q,\chi}$  assumption, states that for any  $\ell = \text{poly}(\lambda)$  it holds that*

$$\{(a_i, a_i \cdot s + e_i)\}_{i \in [\ell]} \approx_c \{(a_i, u_i)\}_{i \in [\ell]},$$

where  $s \leftarrow R$ ,  $a_i \leftarrow R$ ,  $e_i \leftarrow \chi$ , and  $u_i \leftarrow R$ .

As shown by Lyubashevsky, Peikert, and Regev [LPR10], the RingLWE assumption with some  $\beta$ -bounded error distribution  $\chi$  is implied by the worst-case hardness of the approximate shortest vector problem in an ideal lattice with approximation factor  $\approx q/\beta$ .

**Theorem 2.4** (Lyubashevsky-Peikert-Regev [LPR10, BLP<sup>+</sup>13]). *For any  $n$  that is a power of 2, ring  $R = \mathbb{Z}[Z]/(Z^n + 1)$ , prime integer  $q = q(n) = 1 \pmod n$ , and  $\beta = \omega(\sqrt{n \log n})$ , there is an efficiently samplable  $\beta$ -bounded distribution  $\chi$  over  $R$ , such that is a quantum reduction from the  $n^{\omega(1)} \cdot (q/\beta)$ -approximate worst-case SVP in ideal lattices over  $R$  to RingLWE $_{n,q,\chi}$ , where the reduction runs in time  $\text{poly}(n, q)$ .*

Later work of [BLP<sup>+</sup>13, PRS17] further generalized this to any (not necessarily prime) modulus  $q$  and also to other choices of the ring.

<sup>13</sup>This is without loss of generality since  $X^d = X$  in  $\mathbb{F}_d$ .

We let *RingLWE with sub-exponential approximation factors* refer to the assumption that, given any security parameter  $\lambda$  and any *gap parameter*  $t$ , we can, in  $\text{poly}(\lambda, t)$  deterministic time, find parameters:  $n = \text{poly}(\lambda, t)$ ,  $\beta = \text{poly}(\lambda, t)$ , a  $\beta$ -bounded distribution  $\chi$  that is efficiently samplable in  $\text{poly}(\lambda, t)$  time, and  $q = \lambda^{\text{poly}(t)}$  with  $q > (2\beta n)^t$ , such that for any  $t(\lambda) = \text{poly}(\lambda)$  the corresponding  $\text{RingLWE}_{n,q,\chi}$  assumption holds. Similarly, we let *RingLWE with sub-sub-exponential approximation factors* refers to the same assumption except with any  $t(\lambda) = \lambda^{o(1)}$ , and *RingLWE with quasi-polynomial approximation factors* refers to the same assumption except with any  $t(\lambda) = \text{poly} \log(\lambda)$ .

The RingLWE assumption with sub-exponential approximation factors is implied by the worst-case sub-exponential  $2^{n^\varepsilon}$  quantum hardness of the approximate shortest vector problem in an ideal lattice with sub-exponential approximation factors  $2^{n^\varepsilon}$  for any  $\varepsilon > 0$ . Similarly, the RingLWE assumption with quasi-polynomial (resp. sub-sub-exponential) approximation factors is implied by the  $2^{\text{poly} \log n}$  (resp.  $2^{n^{o(1)}}$ ) quantum hardness of the approximate shortest vector problem in an ideal lattice with approximation factors  $2^{\text{poly} \log n}$  (resp.  $2^{n^{o(1)}}$ ).

The  $\text{RingLWE}_{n,q,\chi}$  assumption is known to be equivalent to a *hermite normal form* variant where we sample  $s \leftarrow \chi$  from the error distribution rather than uniformly at random from the ring [ACPS09, LPR10]. It is also equivalent to a *scaled error* variant where instead of adding the errors  $e_i \leftarrow \chi$  we add  $d \cdot e_i$  for some integer  $d$  relatively prime to  $q$  [BV11b].

### 3 Algebraic Somewhat Homomorphic Encryption (ASHE)

This section defines *algebraic somewhat homomorphic encryption (ASHE)* and shows how to cast the SHE scheme of Brakerski and Vaikuntanathan [BV11b] based on RingLWE as an ASHE. See Appendix C for alternative constructions from the approximate GCD assumption or the mouldle LWE assumption with constant rank.

An ASHE is a symmetric-key CPA-secure encryption scheme, where the plaintext space is  $\mathbb{Z}_d$  for a prime  $d$  of our choosing, and the ciphertext space is some corresponding polynomial ring  $R$ . An ASHE allows us to homomorphically evaluate low-degree multivariate polynomials  $f$  over encrypted data, just by evaluating the same polynomial (appropriately lifted) over the ciphertexts.

**Definition 3.1 (ASHE).** *An algebraic somewhat homomorphic encryption scheme (ASHE) is a tuple of PPT algorithms (Setup, Gen, Enc, Dec, Lift) with the following syntax:*

- $\text{params} := \text{Setup}(1^\lambda, 1^d, 1^D, N)$ : On input a security parameter  $\lambda$ , total degree  $D$ , number of terms  $N$ , plaintext space  $d$ , it deterministically fixes some public parameters  $\text{params}$ , which implicitly define a ring  $R$  of the form  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for some  $q \in \mathbb{N}$  and non-constant monic polynomials  $E_1, E_2$ .<sup>14</sup> All other algorithms, Gen, Enc, Dec, Lift, implicitly take  $\text{params}$  as input even when not explicitly stated.
- $s \leftarrow \text{Gen}(\text{params})$ : Output a secret key  $s$ .
- $\text{ct} \leftarrow \text{Enc}(s, \mu)$ : Given a secret key  $s$  and a message  $\mu \in \mathbb{Z}_d$ , outputs a ciphertext  $\text{ct} \in R$ .
- $\mu := \text{Dec}(s, \text{ct})$ : Given a secret key  $s$  and a ciphertext  $\text{ct} \in R$ , outputs a message  $\mu \in \mathbb{Z}_d$ .

<sup>14</sup> We restrict to these rings to match the rings for which we have fast polynomial evaluation with preprocessing (Theorem 2.1). As noted, this can be generalized further; for example, to rings  $R = \mathbb{Z}_q[Y_1, \dots, Y_t]/(E_1(Y), \dots, E_t(Y))$  for some monic non-constant polynomials  $E_1, \dots, E_t$  with  $t = O(1)$  (see Remark A.1).

- $\bar{\mu} := \text{Lift}(\mu)$ : Lifts  $\mu \in \mathbb{Z}_d$  to  $\bar{\mu} \in R$ . For any polynomial  $f$  over  $\mathbb{Z}_d$ , we let  $\bar{f} := \text{Lift}(f)$  denote the analogous polynomial over  $R$  derived by applying  $\text{Lift}$  to every coefficient of  $f$ .

We require that the scheme satisfies the following properties.

**Correctness:** We require that for all plaintexts  $\mu_1, \dots, \mu_m \in \mathbb{Z}_d$  and for any polynomial  $f(X_1, \dots, X_m)$  over  $\mathbb{Z}_d$  consisting of at most  $N$  terms and total degree  $< D$ , it holds that

$$\Pr \left[ \begin{array}{l} \text{Dec}(s, \text{ct}') = f(\mu_1, \dots, \mu_m) : \\ \text{params} := \text{Setup}(1^\lambda, 1^d, 1^D, N) \\ s \leftarrow \text{Gen}(\text{params}) \\ \text{ct}'_j \leftarrow \text{Enc}(s, \mu_j) \text{ for all } j \in [m] \\ \bar{f} := \text{Lift}(f) \\ \text{ct}' := \bar{f}(\text{ct}'_1, \dots, \text{ct}'_m) \end{array} \right] = 1.$$

**Security:** We require the standard symmetric-key IND-CPA security for the encryption scheme  $(\text{Gen}, \text{Enc}, \text{Dec})$  when  $\text{params} := \text{Setup}(1^\lambda, 1^d, 1^D, N)$  for any  $N = \text{poly}(\lambda)$ ,  $d = \text{poly}(\lambda)$ ,  $D = \text{poly}(\lambda)$ .

We also define weaker notions of ASHE for polylogarithmic degree where we only require the above to hold for  $d = \text{poly} \log(\lambda)$ ,  $D = \text{poly} \log(\lambda)$ . Analogously, we define ASHE for sub-polynomial degree with  $d = \lambda^{o(1)}$ ,  $D = \lambda^{o(1)}$ .

**Efficiency:** We require that the description length of ring elements, the run-time of the ring operations, and the run-time of  $\text{Setup}$ ,  $\text{Gen}$ ,  $\text{Enc}$ ,  $\text{Dec}$ ,  $\text{Lift}$  are all bounded by  $\text{poly}(\lambda, D, d, \log N)$ .

### 3.1 ASHE from RingLWE

Our ASHE construction is essentially the same as the somewhat homomorphic encryption (SHE) scheme of Brakerski and Vaikuntanathan [BV11b, Theorem 2], which is based on the RingLWE assumption. Their scheme works over a ring  $Q = \mathbb{Z}_q[Z]/(Z^n + 1)$ , and ciphertexts are polynomials in  $Q[Y]$  of degree  $< D$ . We observe that we can equivalently interpret ciphertexts as elements of the ring  $R$  defined as  $R = Q[Y]/(Y^D + 1) \cong \mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1)$ .

**Construction.** For the construction of ASHE, we will identify elements of  $\mathbb{Z}_q$  with their representative in the range  $(-q/2, \dots, q/2] \cap \mathbb{Z}$ , and similarly for  $\mathbb{Z}_d$ . This allows us to *reinterpret* an element  $\mu \in \mathbb{Z}_d$  as an element of  $\mathbb{Z}_q$  by taking the representative of  $\mu$  and reducing it modulo  $q$  (and vice versa). Similarly, we can naturally reinterpret elements  $\mu \in \mathbb{Z}_d$  as elements of  $Q = \mathbb{Z}_q[Z]/(Z^n + 1)$  or  $R = \mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1)$  by first reinterpreting  $\mu$  as an element of  $\mathbb{Z}_q$  and then interpreting it as a constant polynomial in  $Q$  or  $R$  respectively. The scheme is described as follows:

$\text{params} := \text{Setup}(1^\lambda, 1^d, 1^D, N)$ : Set the gap parameter  $t := D \log d + \log N + \log d + 1$  and choose  $n = \text{poly}(\lambda, t)$ ,  $q = \lambda^{\text{poly}(t)}$  and a  $\beta$ -bounded error distribution  $\chi$  as in Section 2.2 so that  $q > (2\beta n)^t > 2Nd(d(\beta + 1)n)^D$  and  $q$  is relatively prime to  $d$ . Define the rings

$$Q := \mathbb{Z}_q[Z]/(Z^n + 1), \quad R := Q[Y]/(Y^D + 1) \cong \mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1).$$

$s \leftarrow \text{Gen}(1^\lambda)$ : Sample  $s \leftarrow Q$  uniformly at random.

$\text{ct} \leftarrow \text{Enc}(s, \mu)$ : Reinterpret  $\mu \in \mathbb{Z}_d$  as an element of  $Q$ . Sample  $a \leftarrow Q, e \leftarrow \chi$ . Let

$$b = a \cdot s + d \cdot e + \mu \in Q.$$

Define  $\text{ct} \in R$  as the formal polynomial with a symbolic variable  $Y$  via:

$$\text{ct}(Y) = -a \cdot Y + b$$

$\mu := \text{Dec}(s, \text{ct})$ : Interpret  $\text{ct} \in R$  as a formal polynomial  $\text{ct}(Y) \in Q[Y]/(Y^D + 1)$  and compute  $g = \text{ct}(s) \in Q$  to be its evaluation on  $s \in Q$ . Interpret  $g \in Q$  as a formal polynomial  $g(Z) \in \mathbb{Z}_q[Z]/(Z^n + 1)$  and let  $h = g(0) \in \mathbb{Z}_q$  be its constant term. Reinterpret  $h$  as an element of  $\mathbb{Z}_d$  and output it.

$\bar{\mu} := \text{Lift}(\mu)$ : Reinterpret  $\mu \in \mathbb{Z}_d$  as an element of  $R$ .

**Theorem 3.2** (ASHE from RingLWE). *The above scheme is an ASHE under RingLWE with sub-exponential approximation factors. Alternately, it is an ASHE for poly-logarithmic (resp. sub-polynomial) degree under RingLWE with quasi-polynomial (resp. sub-sub-exponential) approximation factors.*

*Proof.* We argue that the scheme satisfies the correctness, security and efficiency properties of Definition 3.1.

*Correctness:* Let  $f$  be any polynomial of total degree  $< D$  over  $\mathbb{Z}_d$  and let  $\bar{f} = \text{Lift}(f)$ . Fresh encryptions  $\text{ct}_i = \text{ct}_i(Y)$  outputted by Enc are degree-1 polynomials in  $Y$ . Therefore  $\text{ct}' = \bar{f}(\text{ct}_1, \dots, \text{ct}_m)$  is the same whether we do the computation over  $R = Q[Y]/(Y^D + 1)$  or simply over  $Q[Y]$ , since in the latter case,  $\text{ct}'(Y)$  has degree  $< D$  in  $Y$  and hence modding out by  $(Y^D + 1)$  does not do anything. We therefore analyze correctness in the latter case where all operations are done in  $Q[Y]$ .

We say that a ciphertext  $\text{ct}(Y)$  is an  $\gamma$  noisy encryption of  $\mu \in \mathbb{Z}_d$  if  $\|\text{ct}(s)\| \leq \gamma$  and  $\text{Dec}(s, \text{ct}) = \mu$ . A freshly encrypted ciphertext  $\text{ct}(Y)$  of a message  $\mu$  is  $\gamma$ -noisy for  $\gamma = d(\beta + 1)$  as long as  $\gamma < q/2$ . Assume  $\text{ct}_1, \text{ct}_2$  are  $\gamma_1, \gamma_2$  noisy encryptions of  $\mu_1, \mu_2$  respectively. Then  $\text{ct}_1 + \text{ct}_2$  is a  $\gamma_+ = (\gamma_1 + \gamma_2)$  noisy encryption of  $\mu_1 + \mu_2$  as long as  $\gamma_+ < q/2$  since

$$(\text{ct}_1 + \text{ct}_2)(s) = \text{ct}_1(s) + \text{ct}_2(s) = (de_1 + \mu_1) + (de_2 + \mu_2) = d(e_1 + e_2) + (\mu_1 + \mu_2)$$

Similarly,  $\text{ct}_1 \cdot \text{ct}_2$  is a  $\gamma_\times = n\gamma_1\gamma_2$  noisy encryption of  $\mu_1 \cdot \mu_2$  as long as  $\gamma_\times < q/2$  since

$$(\text{ct}_1 \cdot \text{ct}_2)(s) = \text{ct}_1(s) \cdot \text{ct}_2(s) = (d \cdot e_1 + \mu_1) \cdot (d \cdot e_2 + \mu_2) = d(de_1e_2 + e_1\mu_2 + e_2\mu_1) + \mu_1\mu_2.$$

For the above, we rely on the fact that for  $a, b \in \mathbb{Z}_q[Z]/(Z^n + 1)$  with  $\|a\| \leq \gamma_a, \|b\| \leq \gamma_b$ , we have  $\|a \cdot b\| \leq n\gamma_a\gamma_b$ . Lastly, if  $a \in \mathbb{Z}_d$  is a constant then  $a \cdot \text{ct}_1$  is a  $\gamma_c = d\gamma_1$  noisy encryption of  $a\mu_1$  as long as  $\gamma_c < q/2$ . Therefore, if  $\text{ct}_i$  are fresh encryptions of  $\mu_i$  then  $\text{ct}' = \bar{f}(\text{ct}_1, \dots, \text{ct}_m)$  is a  $\gamma_{\bar{f}} = N \cdot d(dn(\beta + 1))^D$  noisy encryption of  $f(\mu_1 \dots, \mu_m)$ , as long as  $\gamma_{\bar{f}} < q/2$ , which is ensured by our choice of parameters. This means that  $\text{Dec}(\text{sk}, \text{ct}') = f(\mu_1 \dots, \mu_m)$  as desired.

*Security:* Security follows directly from the (scaled error variant of the) RingLWE assumption. In particular, the ciphertexts consist of  $a_i \leftarrow Q, b_i = a_i \cdot s + d \cdot e_i + \mu_i$  which is indistinguishable from uniform under RingLWE. In general, we can bound  $t = D \log d + \log N + \log d + 1 = \text{poly}(\lambda)$  and therefore we need to rely on RingLWE with sub-exponential approximation factors. However, in the case of ASHE for sub-polynomial (resp. polylogarithmic) degree, we can bound  $t = \lambda^{o(1)}$  (resp.  $t = \text{poly} \log \lambda$ ) and therefore we only need to rely on RingLWE with sub-sub-exponential (resp. quasi-polynomial) approximation factors.

*Efficiency:* By the definition of RingLWE, we can determine the parameters  $n, q, \chi$  in  $\text{poly}(\lambda, t) = \text{poly}(\lambda, D, \log d, \log N)$  time, which also bounds the efficiency of sampling from  $\chi$ . We have  $q = \lambda^{\text{poly}(t)} = \lambda^{\text{poly}(D, \log d, \log N)}$  and  $n = \text{poly}(\lambda, t) = \text{poly}(\lambda, D, \log d, \log N)$ . The cardinality of the ring  $R$  is  $q^{Dn}$  and therefore the ring elements have description length  $Dn \log q = \text{poly}(\lambda, D, \log d, \log N)$  bits. The run-time of ring operations and the algorithms Setup, Gen, Enc, Dec is therefore also bounded by  $\text{poly}(\lambda, D, \log d, \log N)$ .<sup>15</sup>  $\square$

## 4 DEPIR

In this section, we formally define *doubly efficient private information retrieval (DEPIR)* and give a generic construction from ASHE, via fast polynomial evaluation with preprocessing.

### 4.1 Definition

At high level, an (unkeyed) DEPIR scheme is a protocol between a Server and an arbitrary Client. The protocol consists of four algorithms, Prep, Query, Resp, Dec, and the algorithms are performed in two phases of the protocol, preprocessing and query, illustrated as follows.

**Preprocessing.** Server has a database  $\text{DB} \in \{0, 1\}^N$  and runs the deterministic preprocessing algorithm  $\widetilde{\text{DB}} := \text{Prep}(1^\lambda, \text{DB})$ . It stores the static data structure  $\widetilde{\text{DB}}$  in random-access memory.

**Query.** Client knows the database size  $N$ , has index  $i \in \llbracket N \rrbracket$  and wants to learn the entry  $\text{DB}[i]$  without revealing  $i$ .

1. Client runs  $(\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N, i)$  to generate query ciphertext  $\text{ct}$  that it sends to Server, and a query-specific secret decoding key  $s$  that it keeps locally.
2. Server responds with the answer  $\text{ans} \leftarrow \text{Resp}(\widetilde{\text{DB}}, \text{ct})$  using random-access to the data structure  $\widetilde{\text{DB}}$ .
3. Client decodes the answer using the algorithm  $b := \text{Dec}(s, \text{ans})$  to learn the bit  $b = \text{DB}[i]$ .

We next describe the definition formally. Our definition is based on those of Boyle et al. [BIPW17, Definition 3.5] and Canetti et al. [CHR17, Definition 1]. The main difference is that our setting is *unkeyed*: the preprocessing algorithm Prep is deterministic and does not require any secret coins, nor does it output any keys needed by clients to run queries in the future. Therefore, the Server can perform the preprocessing on its own, and there is no need for a trusted third party to do it.

**Definition 4.1 (DEPIR).** *An (unkeyed) doubly efficient private information retrieval (DEPIR) is a tuple of algorithms (Prep, Query, Resp, Dec) with the following syntax.*

- $\widetilde{\text{DB}} := \text{Prep}(1^\lambda, \text{DB})$  takes the security parameter  $1^\lambda$  and a database  $\text{DB} \in \{0, 1\}^N$ , and deterministically outputs a preprocessed database  $\widetilde{\text{DB}}$ .
- $(\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N, i)$  takes the security parameter  $1^\lambda$  a database size  $N$ , and index  $i \in \llbracket N \rrbracket$ , and it outputs a query ciphertext  $\text{ct}$  and a query-specific decoding key  $s$ .

<sup>15</sup>Note that the definition of ASHE allows efficiency to scale polynomially with  $d$ , while the above construction “overachieves” by only scaling polylogarithmically with  $d$ .

- $\text{ans} := \text{Resp}(\widetilde{\text{DB}}, \text{ct})$  takes the preprocessed database  $\widetilde{\text{DB}}$  stored in random-access memory and a query  $\text{ct}$ , and it responds with an answer  $\text{ans}$ .
- $b := \text{Dec}(s, \text{ans})$  takes the query-specific decoding key  $s$  and the answer  $\text{ans}$ , and it outputs a decoded bit  $b \in \{0, 1\}$ .

The algorithms  $(\text{Prep}, \text{Query}, \text{Resp}, \text{Dec})$  should satisfy the following correctness, security, and efficiency:

**Correctness:** Honest execution of  $\text{Prep}, \text{Query}, \text{Resp}$ , and  $\text{Dec}$  successfully recovers requested data items with probability 1. That is, for every  $\text{DB} \in \{0, 1\}^N$  and every  $i \in \llbracket N \rrbracket$ , it holds that

$$\Pr \left[ \begin{array}{l} \widetilde{\text{DB}} := \text{Prep}(1^\lambda, \text{DB}); \\ \text{Dec}(s, \text{ans}) = \text{DB}[i] : (\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N, i); \\ \text{ans} := \text{Resp}(\widetilde{\text{DB}}, \text{ct}) \end{array} \right] = 1.$$

**Security:** No efficient adversary can distinguish the queries output by  $\text{Query}$  on input index  $i_0$  and  $i_1$ . Namely, we define the following game between a challenger and an adversary  $\mathcal{A}$ :

1.  $(i_0, i_1, 1^N, \text{aux}) \leftarrow \mathcal{A}(1^\lambda)$ :  $\mathcal{A}$  selects a size  $N$ , a challenge index pair  $i_0, i_1 \in \llbracket N \rrbracket$ , and auxiliary information  $\text{aux}$ .
2.  $b \leftarrow \{0, 1\}; (s, \text{ct}) \leftarrow \text{Query}(1^\lambda, N, i_b)$ : The challenger selects a random bit  $b$  and generates a sample query  $\text{ct}$  for the chosen index  $i_b$ .
3.  $b' \leftarrow \mathcal{A}(\text{aux}, \text{ct})$ :  $\mathcal{A}$  outputs a guess for  $b$ , given the query  $\text{ct}$ .

We require that for every PPT adversary  $\mathcal{A}$ , there exists a negligible function  $\text{negl}$  such that the distinguishing advantage of  $\mathcal{A}$  in the above security game is  $|\Pr[b' = b] - 1/2| \leq \text{negl}(\lambda)$ .

**Efficiency:** Suppose that  $\text{Resp}$  is given random accesses to  $\widetilde{\text{DB}}$ . We say the scheme  $(\text{Prep}, \text{Query}, \text{Resp}, \text{Dec})$  is doubly efficient if  $\text{Prep}$  runs in time  $\text{poly}(\lambda, N)$ , and  $\text{Query}, \text{Resp}, \text{Dec}$  run in time sublinear in  $N$ . Ideally, we want  $\text{Prep}$  to run in quasilinear time in  $N$ , and  $\text{Query}, \text{Resp}, \text{Dec}$  run in polylogarithmic time in  $N$ .

We remark that the security game does not talk about the preprocessing step  $\text{Prep}$  at all, and that this step is only needed for efficiency/correctness. When the client issues a query, its security is guaranteed no matter what the server does, even if it does not preprocess the database correctly. This is contrast to secret-key or even public-key DEPIR, where the client gets some key associated with the preprocessed database and it is essential that this key is generated honestly. Also, we remark that the above security game explicitly defines security for only one query, but the security generically extends to any polynomial number of queries via a straightforward hybrid argument.

Our syntax is slightly different from the “passive” syntax of Boyle et al. [BIPW17, Definition 3.5], which is more restrictive. Their  $\text{Query}$  algorithm outputs a list of indices  $I$ , and the server passively responds  $\widetilde{\text{DB}}[I]$  denoting the entries of  $\widetilde{\text{DB}}$  at locations  $I$ . In our syntax (and also that of Canetti et al.), the server’s response  $\text{Resp}$  may perform arbitrary computation using  $\widetilde{\text{DB}}$ . In particular, our syntax allows  $\text{Resp}$  to read various locations of  $\widetilde{\text{DB}}$  adaptively for many rounds, where the choice of each location may depend on the contents of the locations read so far. However, our actual construction (presented later) is compatible with the passive syntax: the query ciphertext  $\text{ct}$  fully determines a non-adaptive list of indices  $I$  and  $\text{Resp}$  only accesses entries of  $\widetilde{\text{DB}}[I]$ .

**Definition 4.2** (Efficiency metrics). *For any DEPIR scheme (Prep, Query, Resp, Dec), the preprocessing time is the running time of Prep, the server storage is the output size of Prep, the query time denotes the total time of (Query, Resp, Dec), and the communication is the total output size of Query and Resp.*

## 4.2 Construction

In this subsection, we construct a DEPIR from an ASHE. The construction relies on some parameters  $d, m$  such that  $d$  is prime and  $d^m > N$ ; we discuss how to set them later. The construction consists of 3 steps that carefully fit together:

- Express the function  $f_{\text{DB}}(i) = \text{DB}[i]$  as an  $m$ -variate polynomial of individual degree  $< d$  over  $\mathbb{F}_d$ , where the inputs are the base- $d$  digits of the index  $i$ .
- Construct a basic PIR scheme by using ASHE to homomorphically evaluate the polynomial  $f_{\text{DB}}$  over the encryptions of the base- $d$  digits of  $i$ ; the ciphertext consists of  $m$  ring elements and the server computation consists of evaluating the “lifted” polynomial  $\bar{f}_{\text{DB}}$  over them.
- Preprocess the polynomial  $\bar{f}_{\text{DB}}$  into a data structure  $\widetilde{\text{DB}}$  that enables fast online evaluation using Theorem 2.1.

We first discuss each of the steps, introducing some useful notation and claims along the way. We then formally describe the entire DEPIR scheme by combining all 3 steps.

**Encoding a database as a polynomial.** For  $i \in \llbracket N \rrbracket$  let  $(i_1, \dots, i_m) = \text{base}_{d,m}(i)$  be the based- $d$  representation of  $i$  such that  $i = \sum_{j=1}^m i_j \cdot d^{j-1}$  with  $i_j \in \llbracket d \rrbracket$ . The following claim says that we can encode the database  $\text{DB}$  as an  $m$ -variate polynomial  $f_{\text{DB}}$  over  $\mathbb{F}_d$  such that  $f_{\text{DB}}(\text{base}_{d,m}(i)) = \text{DB}[i]$ , and moreover we can find the coefficient representation of this polynomial efficiently in quasi-linear time.

**Claim 4.2.1.** *For any  $\text{DB} \in \{0, 1\}^N$ , any prime  $d \in \mathbb{N}$ , and any  $m \in \mathbb{N}$  such that  $d^m \geq N$ , there exists some  $m$ -variate polynomial  $f_{\text{DB}}(X_1, \dots, X_m)$  over  $\mathbb{F}_d$  with individual degree  $< d$  in each variable such that, for all  $i \in \llbracket N \rrbracket$ , it holds that  $f_{\text{DB}}(i_1, \dots, i_m) = \text{DB}[i]$  where  $(i_1, \dots, i_m) = \text{base}_{d,m}(i)$ . Moreover, there is an algorithm  $f_{\text{DB}} := \text{ToPoly}_{d,m}(\text{DB})$  that outputs the coefficients of  $f_{\text{DB}}$  in time  $O(d^m \cdot m \cdot \text{poly log}(d))$ .*

*Proof.* We invoke multivariate polynomial interpolation from Lemma 2.2. Define the  $d^m$  evaluation points  $\{y_{(x_1, \dots, x_m)} \in \mathbb{F}_d\}_{(x_1, \dots, x_m) \in \mathbb{F}_d^m}$  via  $y_{(x_1, \dots, x_m)} = \text{DB}[i]$  when  $(x_1, \dots, x_m) = \text{base}_{d,m}(i)$  for  $i \in \llbracket N \rrbracket$  and  $y_{(x_1, \dots, x_m)} = 0$  else. The Lemma says that in time  $O(d^m \cdot m \cdot \text{poly log}(d))$  we can interpolate the coefficients of a polynomial  $f_{\text{DB}}(X_1, \dots, X_m)$  such that  $f_{\text{DB}}(x_1, \dots, x_m) = y_{x_1, \dots, x_m}$  for all  $(x_1, \dots, x_m) \in \mathbb{F}_d^m$ , which satisfies the Claim.  $\square$

**Basic PIR.** We construct a basic PIR scheme (without preprocessing) by using an ASHE scheme; the client encrypts the  $m$  base- $d$  digits of  $i$  and the server homomorphically evaluates the polynomial  $f_{\text{DB}}$  over them. Concretely, we use ASHE with plaintext space  $\mathbb{F}_d$ , total degree  $D = dm$ , and number of terms  $N$ , with some corresponding ciphertext space consisting of a ring  $R$ . The Query algorithm, given an index  $i$ , finds its base- $d$  representation  $(i_1, \dots, i_m) = \text{base}_{d,m}(i)$ , chooses an ASHE secret  $s$  and uses it to encrypt each of the  $m$  plaintexts  $i_j \in \mathbb{Z}_d$  resulting in ciphertexts

$ct_j \in R$ , which it sends to the server. The server responds by taking the polynomial  $f_{DB}$  defined above, lifting it to  $R$  to get  $\bar{f}_{DB}$ , and sending the homomorphically evaluated ciphertext  $ct^* := \bar{f}_{DB}(ct_1, \dots, ct_m)$  to the client. The client uses ASHE decryption  $\text{Dec}(s, ct^*)$  to recover  $\text{DB}[i] = f_{DB}(i_1, \dots, i_m)$ .

**Preprocessing.** Lastly, we take the basic PIR scheme above and convert it to DEPIR by preprocessing the polynomial  $\bar{f}_{DB}$ . Our DEPIR scheme uses the same Query, Dec algorithms as the basic PIR above. In particular, from the client's point of view, there is no difference between the DEPIR scheme and the basic PIR scheme, and the security of the former directly follows from that of the latter, which in turn follows from that of the ASHE, which in turn follows from RingLWE. However, we improve the server's efficiency and accelerate the computation of  $ct^* := \bar{f}_{DB}(ct_1, \dots, ct_m)$  by preprocessing  $\bar{f}_{DB}$  using Theorem 2.1. The preprocessing results in some data structure  $\widetilde{\text{DB}}$  that lets us evaluate  $\bar{f}_{DB}$  on any input  $ct_1, \dots, ct_m$  in time sublinear in  $N$ . Here we have to be careful with the parameter choice of  $d, m$  and different options will result in different trade-offs between preprocessing time/size and evaluation time.

**Full DEPIR Construction.** We now give a full construction of DEPIR from an ASHE scheme (ASHE.Setup, ASHE.Gen, ASHE.Enc, ASHE.Dec, ASHE.Lift). We leave the choice of the parameters  $d$  and  $m$  flexible and will later plug in concrete choices of  $d$  and  $m$  to achieve different trade-offs.

---

#### Algorithm 4.3: DEPIR from ASHE

---

**Parameters:** The database size  $N$ , determines some parameters  $d, m \in \mathbb{N}$  such that  $d$  prime and  $d^m \geq N$ , as specified later. Let  $\text{params} := \text{ASHE.Setup}(1^\lambda, 1^d, 1^D, d^m)$  with  $D = dm$ , which determines a ring  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$ . Without loss of generality, we implicitly assume all algorithms have access to these parameters, which they can derive from  $\lambda, N$ .

Prep( $1^\lambda, \text{DB}$ ):

1. Let  $f_{DB} := \text{ToPoly}_{d,m}(\text{DB})$  be computed using the algorithm in Claim 4.2.1.  
Note that  $f_{DB}(X_1, \dots, X_m)$  is an  $m$ -variate polynomial over  $\mathbb{F}_d$  with individual degree  $< d$  in each variable such that  $f_{DB}(\text{base}_{d,m}(i)) = \text{DB}[i]$  for all  $i \in \llbracket N \rrbracket$ . The total degree of  $f_{DB}$  is  $< dm$  and the total number of terms is  $\leq d^m$ .
2. Lift  $f_{DB} \in \mathbb{F}_d[X_1, \dots, X_m]$  to  $\bar{f}_{DB} \in R[X_1, \dots, X_m]$  via  $\bar{f}_{DB} := \text{ASHE.Lift}(f_{DB})$ .
3. Invoke the preprocessing algorithm from Theorem 2.1 on the polynomial  $\bar{f}_{DB}$  over  $R$  and let the resulting data structure be  $\widetilde{\text{DB}}$ .
4. Output  $\widetilde{\text{DB}}$ .

Query( $1^\lambda, N, i$ ):

1. Let  $(i_1, \dots, i_m) = \text{base}_{d,m}(i)$  be the base- $d$  digits of  $i$ .
2. Sample  $s \leftarrow \text{ASHE.Gen}(1^\lambda)$ .
3. For each  $j \in [m]$ , encrypt  $i_j$  by invoking  $ct_j \leftarrow \text{ASHE.Enc}(s, i_j)$ .
4. Output  $(ct = (ct_1, \dots, ct_m), s)$ .

Resp( $\widetilde{\text{DB}}, \text{ct}$ ):

1. Parse  $\text{ct} = (\text{ct}_1, \dots, \text{ct}_m)$ .
2. Invoke the evaluation algorithm from Theorem 2.1 to evaluate  $\text{ans} = \bar{f}_{\text{DB}}(\text{ct}_1, \dots, \text{ct}_m)$  using random-access to the data structure  $\widetilde{\text{DB}}$ .
3. Output  $\text{ans}$ .

Dec( $s, \text{ans}$ ):

1. Output ASHE.Dec( $s, \text{ans}$ ).

**Correctness.** Consider any  $\text{DB} \in \{0, 1\}^N$  and any  $i \in \llbracket N \rrbracket$  with  $(i_1, \dots, i_m) = \text{base}_{d,m}(i)$ . Let  $\widetilde{\text{DB}} := \text{Prep}(1^\lambda, \text{DB})$ ,  $(\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N, i)$ ,  $\text{ans} := \text{Resp}(\widetilde{\text{DB}}, \text{ct})$ ,  $b = \text{Dec}(s, \text{ans})$ . By Claim 4.2.1, we know that the polynomial  $f_{\text{DB}}$  computed during preprocessing satisfies  $f_{\text{DB}}(i_1, \dots, i_m) = \text{DB}[i]$ . Also, by the correctness of polynomial evaluation with preprocessing (Theorem 2.1), we have that the value  $\text{ans}$  computed during Resp satisfies  $\text{ans} = \bar{f}_{\text{DB}}(\text{ct}_1, \dots, \text{ct}_m)$ . Hence, by the definition of correctness for ASHE (Definition 3.1), we have that

$$b = \text{ASHE.Dec}(s, \text{ans} = \bar{f}_{\text{DB}}(\text{ct}_1, \dots, \text{ct}_m)) = f_{\text{DB}}(i_1, \dots, i_m) = \text{DB}[i].$$

**Security.** The security of the DEPIR follows directly from that of ASHE, since the adversary only sees  $m$  ASHE ciphertexts. Notice that the adversary runs in time  $\text{poly}(\lambda)$  and chooses the bound  $1^N$ , meaning that  $N = \text{poly}(\lambda)$ . Depending on the choice of  $d, m$  as functions of  $N$ , we can rely on restricted forms of ASHE security. In particular, if  $d, m = \text{poly} \log(N) = \text{poly} \log(\lambda)$  then we only need ASHE security for polylogarithmic degree, and if  $d, m = N^{o(1)} = \lambda^{o(1)}$  then we only need ASHE security for sub-polynomial degree.

**Efficiency.** We calculate the computation time and output size for each algorithm. Recall that, by the definition of ASHE efficiency, the bit-length of ring elements and the run-time of the ring operations are bounded by  $\text{poly}(\lambda, d, m)$ .

- Prep: The interpolation takes time  $d^m \cdot m \cdot \text{poly} \log d$  (Lemma 2.2), and ASHE.Lift takes time  $d^m \cdot \text{poly}(\lambda, d, m)$ . By Theorem 2.1, computing the data structure  $\widetilde{\text{DB}}$  takes time

$$\begin{aligned} & d^m \cdot m^m \cdot \text{poly}(m, d, \log |R|) \cdot O(\log m + \log d + \log \log |R|)^m \\ & = d^m \cdot m^m \cdot \text{poly}(m, d, \lambda) \cdot O(\log m + \log d + \log \lambda)^m, \end{aligned}$$

which therefore dominates the run-time of Prep and also bounds the size of  $\widetilde{\text{DB}}$ .

- Query: The run-time of Query is bounded by that of ASHE.Gen and that of running  $m$  copies of ASHE.Enc, which is bounded by  $\text{poly}(\lambda, d, m)$  by the definition of ASHE efficiency.
- Resp: By Theorem 2.1, the run-time of Resp is bounded by  $\text{poly}(d, m, \log |R|) = \text{poly}(\lambda, d, m)$ .
- Dec: By the definition of ASHE efficiency, the run-time of Dec is bounded by  $\text{poly}(\lambda, d, m)$ .

We consider two potential options of how to choose  $d, m$  with different tradeoffs between preprocessing time/size and a query time/communication.

- **Option A:** For any constant  $\varepsilon > 0$ , choose  $d$  to be the first prime  $d > \log^{2/\varepsilon} N$ . By Bertrand's postulate  $d < 2 \cdot \lceil \log^{2/\varepsilon} N \rceil = O(\log^{2/\varepsilon} N)$ . We can find  $d$  in deterministic poly  $\log N$  time using deterministic primality testing. Let  $m := \lceil \log_d N \rceil = \frac{\varepsilon}{2} \log N / \log \log N + O(1)$ .

The preprocessing run-time and the server storage are bounded by:

$$\begin{aligned} & d^m \cdot m^m \cdot \text{poly}(m, d, \lambda) \cdot O(\log m + \log d + \log \lambda)^m \\ & = O(N^{1+\varepsilon}) \text{poly}(\lambda, \log N), \end{aligned}$$

where we bound  $m^m \leq O(\log N)^m \leq (\log N)^{\frac{\varepsilon}{2} \log N / \log \log N + O(1)} \leq N^{\varepsilon/2} \text{poly} \log N$ , and similarly  $O(\log m + \log d + \log \lambda)^m \leq O(\log N)^m \leq N^{\varepsilon/2} \text{poly} \log N$ .<sup>16</sup>

The query time (i.e., run-time of Query, Resp, Dec) and communication are bounded by:

$$\text{poly}(\lambda, d, m) = \text{poly}(\lambda, \log N).$$

- **Option B:** Choose  $d$  to be the first prime  $d > 2^{\sqrt{\log N}}$ . By Bertrand's postulate  $d = O(2^{\sqrt{\log N}})$ . We can find  $d$  in deterministic  $2^{O(\sqrt{\log N})}$  time. Let  $m := \lceil \log_d N \rceil \leq \sqrt{\log N} + O(1)$ .

The preprocessing run-time and server storage are bounded by:

$$\begin{aligned} & d^m \cdot m^m \cdot \text{poly}(m, d, \lambda) \cdot O(\log m + \log d + \log \lambda)^m \\ & = N \cdot 2^{\tilde{O}(\sqrt{\log N})} \text{poly}(\lambda) = N^{1+o(1)} \text{poly}(\lambda) \end{aligned}$$

The query time (i.e., run-time of Query, Resp, Dec) and communication are bounded by:

$$\text{poly}(\lambda, d, m) = 2^{O(\sqrt{\log N})} \text{poly}(\lambda) = N^{o(1)} \text{poly}(\lambda).$$

Summarizing, the above gives us the following theorem, where the two parts correspond to parameter options A and B respectively.

**Theorem 4.4.** *Assuming ASHE for polylogarithmic degree, for any constant  $\varepsilon > 0$  there is a DEPIR scheme such that, for a database of size  $N$  and security parameter  $\lambda$ , the preprocessing run-time and the server storage are bounded by  $O(N^{1+\varepsilon}) \text{poly}(\lambda, \log N)$  and the query time and communication are bounded by  $\text{poly}(\lambda, \log N)$ . In particular, such a scheme exists assuming RingLWE with quasi-polynomial approximation factors.*

*Alternatively, assuming ASHE for sub-polynomial degree, there is a DEPIR scheme such that, for a database of size  $N$  and security parameter  $\lambda$ , the preprocessing run-time and the server storage are bounded by  $N \cdot 2^{\tilde{O}(\sqrt{\log N})} \text{poly}(\lambda) = N^{1+o(1)} \text{poly}(\lambda)$  and the query time and communication are bounded by  $2^{O(\sqrt{\log N})} \text{poly}(\lambda) = N^{o(1)} \text{poly}(\lambda)$ . In particular, such a scheme exists assuming RingLWE with sub-sub-exponential approximation factors.*

---

<sup>16</sup>Without loss of generality, we assume  $N \geq \lambda$ , as otherwise we can switch to "trivial PIR", where the server sends the entire database in each query to get the required efficiency.

**Remark on Passive Syntax.** As mentioned in Section 4.1, Boyle et al. defines a passive syntax for DEPIR, where the client’s query consists of a set of indices  $I$  and the server’s response consists of  $\widetilde{\text{DB}}[I] = (\widetilde{\text{DB}}[i] : i \in I)$ . We can modify our scheme, described in Algorithm 4.3, to satisfy the passive syntax while preserving the efficiency. To see this, we have to go under the hood of polynomial evaluation with preprocessing (Theorem 2.1). The preprocessing of the polynomial  $\widetilde{f}_{\text{DB}}$  creates a data structure  $\widetilde{\text{DB}}$  consisting of some tables. To evaluate the polynomial at some point  $(\text{ct}_1, \dots, \text{ct}_m) \in R^m$ , the point fully determines a small set  $I$  of locations in the tables to look up. Using the values in these locations, one can efficiently reconstruct  $\text{ans} = \widetilde{f}_{\text{DB}}(\text{ct}_1, \dots, \text{ct}_m)$ . In our DEPIR scheme as described in Algorithm 4.3, the client sends  $(\text{ct}_1, \dots, \text{ct}_m)$  to the server, and the server then uses these to figure out the set  $I$ , looks up  $\widetilde{\text{DB}}[I]$  and uses it to compute  $\text{ans}$ . However, we can convert this to passive syntax by having the client use  $(\text{ct}_1, \dots, \text{ct}_m)$  to figure out  $I$  locally, send only the set  $I$  to the server who responds with  $\widetilde{\text{DB}}[I]$ , and then have the client use this to compute  $\text{ans}$ . Moreover, since our notion of *query time* already combines the total of server + client work, shifting some of the work from server to client does not change the overall cost.

**Remark on Efficiency Optimizations.** We are being wasteful by thinking of the database as  $N$  bits and only retrieving a single bit in each DEPIR query. We could optimize the scheme by thinking of the database as  $N$  blocks in  $\mathbb{F}_d$ , and retrieve an entire block in each query.

We can also achieve some minor efficiency optimizations by going under the hood of our ASHE construction from RingLWE and polynomial preprocessing. While we think of ASHE ciphertexts  $\text{ct}_1, \dots, \text{ct}_m$  and the coefficients of the polynomial  $\widetilde{f}_{\text{DB}}$  as being elements of the ring  $R = \mathbb{Z}_q[Y, Z]/(Z^n + 1, Y^D + 1)$ , they actually live in a small subset. In particular, all the coefficients are just constant polynomials and all the ciphertext polynomials only have degree 1 in  $Y$ . This observation would allow us to optimize the preprocessing (see Appendix A) by getting better bounds on the degree/magnitude of the polynomial outputted by  $\widetilde{f}_{\text{DB}}(\text{ct}_1, \dots, \text{ct}_m)$  when evaluated without reducing modulo  $q$  or  $Y^D + 1$ . While this would provide an efficiency improvement, it does not affect our asymptotic statement.

## 5 Updatable DEPIR

We show how to construct an updatable DEPIR scheme where the server can efficiently update bits of the database, by setting  $\text{DB}[i] := b$ , and correspondingly update the preprocessed data structure  $\widetilde{\text{DB}}$  in sublinear time.

**Definition.** An updatable DEPIR inherits the same syntax, correctness and efficiency properties as standard DEPIR. However, it also has an additional Update algorithm such that  $\widetilde{\text{DB}}' := \text{Update}(\widetilde{\text{DB}}, i, b)$  updates the data structure in a way that corresponds to setting  $\text{DB}[i] := b$ . Moreover, it does so in sub-linear time given read/write random-access to  $\widetilde{\text{DB}}$ .

**Definition 5.1** (Updatable DEPIR). *An (unkeyed) updatable DEPIR is a tuple of algorithms (Prep, Query, Resp, Dec, Update) such that (Prep, Query, Resp, Dec) satisfy the definition of standard DEPIR (Definition 4.1). Moreover Update has the following syntax:*

- $\widetilde{\text{DB}}' := \text{Update}(\widetilde{\text{DB}}, i, b)$ : Takes as input a location  $i \in \llbracket N \rrbracket$  and a bit  $b \in \{0, 1\}$ . Uses random-access to the data structure  $\widetilde{\text{DB}}$  and updates it to  $\widetilde{\text{DB}}'$ .

We require the following notion of correctness with updates. There is a predicate  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB})$  that corresponds to  $\widetilde{\text{DB}}$  being a “good” preprocessing of  $\text{DB}$  with respect to security parameter  $\lambda$ , such that the following properties hold:

1. For any  $\text{DB} \in \{0, 1\}^N$  and for  $\widetilde{\text{DB}} := \text{Prep}(1^\lambda, \text{DB})$  we have  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$ .
2. For any  $\text{DB} \in \{0, 1\}^N$ , any  $i \in \llbracket N \rrbracket$ , and any  $b \in \{0, 1\}$ , let  $\text{DB}'$  be the same as  $\text{DB}$  except with  $\text{DB}'[i] := b$ . For any  $\widetilde{\text{DB}}$  such that  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$ , if  $\widetilde{\text{DB}}' := \text{Update}(\widetilde{\text{DB}}, i, b)$  then  $\text{GOOD}_\lambda(\widetilde{\text{DB}}', \text{DB}') = 1$ .
3. For any  $\text{DB} \in \{0, 1\}^N$  any  $\lambda \in \mathbb{N}$ , any  $\widetilde{\text{DB}}$  such that  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$ , any  $i \in \llbracket N \rrbracket$ ,

$$\Pr \left[ \text{Dec}(s, \text{ans}) = \text{DB}[i] : \begin{array}{l} (\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N, i); \\ \text{ans} := \text{Resp}(\widetilde{\text{DB}}, \text{ct}) \end{array} \right] = 1.$$

**Definition 5.2** (Efficiency metrics with updates). For any updatable DEPIR ( $\text{Prep}$ ,  $\text{Query}$ ,  $\text{Resp}$ ,  $\text{Dec}$ ,  $\text{Update}$ ), the preprocessing time, server storage, query time and communication are defined the same way as in standard DEPIR. Further, we define the update time to be the run-time of the  $\text{Update}$  procedure.

**Construction overview.** Our previous construction of DEPIR in Section 4.2 does not appear to be updatable directly; every bit of  $\text{DB}$  affects almost every bit of  $\widetilde{\text{DB}}$ . In fact, there seems to be an inherent difficulty in constructing updatable DEPIR. If an update for an index  $i$  touches some set of locations  $I_i$  in the data structure  $\widetilde{\text{DB}}$ , then a PIR query for  $i$  must touch some of the same locations in  $I_i$ , else it will be unaware of the update. But then, it may seem that by observing the set of locations accessed during the PIR query and checking if they coincide with  $I_i$ , the server can learn whether the queried index is  $i$  or not! It turns out that the above intuition is false, and in fact, we can ensure that the set  $I_i$  of locations accessed during an update is completely independent of  $i$ . To do so, we borrow ideas from *hierarchical ORAM* [GO96], which were also used previously used in a related manner in conjunction with multi-server DEPIR [HOWW19] and with secret-key DEPIR [HHWW19]. We show how to use these ideas to generically upgrade any basic (non-updatable) DEPIR into an updatable DEPIR. Essentially, our updatable DEPIR will consist of a hierarchy of  $L = \log N$  levels of exponentially increasing size, where each level  $\ell$  contains a preprocessed database  $\widetilde{\text{DB}}_\ell$  under the standard DEPIR. Whenever we update the database, we will put this data in the smallest level, no matter what index  $i$  we are updating, and therefore the set of accessed locations  $I_i$  is independent of  $i$ . After every  $2^\ell$  updates, we move all the information from the smallest  $\ell - 1$  levels into level  $\ell$ . To read from the updatable DEPIR, the client makes a small number of basic DEPIR queries at every level of the hierarchy.

As a first step, we construct a *multi-round updatable DEPIR*, where the PIR protocol, in which the client privately reads some location  $\text{DB}[i]$ , now requires several rounds of interaction instead of the default 2 rounds. The update procedure in which the server updates the data structure  $\widetilde{\text{DB}}$  is still non-interactive and performed by the server on its own without any involvement from the clients. We then show how to remove the need for many rounds and give a construction that achieves the above round-optimal definition, where the entire protocol consists of the client sending a query ciphertext and the server responding with an answer.

## 5.1 Multi-Round Updatable DEPIR

We generalize the syntax of updatable DEPIR to allow a multi-round protocol  $\Pi$  between the server  $\mathcal{S}$  who holds the preprocessed data structure  $\widetilde{\text{DB}}$  and the client  $\mathcal{C}$  who holds an index  $i$ , such that the client should output  $\text{DB}[i]$  at the end of the protocol. This protocol  $\Pi$  replaces the functions  $\text{Query}$ ,  $\text{Resp}$ ,  $\text{Dec}$ , which implicitly defined a basic 2-round protocol. The correctness, security and efficiency of this generalized syntax are defined analogously in the natural way. In particular, security says that a (malicious) server cannot learn anything about the client's index  $i$  during the execution of  $\Pi$ .

**Definition 5.3** (Updatable Multi-Round DEPIR). *An updatable multi-round DEPIR consists of a tuple of algorithms  $(\text{Prep}, \text{Update})$  with the same syntax as Definition 5.1, together with a protocol  $\Pi = (\mathcal{S}, \mathcal{C})$  between a server  $\mathcal{S}(\widetilde{\text{DB}})$  and a client  $\mathcal{C}(1^\lambda, N, i)$ .*

**Correctness:** *There is a predicate  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB})$  that satisfies properties (1),(2) of the correctness requirement of Definition 5.1 as well as the following modification of property (3): For any  $\text{DB} \in \{0, 1\}^N$ , any  $\lambda \in \mathbb{N}$ , any  $\widetilde{\text{DB}}$  such that  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$ , and any  $i \in \llbracket N \rrbracket$ , an honest execution of the protocol  $\Pi$  between the server  $\mathcal{S}(\widetilde{\text{DB}})$  and the client  $\mathcal{C}(1^\lambda, N, i)$  results in the client outputting  $\text{DB}[i]$  with probability 1.*

**Security:** *We define the following game between a challenger and a stateful adversary  $\mathcal{A}$ :*

1.  $\mathcal{A}(1^\lambda)$  selects a size  $1^N$  and a challenge index pair  $i_0, i_1 \in \llbracket N \rrbracket$ .
2. The challenger selects a random bit  $b \leftarrow \{0, 1\}$ .
3. The adversary participates in a protocol execution  $\mathcal{A} \leftrightarrow \mathcal{C}(1^\lambda, N, i_b)$  with the honest client  $\mathcal{C}(1^\lambda, N, i_b)$ . The honest client follows the protocol specification, but  $\mathcal{A}$  can act arbitrarily. At the end of the protocol execution  $\mathcal{A}$  outputs a guess for  $b'$ .

*We require that for every PPT adversary  $\mathcal{A}$ , there exists a negligible function  $\text{negl}$  such that the distinguishing advantage of  $\mathcal{A}$  in the above security game is  $|\Pr[b' = b] - 1/2| \leq \text{negl}(\lambda)$ .*

**Efficiency:** *The preprocessing time, server storage, and update time are defined the same way as Definition 5.1. We now define query time (resp. communication) to be the total run time (resp. total communication complexity) of the server and the client in the protocol  $\Pi$ .*

**Construction.** We show how to generically upgrade any (non-updatable) DEPIR ( $\text{Prep}$ ,  $\text{Query}$ ,  $\text{Resp}$ ,  $\text{Dec}$ ) with sufficiently good parameters into a multi-round updatable DEPIR ( $\text{Prep}'$ ,  $\text{Update}$ ,  $\Pi = (\mathcal{S}, \mathcal{C})$ ). The construction follows the high-level description in Section 1.2.4.

---

### Algorithm 5.4: Multi-Round Updatable DEPIR

---

$\widetilde{\text{DB}} := \text{Prep}'(1^\lambda, \text{DB})$ : Let  $N := |\text{DB}|$ . Let  $L := \lceil \log N \rceil$ .

1. Construct  $\text{DB}_L$  to consist of a list of  $2^L$  pairs  $(i, b)$  with  $i \in \llbracket N \rrbracket \cup \{\infty\}$ ,  $b \in \{0, 1\}$ , where the list contains all pairs  $(i, \text{DB}[i])$  for  $i \in \llbracket N \rrbracket$ , and is padded with  $2^L - N$  additional "dummy pairs"  $(\infty, 0)$ . The list is sorted according to  $i$ .
2. Let  $\widetilde{\text{DB}}_L := \text{Prep}(1^\lambda, \text{DB}_L)$ .

3. For  $\ell < L$ : Let  $\text{DB}_\ell = \perp$  and  $\widetilde{\text{DB}}_\ell := \perp$ .
4. Set  $\text{count} := 0$ . Output  $\widetilde{\text{DB}} = (\text{DB}_0, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_L, \text{count})$ .

$\widetilde{\text{DB}}' := \text{Update}(\widetilde{\text{DB}}, i^*, b^*)$ : Update  $\text{count} := \text{count} + 1 \pmod{2^L}$ .

Let  $\ell^*$  be the max of  $\ell \in \{0, \dots, L\}$  such that  $2^\ell$  divides  $\text{count}$ .

1. Construct an updated list  $\text{DB}'_{\ell^*}$  as follows:
  - (a) Define  $\text{DB}_{-1} := \{(i^*, b^*)\}$  to be a list containing a single tuple.
  - (b) Take all the non-dummy pairs  $(i, b)$  contained in the lists  $\text{DB}_{-1}, \text{DB}_0, \dots, \text{DB}_{\ell^*}$  and sort them by index  $i$  into  $\text{DB}'_{\ell^*}$ .<sup>17</sup> If there are multiple pairs with the same index  $i$ , take only the one from  $\text{DB}_\ell$  with the smallest  $\ell$  and discard the rest.
  - (c) Append additional dummy pairs  $(\infty, 0)$  to  $\text{DB}'_{\ell^*}$  until the final list is of size  $2^{\ell^*}$ .
2. Set  $\text{DB}_{\ell^*} := \text{DB}'_{\ell^*}$ ,  $\widetilde{\text{DB}}_{\ell^*} := \text{Prep}(1^\lambda, \text{DB}'_{\ell^*})$ .
3. For  $\ell \in \{0, \dots, \ell^* - 1\}$ : Set  $\text{DB}_\ell := \perp$ ,  $\widetilde{\text{DB}}_\ell := \perp$ .

$\Pi = (\mathcal{S}(\widetilde{\text{DB}}), \mathcal{C}(1^\lambda, N, i))$ : Let  $L := \lceil \log N \rceil$ . Let  $N_\ell := 2^\ell(\lceil \log(N+1) \rceil + 1)$  be the bit-size of  $\text{DB}_\ell$ .

Let  $\text{BinSearch}$  be a RAM program implementing binary search, such that  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  has read-only random-access to a sorted list  $\text{DB}_\ell$  consisting of  $2^\ell$  tuples, and checks if the list contains some tuple of the form  $(i, b)$ : if so it outputs the first such tuple, else it outputs  $\perp$ . The program is padded to always run in worst-case time  $T_{N,\ell} = \text{poly log}(N)$ .

The server  $\mathcal{S}(\widetilde{\text{DB}})$  and client  $\mathcal{C}(1^\lambda, N, i)$  run the following protocol:

1. For  $\ell = 0, \dots, L$ :

The client  $\mathcal{C}(1^\lambda, N, i)$  starts running the program  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$ , without having a copy of  $\text{DB}_\ell$ . Each time the program wants to read some bit  $\text{DB}_\ell[j]$  in location  $j \in \llbracket N_\ell \rrbracket$  of  $\text{DB}_\ell$ , the client/server run the following “basic DEPIR” sub-protocol:

- i. The client runs  $(\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N_\ell, j)$  and sends  $(\text{ct}, \ell)$  to the server, and keeps  $s$  locally.
- ii. If  $\widetilde{\text{DB}}_\ell \neq \perp$ , the server responds with the answer  $\text{ans} \leftarrow \text{Resp}(\widetilde{\text{DB}}_\ell, \text{ct})$  using random-access to the data structure  $\widetilde{\text{DB}}_\ell$ , else the server responds with  $\perp$ .
- iii. If  $\text{ans} \neq \perp$ , the client runs  $b := \text{Dec}(s, \text{ans})$  else it sets  $b := \perp$ .

If  $b \neq \perp$ , the client continues running the program  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  with  $\text{DB}_\ell[j] = b$ . Else the client terminates the execution and sets the output to be  $\perp$ .

Eventually each of the programs outputs some  $b_\ell = \text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  such that  $b_\ell \in \{0, 1\} \cup \{\perp\}$ .

2. Let  $\ell^*$  be the minimal  $\ell \in \{0, \dots, L\}$  such that  $b_\ell \neq \perp$ . The client outputs  $b_{\ell^*}$ .

**Theorem 5.5.** *Assume there is a DEPIR scheme with preprocessing time / server storage  $\eta_p(\lambda, N)$  and query time / communication  $\eta_q(\lambda, N)$ . Then the above construction yields a multi-round updatable DEPIR with preprocessing time / server storage  $\eta_p(\lambda, O(N \log N)) \cdot \text{poly log } N$ , query time / communication  $\eta_q(\lambda, O(N \log N)) \cdot \text{poly log } N$ , and amortized update time  $\text{poly log } N + \sum_{\ell=0}^{\lceil \log N \rceil} \frac{1}{2^\ell} \eta_p(\lambda, O(2^\ell \log N))$ .*

<sup>17</sup>If  $\ell^* < L$ , then  $\text{DB}_{\ell^*}$  must be  $\perp$ , an empty list, as we will argue in the correctness.

In particular, assuming RingLWE with quasi-polynomial approximation factors, for any  $\varepsilon > 0$ , there is a multi-round updatable DEPIR with preprocessing time / server storage  $N^{1+\varepsilon}\text{poly}(\lambda)$ , query time / communication  $\text{poly}(\lambda, \log N)$ , and amortized update time  $N^\varepsilon\text{poly}(\lambda, \log N)$ .

Alternatively, assuming RingLWE with sub-sub-exponential approximation factors, there is a multi-round updatable DEPIR with preprocessing time / server storage  $N^{1+o(1)}\text{poly}(\lambda)$ , query time / communication  $N^{o(1)}\text{poly}(\lambda)$ , and amortized update time  $N^{o(1)}\text{poly}(\lambda)$ .

*Proof.* We show correctness, security and efficiency in turn. Throughout, for database size  $N$ , we define  $L := \lceil \log N \rceil$  and for  $\ell \in \{0, \dots, L\}$  we define  $N_\ell := 2^\ell(\lceil \log(N+1) \rceil + 1)$ .

**Correctness.** Define the predicate  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB})$  to hold if

$$\widetilde{\text{DB}} = (\text{DB}_0, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_L, \text{count})$$

with  $\text{DB}_\ell \in \{0, 1\}^{N_\ell}$  such that:

- i. For all  $\ell \in \{0, \dots, L\}$ ,  $\widetilde{\text{DB}}_\ell = \text{Prep}(1^\lambda, \text{DB}_\ell)$ .
- ii. For all  $\ell \in \{0, \dots, L\}$ , either  $\text{DB}_\ell = \perp$  or  $\text{DB}_\ell$  consists of a list of  $2^\ell$  pairs  $(i, b)$  with  $i \in \llbracket N \rrbracket \cup \{\infty\}$ ,  $b \in \{0, 1\}$  and the pairs are sorted by the index  $i$ . Furthermore, for any  $i \in \llbracket N \rrbracket$ , there is at most one tuple of the form  $(i, b)$  in  $\text{DB}_\ell$ .
- iii. For each  $i \in \llbracket N \rrbracket$ , let  $\ell \in \{0, \dots, L\}$  be the minimal value such that  $\text{DB}_\ell$  contains a tuple of the form  $(i, b)$ . Then such an  $\ell$  always exists and  $b = \text{DB}[i]$ .
- iv. Let  $\text{count} = (\text{count}_0, \dots, \text{count}_{L-1})$  be the binary representation of  $\text{count} \in \llbracket 2^L \rrbracket$  with  $\text{count} = \sum_\ell \text{count}_\ell \cdot 2^\ell$  and  $\text{count}_i \in \{0, 1\}$ . For each  $\ell$  such that  $\text{count}_\ell = 0$  we have  $\text{DB}_\ell = \perp$ .

With the above predicate, we show that the 3 correctness properties (Definition 5.1) hold.

By the definition of  $\widetilde{\text{DB}} := \text{Prep}'(1^\lambda, \text{DB})$ , it is easy to see that correctness property (1) holds, meaning that  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$ .

By the definition of  $\widetilde{\text{DB}} := \text{Update}(\widetilde{\text{DB}}, i^*, b^*)$ , it is also relatively easy to see that correctness property (2) holds, meaning that if  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$  for some  $\text{DB}$  then  $\text{GOOD}_\lambda(\widetilde{\text{DB}}', \text{DB}') = 1$ , where  $\text{DB}'$  is the same as  $\text{DB}$  except  $\text{DB}'[i^*] = b^*$ . There are two subtleties to check. Firstly, we verify that property (iv) is preserved by the update. Let  $\ell^*$  be the value defined during the update. Then it must be the case that, after the update, we have  $\text{count}_\ell = 0$  for  $\ell < \ell^*$ ,  $\text{count}_{\ell^*} = 1$  or  $\ell^* = L$ , and  $\text{count}_\ell$  remain the same as before the update for  $\ell > \ell^*$ . The update procedure ensures that for  $\ell < \ell^*$  we have  $\text{DB}_\ell = \perp$  and for  $\ell > \ell^*$  the lists  $\text{DB}_\ell$  are unaffected by the update. Therefore property (iv) is preserved. Secondly, in step 1b of the update procedure, when we take all the non-dummy pairs in  $\text{DB}_{-1}, \text{DB}_0, \dots, \text{DB}_{\ell^*}$ , we need to make sure there are  $\leq 2^{\ell^*}$  of them with distinct indices  $i$ , otherwise we would “overflow”  $\text{DB}_{\ell^*}$ . If  $\ell^* < L$  then, before the update starts, we must have  $\text{count}_{\ell^*} = 0$  and therefore  $\text{DB}_{\ell^*} = \perp$  by (iv). The total number of pairs in  $\text{DB}_{-1}, \text{DB}_0, \dots, \text{DB}_{\ell^*-1}$  is  $\leq 1 + 1 + 2 + 4 + \dots + 2^{\ell^*-1} = 2^{\ell^*}$  and therefore the total number of non-dummy pairs is  $\leq 2^{\ell^*}$ . If  $\ell^* = L$  then the total number of possible non-dummy pairs with distinct indices is  $N \leq 2^L$ .

Finally, by the definition of the protocol  $\Pi$  and the correctness of the underlying non-updatable DEPIR, it is easy to see that correctness property (3) holds, meaning that if  $\text{GOOD}_\lambda(\widetilde{\text{DB}}, \text{DB}) = 1$

for some DB then the execution of  $\Pi$  with  $\mathcal{S}(\widetilde{\text{DB}})$  and  $\mathcal{C}(1^\lambda, N, i)$  results in the latter outputting  $\text{DB}[i]$ . In particular, the correctness of the underlying DEPIR and properties (i) and (ii) above ensure that each of the program executions correctly outputs  $b_\ell = \text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  such that  $b_\ell \neq \perp$  iff the tuple  $(i, b_\ell)$  is contained in  $\text{DB}_\ell$ . Property (iii) then ensures that for  $\ell^*$  being the minimal  $\ell \in \{0, \dots, L\}$  such that  $b_\ell \neq \perp$  we have the client output  $b_{\ell^*} = \text{DB}[i]$ .

**Security.** The security of the protocol follows directly from that of the underlying non-updatable DEPIR, via a standard hybrid argument. In particular, the adversary only sees pairs of the form  $(\ell, \text{ct})$  with a level index  $\ell$  and a query ciphertexts  $\text{ct}$  from the underlying DEPIR, which ensures that the ciphertexts are computationally indistinguishable whether the client's input is  $i_0$  or  $i_1$ . Furthermore, by padding the programs  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  to always make the worst-case number of accesses, the number of pairs  $(\ell, \text{ct})$  that the adversary sees for each  $\ell$  is always the same.

**Efficiency.** The preprocessing algorithm  $\text{Prep}'$  of the updatable DEPIR makes  $L = O(\log N)$  calls to the preprocessing algorithm  $\text{Prep}$  of the underlying DEPIR on databases of size  $N_\ell = 2^\ell \log N = O(N \log N)$  for a total run time of  $\eta_p(\lambda, O(N \log N)) \text{poly} \log(N)$ . The query protocol of the updatable DEPIR runs  $L = O(\log N)$  copies of binary search over databases of size  $N_\ell = O(N \log N)$ , where accessing each bit of the database is done by running a query protocol of the underlying DEPIR which takes time  $\eta_q(\lambda, N_\ell) = \eta_q(\lambda, O(N \log N))$ , for a total run time  $\eta_q(\lambda, O(N \log N)) \cdot \text{poly} \log(N)$ . For the amortized complexity of the update protocol, consider any sequence of  $T$  updates. For each level  $\ell \in \{0, \dots, L\}$  we have  $\leq T/2^\ell$  updates where the value of  $\ell^*$  in the update is  $\ell^* = \ell$ . Each such update takes  $N_\ell \cdot \text{poly} \log N + \sum_{i=0}^{\ell} \eta_p(\lambda, N_i) \leq 2^\ell \text{poly} \log N + \eta_p(\lambda, O(2^\ell \log N)) \text{poly} \log N$  time. Therefore the total cost of the  $T$  updates is

$$\sum_{\ell=0}^{L=\lceil \log N \rceil} \frac{T}{2^\ell} (2^\ell \text{poly} \log N + \eta_p(\lambda, O(2^\ell \log N)) \cdot \text{poly} \log N) = T \cdot \text{poly} \log(N) \cdot \sum_{\ell=0}^{\lceil \log N \rceil} \frac{1}{2^\ell} \eta_p(\lambda, O(2^\ell \log N)).$$

which gives the amortized update time per operation as claimed.  $\square$

**Remark 5.1** (Deamortization). The above scheme only achieves good *amortized* efficiency of the updates, but the worst-case run-time of an update can be  $\geq N$ . However, we can deamortize the scheme and get the same parameters as above with worst-case efficiency. The idea follows the technique of [OS97] and is simple conceptually, but slightly cumbersome to describe. In the scheme above, each time we do an update with some  $\ell^*$  we have to (1) sort the data from  $\text{DB}_{-1}, \text{DB}_0, \dots, \text{DB}_{\ell^*}$  into  $\text{DB}_{\ell^*}$ , (2) redo the preprocessing of  $\text{DB}_{\ell^*}$  to get an updated  $\widetilde{\text{DB}}_{\ell^*}$ . We do this every  $\leq 2^{\ell^*}$  steps and, therefore, although all the work is performed during one update, the amortized analysis spreads the accounting of this cost over the subsequent  $2^{\ell^*}$  updates. In the deamortized scheme, instead of doing the work of steps (1) and (2) during one update itself, we will actually spread the execution (not just the accounting) of these steps across the subsequent  $2^{\ell^*}$  updates. This however raises the issue that the updated  $\text{DB}_{\ell^*}, \widetilde{\text{DB}}_{\ell^*}$  are not available during the subsequent  $2^{\ell^*}$  updates. To solve this we keep an "old auxiliary copy" of  $\text{DB}_{-1}, \text{DB}_0, \dots, \text{DB}_{\ell^*}$  and  $\widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_{\ell^*}$  around and use these to handle accesses to  $\text{DB}_{\ell^*}$  for the next  $2^{\ell^*}$  steps.

## 5.2 Round-Optimal Updatable DEPIR

We now describe how to adapt our multi-round updatable DEPIR construction above to get a round-optimal construction that matches the syntax of definition 5.1. In particular, query protocol consists of the client sending a query ciphertext and the server responds with an answer ciphertext that the client decrypts. To do so, we will rely on RAM-FHE (Section 7); namely, we rely on the simplest variant (Section 7.2), where the RAM program is just binary search  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  over a large plaintext input  $y = \text{DB}_\ell$  of size  $\tilde{O}(N)$ , and a small encrypted input  $x = i$  of size  $O(\log N)$ ; furthermore the program only need  $O(\log N)$  bits of read/write memory. On a high level, we modify the previous protocol  $\Pi$  so that, instead of the client running the programs  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  locally on her input  $i$  and interactively using the basic DEPIR scheme to read from the preprocessed databases  $\text{DB}_\ell$  on the server, we have the client send the RAM-FHE encryption of  $i$  and have the server homomorphically execute the programs  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$  using the preprocessed databases  $\text{DB}_\ell$  without any additional interaction.

Let  $(\text{Setup}, \text{Gen}, \text{Prep}, \text{Enc}, \text{Eval}, \text{Dec})$  be a RAM-FHE. The updatable DEPIR scheme  $(\text{Prep}', \text{Update}, \text{Query}, \text{Resp}, \text{Dec}')$  is defined as follows.

---

### Algorithm 5.6: Updatable DEPIR

---

$\widetilde{\text{DB}} := \text{Prep}'(1^\lambda, \text{DB})$ : Let  $N := |\text{DB}|$ . Let  $L := \lceil \log N \rceil$ . Let  $\text{BinSearch}_{N,\ell}$  be the binary search algorithm as described in the multi-round scheme (5.4). Let  $N_x := L$  and let  $N_y := 2^L(\lceil \log(N+1) \rceil + 1)$  be a bound on the bit-size of the databases  $\text{DB}_\ell$ . Let  $N_S = \text{poly log } N$  be the maximal size of read/write memory needed for  $\text{BinSearch}_{N,\ell}$  for  $\ell \leq L$ . Let  $\text{params} := \text{Setup}(1^\lambda, N_x, N_y, N_S)$  be the parameters of the RAM-FHE. We assume  $\text{params}$  is available to all other algorithms, as they can re-compute it deterministically given  $\lambda, N$ .

The  $\text{Prep}'$  procedure is otherwise the same as in the multi-round scheme (5.4), but using the  $\text{Prep}$  algorithm of RAM-FHE instead of the one of DEPIR. It outputs

$$\widetilde{\text{DB}} = (\text{DB}_0, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_L, \text{count}).$$

$\widetilde{\text{DB}}' := \text{Update}(\widetilde{\text{DB}}, i^*, b^*)$ : This is the same as in the multi-round scheme (5.4), but using the  $\text{Prep}$  algorithm of RAM-FHE instead of the one of DEPIR.

$(\text{ct}, s) \leftarrow \text{Query}(1^\lambda, N, i)$ : Let  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params})$  and let  $\text{ct}' \leftarrow \text{Enc}(\text{pk}, i)$ . Output  $s := \text{sk}$ ,  $\text{ct} := (\text{pk}, \text{ct}')$ .

$\text{ans} := \text{Resp}(\widetilde{\text{DB}}, \text{ct})$ : Parse  $\text{ct} = (\text{pk}, \text{ct}')$ . For  $\ell = 0, \dots, L$ , let  $\text{ct}_\ell := \text{Eval}(\text{pk}, \text{BinSearch}_{N,\ell}, \text{ct}', \widetilde{\text{DB}}_\ell)$ . Output  $\text{ans} := (\text{ct}_0, \dots, \text{ct}_L)$ .

$b = \text{Dec}'(s, \text{ans})$ : Parse  $s = \text{sk}$ ,  $\text{ans} = (\text{ct}_0, \dots, \text{ct}_L)$ . For  $\ell = 0, \dots, L$ , let  $b_\ell := \text{Dec}(\text{sk}, \text{ct}_\ell)$ . Take the smallest  $\ell$  such that  $b_\ell \neq \perp$  and output and output  $b_\ell$ .

---

**Theorem 5.7.** *Assume there is a RAM-FHE scheme for programs with run-time  $T = \text{poly log}(N)$ , encrypted input size  $N_x = O(\log N)$ , maximal plaintext input size  $N_y = O(N \log N)$ , space complexity  $N_S = \text{poly log}(N)$  and output size  $M = O(1)$ , such that the preprocessing of the plaintext input  $y$  runs in time  $\eta_p(\lambda, N, |y|)$ , and the sum of the encryption, evaluation and decryption time is  $\eta_q(\lambda, N)$ . Then*

the above construction yields an updatable DEPIR with preprocessing time / server storage bounded by  $\eta_p(\lambda, N, O(N \log N)) \cdot \text{poly} \log N$ , query time / communication bounded by  $\eta_q(\lambda, N) \cdot \text{poly} \log N$ , and amortized update time bounded by  $\text{poly} \log N + \sum_{\ell=0}^{\lceil \log N \rceil} \frac{1}{2^\ell} \eta_p(\lambda, N, O(2^\ell \log N))$ .

In particular, assuming RingLWE with quasi-polynomial approximation factors, for any  $\varepsilon > 0$ , there is an updatable DEPIR with preprocessing time / server storage bounded by  $N^{1+\varepsilon} \text{poly}(\lambda)$ , query time / communication bounded by  $\text{poly}(\lambda, \log N)$ , and amortized update time bounded by  $N^\varepsilon \text{poly}(\lambda, \log N)$ .

Alternatively, assuming RingLWE with sub-sub-exponential approximation factors, there is an updatable DEPIR with preprocessing time / server storage bounded by  $N^{1+o(1)} \text{poly}(\lambda)$ , query time / communication bounded by  $N^{o(1)} \text{poly}(\lambda)$ , and amortized update time bounded by  $N^{o(1)} \text{poly}(\lambda)$ .

*Proof.* The correctness proof is the same as in the proof of Theorem 5.5, but now we rely on the correctness of the RAM-FHE evaluation of the programs  $\text{BinSearch}_{N,\ell}(i, \text{DB}_\ell)$ . Security follows directly from that of the RAM-FHE via a standard hybrid argument, since the DEPIR query ciphertext  $\text{ct}$  consists of  $L$  RAM-FHE ciphertexts. Lastly, the efficiency analysis is the same as in Theorem 5.5.

Note that our RAM-FHE construction generally relies on an additional circular security assumption. However, in the above we only need RAM-FHE for programs with run-time  $T = \text{poly} \log N$ . Following Theorem 8.3, we can use leveled RAM-FHE and avoid circular security in this case.  $\square$

**Remark 5.2** (Deamortization). We can deamortize the scheme and get the same parameters as above with worst-case efficiency, same as in Remark 5.1.

**Remark: Alternative Construction of Round-Optimal Updatable DEPIR.** We briefly mention an alternative method for constructing round-optimal updatable DEPIR from a basic non-updatable DEPIR. This method avoids the need for RAM-FHE, but has other deficiencies discussed below. The main idea is to use *hash-based sorting* to replace binary search by a single non-adaptive lookup (which also closely follows the approach in [HOWW19, HHWW19]). In other words, the data in each level  $\text{DB}_\ell$  now consists of  $2^\ell$  buckets, and each bucket contains  $\lambda$  pairs  $(i, b)$  with  $i \in \llbracket N \rrbracket \cup \{\infty\}$  and  $b \in \{0, 1\}$ . The pairs  $(i, b)$  in each level  $\ell$  are now assigned to buckets via a hash function  $h_\ell : \llbracket N \rrbracket \rightarrow \llbracket 2^\ell \rrbracket$ , and the buckets are padded with additional dummy pairs  $(\infty, 0)$  to make each bucket of size  $\lambda$ . Otherwise the data structure and the update procedure are analogous. Now, to query some location  $i$ , the client needs to retrieve the buckets  $h_\ell(i)$  at every level  $\ell$ , which it can do by making  $\text{poly}(\lambda, \log N)$  DEPIR queries of the basic scheme. The main advantage of this approach over the prior approach using binary search, is that the client can make all these queries non-adaptively in parallel, and therefore the protocol still runs in the optimal 2 rounds. By choosing the hash function  $h_\ell$  to be  $\lambda$ -wise independent, we can ensure a negligible probability of a bucket overflow if the updates are independent of the hash functions.

The main issues with the above approach stem from the reliance on hash functions  $h_\ell$ . Firstly, if the server chooses them, it will need to communicate them somehow to the client, which would add an extra round of communication leading to a 3 round protocol instead of a 2 round one. Alternatively, we can think of them as part of some *common random string* (CRS) chosen externally, but now we need to rely on the CRS model. Secondly, in either case, the use of hash functions adds a correctness error. This error is negligible only if the updates are chosen non-adaptively and independently of the hash functions. However, since these hash functions are public, an adversary who can adaptively influence the updates after seeing the hash functions can cause an error with

non-negligible probability. For these reasons, we prefer our default approach using RAM-FHE, which does not suffer from these deficiencies.

## 6 ASHE-FHE

In this section, we define and construct ASHE-FHE encryption, which is a hybrid of an ASHE (see Definition 3.1) and fully homomorphic encryption (FHE), providing the best of both worlds. We will use ASHE-FHE as a building block to construct RAM-FHE.

In more detail, an ASHE-FHE is a public-key encryption scheme where we can set the plaintext space to  $\mathbb{Z}_d$  for  $d$  of our choice, and the ciphertexts are elements of some corresponding ring  $R$ . The scheme is an FHE and allows us to evaluate arbitrary (arithmetic) circuits  $C$  over encrypted data. This FHE evaluation procedure can work arbitrarily, and we do not impose any algebraic structure on it. The output of FHE evaluation is a “good” ciphertext, which decrypts correctly and can also be used as an input to future FHE evaluations (i.e., the scheme is multi-hop). The scheme is also an ASHE that allows us to evaluate low-degree multivariate polynomials  $f$  over the encrypted data just by evaluating a lifted version of the same polynomial over the ciphertexts. The input to an ASHE evaluation can be any “good” ciphertext – either a fresh encryption or the output of an FHE evaluation. The output of an ASHE evaluation is no longer directly a “good” ciphertext (i.e., it cannot be used directly as an input to future FHE or ASHE evaluations), but there is some “refresh” procedure that converts it into a good ciphertext. We show how to construct such a ASHE-FHE scheme using a modified version of the FHE scheme of Brakerski, Gentry and Vaikuntanathan [BGV12] based on RingLWE.

**Definition 6.1** (ASHE-FHE). *An ASHE-FHE scheme is a tuple of PPT algorithms (Setup, Gen, Enc, Eval, Dec, Refresh, Lift) with the following syntax:*

- $\text{params} := \text{Setup}(1^\lambda, 1^d, 1^D, N)$ : On input a security parameter  $\lambda$ , total degree  $D$ , number of terms  $N$ , plaintext space  $d$ , it outputs public parameters  $\text{params}$  that implicitly define a ring  $R$  of the form  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for some  $E_1, E_2$ .<sup>18</sup> All other algorithms implicitly take  $\text{params}$  as input even when not explicitly stated.
- $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params})$ : Outputs a public key  $\text{pk}$ , a secret key  $\text{sk}$ .
- $\text{ct} \leftarrow \text{Enc}(\text{pk}, \mu)$ : Given a public key  $\text{pk}$  and a message  $\mu \in \mathbb{Z}_d$ , outputs a ciphertext  $\text{ct} \in R$ .
- $(\text{ct}'_1, \dots, \text{ct}'_m) := \text{Eval}(\text{pk}, C, \text{ct}_1, \dots, \text{ct}_\ell)$ : Given  $\text{pk}$ , an arithmetic circuit  $C : \mathbb{Z}_d^\ell \rightarrow \mathbb{Z}_d^m$ , and ciphertexts  $(\text{ct}_1, \dots, \text{ct}_\ell)$ , deterministically outputs ciphertexts  $(\text{ct}'_1, \dots, \text{ct}'_m)$ .
- $\mu := \text{Dec}(\text{sk}, \text{ct})$ : Given a secret key  $\text{sk}$  and a ciphertext  $\text{ct} \in R$ , outputs a message  $\mu \in \mathbb{Z}_d$ .
- $\text{ct}' := \text{Refresh}(\text{pk}, \text{ct})$ : Given a public key and a ciphertext  $\text{ct} \in R$ , deterministically outputs a new ciphertext  $\text{ct}' \in R$ .
- $\bar{\mu} := \text{Lift}(\mu)$ : Lifts  $\mu \in \mathbb{Z}_d$  to  $\bar{\mu} \in R$ . For any polynomial  $f$  over  $\mathbb{Z}_d$ , we let  $\bar{f} := \text{Lift}(f)$  denote the analogous polynomial over  $R$  derived by applying Lift to every coefficient of  $f$ .

<sup>18</sup>We restrict to these rings to match the rings for which we have fast polynomial evaluation with preprocessing (Theorem 2.1). As noted, this can be generalized further.

We require that the scheme satisfies the following properties.

**Correctness:** For all  $\text{params}$  in the support of  $\text{Setup}$  and all  $(\text{pk}, \text{sk})$  in the support of  $\text{Gen}(\text{params})$ , there is a well-defined class of ciphertexts  $\text{GOOD}_{\text{sk}} \subseteq R$  depending on  $\text{sk}$  (and  $\text{params}$ ), such that that

1. For any plaintext  $\mu \in \mathbb{Z}_d$ ,

$$\Pr \left[ \begin{array}{l} \text{ct} \in \text{GOOD}_{\text{sk}} \wedge \\ \text{Dec}(\text{sk}, \text{ct}) = \mu \end{array} : \text{ct} \leftarrow \text{Enc}(\text{pk}, \mu) \right] = 1.$$

2. For all ciphertexts  $\text{ct}_1, \dots, \text{ct}_m \in \text{GOOD}_{\text{sk}}$  with  $\mu_i := \text{Dec}(\text{sk}, \text{ct}_i)$  for  $i \in [m]$ , and for any polynomial  $f(X_1, \dots, X_m)$  over  $\mathbb{Z}_d$  consisting of at most  $N$  terms and total degree  $< D$ ,

$$\Pr \left[ \begin{array}{l} \text{ct}^* \in \text{GOOD}_{\text{sk}} \wedge \\ \text{Dec}(\text{sk}, \text{ct}^*) = f(\mu_1, \dots, \mu_m) \end{array} : \begin{array}{l} \bar{f} := \text{Lift}(f) \\ \text{ct}' := \bar{f}(\text{ct}_1, \dots, \text{ct}_m) \\ \text{ct}^* := \text{Refresh}(\text{pk}, \text{ct}') \end{array} \right] = 1.$$

3. For all ciphertexts  $\text{ct}_1, \dots, \text{ct}_\ell \in \text{GOOD}_{\text{sk}}$  such that  $\mu_i := \text{Dec}(\text{sk}, \text{ct}_i)$  for  $i \in [\ell]$ , for any arithmetic circuit  $C : \mathbb{Z}_d^\ell \rightarrow \mathbb{Z}_d^m$ , if we compute

$$(\text{ct}'_1, \dots, \text{ct}'_m) := \text{Eval}(\text{pk}, C, \text{ct}_1, \dots, \text{ct}_\ell) \text{ and } \nu_i := \text{Dec}(\text{sk}, \text{ct}'_i) \text{ for } i \in [m]$$

then it holds that  $\text{ct}'_i \in \text{GOOD}_{\text{sk}}$  for all  $i \in [m]$  and  $(\nu_1, \dots, \nu_m) = C(\mu_1, \dots, \mu_\ell)$ .

**Security:** We require the standard public-key IND-CPA security for the encryption scheme  $(\text{Gen}, \text{Enc}, \text{Dec})$  when  $\text{params} \leftarrow \text{Setup}(1^\lambda, 1^d, 1^D, N)$  for any  $N = \text{poly}(\lambda)$ ,  $d = \text{poly}(\lambda)$ ,  $D = \text{poly}(\lambda)$ .

We also define weaker notions of ASHE-FHE for polylogarithmic degree where we only require the above to hold for  $d = \text{poly} \log(\lambda)$ ,  $D = \text{poly} \log(\lambda)$ . Analogously, we define ASHE-FHE for sub-polynomial degree with  $d = \lambda^{o(1)}$ ,  $D = \lambda^{o(1)}$ .

**Efficiency:** We require that the description length of ring elements, the run-time of the ring operations and the run-time of  $\text{Gen}$ ,  $\text{Enc}$ ,  $\text{Dec}$ ,  $\text{Refresh}$ ,  $\text{Lift}$  are all bounded by  $\text{poly}(\lambda, D, d, \log N)$ . Additionally we require that the run-time of  $\text{Eval}$  is bounded by  $|C| \cdot \text{poly}(\lambda, D, d, \log N)$  where  $C$  is the circuit being evaluated.

**Construction Overview.** We start with the leveled FHE from RingLWE of Brakerski, Gentry and Vaikuntanathan (BGV) [BGV12] and leverage the fact that the ciphertexts in their scheme have the same format as the ciphertexts of the ASHE described in Section 3.1, which is in turn based on [BV11b]. We only make two light modifications to the BGV scheme: (1) we set the plaintext space to be  $\mathbb{Z}_d$ ; as remarked by BGV, all their techniques generalize to this setting, (2) we slightly modify the parameters of the scheme to make sure the evaluated ciphertexts have a larger modulus-to-error ratio to leave room for us to perform an additional ASHE evaluation over them after an FHE evaluation. The resulting scheme is described in Section 6.1. We then convert this leveled FHE scheme into an ASHE-FHE via bootstrapping, where we include a *key cycle* (i.e., an encryption of the secret key under the public key) as part of the public key and set the parameters such that the leveled scheme is capable of evaluating its own decryption function plus one additional arithmetic operation. Security now relies on an additional circular security assumption. This gives us a true (unleveled) FHE where we perform bootstrapping to refresh the ciphertext after each operation.

The ciphertexts maintain the same structure as in our ASHE, and therefore we can also perform ASHE evaluation on them. After an ASHE evaluation, the structure of the ciphertext changes (i.e., it is a higher degree polynomial, the noise level is higher), but there is still a way to decrypt it with the secret key. Therefore, we can refresh the output of an ASHE evaluation back into a good FHE ciphertext via bootstrapping, by using the key cycle to decrypt the ASHE ciphertext under FHE. This is described in detail in Section 6.2.

## 6.1 Leveled FHE from RingLWE

We start by summarizing the result of Brakerski, Gentry and Vaikuntanathan [BGV12], which constructs a leveled FHE from RingLWE. We rely on a light adaptation of their scheme with flexible plaintext space  $\mathbb{Z}_d$ , and an extra gap in the modulus-to-noise ratio of the evaluated ciphertext, as parameterized by  $t$ . The adaptation of their result shows the following.

**Claim 6.1.1** (Leveled Ring FHE [BGV12]). *There is a leveled FHE scheme (Setup, Gen, Enc, Eval, Dec) with the following syntax:*

- $\text{params} := \text{Setup}(1^\lambda, 1^L, 1^d, 1^t)$ : Given a max depth bound  $L$ , plaintext space  $d$ , and a gap parameter  $t$  that allows us to control the modulus-to-noise ratio of the evaluated ciphertext, generates parameters  $\text{params}$  that define some ring  $Q = \mathbb{Z}_q[Z]/(Z^n + 1)$ .
- Gen, Enc, Eval, Dec have the same syntax as in ASHE-FHE 6.1. Furthermore, the secret key  $\text{sk} = s$  is an element of the ring  $Q$ .
- $\text{Dec}(\text{sk} = s, \text{ct})$ : For  $\text{ct} = (a, b) \in Q^2$ , compute  $b - a \cdot s$ , interpret it as a polynomial over a formal variable  $Z$ , and output the constant term of the polynomial taken modulo  $d$ .

The scheme satisfies the following correctness, security and efficiency properties:

**Correctness:** For any  $\text{params} := \text{Setup}(1^\lambda, 1^L, 1^d, 1^t)$  defining the ring  $Q = \mathbb{Z}_q[Z]/(Z^n + 1)$ , any  $\mu_1, \dots, \mu_\ell \in \mathbb{Z}_d^m$ , any arithmetic circuit  $C : \mathbb{Z}_d^\ell \rightarrow \mathbb{Z}_d$  of depth  $\leq L$ : if we choose  $(\text{pk}, \text{sk} = s) \leftarrow \text{Gen}(\text{params})$ ,  $(\text{ct}_i \leftarrow \text{Enc}(\text{pk}, \mu_i))_{i \in [m]}$  and  $\text{ct}' := \text{Eval}(\text{pk}, C, \text{ct}_1, \dots, \text{ct}_\ell)$  then, with probability 1:

- $\text{ct}' = (a, b) \in Q^2$  such that  $b = a \cdot s + d \cdot e + C(\mu_1, \dots, \mu_\ell)$  for some  $e$  such that  $(2n\|e\|)^t \leq q$ . For a sufficiently large  $t = \Omega(1)$  this implies that  $\text{Dec}(\text{sk}, \text{ct}') = C(\mu_1, \dots, \mu_\ell)$ .

**Security:** The scheme satisfies standard public-key IND-CPA security when  $\text{params} \leftarrow \text{Setup}(1^\lambda, 1^L, 1^d, 1^t)$  for any  $d = \text{poly}(\lambda)$ ,  $L = \text{poly}(\lambda)$ ,  $t = \text{poly}(\lambda)$  under RingLWE with sub-exponential approximation factors. If we restrict to  $L = \lambda^{o(1)}$ ,  $t = \lambda^{o(1)}$  then the security holds under RingLWE with sub-sub-exponential approximation factors. If we restrict to  $L = \text{poly} \log \lambda$ ,  $t = \text{poly} \log \lambda$  then the security holds under RingLWE with quasi-polynomial approximation factors.

**Efficiency:** The run-time of Setup, Gen, Enc as well as the description size of ring elements in  $Q$  and the cost of ring-operations over  $Q$  is bounded by  $\text{poly}(\lambda, L, t, \log d)$ . The run-time of Eval is  $|C| \cdot \text{poly}(\lambda, L, t, \log d)$ . For any  $\text{ct} \in Q^2$  the decryption circuit  $C_{\text{ct}}(s) = \text{Dec}(s, \text{ct})$ , represented as an arithmetic circuit with the input  $s$  given in bits, has depth  $\text{poly}(\log \lambda + \log L + \log t + \log \log d)$ .

*Proof overview of [BGV12].* We use the BGV leveled FHE scheme described in [BGV12, Section 3.4] with two light modifications:

- *Plaintext space  $\mathbb{Z}_d$* : The BGV scheme is formally described with plaintext space  $\mathbb{Z}_2$  (or more generally  $\mathbb{Z}_2[Z]/(Z^n+1)$ ), while we need plaintext space  $\mathbb{Z}_d$ . To implement this, we just need to set all the RingLWE noise to  $de$  instead of  $2e$  for  $e \leftarrow \chi$  and all the RingLWE moduli  $q_i$  are chosen to satisfy  $q_i = 1 \pmod d$ .
- *Extra Modulus-to-Noise Gap*: The BGV scheme chooses a “ladder” of  $L + 1$  moduli  $q_L, \dots, q_0$  of decreasing size, where  $q_0 = q$  is the modulus of the evaluated ciphertext. In BGV, the moduli are carefully chosen so that  $q_0 = \text{poly}(\lambda, L, d)$  and  $q_i = \text{poly}(\lambda, L, d) \cdot q_{i-1}$ . The only difference in our case is that we will start with a larger initial modulus  $q_0 = \text{poly}(\lambda, L, d, t)^t$  and also modestly up the increments to  $q_i = \text{poly}(\lambda, L, d, t) \cdot q_{i-1}$ . This will give us extra “gap” in the modulus-to-noise ratio of the evaluated ciphertext, as set by the parameter  $t$ .

Let us now recall the high-level structure of the BGV scheme, with the above adaptations. The parameters define a ladder of moduli  $q_L, \dots, q_0$  as described above with  $q_L = \text{poly}(\lambda, L, d, t)^{L+t}$  and corresponding rings  $Q_\ell = \mathbb{Z}_{q_\ell}[Z]/(Z^n + 1)$ . We also have a ladder of RingLWE secrets  $s_L, \dots, s_0$  with  $s_\ell \in Q_\ell$ . A ciphertext  $\text{ct}$  encrypting a message  $\mu$  is associated with a level  $\ell$  and a noise-amount  $\beta$ ; such a ciphertext is a tuple  $\text{ct} = (a, b) \in Q_\ell$  such that  $b = a \cdot s_\ell + d \cdot e + \mu$  where  $\|e\| \leq \beta$ . A fresh encryption of  $\mu$  is a ciphertext at level  $L$  and noise  $\beta = \text{poly}(\lambda, L, d, t)$ . To evaluate a leveled arithmetic circuit  $C$  of depth  $\leq L$  over encrypted data, the BGV scheme performs the computation level-by-level and gate-by-gate via a sequence of 3 steps:

- **Multiplication/Addition**: Interpret level- $\ell$  ciphertexts  $\text{ct} = (a, b) \in Q_\ell^2$  as formal polynomials  $\text{ct}(Y) = -a \cdot Y + b$ . To add/multiply the messages we add/multiply the polynomials. This results in a polynomial  $\text{ct}_0^*(Y)$  such that  $\text{ct}_0^*(s_\ell) = de^* + \mu^*$  where  $\mu^*$  is the correct output of the gate and  $e^*$  is small. After multiplication, the ciphertext  $\text{ct}_0^*(Y)$  is a degree-2 polynomial in  $Y$ .
- **Relinearization**: Perform a key-switching/relinearization step ([BGV12, Section 3.2]) on  $\text{ct}_0^*$  to get a degree-1 polynomial  $\text{ct}_1^*(Y)$  such that  $\text{ct}_1^*(s_{\ell-1}) = de_1^* + \mu^* \in Q_\ell$  (where  $s_{\ell-1}$  is interpreted as an element in  $Q_\ell$ ). This switches the RingLWE secret from  $s_\ell$  to  $s_{\ell-1}$  but preserves the ring as  $Q_\ell$ .
- **Modulus Reduction**: Perform a modulus reduction step ([BGV12, Section 3.3]) on  $\text{ct}_1^* \in Q_\ell[Y]$  to get  $\text{ct}_2^* \in Q_{\ell-1}[Y]$  such that  $\text{ct}_2^*(s_{\ell-1}) = de_2^* + \mu^* \in Q_{\ell-1}$ . This changed the modulus from  $q_\ell$  to  $q_{\ell-1}$  and also reduces the noise from  $e_1^*$  to  $e_2^*$ .

The choice of parameters is such that the resulting ciphertext  $\text{ct}_2^*$  is an encryption of the the gate’s output  $\mu^*$  at level  $\ell - 1$  with the same noise level  $\beta = \text{poly}(\lambda, L, d, t)$ . By continuing this process for  $L$  levels (if the circuit has depth  $\leq L$  we pad it to make it exactly  $L$ ) the output is a ciphertext  $\text{ct}^*$  that encrypts the output of the circuit  $C(\mu_1, \dots, \mu_\ell)$  at level 0 with noise level  $\beta = \text{poly}(\lambda, L, d, t)$ . We can choose  $q_0 = \text{poly}(\lambda, L, d, t)^t$  sufficiently large that  $(2n\beta)^t \leq q_0$  as needed for correctness. With that, Setup chooses  $Q = Q_0$ , and Gen outputs secret key  $s = s_0$  at level 0.  $\square$

We will also rely on the following circular security assumption, which is the same assumption as needed to get unlevelled FHE from (Ring)LWE in [BGV12].

**Assumption 6.2** (Circular Security Assumption). *The circular security assumption says that the leveled FHE encryption scheme from the above Claim 6.1.1 remains secure, even if we give out the encryptions  $(\text{Enc}(\text{pk}, \text{sk}_i))_{i \in |\text{sk}|}$  of each of the bits  $\text{sk}_i$  of the secret key  $\text{sk}$  under the public key  $\text{pk}$ .*

## 6.2 ASHE-FHE Construction

We now show how to leverage the leveled Ring FHE from Claim 6.1.1 to construct an ASHE-FHE. The main ideas are: (1) use an encrypted key cycle and bootstrapping to go from leveled FHE to non-leveled FHE, (2) use the fact that the FHE-evaluated ciphertexts have the same structure as the ciphertexts of the ASHE scheme in Section 3.1 to make the scheme an ASHE, (3) after performing an ASHE-evaluation of a low degree polynomial, we get some ASHE ciphertext  $\text{ct}_{\text{ASHE}}$  that is decryptable using the secret key  $\text{sk}$ ; refresh it into a good FHE ciphertext by homomorphically decrypting  $\text{ct}_{\text{ASHE}}$  under FHE, using the FHE encryption of the secret key  $\text{sk}$ .

Let  $(\text{Setup}, \text{Gen}, \text{Enc}, \text{Eval}, \text{Dec})$  be the (lightly modified) BGV leveled Ring FHE from Claim 6.1.1. We construct an ASHE-FHE  $(\text{Setup}', \text{Gen}', \text{Enc}', \text{Eval}', \text{Dec}, \text{Refresh}, \text{Lift})$  as follows:

---

### Algorithm 6.3: ASHE-FHE

---

**params** :=  $\text{Setup}'(1^\lambda, 1^d, 1^D, N)$ : Set  $t := D \log d + \log N + \log d + 1$ . Choose  $L^*(\lambda, d, t) = \text{poly}(\log \lambda + \log t + \log \log d)$  such that, for **params** :=  $\text{Setup}(1^\lambda, 1^{L^*}, 1^d, 1^t)$ , the decryption circuit  $C_{\text{ct}}(s) = \text{Dec}(s, \text{ct})$  has depth  $\leq L^* - 1$ , which is always possible since the depth is bounded by  $\text{poly}(\log \lambda + \log L^* + \log t + \log \log d)$ . Output **params** :=  $\text{Setup}(1^\lambda, 1^{L^*}, 1^d, 1^t)$ . This ensures the BGV scheme can evaluate its own decryption.

The **params** define the ring  $Q = \mathbb{Z}_q[Z]/(Z^n + 1)$ . Let

$$R = Q[Y]/(Y^D + 1) \cong \mathbb{Z}_q[Y, Z]/(Y^D + 1, Z^n + 1)$$

be the ring of the ASHE-FHE.

**(pk', sk)**  $\leftarrow$   $\text{Gen}'(\text{params})$ : Sample  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params})$ . For  $i = 1, \dots, |\text{sk}|$ , let  $\text{sk}[i]$  be the  $i$ 'th bit of  $\text{sk}$  and sample  $\text{ct}_{\text{sk}[i]} \leftarrow \text{Enc}(\text{pk}, \text{sk}[i])$ . Let  $\overline{\text{ct}}_{\text{sk}} = (\text{ct}_{\text{sk}[i]})_{i \in |\text{sk}|}$ . Output  $(\text{pk}' = (\text{pk}, \overline{\text{ct}}_{\text{sk}}), \text{sk})$ .

**ct'**  $\leftarrow$   $\text{Enc}'(\text{pk}, \mu)$ : Run the leveled FHE encryption  $\text{ct} \leftarrow \text{Enc}(\text{pk}, \mu)$ . Evaluate the ‘‘dummy’’ identity circuit  $I_{L^*}(\mu) = \mu$ , which is padded to depth  $L^*$ , and output  $\text{ct}' := \text{Eval}(\text{pk}, I_{L^*}, \text{ct})$ .<sup>19</sup>

**Dec**: This is the same as in the BGV leveled Ring FHE.

**(ct'\_1, \dots, ct'\_m)** :=  $\text{Eval}'(\text{pk}', C, \text{ct}_1, \dots, \text{ct}_\ell)$ : Given an arithmetic circuit  $C : \mathbb{F}_d^\ell \rightarrow \mathbb{F}_d^m$ , evaluate it gate-by-gate starting from the input level by computing a ciphertext for each wire in the circuit. The ciphertexts for the input wires are  $\text{ct}_1, \dots, \text{ct}_\ell$ . For any arithmetic gate  $g$  with input wires  $a, b$  and output wire  $c$ , assume we have ciphertexts  $\text{ct}_a, \text{ct}_b$  for the input wires. Let  $C_{g, \text{ct}_a, \text{ct}_b}(\text{sk})$  be the arithmetic circuit of depth  $L^*$  that takes the bits of  $\text{sk}$  as an input, evaluates  $\mu_a := \text{Dec}(\text{sk}, \text{ct}_a), \mu_b := \text{Dec}(\text{sk}, \text{ct}_b)$  and outputs  $g(\mu_a, \mu_b)$ . Let  $\text{ct}_c = \text{Eval}(\text{pk}, C_{g, \text{ct}_a, \text{ct}_b}, \overline{\text{ct}}_{\text{sk}})$ .

**ct'** :=  $\text{Refresh}(\text{pk}', \text{ct})$ : Parse  $\text{ct} \in R = Q[Y]/(Y^D + 1)$  as a polynomial  $\text{ct}(Y)$  of degree  $< D$ . Let  $C_{\text{ct}}(\text{sk})$  be the arithmetic circuit that performs ASHE decryption: it takes as input the bits of  $\text{sk} = s \in Q$ , computes  $g = \text{ct}(s) \in Q = \mathbb{Z}_q[Z]/(Z^n + 1)$  and outputs the constant term of  $g$  modulo  $d$ . Output  $\text{ct}' := \text{Eval}'(\text{pk}', C_{\text{ct}}, \overline{\text{ct}}_{\text{sk}})$ .<sup>20</sup>

---

<sup>19</sup>We perform the evaluation of the identity circuit as a technicality to ensure that the outputs of  $\text{Enc}'$  have the same format as the outputs of  $\text{Eval}$ .

<sup>20</sup>Note that here we are using  $\text{Eval}'$  rather than  $\text{Eval}$ , so we do not need to worry the depth of the circuit  $C_{\text{ct}}$ , which may be (slightly) larger than  $L^*$ . Alternately, we could have chosen  $L^*$  slightly larger so that it is bigger than the depth of  $C_{\text{ct}}$ , which is still  $\text{poly}(\log \lambda + \log L + \log t + \log \log d)$ , in which case we could have just used  $\text{Eval}$  instead of  $\text{Eval}'$  to avoid many bootstrapping operations.

Lift: Interpret  $\mu \in \mathbb{Z}_d$  as an element of  $R$ .

---

**Theorem 6.4.** *The above is an ASHE-FHE assuming RingLWE with sub-exponential approximation factors and the circular-security assumption (6.2). Alternately, it is an ASHE-FHE for poly-logarithmic (resp. sub-polynomial) degree under RingLWE with quasi-polynomial (resp. sub-sub-exponential) approximation factors and the circular-security assumption (6.2).*

*Proof.* We analyze correctness, security and efficiency in turn.

For correctness, we define the set  $\text{GOOD}_{\text{sk}}$  for  $\text{sk} = s$  to consist of ciphertexts  $\text{ct} = (a, b) \in Q^2$  such that  $b = a \cdot s + d \cdot e + \mu$  for some  $e$  such that  $(2n\|e\|)^t \leq q$  and some  $\mu \in \mathbb{Z}_d$ . With this definition, it is easy to see that correctness properties (1) and (3) hold by the correctness property of the leveled FHE. Correctness property (2) holds by first applying the exact same analysis as in our ASHE from Theorem 3.2 to argue that if  $\text{ct}' = \bar{f}(\text{ct}_1, \dots, \text{ct}_\ell)$  is the output of the ASHE computation over ciphertexts  $\text{ct}_i \in \text{GOOD}_{\text{sk}}$  encrypting plaintexts  $\mu_i$ , then we have  $C_{\text{ct}'}(\text{sk}) = f(\mu_1, \dots, \mu_\ell)$ , where  $C$  is the arithmetic circuit defined in our Refresh procedure, which corresponds to the ASHE decryption defined there. Then we rely on correctness property (3) to argue that  $\text{ct}^* = \text{Eval}'(\text{pk}', C_{\text{ct}'}, \bar{\text{ct}}_{\text{sk}}) \in \text{GOOD}_{\text{sk}}$  and that it decrypts to the correct value  $f(\mu_1, \dots, \mu_\ell)$ .

Security follows directly from that of the BGV leveled Ring FHE from Claim 6.1.1, modulo the fact that the adversary also gets a key cycle, which preserves security via the circular security assumption.

The efficiency of the ASHE-FHE follows directly from the efficiency properties of the BGV leveled Ring FHE from Claim 6.1.1.  $\square$

**Remark 6.1** (ASHE-FHE with Key Cycle). In the above scheme, the public key of the ASHE-FHE contains an *encrypted key cycle*  $\bar{\text{ct}}_{\text{sk}}$ , which is a vector of ciphertexts, each of which is in  $\text{GOOD}_{\text{sk}}$ , that decrypt to  $\text{sk}$ . We refer to an ASHE-FHE with this property as an *ASHE-FHE with a key cycle*, and will rely on it later.

**Remark 6.2** (Leveled ASHE-FHE). Note that an ASHE-FHE scheme can generically be used as an unleveled FHE, allowing for the evaluation of arbitrary boolean circuits of unbounded depth. Currently, all known constructions of such unleveled FHE schemes rely on a “circular security” assumption, and we do as well. However, we can also define a *leveled ASHE-FHE* and construct it using the same template as above.

For the definition of leveled ASHE-FHE, the Setup algorithm is the same, but Gen takes in an additional parameter  $1^L$  denoting the maximal number of levels the scheme supports. Each ASHE-FHE ciphertext is now also associated with a level  $i$  and the sets  $\text{GOOD}_{\text{sk}}^i$  now denote the set of good ciphertexts at level  $i$ . Correctness property 1 is now modified to ensure that a fresh encryption is a good ciphertext at level 1 with  $\text{ct} \in \text{GOOD}_{\text{sk}}^1$ . Correctness property 2 is modified so that, if  $\text{ct}_1, \dots, \text{ct}_m \in \text{GOOD}_{\text{sk}}^i$  are good level- $i$  ciphertexts then the results of ASHE evaluation followed by a refresh satisfies  $\text{ct}^* \in \text{GOOD}_{\text{sk}}^{i+1}$  is a good ciphertext at level  $i + 1$ . Correctness property 3 is modified so that, if  $\text{ct}_1, \dots, \text{ct}_\ell \in \text{GOOD}_{\text{sk}}^i$  are good level- $i$  ciphertexts then the results of FHE evaluation for an arithmetic circuit  $C$  of depth  $\rho$  results in a ciphertext  $\text{ct}' \in \text{GOOD}_{\text{sk}}^{i+\rho}$ , which is a good ciphertext at level  $i + \rho$ .

For the construction of leveled ASHE-FHE, instead of an *encrypted key-cycle*, we choose  $L + 1$  public/secret keys  $(\text{pk}_i, \text{sk}_i)$  of the BGV scheme and add an *encrypted key ladder*  $\text{Enc}(\text{pk}_{i+1}, \text{sk}_i)$  for  $i \in [L]$  to the public key. The secret key consists of  $\text{sk}_L$ . The scheme is otherwise the same, but

each time we perform bootstrapping during Eval', Refresh we use the next secret key in the ladder, which results in a ciphertext under  $\text{pk}_{i+1}$ .<sup>21</sup> The efficiency of the leveled ASHE-FHE is the same as the unlevelled scheme *except* that the run-time of Gen and the size of the public key  $\text{pk}$  are now also linear in  $L$ , and bounded by  $L \cdot \text{poly}(\lambda, D, d, \log N)$ .

## 7 RAM-FHE

We define and construct *fully homomorphic encryption for RAMs (RAM-FHE)* in this section. A RAM-FHE is a public-key encryption scheme, where a client can encrypt some input  $x$  to derive a ciphertext  $\tilde{\text{ct}}_x$ . The server can independently preprocess some plaintext input  $y$  into a static data structure  $\tilde{y}$ . The server can then homomorphically evaluate an arbitrary RAM program  $P(x, y)$  by operating over  $\tilde{\text{ct}}_x, \tilde{y}$  to derive an encryption of the output. Our notion generalizes that of [HHWW19] by allowing the server to incorporate a preprocessed plaintext input  $y$  into the computation. We give a formal definition in Section 7.1.

We present our construction via a sequence of three schemes in Sections 7.2, 7.3, and 7.4, with each latter scheme building on the previous to improve efficiency. The first scheme only allows read-only random-access to the plaintext input  $y$ , but not to the encrypted input  $x$ , or to any additional read/write memory. The second scheme also allows read-only random-access to  $x$ . Our final third scheme also allows additional random-access read/write memory  $z$ .

### 7.1 Definition of the RAM Model and RAM-FHE

In this subsection, we define the abstraction of RAM-FHE. We will start by defining a random-access machine (RAM) model that captures the efficiency requirements of RAM-FHE.

#### 7.1.1 The RAM model

In our RAM model, we consider programs  $P$  that take two inputs  $x, y$ , and we denote the output of the evaluation by  $P(x, y)$ . We think of  $x, y$  as stored in read-only random access memory. The program also has random access to additional read-write enabled memory  $z$ . The evaluation of a RAM program consists of a series of steps where, in each step, the program may write one bit to an arbitrary position in  $z$  and then may read one bit each from arbitrary positions in  $x, y$  and  $z$ . The specification of what a step does is given by some step circuit  $C$ , which we can think of as implementing some simple "CPU".

**Definition 7.1.** A RAM program  $P$  with some fixed input size is defined by a tuple  $(C, T, S, M)$ , where  $C$  is a binary circuit called the *step circuit* and  $T, S, M \in \mathbb{N}$  are the *worst-case time complexity, space complexity and output length of  $P$  respectively*. The execution of  $P(x, y)$  is performed iteratively. We first initialize state<sup>0</sup> to be the all-zeros string,  $b_x^0 := 0, b_y^0 := 0, b_z^0 := 0$  and  $z := 0^S$ . Then, for each step  $t \in [T]$ :

1. Call  $(\text{state}^t, i_x^t, i_y^t, i_z^t, i_w^t, b_w^t) := C(\text{state}^{t-1}, b_x^{t-1}, b_y^{t-1}, b_z^{t-1})$ .

<sup>21</sup>Note that the encrypted key ladder is separate from the ladder of moduli used in the construction of the BGV scheme discussed in Section 6.1, which we still only choose large enough for the scheme to be able to evaluate its own decryption. Instead, with the encrypted key ladder, the choice of  $L$  only affects the size of the public-key but not the cost of each homomorphic operation. Also, we assume we are using the optimization discussed in footnote 20, so that a refresh only increases the level by 1.

2. Write  $z[i_w^t] := b_w^t$  to memory.
3. Read  $b_x^t := x[i_x^t]$ ,  $b_y^t := y[i_y^t]$ , and  $b_z^t := z[i_z^t]$ .

The output of the evaluation  $P(x, y)$  is given by the first  $M$  bits of the final memory contents  $z^T$  at the end of the execution.

**Remark on the Step Circuit and Universal RAM.** The above gives us a lot of flexibility over what information to put in the circuit  $C$  vs. encrypted input  $x$  vs. the plaintext input  $y$  vs. the state vs. the memory  $z$ . Most naturally, we can implement a universal RAM computer, where  $C$  is the circuit performing a single generic CPU step and state corresponds to some constant number of registers in the CPU. In that case, we have  $|C| = \text{poly log}(|x| + |y| + S)$ . We can then store the “code” of the actual program that we want to execute as part of the inputs  $x$  or  $y$ , depending on whether the client or the server chooses it.

**Remark on Uniform vs. Non-Uniform Model.** We also note that the above is a non-uniform model of computation, where the program  $P$  is tailored to some particular input sizes  $|x|, |y|$  and needs to specify a corresponding run-time  $T$  and space complexity  $S$ . This fits our setting where the server knows the sizes  $|x|, |y|$  when homomorphically evaluating the program, and can therefore tailor the description of the program  $P$  to these particular sizes. Naturally, we expect that there is some simple uniform computation that the server can perform to come up with the above specification of  $P$  given  $|x|, |y|$ , but we leave this outside of our formalism.

### 7.1.2 RAM-FHE

Under the above model of RAM computation, we define fully homomorphic encryption for RAMs (RAM-FHE). The definition is analogous to that of circuit FHE: any RAM program  $P$  can be evaluated homomorphically on an encrypted input  $x$  and a preprocessed input  $y$ , and the running time of this homomorphic evaluation is proportional to that of running  $P$  in the clear. In particular, because  $P$  is a RAM program, this means that the evaluation may take time significantly less than the length of the inputs.

**Definition 7.2 (RAM-FHE).** A RAM-FHE scheme is a tuple of algorithms (Setup, Gen, Prep, Enc, Eval, Dec) with the following syntax:

- $\text{params} := \text{Setup}(1^\lambda, N_x, N_y, N_S)$ : On input security parameter  $\lambda$  and space bounds  $N_x, N_y$  and  $N_S$ , it deterministically outputs public parameters  $\text{params}$ .
- $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params})$ : Given public parameters  $\text{params}$ , it samples and outputs a public key  $\text{pk}$  and a secret key  $\text{sk}$ .
- $\tilde{y} := \text{Prep}(\text{params}, y)$ : Given public parameters  $\text{params}$  and a database  $y$ , it outputs a preprocessed database  $\tilde{y}$ .
- $\tilde{\text{ct}}_x \leftarrow \text{Enc}(\text{pk}, x)$  Given a public key  $\text{pk}$  and a database  $x$ , it outputs a preprocessed ciphertext  $\tilde{\text{ct}}_x$ .
- $\text{ct}_{\text{out}} := \text{Eval}(\text{pk}, P, \tilde{\text{ct}}_x, \tilde{y})$ : Given a public key  $\text{pk}$ , RAM program  $P$ , preprocessed ciphertext  $\tilde{\text{ct}}_x$ , and preprocessed database  $\tilde{y}$ , it outputs an output ciphertext  $\text{ct}_{\text{out}}$ .
- $\mu := \text{Dec}(\text{sk}, \text{ct})$ : Given a secret key  $\text{sk}$  and a ciphertext  $\text{ct}$ , it outputs a plaintext  $\mu$ .

**Correctness.** For any  $\lambda, N_x, N_y, N_S \in \mathbb{N}$ , any RAM program  $P = (C, T, S, M)$  and any input pair  $(x, y)$  such that  $|x| \leq N_x$ ,  $|y| \leq N_y$ , and  $S \leq N_S$ , it must hold that

$$\Pr \left[ \begin{array}{l} \text{Dec}(\text{sk}, \text{ct}_{\text{out}}) = P(x, y) : \\ \text{params} := \text{Setup}(1^\lambda, N_x, N_y, N_S) \\ (\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params}) \\ \tilde{y} := \text{Prep}(\text{params}, y) \\ \tilde{\text{ct}}_x \leftarrow \text{Enc}(\text{pk}, x) \\ \text{ct}_{\text{out}} := \text{Eval}(\text{pk}, P, \tilde{\text{ct}}_x, \tilde{y}) \end{array} \right] = 1.$$

**Security.** We require that standard public-key IND-CPA security holds for all  $\lambda \in \mathbb{N}$  and all space bounds  $N_x, N_y, N_S = \text{poly}(\lambda)$ . Namely, we define the following game between a challenger and a stateful adversary  $\mathcal{A}$ :

1. The adversary  $\mathcal{A}(1^\lambda)$  selects sizes  $1^{N_x}, 1^{N_y}, 1^{N_S}$ .
2. The challenger computes  $\text{params} := \text{Setup}(1^\lambda, N_x, N_y, N_S)$  and samples  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params})$ .
3.  $\mathcal{A}$  gets  $\text{pk}$  and selects a challenge pair  $x_0, x_1 \in \{0, 1\}^*$  with  $|x_0| = |x_1|$ .
4. The challenger samples a random bit  $b \leftarrow \{0, 1\}$ , and runs  $\tilde{\text{ct}}_x \leftarrow \text{Enc}(\text{pk}, x_b)$ .
5.  $\mathcal{A}$  gets  $\tilde{\text{ct}}_x$  outputs a guess  $b'$ .

The advantage of  $\mathcal{A}$  is defined to be  $|\Pr[b' = b] - 1/2|$ , over the randomness of the challenger and  $\mathcal{A}$ . We require that for every stateful PPT adversary  $\mathcal{A}$ , there exists a negligible function  $\text{negl}$  such that the distinguishing advantage of  $\mathcal{A}$  is bounded by  $\text{negl}(\lambda)$ :

**Remark on Efficiency.** Note that we did not specify a formal efficiency requirement for RAM-FHE, but will clearly state the run-times of the procedures (Setup, Gen, Prep, Enc, Eval, Dec) in our construction(s). We measure efficiency of these procedures in the context of the correctness experiment in the above definition, where  $\lambda, N_x, N_y, N_S \in \mathbb{N}$ ,  $P = (C, T, S, M)$  is some RAM program, and  $(x, y)$  are some inputs such that  $|x| < N_x$ ,  $|y| < N_y$ , and  $S < N_S$ . Let  $N := \max\{N_x, N_y, N_S\}$ . Ideally, we want the run-time of Setup, Gen to just be  $\text{poly}(\lambda, \log N)$ , the run-time of Enc should be nearly linear in  $|x|$ , the run-time of Prep should be nearly linear in  $|y|$ , the run-time of Dec should be nearly linear in  $|P(x, y)| = M$  and the run-time of Eval should be nearly linear in  $T$ .

**Remark on Reusable Plaintext Input.** Notice that the preprocessing of plaintext  $y$  in our definition is *unkeyed*, and does not depend on the public-key  $\text{pk}$ . This means that the server can reuse the same preprocessed data structure  $\tilde{y}$  in many different homomorphic computations with different clients that encrypt their inputs  $x$  under different keys.

**Remark on the Bounds  $N_x, N_y, N_S$ .** Our schemes require setting some upper bounds on the inputs sizes  $N_x, N_y$  and the space-complexity  $N_S$  already during Setup and the efficiency of our schemes can depend on these bounds, even if the actual values of  $|x|, |y|, S$  used later are smaller. However, the complexity of our eventual schemes will only either scale with  $\text{poly} \log(N)$ , or with  $N^{o(1)}$  where  $N := \max\{N_x, N_y, N_S\}$ . Therefore, we can even set these bounds to some slightly super-polynomial value  $\lambda^{\omega(1)}$  while only incurring some fixed  $\text{poly}(\lambda)$  overhead. This means that the requirement that we know these upper bounds ahead of time does not impose much of a restriction.

**Remark on DEPIR as a Special Case of RAM-FHE.** Note that RAM-FHE gives a DEPIR as a special case. In particular, let  $P = (C, T, S, M)$  be the RAM program that takes as input  $x \in \llbracket N \rrbracket$  and  $y = \text{DB} \in \{0, 1\}^N$  and outputs  $\text{DB}[x]$ . This program has  $|C| = O(\log N)$ ,  $S = O(\log N)$ ,  $T = O(\log N)$  and  $M = 1$ . The Prep algorithm of the DEPIR is the same as that of the RAM-FHE. The  $\text{Query}(1^\lambda, N, i)$  algorithm of DEPIR runs the  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params}), \tilde{\text{ct}}_x \leftarrow \text{Enc}(\text{pk}, x)$  algorithms of RAM-FHE and sets the DEPIR ciphertext  $\text{ct} = (\text{pk}, \tilde{\text{ct}}_x)$  and the key as  $s = \text{sk}$ . The  $\text{Resp}(\tilde{\text{DB}}, \text{ct})$  algorithm of the DEPIR runs  $\text{ct}_{\text{out}} := \text{Eval}(\text{pk}, P, \tilde{\text{ct}}_x, \tilde{\text{DB}})$ , and the Dec algorithm of the DEPIR is the same as that of the RAM-FHE.

**Remark on Leveled RAM-FHE.** We can also define a relaxed variant of RAM-FHE that can only evaluate RAM programs  $P$  with some a-priori bounded run-time  $T \leq T_{\text{max}}$ , where  $T_{\text{max}}$  is chosen during key generation, and the efficiency of key generation and the size of the public-key can grow linearly with  $T_{\text{max}}$ . We discuss this variant in Section 8.2 and show that we can instantiate it without circular security.

## 7.2 RAM-FHE with Random Access to $y$

We start with a construction of RAM-FHE for programs  $P(x, y)$  that only have random-access to the plaintext input  $y$ , but not the encrypted input  $x$  nor the additional read/write memory  $z$ . We will then upgrade it in future sections to get our full construction.

Instead of modifying the definition of the RAM model or the notion of RAM-FHE to capture this setting, we leave the definitions as is, but settle for poor efficiency in terms of the dependence on the size of the encrypted input  $|x|$  and on the space complexity  $S = |z|$ . In particular, our homomorphic evaluation procedure will simply read the entire encrypted input and the entire encrypted memory in each step of the program execution. However, it will still only access a small number of locations in the preprocessed plaintext input  $\tilde{y}$  in each step. Note that this is already useful in many applications where the program has small (e.g., poly-logarithmic) space complexity and a small encrypted input. For example, this already suffices for binary search with a small encrypted search term over a large preprocessed plaintext database, as was needed to achieve round-optimal updatable DEPIR in Section 5.2.

**Construction Overview.** We rely on an ASHE-FHE scheme. The server preprocesses the long input  $y$  into a data structure  $\tilde{y}$  the same way as in our DEPIR. The client encrypts the input  $x$  using the ASHE-FHE. To evaluate the program  $P$ , the server proceeds in a sequence of steps, where in each step it evaluates the circuit  $C$  under encryption using the fully homomorphic evaluation of the ASHE-FHE. Initially, it starts with an encryption of the all-0's state  $\text{state}^0$  and an all-0s memory  $z$ . After performing the FHE evaluation of  $C$  in step  $t$ , it gets an encryption of the updated state  $\text{state}^t$  as well as encryptions of the values  $i_x^t, i_y^t, i_z^t, i_w^t, b_w^t$  denoting the indices to read from  $x, y, z$  and the index/bit to write to  $z$ . The server can perform simple FHE evaluations to derive encryptions of  $x[i_x^t]$  and  $z[i_z^t]$ , as well as to update the encryption of  $z$  by setting  $z[i_w^t] = b_w^t$ , all in time linear in  $|x| + |z|$  (without relying on random access). To read from  $y$ , the server can first convert the FHE encryption of the bits of  $i_y^t$  into encryptions of the base- $d$  digits of  $i_y^t$  (relying on the fact that the fully homomorphic evaluation of the ASHE-FHE supports algebraic circuits over  $\mathbb{Z}_d$ ), which is exactly the same as a DEPIR query for the index  $i_y^t$  in our DEPIR scheme. The server can then use the DEPIR to compute the answer for this query using random access to  $\tilde{y}$ ,

which results in some ciphertext  $ct'$  encrypting  $y[i_y^t]$ . The server refreshes the ciphertext  $ct'$  so it can be used in future evaluations. It then uses the encryptions of  $state^t, x[i_x^t], y[i_y^t], z[i_z^t]$  and the encrypted memory contents  $z^t$  to evaluate the next step of the computation, and so on.

**Construction.** The construction of this RAM-FHE (Setup, Gen, Prep, Enc<sub>p</sub>, Eval<sub>p</sub>, Dec) is given below. We denote the encryption and evaluation of this scheme using subscripts Enc<sub>p</sub> and Eval<sub>p</sub> respectively, and will eventually replace them in subsequent sections to improve efficiency for our final construction. As a building block, we rely on a generic ASHE-FHE scheme (Setup\*, Gen\*, Enc\*, Eval\*, Dec\*, Refresh\*, Lift\*). We also rely on our construction of DEPIR (Prep', Query', Resp', Dec') from ASHE from Algorithm 4.3, where we plug in the ASHE-FHE scheme in the place of the ASHE. We abuse notation and, for a vector for messages  $x = (x_1, \dots, x_\ell)$  we define  $Enc^*(pk, x) := (Enc^*(pk, x_i))_i$  to work component-wise. Similarly, for a vector of ciphertexts  $ct = (ct_1, \dots, ct_\ell)$  we define  $Dec^*(sk, ct) := (Dec^*(sk, ct_i))_i$  component-wise.

---

**Algorithm 7.3: RAM-FHE scheme with random access to  $y$ .**

---

**Building blocks:** We use a generic ASHE-FHE scheme (Setup\*, Gen\*, Enc\*, Eval\*, Dec\*, Refresh\*, Lift\*), and the specific construction of DEPIR (Prep', Query', Resp', Dec') from ASHE in Algorithm 4.3, where we plug in the ASHE-FHE scheme in place of the ASHE.

**Algorithms:**

Setup( $1^\lambda, N_x, N_y, N_S$ ): Let  $N := \max\{N_x, N_y, N_S\}$ . Choose a prime  $d$  as specified later. Let  $m := \lceil \log_d N \rceil$  and let  $D := d \cdot m$ . Set parameters for ASHE-FHE scheme as  $params^* := Setup^*(1^\lambda, 1^d, 1^D, d^m)$  and output  $params := (params^*, d)$ . Let  $R$  be the ASHE-FHE ring.

Gen( $params$ ): Output  $(pk, sk) \leftarrow Gen^*(params^*)$ .

Dec( $sk, ct$ ): Output  $Dec^*(sk, ct)$ .

Enc<sub>p</sub>( $pk, x$ ): Output  $\tilde{ct}_x \leftarrow Enc^*(pk, x)$ , where we interpret  $x$  as a vector of bits.

Prep( $params, y$ ): Perform DEPIR preprocessing  $\tilde{y} := Prep'(1^\lambda, y)$ , with the modification that the polynomial  $f_y := ToPoly_{d, m_y}(y)$  in step 1 is computed using  $m_y := \lceil \log_d |y| \rceil$  instead of  $m$  for the number of variables. Output  $\tilde{y}$ .<sup>22</sup>

Eval<sub>p</sub>( $pk, P, \tilde{ct}_x, \tilde{y}$ ): Let  $P = (C, T, S, M)$  consists of the step circuit  $C$ , the running time  $T$ , the space  $S$ , and the output size  $M$ . The homomorphic evaluation of  $P$  proceeds as follows:

1. Initialize  $state^0, b_x^0, b_y^0, b_z^0, z^0$  to all 0's strings of appropriate size. Let  $ct_{state}^0, \hat{b}_x^0, \hat{b}_y^0, \hat{b}_z^0, \hat{z}^0$  be public-key encryptions of the corresponding values under the ASHE-FHE, with random coins set to all 0's.
2. For each step  $t$  from 1 to  $T$ :

---

<sup>22</sup>Without loss of generality, we will assume that we can recover  $|y|$  from  $\tilde{y}$ , which also lets us compute the value  $m_y$ . Note that in DEPIR, we knew the size of DB already when we chose the parameters of the ASHE, while here we only know an upper bound  $N_y$  on the size of  $y$  when we choose ASHE-FHE parameters. Therefore, the values  $d, m$  chosen at setup corresponds to the maximal size of  $y$ . However, when we actually preprocess  $y$ , the number of variables in the polynomial  $f_y$  can be set to  $m_y \leq m$ .

- (a) Using ASHE-FHE, homomorphically evaluate the step circuit  $C$  on the encrypted state and values at the  $t$ 'th step. That is,

$$(\text{ct}_{\text{state}}^t, \hat{i}_x^t, \hat{i}_y^t, \hat{i}_z^t, \hat{i}_w^t, \hat{b}_w^t) := \text{Eval}^*(\text{pk}, C, \text{ct}_{\text{state}}^{t-1}, \hat{b}_x^{t-1}, \hat{b}_y^{t-1}, \hat{b}_z^{t-1}), \quad (1)$$

where we interpret the boolean circuit  $C$  as an arithmetic circuit over  $\mathbb{F}_d$  that works correctly with inputs/outputs in binary.

- (b) Read from  $\tilde{\text{ct}}_x$  by setting

$$\hat{b}_x^t := \text{Eval}^*(\text{pk}, C_{\text{read}}, \tilde{\text{ct}}_x, \hat{i}_x^t),$$

where  $C_{\text{read}}$  is a boolean circuit such that  $C_{\text{read}}(x, i) = x[i]$ , and we naturally reinterpret it as an arithmetic circuit over  $\mathbb{F}_d$ .

- (c) Convert the encrypted index  $\hat{i}_y^t$  to base- $d$ , and then read from  $\tilde{y}$  using DEPIR, and finally refresh the resulting ciphertext to allow for future use in FHE evaluation:

$$\hat{i}' := \text{Eval}^*(\text{pk}, \text{base}_{d, m_y}, \hat{i}_y^t), \quad \text{ct}' := \text{Resp}'(\tilde{y}, \hat{i}'), \quad \hat{b}_y^t := \text{Refresh}^*(\text{pk}, \text{ct}')$$

where  $\text{base}_{d, m_y} : \llbracket |y| \rrbracket \rightarrow \mathbb{F}_d^{m_y}$  is an arithmetic circuit of size  $\text{poly} \log(N)$  that takes an index  $i \in \llbracket |y| \rrbracket$  written in binary and outputs the base- $d$  representation of  $i$ .<sup>23</sup> Note that  $\hat{i}' \in R^{m_y}$  and  $\text{ct}' = \bar{f}_y(\hat{i}')$ , where  $\bar{f}_y$  is the lifted version of the polynomial  $f_y(i) = y[i]$ .

- (d) Write to and then read from the memory  $\hat{z}$ :

$$\hat{z}^t := \text{Eval}^*(\text{pk}, C_{\text{write}}, \hat{z}^{t-1}, \hat{i}_w^t, \hat{b}_w^t), \quad \hat{b}_z^t := \text{Eval}^*(\text{pk}, C_{\text{read}}, \hat{z}^t, \hat{i}_z^t),$$

where  $z' := C_{\text{write}}(z, i^*, b^*)$  is the circuit that updates the array  $z$  by setting  $z[i^*] := b^*$  and then outputs the updated array  $z'$ , and  $C_{\text{read}}$  is defined as in 2a.

3. Let  $\text{ct}_{\text{out}}$  be the first  $M$  ciphertext ring elements in the final encrypted memory  $\hat{z}^T$ . Output  $\text{ct}_{\text{out}}$ .

**Analysis.** The **security** directly follows by the IND-CPA security of the ASHE-FHE encryption.

The **correctness** follows by induction on  $t$ . We say that the  $t$ 'th *encrypted configuration* consisting of the ciphertexts

$$(\text{ct}_{\text{state}}^t, \hat{b}_x^t, \hat{b}_y^t, \hat{b}_z^t, \hat{z}^t)$$

is *correct* if all the ciphertexts are in  $\text{GOOD}_{\text{sk}}$  and if they decrypt to the correct values matching the  $t$ 'th configuration of the the computation  $P(x, y)$ . In the base case, correctness holds for  $t = 0$  since outputs of  $\text{Enc}^*(\text{pk}, \cdot)$  are in  $\text{GOOD}_{\text{sk}}$  with probability 1 over the encryption randomness, and therefore even if we use the all 0's randomness. Suppose by induction that correctness holds for  $t$ . Then it also holds for  $t + 1$ . In particular, steps 2a, 2b, 2d produce the correct values by the correctness of ASHE-FHE  $\text{Eval}^*$ . In step 2c, the ciphertext  $\hat{i}' = (\hat{i}'_1, \dots, \hat{i}'_{m_y})$  consists of good ASHE-FHE encryptions of the base- $d$  digits of  $i_y^t$ , by the correctness of  $\text{Eval}^*$ . Then, going under

<sup>23</sup>In more detail, we can implement the circuit for the function  $\text{base}_{d, m_y}(i) = (i_1, \dots, i_{m_y})$  by first applying a boolean circuit that computes the binary representation  $i_j = (i_{j,0}, \dots, i_{j, \lceil \log d \rceil - 1})$  of each digit  $i_j$  and then applying arithmetic gates to compute  $i_j = \sum_{k=0}^{\lceil \log d \rceil - 1} i_{j,k} \cdot 2^k$ .

the hood of the DEPIR from ASHE construction, we have  $\text{ct}' = \bar{f}_y(\hat{i}'_1, \dots, \hat{i}'_{m_y})$ , and by ASHE-FHE correctness property (2), we then have that  $\text{Refresh}(\text{ct}')$  is a good ASHE-FHE encryption of  $f_y(i_1, \dots, i_{m_y}) = y[i_y^t]$ . To use the ASHE-FHE correctness property (2), we rely on the fact that the polynomial  $f_y$  has total degree  $< d \cdot m_y \leq d \cdot m = D$  and the number of terms is  $< d^{m_y} \leq d^m$ . Therefore, by induction the final encrypted configuration  $T$  is correct, meaning that  $\hat{z}^T$  is a good encryption of the final memory contents  $z^T$ . This implies that  $\text{ct}_{\text{out}}$ , consisting of the first  $M$  components of  $\hat{z}^T$ , correctly decrypts to  $P(x, y)$ .

To analyze **efficiency**, we now set the parameter  $d$ . For any constant  $\varepsilon > 0$ , we choose  $d$  to be the first prime  $d > \log^{2/\varepsilon} N$ , which we can find efficiently. This matches parameter option A from our DEPIR construction in Section 4.2. By the efficiency of ASHE-FHE, the bit-length of elements in the ring  $R$ , the run-time of ring operations and the run-time of  $\text{Setup}^*$ ,  $\text{Gen}^*$ ,  $\text{Enc}^*$ ,  $\text{Dec}^*$  (for a single message in  $\mathbb{F}_d$ ) are bounded by  $\text{poly}(\lambda, D, \log d, \log N) = \text{poly}(\lambda, \log N)$ . Therefore, the run-time of  $\text{Gen}$  is  $\text{poly}(\lambda, \log N)$  and the run-time of  $\text{Enc}_p$  is  $|x| \cdot \text{poly}(\lambda, \log N)$  and the run-time of  $\text{Dec}$  is  $M \cdot \text{poly}(\lambda, \log N)$ . The run-time of  $\text{Prep}$  is

$$\begin{aligned} & d^{m_y} \cdot m_y^{m_y} \cdot \text{poly}(m_y, d, \log |R|) \cdot O(\log m_y + \log d + \log \log |R|)^{m_y} \\ &= |y|^{1+\varepsilon} \text{poly}(\lambda, \log N), \end{aligned}$$

where we plug in  $\log |R| = \text{poly}(\lambda, \log N)$ ,  $m_y = \lceil \log_d |y| \rceil = \log |y| / \log d + O(1)$  and follow the calculation of Theorem 4.4, option A. Finally, to analyze the run-time of  $\text{Eval}_p$ , note that in each step  $t$ , all the calls to  $\text{Eval}^*$  take time  $\text{poly}(\lambda, \log N) \cdot (|C| + |x| + S)$ , and the call to the DEPIR procedure  $\text{Resp}'$  and to  $\text{Refresh}^*$  take time  $\text{poly}(\lambda, \log N)$ .

**Claim 7.3.1.** *Assume there exists a secure ASHE-FHE encryption scheme for poly-logarithmic degree. In particular, this holds assuming RingLWE with quasi-polynomial approximation factors and the circular-security assumption (6.2). Then, for any constant  $\varepsilon > 0$ , Algorithm 7.3 is a secure RAM-FHE scheme. For security parameter  $\lambda$ , any bounds  $N_x, N_y, N_S$  with  $N := \max\{N_x, N_y, N_S\}$ , any program  $P = (C, T, S, M)$  and any inputs  $x, y$  such that  $S \leq N_S, |x| \leq N_x, |y| \leq N_y$ , the achieved efficiency is:*

- $\text{Setup}$  and  $\text{Gen}$  take time  $\text{poly}(\lambda, \log N)$ .
- $\text{Enc}_p$  takes time  $|x| \cdot \text{poly}(\lambda, \log N)$ .
- $\text{Prep}$  takes time  $|y|^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$ .
- $\text{Eval}_p$  takes time  $T \cdot \text{poly}(\lambda, \log N) \cdot (|C| + |x| + S)$ .
- $\text{Dec}$  takes time  $M \cdot \text{poly}(\lambda, \log N)$ .

### 7.3 RAM-FHE with Random Access to $x$

We now augment the RAM-FHE scheme from the previous section to also handle random access to the encrypted input  $x$ . We will modify the algorithms  $\text{Enc}_p$  and  $\text{Eval}_p$  while keeping the other algorithms the same.

**Construction Overview.** Instead of directly encrypting  $x$  via the ASHE-FHE, the client first chooses a  $s$  for a pseudorandom function  $\text{PRF}_s$  and one-time pads  $x$  with the PRF outputs to get  $\text{ct}_x = (x[i] \oplus \text{PRF}_s(i))_i$ . It then applies the DEPIR preprocessing on  $\text{ct}_x$  to get a DEPIR preprocessed data structure  $\tilde{\text{ct}}'_x$ . Finally it encrypts  $s$  under the ASHE-FHE scheme to get  $\text{ct}_s$  and sends  $\tilde{\text{ct}}_x = (\tilde{\text{ct}}'_x, \text{ct}_s)$  to the server as an encryption of  $x$ . The server evaluates the program  $P$  under ASHE-FHE the same way as before. The only difference is that now, when it gets an encryption of an index  $i_x^t$ , it uses the DEPIR (the same way it previously did for  $y$ ) to get an encryption of  $(x[i_x^t] \oplus \text{PRF}_s(i_x^t))$  using random-access to  $\tilde{\text{ct}}'_x$ . It then performs an FHE evaluation using  $\text{ct}_s$  to convert that to an encryption of just  $x[i_x^t]$ . It then continues the computation as before.

**Construction.** Let  $\text{PRF}$  be a standard *pseudorandom function* (PRF) that takes a key  $\in \{0, 1\}^\lambda$  and an input  $i \in \{0, 1\}^*$  and outputs one bit  $b = \text{PRF}_{\text{key}}(i) \in \{0, 1\}$ .

---

**Algorithm 7.4: RAM-FHE with random-access to  $x$ .**

---

Setup, Gen, Dec, Prep: These are identical to the RAM-FHE construction from the previous section (Algorithm 7.3).

**Modifications:** We replace  $\text{Enc}_p$  and  $\text{Eval}_p$  with  $\text{Enc}$  and  $\text{Eval}_e$  given below.

$\text{Enc}(\text{pk}, x)$ : Sample a PRF key:  $\text{key} \leftarrow \{0, 1\}^\lambda$ . Set  $\text{ct}_x[i] := \text{PRF}_{\text{key}}(i) \oplus x[i]$  for each  $i \in [|x|]$ .  
Compute:

$$\tilde{\text{ct}}'_x := \text{Prep}(\text{params}, \text{ct}_x), \quad \text{ct}_{\text{key}} := \text{Enc}^*(\text{pk}, \text{key}),$$

where  $\text{Enc}^*$  is the encryption algorithm of ASHE-FHE. Output  $\tilde{\text{ct}}_x := (\text{ct}_{\text{key}}, \tilde{\text{ct}}'_x)$ .

$\text{Eval}_e(\text{pk}, P, \tilde{\text{ct}}_x, \tilde{y})$ : This algorithm is modified from  $\text{Eval}_p$  (Algorithm 7.3) in the previous section, by replacing Step 2b with the following subroutine:

- Convert the encrypted index  $\hat{i}_x^t$  to base- $d$ , and then use it to read from  $\tilde{\text{ct}}'_x$  via DEPIR. Refresh the resulting DEPIR output, then remove the one-time pad under FHE:

$$\begin{aligned} \hat{i}' &:= \text{Eval}^*(\text{pk}, \text{base}_{d, m_x}, \hat{i}_x^t), & \text{ct}' &:= \text{Resp}'(\tilde{\text{ct}}'_x, \hat{i}'), \\ \hat{b}' &:= \text{Refresh}^*(\text{pk}, \text{ct}'), & \hat{b}_x^t &:= \text{Eval}^*(\text{pk}, \text{OTPDec}, \hat{i}_x^t, \hat{b}', \text{ct}_{\text{key}}) \end{aligned}$$

where  $m_x := \lceil \log_d x \rceil$ ,  $\text{base}_{d, m_x}$  is the circuit describe in Step 2c of Algorithm 7.3, and  $\text{OTPDec}$  is the circuit defined via  $\text{OTPDec}(i, b, \text{key}) = \text{PRF}_{\text{key}}(i) \oplus b$ .

---

**Analysis.** The analysis is similar to that of Algorithm 7.3 from the previous section.

**Security** follows by a standard hybrid argument that first uses the IND-CPA security of ASHE-FHE to replace  $\text{ct}_{\text{key}}$  by an encryption of an all 0's string, and then uses the pseudorandomness of PRF to replace  $\text{ct}_x$  by a uniformly random string.

**Correctness** requires us to show that  $\hat{b}_x^t$  computed by the modified version of Step 2b is a good ASHE-FHE encryption of  $b_x^t = x[i_x^t]$ . The analysis is similar to the previous analysis of Step 2c. In particular, the same analysis tells us that  $\hat{b}'$  is a good ASHE-FHE encryption of  $b' =$

$x[i_x^t] \oplus \text{PRF}_{\text{key}}(i_x^t)$ . Then, by the correctness of the ASHE-FHE Eval\* procedure,  $\hat{b}_x^t$  is a good ASHE-FHE encryption of  $\text{PRF}_{\text{key}}(i_x^t) \oplus b' = x[i_x^t]$  as desired.

The run-time of Enc is bounded by  $|x|^{1+\varepsilon} \text{poly}(\lambda, \log N)$  because it invokes Prep, which dominates PRF evaluation. The run-time of the modified Step 2b is  $\text{poly}(\lambda, \log N) \cdot \text{poly}(\lambda + \log |x|)$ . Hence, the per-step run-time of Eval<sub>e</sub> is bounded by  $\text{poly}(\lambda, \log N) \cdot (|C| + \text{poly}(\log(|x| + |y|)) + S)$ .

**Claim 7.4.1.** *Assume there exists a secure ASHE-FHE encryption scheme for poly-logarithmic degree. In particular, this holds assuming RingLWE with quasi-polynomial approximation factors and the circular-security assumption (6.2). Then, for any constant  $\varepsilon > 0$ , Algorithm 7.4 is a secure RAM-FHE scheme. For security parameter  $\lambda$ , any bounds  $N_x, N_y, N_S$  with  $N := \max\{N_x, N_y, N_S\}$ , any program  $P = (C, T, S, M)$  and any inputs  $x, y$  such that  $S \leq N_S, |x| \leq N_x, |y| \leq N_y$ , the achieved efficiency is:*

- The run-time of Setup, Gen, Prep, and Dec are identical to those given in Claim 7.3.1.
- Enc takes time  $|x|^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$ .
- Eval<sub>e</sub> takes time  $T \cdot \text{poly}(\lambda, \log N) \cdot (|C| + S)$  for any  $(P = (C, T, S, M), x, y)$ .

## 7.4 RAM-FHE with Random Access to $z$

In the previous two sections, we constructed RAM-FHE schemes with read-only random-access to  $x, y$ . However, the schemes only had sequential access to the read/write memory  $z$ , and the homomorphic evaluation operated over the entire content  $z$  in every step of the computation, incurring a multiplicative factor of  $S$  in the run time. We now show how to remove this. We first construct an efficient read-writable data structure called *homomorphic memory* that allows us to homomorphically perform encrypted read/write operations to an encrypted memory. Then we plug the homomorphic memory into our construction of RAM-FHE to achieve the desired efficiency.

### 7.4.1 Homomorphic Memory

A homomorphic memory is a data structure that is parameterized by its maximal space  $S$  bits. It provides two algorithms, (ReadMem, WriteMem), and its functionality is to maintain and access an encrypted version of an array  $z \in \{0, 1\}^S$ , which is initially zeroed-out. The ReadMem algorithm takes as input an encryption of an index  $i$  and outputs an encryption of  $z[i]$ , and WriteMem takes as input an encryption of a pair  $(i^*, b^*)$  and updates  $z[i^*] := b^*$ . Both of these algorithms needs to be efficient and run time sublinear in  $S$ . Roughly speaking, our implementation of homomorphic memory resembles the updatable DEPIR in both abstraction and construction (see Section 5), with the main difference being that the data itself is encrypted. When we perform a DEPIR access to this data, we get a “double encryption” under the ASHE-FHE. We will rely on an ASHE-FHE with a *key cycle* (see Remark 6.1) to go from a double encryption to a single encryption.

Since homomorphic memory is a part of our RAM-FHE scheme, the construction depends on the other algorithms of the scheme as well as the underlying ASHE-FHE scheme. Hence in the following, we list the algorithms that will be invoked, then describe the data structure, and then give the procedures of ReadMem and WriteMem.

---

**Algorithm 7.5: Homomorphic memory,  $\tilde{z}$ .**

---

**Building blocks:** We use the RAM-FHE algorithms  $\text{Prep}$ ,  $\text{Eval}_p$ , and  $\text{Dec}$  from Algorithm 7.3 as well as the circuit evaluation algorithm  $\text{Enc}^*$ ,  $\text{Eval}^*$  from the underlying ASHE-FHE scheme. We rely on an ASHE-FHE scheme with a key cycle, and denote the ciphertext containing the encrypted secret key by  $\text{ct}_{\text{sk}}$ , which is part of  $\text{pk}$ . We will specify how to pick parameters for Setup in Algorithm 7.6.

**Data structure:** The data structure is parameterized by a space bound  $S$ , and it consists of:

$$\tilde{z} = (\text{DB}_0, \text{DB}_1, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \widetilde{\text{DB}}_1, \dots, \widetilde{\text{DB}}_L, \text{count}),$$

where  $L := \lceil \log S \rceil$ , each  $\text{DB}_\ell$  is either empty, or a sorted list consisting of  $2^\ell$  encrypted index-value pairs  $(i, b)$ , and  $\widetilde{\text{DB}}_\ell$  is the DEPIR preprocessed copy of  $\text{DB}_\ell$ . When  $\text{DB}_\ell$  is empty, we set  $(\text{DB}_\ell, \widetilde{\text{DB}}_\ell)$  to the symbol  $(\perp, \perp)$ . Each pair  $(i, b) \in (\llbracket S \rrbracket \times \{0, 1\}) \cup \{(\infty, 0)\}$  stores the entry  $z[i] = b$  for the virtual array  $z$ , where  $(\infty, 0)$  denotes a “dummy” pair. The counter count denotes the number of writes taken modulo  $2^L$ .

$\text{Init}(S)$ : Let  $L := \lceil \log S \rceil$ , set  $(\text{DB}_\ell, \widetilde{\text{DB}}_\ell) := (\perp, \perp)$  for all  $\ell \leq L$ , set  $\text{count} := 0$ , and then output  $\tilde{z} := (\text{DB}_0, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_L, \text{count})$ .

$\text{ReadMem}(\text{pk}, \tilde{z}, \hat{i})$ : Takes as input the public key  $\text{pk}$ , the data structure  $\tilde{z}$ , and an ASHE-FHE encryption  $\hat{i}$  of an index  $i \in \llbracket S \rrbracket$ .

Let  $P_{\text{search}}$  be a RAM program implementing binary search, such that

$$P_{\text{search}}(x = (\text{sk}, i^*), y = \text{DB}_\ell)$$

has read-only random-access to a list  $\text{DB}_\ell$  consisting of  $2^\ell$  encrypted tuples  $(i, b)$  sorted by  $i$ . The program  $P_{\text{search}}$  performs a binary search and check if the list contains some encryption of a tuple of the form  $(i^*, b)$ : if so  $P_{\text{search}}$  outputs  $b$  in the first such tuple, else it outputs  $\perp$ . The program uses  $\text{sk}$  to decrypt any encrypted tuples that it touches. The program is padded to always run in worst-case time  $T = \text{poly}(\lambda, \log N)$ .

$\text{ReadMem}$  performs the following procedure.

1. For  $\ell = 0, \dots, L$ :

If  $\widetilde{\text{DB}}_\ell = \perp$ , set  $\hat{b}_\ell := \text{Enc}^*(\text{pk}, \perp; 0^*)$  to be an encryption of the special symbol  $\perp$  using all 0's randomness. Else, perform

$$\hat{b}_\ell := \text{Eval}_p(\text{pk}, P_{\text{search}}, (\text{ct}_{\text{sk}}, \hat{i}), \widetilde{\text{DB}}_\ell).$$

where  $\text{Eval}_p$  is the homomorphic evaluation from Algorithm 7.3 in Section 7.2, that only relies on random-access to the second input  $y = \text{DB}_\ell$ .

Eventually each of  $\hat{b}_\ell$  is an encryption of 0, 1 or  $\perp$ .

2. Output  $\hat{b} := \text{Eval}^*(\text{pk}, C_{\text{first}}, \hat{b}_0, \hat{b}_1, \dots, \hat{b}_L)$ , where  $C_{\text{first}}$  is the following circuit:
  - $C_{\text{first}}(b_0, \dots, b_L)$  finds the minimal  $\ell^* \in \{0, \dots, L\}$  such that  $b_{\ell^*} \neq \perp$ , and then it outputs  $b_{\ell^*}$ ; it outputs 0 if all inputs are  $\perp$ .

$\text{WriteMem}(\text{pk}, \tilde{z}, \hat{i}^*, \hat{b}^*)$ : Takes as input the public key  $\text{pk}$ , the data structure  $\tilde{z}$ , and ASHE-FHE encryptions  $\hat{i}^*$  of an index  $i^* \in \llbracket S \rrbracket$  and  $\hat{b}^*$  of a bit  $b^*$  to write into location  $i^*$ .

1. Update  $\text{count} := \text{count} + 1 \pmod{2^L}$ .  
Let  $\ell^*$  be the max of  $\ell \in \{0, \dots, L\}$  such that  $2^\ell$  divides  $\text{count}$ .
  2. Construct an updated list  $\text{DB}'_{\ell^*}$  as follows:
    - (a) Define  $\text{DB}_{-1} := \{(\hat{i}^*, \hat{b}^*)\}$  to be a list containing a single encrypted tuple.
    - (b) Perform  $\text{DB}'_{\ell^*} := \text{Eval}^*(\text{pk}, C_{\text{merge}}, (\text{DB}_{-1}, \text{DB}_0, \dots, \text{DB}_{\ell^*}))$ , where  $\text{Eval}^*$  is the evaluation algorithm of ASHE-FHE,  $C_{\text{merge}}$  is a circuit defined as follows:  
 $C_{\text{merge}}(D_{-1}, D_0, \dots, D_{\ell^*})$ :
      - i. Take input the lists  $D_{-1}, D_0, \dots, D_{\ell^*}$ , where  $D_\ell$  is a list of  $2^\ell$  pairs of  $(i, b)$  for  $\ell \geq 0$ , and  $D_{-1}$  is single-element.
      - ii. Take all the non-dummy pairs  $(i, b)$  contained in the lists  $D_{-1}, D_0, \dots, D_{\ell^*}$  and sort them by index  $i$ . If there are multiple pairs with the same index  $i$ , take only the one from  $D_\ell$  with the smallest  $\ell$  and discard the rest. Here, we instantiate a sorting circuit with circuit size  $n \text{poly} \log n$  for input size  $n$ , e.g., [Bat68, AKS83, Goo14].
      - iii. Append additional dummy pairs  $(\infty, \perp)$  until the final list is of size  $2^{\ell^*}$  and output it.

Remark: In the above, if  $\ell^* < L$ , then  $\text{DB}_{\ell^*}$  is empty, and we omit the input  $\text{DB}_{\ell^*}$  from the computation and reduce the input size of  $C_{\text{merge}}$  accordingly.
  3. Set  $\text{DB}_{\ell^*} := \text{DB}'_{\ell^*}$ ,  $\widetilde{\text{DB}}_{\ell^*} := \text{Prep}(\text{params}, \text{DB}'_{\ell^*})$ .
  4. For  $\ell \in \{0, \dots, \ell^* - 1\}$ : Set  $\text{DB}_\ell := \perp$ ,  $\widetilde{\text{DB}}_\ell := \perp$ .
  5. Output  $\tilde{z}' := (\text{DB}_0, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_L, \text{count})$ .
- 

**Claim 7.5.1 (Correctness).** *There is a predicate  $\text{GoodMem}_{\text{sk}}(\tilde{z}, z)$  that corresponds to  $\tilde{z}$  being a “good” homomorphic memory representation of  $z$  with respect to the secret key  $\text{sk}$ , such that the following holds:*

1. For any  $S \leq N_S$ ,  $\tilde{z} := \text{Init}(S)$  satisfies  $\text{GoodMem}_{\text{sk}}(\tilde{z}, 0^S) = 1$ .
2. Let  $z \in \{0, 1\}^S$ ,  $i^* \in \llbracket S \rrbracket$ , and  $b^* \in \{0, 1\}$ , and let  $z'$  be the same as  $z$  except with  $z'[i] := b$ . For any  $\tilde{z}$  satisfying  $\text{GoodMem}_{\text{sk}}(\tilde{z}, z) = 1$ , for any good ASHE ciphertexts  $(\hat{i}^*, \hat{b}^*) \in \text{GOOD}_{\text{sk}}$  such that  $\text{Dec}(\text{sk}, (\hat{i}^*, \hat{b}^*)) = (i^*, b^*)$ , if  $\tilde{z}' := \text{WriteMem}(\text{pk}, \tilde{z}, \hat{i}^*, \hat{b}^*)$  then  $\text{GoodMem}_{\text{sk}}(\tilde{z}', z') = 1$ .<sup>24</sup>
3. Let  $z \in \{0, 1\}^S$  and  $\tilde{z}$  satisfy  $\text{GoodMem}_{\text{sk}}(\tilde{z}, z)$ , and let  $i \in \llbracket S \rrbracket$ . For any ciphertext  $\hat{i} \in \text{GOOD}_{\text{sk}}$  such that  $\text{Dec}(\text{sk}, \hat{i}) = i$ , if  $\hat{b} := \text{ReadMem}(\text{pk}, \tilde{z}, \hat{i})$  then  $\hat{b} \in \text{GOOD}_{\text{sk}}$  and  $\text{Dec}(\text{sk}, \hat{b}) = z[i]$ .

*Proof.* The proof closely follows the correctness of updatable DEPIR in Theorem 5.5, but now everything is under ASHE-FHE encryption. We define the predicate  $\text{GoodMem}_{\text{sk}}(\tilde{z}, z)$  to hold if  $\tilde{z} = (\text{DB}_0, \dots, \text{DB}_L, \widetilde{\text{DB}}_0, \dots, \widetilde{\text{DB}}_L, \text{count})$  such that:

- i. For all  $\ell \in \{0, \dots, L\}$ ,  $\widetilde{\text{DB}}_\ell = \text{Prep}(\text{params}, \text{DB}_\ell)$ .
- ii. For all  $\ell \in \{0, \dots, L\}$ , either  $\text{DB}_\ell = \perp$  or  $\text{DB}_\ell$  consists of a list of  $2^\ell$  encrypted pairs  $\hat{i}, \hat{b} \in \text{GOOD}_{\text{sk}}$  such that  $\text{Dec}(\text{sk}, (\hat{i}, \hat{b})) = (i, b)$  for some  $i \in \llbracket S \rrbracket \cup \{\infty\}$  and  $b \in \{0, 1\}$ . The encrypted pairs are sorted by the index  $i$ . Furthermore, for any  $i \in \llbracket S \rrbracket$ , there is at most one encrypted tuple in  $\text{DB}_\ell$  that decrypts to a pair of the form  $(i, b)$ .

<sup>24</sup>We abuse notation and extend the definition of  $\text{GOOD}_{\text{sk}}$  from ASHE-FHE to contain vectors of ASHE-FHE ciphertexts where each component is in  $\text{GOOD}_{\text{sk}}$ .

- iii. For each  $i \in \llbracket S \rrbracket$ , let  $\ell \in \{0, \dots, L\}$  be the minimal value such that  $\text{DB}_\ell$  contains an encrypted tuple that decrypts to a pair of the form  $(i, b)$ , or if no such  $\ell$  exists let  $\ell = \perp$ . If  $\ell \neq \perp$ , then  $z[i] = b$ , otherwise  $z[i] = 0$ .
- iv. Let  $\text{count} = (\text{count}_0, \dots, \text{count}_{L-1})$  be the binary representation of  $\text{count} \in \llbracket 2^L \rrbracket$ . For each  $\ell$  such that  $\text{count}_\ell = 0$  we have  $\text{DB}_\ell = \perp$ .

The fact that this predicate satisfies the three properties of the claim follows by essentially the same argument as in the proof of Theorem 5.5, with the main difference that we additionally rely on the correctness of the RAM-FHE evaluation  $\text{Eval}_p$  from Algorithm 7.3 with the RAM program  $P_{\text{search}}$ , as well as the ASHE-FHE evaluation  $\text{Eval}^*$  with the circuits  $C_{\text{merge}}$  and  $C_{\text{first}}$ .  $\square$

**Claim 7.5.2 (Efficiency).** *The run-time of ReadMem is bounded by  $\text{poly}(\lambda, \log N)$ . Moreover, for any sequence of  $T$  calls  $\tilde{z}^t := \text{WriteMem}(\text{pk}, \tilde{z}^{t-1}, \star, \star)$  for  $t \in [T]$ , the total run-time is bounded by  $T \cdot (\min\{T, S\})^\varepsilon \cdot \text{poly}(\lambda, \log N)$ .*

*Proof.* The run-time of ReadMem is dominated by  $L$  calls to  $\text{Eval}_p$  on the program  $P_{\text{search}}$ . Since  $P_{\text{search}}$  runs in time  $\text{poly}(\lambda, \log N)$ , the efficiency of  $\text{Eval}_p$  (Claim 7.3.1) implies the total run time of ReadMem is bounded by  $\text{poly}(\lambda, \log N)$ .<sup>25</sup>

To argue the total run-time of WriteMem, observe that each level  $\ell \in \{0, \dots, L\}$  is only chosen to be merged into once out of every  $2^\ell$  calls to WriteMem (i.e., when  $\ell^* = \ell$ ). The run time of WriteMem when merging into  $\text{DB}_\ell$  is dominated by the call to Prep in Step 3 which takes time  $S_\ell^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$  by Claim 7.3.1, where  $S_\ell = 2^\ell \cdot \text{poly}(\lambda, \log N)$  is the size of  $\text{DB}_\ell$  in bits. Thus the total run time of any sequence of  $T$  calls to WriteMem is at most

$$\begin{aligned} \sum_{\ell=0}^L \left\lfloor T/2^\ell \right\rfloor \cdot S_\ell^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N) &\leq \sum_{\ell=0}^{\lfloor \log(\min\{T, S\}) \rfloor} (T/2^\ell) \cdot (2^\ell)^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N) \\ &\leq T \cdot (\min\{T, S\})^\varepsilon \cdot \text{poly}(\lambda, \log N), \end{aligned}$$

where the first inequality follows from the fact that the terms for which  $2^\ell > T$  become zero after taking the floor of  $T/2^\ell$ .  $\square$

## 7.4.2 Full-fledged RAM-FHE Scheme

Now we plug the homomorphic memory data structure into our earlier RAM-FHE scheme from the previous section, which already leverages random access to  $x, y$ , but not  $z$ . We simply modify the steps of the homomorphic evaluation procedure that reads/writes to  $z$ . At the beginning of the homomorphic evaluation, we initialize the homomorphic memory  $\tilde{z}$ . In each step, to read and write to  $z$  we now perform the homomorphic ReadMem, WriteMem operations on  $\tilde{z}$ .

---

### Algorithm 7.6: Final RAM-FHE Scheme (Setup, Gen, Enc, Prep, Eval, Dec)

---

**Setup**( $1^\lambda, N_x, N_y, N_S$ ): Let  $N := \max\{N_x, N_y, N_S\}$  and let  $N' := N \cdot \eta(\lambda, \log N)$  for some sufficiently large polynomial  $\eta$  to be specified later. Choose a prime  $d$  as specified later. Let  $m := \lceil \log_d N' \rceil$  and let  $D := d \cdot m$ . Set parameters for ASHE-FHE scheme as  $\text{params}^* := \text{Setup}^*(1^\lambda, 1^d, 1^D, d^m)$  and output  $\text{params} := (\text{params}^*, d)$ . Let  $R$  be the ASHE-FHE ring.

---

<sup>25</sup>The run-time of  $\text{Eval}_p$  (and also Prep) is actually derived from the parameters picked by Setup that we deferred to Algorithm 7.6, but the time will be identical to what we claimed here.

Gen, Dec, Prep, Enc: These are identical to the previous scheme (Algorithm 7.4).

Eval(pk,  $P, \tilde{x}, \tilde{y}$ ): This algorithm is modified from Eval<sub>e</sub> (Algorithm 7.4), which in turn is a modification of Eval<sub>p</sub> (Algorithm 7.3) by replacing Step 2b. We retain this modification and further modify Eval<sub>p</sub> as follows:

1. In Step 1, instead of setting  $z^0 := 0^S$  and  $\hat{z}^0$  to be an encryption of  $z^0$ , we now initialize the homomorphic memory data structure  $\tilde{z}^0 := \text{Init}(S)$ .
  2. Replace Step 2d with the following subroutine. Recall that at the beginning of that step we already derived ciphertexts  $(\hat{i}_w^t, \hat{b}_w^t, \hat{i}_z^t)$  that encrypt the index  $i_w^t \in \llbracket S \rrbracket$  and bit  $b_w^t$  to write, as well as the index  $i_z^t$  to read from in the memory  $z$ .
    - Perform  $\tilde{z}^t := \text{WriteMem}(\text{pk}, \tilde{z}^{t-1}, \hat{i}_w^t, \hat{b}_w^t)$ . This step uses read/write random-access to the data structure  $\tilde{z}^{t-1}$  and updates it to  $\tilde{z}^t$ .
    - Perform  $\hat{b}_z^t := \text{ReadMem}(\text{pk}, \tilde{z}^t, \hat{i}_z^t)$ . This step uses read-only random-access to the data structure  $\tilde{z}^t$ .
  3. Replace Step 3 with the following: For each  $j \in \llbracket M \rrbracket$ , let  $\text{ct}_{\text{out},j} := \text{ReadMem}(\text{pk}, \tilde{z}^T, \hat{j})$ , where  $\hat{j} := \text{Enc}^*(\text{pk}, j; 0^*)$  is an ASHE-FHE encryption of  $j$  under all  $0$ 's randomness, and  $M$  is the output length of  $P$ . Output  $\text{ct}_{\text{out}} := (\text{ct}_{\text{out},j})_{j \in \llbracket M \rrbracket}$ .
- 

**Analysis.** We begin by setting the **parameters**. As before, for any constant  $\varepsilon > 0$ , we choose  $d$  to be the first prime  $d > \log^{2/\varepsilon} N$ , which we can find efficiently. We choose a sufficiently large  $\eta(\lambda, N) = \text{poly}(\lambda, \log N)$  so that the size of  $\text{DB}_L$  in the homomorphic memory is  $\leq N' = N \cdot \eta(\lambda, N)$  bits. Recall that  $\text{DB}_L$  consists of  $2^L \leq 2N_S \leq 2N$  ASHE-FHE ciphertexts encrypting values  $(i, b)$  with  $i \in \llbracket S \rrbracket$  and  $b \in \{0, 1\}$ . Each such ASHE-FHE ciphertext is of some size  $\text{poly}(\lambda, D, \log d, \log N') = p(\lambda, \log N')$  for some polynomial  $p$ . This means that we can choose a sufficiently large  $\eta(\lambda, N) = \text{poly}(\lambda, \log N)$  to ensure that  $2Np(\lambda, \log(N \cdot \eta(\lambda, N))) \leq N\eta(\lambda, N)$ .

The **security** of the scheme is the same as Claim 7.4.1 since the same key generation and encryption procedures were unchanged.

The **correctness** of the scheme follows via the same arguments as in the previous schemes (Algorithm 7.3, Algorithm 7.4), but now we also rely on the correctness of the homomorphic memory (Claim 7.5.1). Recall that in the previous scheme, the correctness of evaluation is argued by an induction on  $t$ . Here we proceed by a similar argument, with the difference that in the step-circuit evaluation (Equation (1)), the ciphertext  $\hat{b}_z^t$  is obtained from the output of ReadMem. The inductive hypothesis is identical, we assume by induction that the overall state  $(\text{ct}_{\text{state}}^t, \hat{b}_x^t, \hat{b}_y^t, \hat{b}_z^t, \tilde{z}^t)$  is correct at step  $t$ , and we want to prove the hypothesis holds for  $t + 1$ . Particularly, we want to prove that both  $\hat{b}_z^{t+1}$  and  $\tilde{z}^{t+1}$  are correct, where we use Properties 1, 2, 3 of Claim 7.5.1 as follows. Let  $z^t$  denote the memory when the given program is performed in plaintext for  $t$  steps, where  $z^0 = 0^S$ . In the base case  $t = 0$ ,  $\hat{b}_z^0$  is correct, and  $\tilde{z}^0$  is correct by Property 1 ( $\text{GoodMem}_{\text{sk}}(\tilde{z}^0, z^0) = 1$ ). Then, suppose the induction hypothesis holds for step  $t \geq 0$ . By the correctness of circuit evaluation, Eval\* outputs correct ciphers  $\hat{i}_z^{t+1}, \hat{i}_w^{t+1}$ , and  $\hat{b}_w^{t+1}$ . Then by Property 2, WriteMem outputs correct  $\tilde{z}^{t+1}$  ( $\text{GoodMem}_{\text{sk}}(\tilde{z}^{t+1}, z^{t+1}) = 1$ ). Next, by Property 3, ReadMem outputs correct  $\hat{b}_z^{t+1}$ , that is,  $\hat{b}_z^{t+1}$  decrypts to  $z^{t+1}[i_z^{t+1}]$ . This concludes the induction.

The **efficiency** is the same as in Claim 7.4.1, except for Eval. By Claim 7.5.2, the run-time of Eval is dominated by WriteMem, that is,  $T \cdot \text{poly}(\lambda, \log N) \cdot ((\min\{T, S\})^\varepsilon + |C|)$ .

Thus we have completed the proof of the following theorem.

**Theorem 7.7 (RAM-FHE).** *Assume there exists a secure ASHE-FHE encryption scheme with a key cycle (Remark 6.1) for poly-logarithmic degree. In particular, this holds assuming RingLWE with quasi-polynomial approximation factors and the circular-security assumption (6.2). Then, for any constant  $\varepsilon > 0$ , Algorithm 7.6 is a secure RAM-FHE scheme. For security parameter  $\lambda$ , any bounds  $N_x, N_y, N_S$  with  $N := \max\{N_x, N_y, N_S\}$ , any program  $P = (C, T, S, M)$  and any inputs  $x, y$  such that  $S \leq N_S, |x| \leq N_x, |y| \leq N_y$ , the achieved efficiency is:*

- Setup and Gen run in time  $\text{poly}(\lambda, \log N)$ ,
- Enc and Prep run in time  $|x|^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$  and  $|y|^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$  respectively,
- Eval runs in time  $T \cdot ((\min\{T, S\})^\varepsilon + |C|) \cdot \text{poly}(\lambda, \log N)$ , and
- Dec runs in time  $M \cdot \text{poly}(\lambda, \log N)$ .

Recall that, in the most natural case where  $P$  is the universal RAM, we have  $|C| = \text{poly} \log(N)$  and therefore the evaluation time becomes  $T \cdot (\min\{T, S\})^\varepsilon \cdot \text{poly}(\lambda, \log N)$ .

## 8 Extensions and Variants of RAM-FHE

We now discuss several potential extensions and variants of the RAM-FHE scheme above. In Section 8.1, we consider *alternative parameters settings* resulting in different efficiency tradeoffs. In Section 8.2, we discuss how to *avoid circular security* at the cost of worse efficiency, where the client’s run-time scales linearly with the RAM run-time  $T$ . In Section 8.3, we discuss how to *update the preprocessed input  $\tilde{y}$*  in sublinear time. Finally in Section 8.4, we discuss how to allow for *multi-hop and multi-input* RAM-FHE, where we can perform homomorphic computations over many individually preprocessed plaintext inputs and many individually encrypted inputs (under the same key), and also use the outputs of previous homomorphic evaluations as inputs to future homomorphic evaluations.

### 8.1 Alternative Efficiency Tradeoffs

Similar to DEPIR (Theorem 4.4), our RAM-FHE is tunable with different parameters to achieve different efficiency tradeoffs. We refer to the default choice from Theorem 7.7 as Option A.

Our second option reduces the run-time of Enc, Prep, Eval by replacing the constant  $\varepsilon$  in the exponent by  $o(1)$ , at the cost of increasing the run-time of Setup, Gen, Dec by replacing  $\text{poly} \log N$  factors by  $N^{o(1)}$  factors. This is essentially the same as the alternative parameterization of DEPIR in Theorem 4.4 (Option B) that “balances” the overhead of preprocessing and evaluating. In the context of RAM-FHE, this choice balances between reads/writes to memory to achieve  $N^{o(1)}$  overhead in both cases.

**Theorem 8.1 (RAM-FHE, Option B).** *Assume there exists a secure ASHE-FHE encryption scheme with a key cycle (Remark 6.1) for sub-polynomial degree. In particular, this holds assuming RingLWE with sub-sub-exponential approximation factors and the circular-security assumption (6.2). Then there*

is a secure RAM-FHE such that, for any security parameter  $\lambda$ , any bounds  $N_x, N_y, N_S$ , any program  $P = (C, T, S, M)$  and any inputs  $x, y$  such that  $S \leq N_S, |x| \leq N_x, |y| \leq N_y$ , the scheme achieves the following efficiency, where  $N := \max\{N_x, N_y, N_S\}$ :

- Setup and Gen run in time  $N^{o(1)} \text{poly}(\lambda)$ ,
- Enc and Prep run in time  $|x| \cdot N^{o(1)} \text{poly}(\lambda)$  and  $|y| \cdot N^{o(1)} \text{poly}(\lambda)$  respectively,
- Eval runs in time  $T \cdot N^{o(1)} \cdot |C| \cdot \text{poly}(\lambda)$ , and
- Dec runs in time  $M \cdot N^{o(1)} \text{poly}(\lambda)$ .

All the  $N^{o(1)}$  factors above can be more precisely replaced with  $2^{\tilde{O}(\sqrt{\log N})}$ .

*Proof.* We now choose  $d$  to be the first prime such that  $d \geq 2^{\sqrt{\log N'}}$ , just as in parameter Option B of DEPIR. We choose  $N' = N^{1+o(1)} \text{poly}(\lambda)$  sufficiently large to ensure that the size of  $\text{DB}_L$  consisting of  $O(N_S)$  ciphertexts, each of size  $\text{poly}(\lambda)(N')^{o(1)}$ , is bounded by  $|\text{DB}_L| \leq N'$ . The other parameters are chosen as before, as functions of  $N', d$ . Correctness is unaffected and security now relies on ASHE-FHE with sub-polynomial degree, but is otherwise the same. The efficiency analysis is analogous to the previous one, but plugging in the “alternative parameters” of DEPIR (Theorem 4.4), and noting that each ASHE-FHE ciphertext is of size  $\text{poly}(\lambda)N^{o(1)}$ .  $\square$

Our third option is somewhere in between A and B. It optimizes for the case when  $|x|, |y|$  is much larger than  $S, T$ . In that case, we may want to avoid paying the  $N^{o(1)} \geq \max\{N_x, N_y\}^{o(1)}$  overhead during setup, decryption, key generation and evaluation (as we need to pay in option B) and would prefer to just pay  $\text{poly} \log(N_x, N_y)$  (as was the case in option A). But we also want to avoid paying the  $\min\{T, S\}^\varepsilon$  overhead during evaluation (as we paid in Option A) and would prefer to just pay  $\min\{T, S\}^{o(1)}$  (as we did in option B). This is also possible.

**Theorem 8.2** (RAM-FHE, Option C). *Under the same conditions as in Theorem 8.1, for any  $\varepsilon > 0$ , there is a secure RAM-FHE scheme with the following efficiency:*

- Setup and Gen run in time  $\text{poly}(\lambda, d)$ ,
- Enc and Prep run in time  $|x|^{1+\varepsilon} \text{poly}(\lambda, d)$  and  $|y|^{1+\varepsilon} \text{poly}(\lambda, d)$  respectively,
- Eval runs in time  $T \cdot \text{poly}(\lambda, d) \cdot |C|$ , and
- Dec runs in time  $M \cdot \text{poly}(\lambda, d)$ ,

where  $d = \text{poly}(2^{\sqrt{\log N'_S}}, \log N_x, \log N_y) = N_S^{o(1)} \text{poly}(\log N_x, \log N_y)$ .

*Proof.* We use different DEPIR parameters when encrypting/preprocessing  $x, y$  vs when preprocessing the homomorphic memory. For  $x, y$  we use degree  $d_x = \log^c N_x, d_y = \log^c N_y$  respectively, for a sufficiently large constant  $c$  depending on  $\varepsilon$ , as in Option A of DEPIR. For the memory, we use degree  $d_S = 2^{\sqrt{\log N'_S}}$  as in option B of DEPIR, where  $N'_S = N_S^{1+o(1)} \cdot \text{poly}(\lambda, \log N)$  is a bound on the bit-size of  $\text{DB}_L$ . To make this work, we need to choose the overall ASHE-FHE parameters to accommodate the largest of these degrees, and therefore we set  $d = \max\{d_x, d_y, d_S\}$ .  $\square$

## 8.2 Leveled RAM-FHE without Circular Security

We now discuss how to construct *leveled* RAM-FHE from RingLWE without relying on a circular security assumption. This comes at a cost of needing to bound the maximal run-time  $T$  of the computation already during key-generation and having the run-time of key generation and the size of the public key  $pk$  scale linearly with this bound. The run-time of the preprocessing, encryption, decryption and evaluation otherwise stays the same. Note that the client needs to generate  $pk$  and send it to the server in order for the server to perform homomorphic evaluation; therefore, leveled RAM-FHE effectively makes the client’s run-time and communication complexity linear in the run-time  $T$  of the program.<sup>26</sup>

**Comparison of Leveled RAM-FHE and DEPIR.** We compare leveled RAM-FHE to the alternative where the client simply runs the program  $P$  locally over her input  $x$ , and uses DEPIR to access the plaintext input  $y$  stored on the server. In both cases, the client’s run-time and the communication complexity are linear in  $T$ . The main advantage of leveled RAM-FHE are: (1) Using RAM-FHE the entire computation only needs 2 rounds of interaction between the client/server whereas using DEPIR the number of rounds would be linear in  $T$ , (2) if the same client wants to outsource many different RAM computations then it only needs to pay the linear cost in  $T$  once to generate  $pk$ , but can reuse it to encrypt different inputs. In contrast, if we use full (non-leveled) RAM-FHE then the client’s run-time is sub-linear in  $T$ .

**Definition of Leveled RAM-FHE.** To define leveled RAM-FHE, we lightly modify the *syntax* of RAM-FHE (Definition 7.2), by letting the Gen algorithm take an additional input  $T_{\max}$ . We then modify the *correctness* requirement to only hold for programs  $P = (C, T, S, M)$  for which  $T \cdot \text{depth}(C) \leq T_{\max}$ , where  $\text{depth}(C)$  denotes the circuit depth of  $C$ . For *efficiency*, the run-time of key generation and the size of the public-key can grow linearly with  $T_{\max}$ ; we assume that all the procedures have random-access to  $pk$  and therefore their efficiency can be sub-linear in  $T_{\max}$ .

**Construction of Leveled RAM-FHE.** Our construction of leveled RAM-FHE without a circular security assumption is similar to our construction of RAM-FHE from ASHE-FHE with light modifications. Most importantly, we rely on Remark 6.2, which discusses how to construct leveled ASHE-FHE without circular security, using a ladder of public/secret keys  $pk_i, sk_i$ , where we have an encryption of  $sk_i$  under  $pk_{i+1}$ , and using this ladder instead of a key cycle to perform bootstrapping. Each ASHE-FHE ciphertext is associated with a level  $i$  and the level increases when performing FHE/ASHE operations.

*Random-Access to  $y$ .* We can plug such a leveled ASHE-FHE directly into our most basic constructions of RAM-FHE with random-access to  $y$  (Algorithms 7.3). We set the number of levels

---

<sup>26</sup>Just like in leveled FHE for circuits, the efficiency scales with the *depth* of the computation. Our RAM model is naturally defined to sequentially execute  $T$  steps of some step circuit  $C$ , and therefore the depth of the computation is linear in the run-time  $T$ . We could define a more flexible model of RAM computation as a “leveled RAM circuit” that has special gates for reading from  $x, y$  and reading/writing to memory  $z$ . For example, a gate for reading from  $x$  would have input wires corresponding to the bits of some location  $i$  and the output wire would contain  $x[i]$ . Each level can have at most one “write to  $z$ ” gate, ensuring that these are executed sequentially. Such a circuit naturally allows for some parallelism and its depth can be much smaller than its overall size. In this more flexible model, we would have the size of  $pk$  scale linearly with the number of levels in such leveled RAM circuit, rather than the overall run-time of the computation.

$L^* := T_{\max} \cdot \text{poly log } N$  for some appropriate  $\text{poly log } N$  terms defined later. The initial encryption of  $x$  is under  $\text{pk}_0$  and therefore at level 0. Recall that the homomorphic evaluation proceeds in steps, where each step computes an updated ASHE-FHE encrypted configuration

$$(\text{ct}_{\text{state}}^t, \hat{b}_x^t, \hat{b}_y^t, \hat{b}_z^t, \hat{z}^t).$$

For  $t = 0$ , the configuration is an ASHE-FHE encryption at level 0. In each subsequent step the level grows by some amount  $\text{depth}(C) + \text{poly log } N$  that depends on the depth of the circuit  $C$  and also the auxiliary circuits  $C_{\text{read}}, C_{\text{write}}, \text{base}_{d,m}$  of depth  $\text{poly log}(N)$ . The level also increases by 1 when we apply Refresh on the DEPIR response. We set  $L^*$  appropriately so that this is the maximal number of levels needed. Note that we can always increase the level of a ciphertext as needed throughout the computation, by evaluating an identity circuit of appropriate depth.

*Random-access to  $x$ .* We can also plug such a leveled ASHE-FHE scheme into the construction of RAM-FHE with random-access to  $x$  (Algorithm 7.4) analogously, where now the initial encryption  $\text{ct}_{\text{key}}$  of the PRF key is at level 0 and we update it accordingly in each step. This time, we need to also evaluate the circuit  $\text{OTPDec}$  of depth up to  $\text{poly}(\lambda, \log N)$  in each step and therefore we set  $L^* := T_{\max} \cdot \text{poly}(\lambda, \log N)$ .

*Random-access to  $z$ .* To handle random-access to read/write memory  $z$ , we need a little more work. Recall that in our homomorphic memory, we relied on the encrypted key cycle  $\text{ct}_{\text{sk}}$  to strip a layer from the doubly encrypted result when we use DEPIR to read from memory. However, we can solve the same issue with the key ladder instead of the key cycle. When we read from memory in some step  $t$ , the ASHE-FHE encryption of the index  $i_z^t$  is at some corresponding level  $t' = t \cdot \text{poly}(\lambda, \log N)$ . When we run  $\text{ReadMem}$  we need to homomorphically evaluate the program  $P_{\text{search}}(x = (\text{sk}_j, i_z^t), y = \text{DB}_\ell)$ , where the secret key  $\text{sk}_j$  needs to be the secret key for some level  $j \ll t'$  that matches the level of the ASHE-FHE ciphertexts contained in  $\text{DB}_\ell$ , which in turn depends on the time when the data was last reshuffled into  $\text{DB}_\ell$ . To homomorphically evaluate  $P_{\text{search}}$  we therefore need to have an *encryption* of  $\text{sk}_j$  at level  $t'$ . We can derive this using the key ladder by upgrading the level of the encryption of  $\text{sk}_j$  from  $j + 1$  to  $t'$ , but naively this would take  $O(t' - j) = O(T)$  work. Doing this at every step would result in run-time  $O(T^2)$  for the homomorphic evaluation. We can do better! During key generation, the client augments the key ladder by adding exponentially increasing “shortcut jumps” consisting of encryptions of  $\text{sk}_i$  under  $\text{pk}_{i+2^k}$  for  $k \in [\log L^*]$ . This does not hurt security since the graph is acyclic. But now, using such shortcut jumps, we can efficiently derive a level  $t'$  encryption of  $\text{sk}_j$  under  $\text{pk}_{i'}$ , for any  $j, t'$ , in  $\text{poly log}(T)$  steps. Overall, we get the following result.

**Theorem 8.3** (Leveled RAM-FHE). *Assuming RingLWE with quasi-polynomial approximation factors, there is a leveled RAM-FHE scheme. For any security parameter  $\lambda$ , any bounds  $N_x, N_y, N_S$  and  $T_{\max}$ , any program  $P = (C, T, S, M)$  and any inputs  $x, y$  such that  $S \leq N_S, |x| \leq N_x, |y| \leq N_y$  and  $T \cdot \text{depth}(C) \leq T_{\max}$ , the scheme achieves the following efficiency, where  $N := \max\{N_x, N_y, N_S\}$ :*

- Setup runs in time  $\text{poly}(\lambda, \log N)$ ,
- Gen runs in time  $T_{\max} \cdot \text{poly}(\lambda, \log N, \log T_{\max})$ , which also bounds the size of  $\text{pk}$ . The size of  $\text{sk}$  is bounded by  $\text{poly}(\lambda, \log N)$ .
- Enc and Prep run in time  $|x|^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$  and  $|y|^{1+\varepsilon} \cdot \text{poly}(\lambda, \log N)$  respectively,
- Eval runs in time  $T \cdot ((\min\{T, S\})^\varepsilon + |C|) \cdot \text{poly}(\lambda, \log N, \log T_{\max})$ , and

- Dec runs in time  $M \cdot \text{poly}(\lambda, \log N)$ .

We can also achieve alternate parametrizations of leveled RAM-FHE analogous to parameter options B and C in Section 8.1.

### 8.3 Updatable RAM-FHE

In Section 5, we showed how to construct updatable DEPIR that allows us to update individual bits of the database DB and correspondingly update the preprocessed data structure  $\widetilde{\text{DB}}$  in sub-linear time. We can use the same ideas to construct an updatable RAM-FHE scheme, where the server can update individual bits of the plaintext input  $y$  and correspondingly update the preprocessed data structure  $\widetilde{y}$  in sublinear time via some procedure  $\widetilde{y}' := \text{Update}(\widetilde{y}, i, b)$ . Indeed, in our current construction of RAM-FHE, the data structure  $\widetilde{y}$  is the same as in DEPIR, and we can simply replace it by the updatable DEPIR from Section 5. The new data structure for  $\widetilde{y}$  now consists of  $\log |y|$  levels, each of which contains a preprocessed data-structure  $\widetilde{\text{DB}}_\ell$  for a database  $\text{DB}_\ell$  under the basic (non-updatable) DEPIR. To perform reads, we now need to homomorphically evaluate the binary search program over the data structure  $\widetilde{\text{DB}}_\ell$  in each level, which we already know how to do using non-updatable RAM-FHE. Overall, we achieve an updatable RAM-FHE with the same security and efficiency as in Theorem 7.7 and the run-time of Update is  $|y|^\varepsilon \cdot \text{poly}(\lambda, \log N)$ .

### 8.4 Multi-Hop / Multi-Input RAM-FHE

In this section, we discuss how to extend our RAM-FHE to handle multi-hop and multi-input evaluation. In a *multi-hop* evaluation, we can homomorphically evaluate many RAM programs with some persistent memory  $z$  between executions; the final contents of the memory  $z$  at the end of one program execution become the initial memory contents for the next program execution. In a *multi-input* homomorphic evaluation, each program can operate over many separately encrypted inputs  $x_i$  under the same key, many separately preprocessed plaintext inputs  $y_i$ , and even many read/write memories  $z_i$  that may have been filled by other prior program executions.

#### 8.4.1 Multi-Hop

**Definition.** As in Section 7.1, we consider a RAM program  $P = (C, T, S, M)$  with step circuit  $C$ , run-time  $T$ , space  $S$ , and output length  $M$ . The only difference is that now we do not necessarily assume that the memory  $z \in \{0, 1\}^S$  starts zeroed out, but instead directly allow for an arbitrary initial memory  $z$  that is explicitly provided as an input. In other words, we write  $(\text{out}, z^T) := P(x, y, z)$  to denote the execution of  $P$  with initial memory content  $z \in \{0, 1\}^S$ , final memory content  $z^T \in \{0, 1\}^S$  and output  $\text{out}$ . We define multi-hop RAM-FHE where the homomorphic evaluation algorithm takes as input a homomorphic memory data-structure  $\widetilde{z}$  and updates it during the execution. The updated data structure  $\widetilde{z}$  can then be used as the initial homomorphic memory of a future homomorphic computations.

**Definition 8.4.** A multi-hop RAM-FHE scheme is a tuple of algorithms (Setup, Gen, Prep, Enc, Init, Eval, Dec). The syntax of Init and Eval is given below, while the syntax of the remaining algorithms are the same as the standard single-hop RAM-FHE (see Definition 7.2):

- $\widetilde{z} := \text{Init}(S)$ : Initializes a homomorphic memory data structure  $\widetilde{z}$  for the empty memory  $z = 0^S$ .

- $(\text{ct}_{\text{out}}, \tilde{z}^T) := \text{Eval}(\text{pk}, P, \tilde{\text{ct}}_x, \tilde{y}, \tilde{z})$ : We augment the syntax of  $\text{Eval}$  from Definition 7.2 to allow us to explicitly provide some initial homomorphic memory data structure  $\tilde{z}$  and to denote the final contents of the data structure at the end of the evaluation by  $\tilde{z}^T$ .

We augment the correctness requirement from Definition 7.2 as follows.

**Correctness:** There is a predicate  $\text{GoodMem}_{\text{sk}}(\tilde{z}, z)$  that corresponds to  $\tilde{z}$  being a “good” homomorphic memory of  $z$  with respect to the given secret key  $\text{sk}$  such that the following requirements hold. For any  $\lambda, N_x, N_y, N_S \in \mathbb{N}$ , for  $\text{params} := \text{Setup}(1^\lambda, N_x, N_y, N_S)$  and for any  $(\text{pk}, \text{sk}) \leftarrow \text{Gen}(\text{params})$ :

- For any  $S < N_S$ ,  $\tilde{z} := \text{Init}(S)$  satisfies  $\text{GoodMem}_{\text{sk}}(\tilde{z}, 0^S) = 1$ .
- Let  $P = (C, T, S, M)$  be a RAM program and  $(x, y)$  be any input pair such that  $|x| < N_x$ ,  $|y| < N_y$ , and  $S < N_S$ . For any  $z \in \{0, 1\}^S$  and any  $\tilde{z}$  such that  $\text{GoodMem}_{\text{sk}}(\tilde{z}, z) = 1$ :

$$\Pr \left[ \begin{array}{l} \text{Dec}(\text{sk}, \text{ct}_{\text{out}}) = \text{out} \\ \wedge \text{GoodMem}_{\text{sk}}(\tilde{z}^T, z^T) = 1 \end{array} \quad ; \quad \begin{array}{l} \tilde{y} := \text{Prep}(\text{params}, y) \\ \tilde{\text{ct}}_x \leftarrow \text{Enc}(\text{pk}, x) \\ (\text{ct}_{\text{out}}, \tilde{z}^T) := \text{Eval}(\text{pk}, P, \tilde{\text{ct}}_x, \tilde{y}, \tilde{z}) \\ (\text{out}, z^T) := P(x, y, z) \end{array} \right] = 1.$$

**Security:** The security requirement is the same as in Definition 7.2.

**Construction and Results.** Our construction from Section 7 essentially achieves the above notion of multi-hop RAM-FHE as is. In particular, we already explicitly define a homomorphic memory data structure  $\tilde{z}$  in Algorithm 7.5, which also gives us the  $\text{Init}$  algorithm. Our  $\text{Eval}$  procedure in Algorithm 7.6 already explicitly operates over a homomorphic memory  $\tilde{z}$ , and we can simply provide some initial  $\tilde{z}$  as an input. Our correctness analysis already explicitly defined the predicate  $\text{GoodMem}$  and directly shows the correctness property needed for multi-hop RAM-FHE.

For efficiency, there is a slight subtlety in that our homomorphic memory data structure, as presented, only achieves good *amortized* update time, but the worst-case update time may be poor. This already suffices to get a good bound on the total run-time of a sequence of homomorphic multi-hop program executions. In particular, we can bound the total run-time of a sequence of operations:

$$\tilde{z}_0 := \text{Init}(S), \quad (\text{ct}_{\text{out}}, \tilde{z}_{i+1}) := \text{Eval}(\text{pk}, P_i, \tilde{\text{ct}}_{x_i}, \tilde{y}_i, \tilde{z}_i)$$

with different programs  $P_i = (C_i, T_i, S, M_i)$  and encrypted/preprocessed inputs  $\tilde{\text{ct}}_{x_i}, \tilde{y}_i$ , by:

$$T \cdot (S^\epsilon + \max\{|C_i|\}) \cdot \text{poly}(\lambda, \log N)$$

where  $T = \sum_i T_i$ . However, the worst-case run-time of the  $i$ -th execution  $\text{Eval}(\text{pk}, P_i, \tilde{\text{ct}}_{x_i}, \tilde{y}_i, \tilde{z}_i)$  may be significantly larger than just  $T_i \cdot (S^\epsilon + |C_i|) \cdot \text{poly}(\lambda, \log N)$ .

To fix this, we can deamortize the homomorphic memory to get essentially the same performance in the worst-case rather than just amortized. The technique for deamortization is identical to that of updatable DEPIR (see Remark 5.1, based on [OS97]). Namely, whenever  $\text{WriteMem}$  would merge the databases  $(\text{DB}_0, \text{DB}_1, \dots, \text{DB}_{\ell^*})$  for some  $\ell^*$ , we spread the work to the subsequent  $O(2^{\ell^*})$  updates; meanwhile, we keep old copies of the  $\tilde{\text{DB}}_\ell$ 's to correctly answer  $\text{ReadMem}$  queries. The asymptotic worst-case run-time is then the same as the previous amortized run-time. We omit the details of deamortization and we conclude with the corollary below. Therefore, we can get the same efficiency of multi-hop RAM-FHE as standard RAM-FHE.

**Corollary 8.5.** *Theorem 7.7 also holds for multi-hop RAM-FHE under the same assumptions and with the same efficiency, except that:*

- *The run-time of  $\text{Init}(S)$  is  $O(\log S)$ .*
- *The run-time of  $\text{Eval}$  for a program  $P = (C, T, S, M)$  is bounded by  $T \cdot (S^\epsilon + |C|) \cdot \text{poly}(\lambda, \log N)$ .*

We can achieve the alternative parameters of Theorems 8.1 and 8.2 for multi-hop RAM-FHE analogously. When we bound the run-time of  $\text{Eval}$ , we need to replace the  $\min\{S, T\}$  term in the original statements by  $S$ , since now the total size of memory used may be  $S$  even if the program's run-time is  $T < S$ .

### 8.4.2 Multi-Input

We can also extend homomorphic evaluation to operate over many separately encrypted inputs  $\tilde{\text{ct}}_{x_i}$  (all under the same key), separately preprocessed plaintext inputs  $\tilde{y}_j$ , and separate homomorphic memory data structures  $\tilde{z}_k$  (also all under the same key). In this case, the syntax of the RAM program  $P = (C, T, \{S_k\}_{k \in [\ell_z]}, M)$  is:

$$(\text{out}, \{z_k^T\}_{k \in [\ell_z]}) := P(\{x_i\}_{i \in [\ell_x]}, \{y_j\}_{j \in [\ell_y]}, \{z_k\}_{k \in [\ell_z]})$$

where each step reads bit from each  $x_i, y_j, z_k$  writes one bit to each memory  $z_k$ . The memories  $z_k$  can be of different sizes  $S_k$ . The homomorphic evaluation is now defined by

$$(\text{ct}_{\text{out}}, \{\tilde{z}_k^T\}_{k \in [\ell_z]}) := \text{Eval}(\text{pk}, P, \{\tilde{\text{ct}}_{x_i}\}_{i \in [\ell_x]}, \{\tilde{y}_j\}_{j \in [\ell_y]}, \{\tilde{z}_k\}_{k \in [\ell_z]}).$$

We assume that each  $\tilde{\text{ct}}_{x_i} \leftarrow \text{Enc}(\text{pk}, x_i)$  is a separately encrypted input under the same key  $\text{pk}$ , and each  $\tilde{y}_j := \text{Prep}(y_j)$  is a separately preprocessed input, and each  $\tilde{z}_k$  is a good homomorphic memory data structure for some corresponding memory  $z_k$  of size  $S_k$  under the same key  $\text{sk}$ , satisfying  $\text{GoodMem}_{\text{sk}}(\tilde{z}_k, z_k) = 1$ . Correctness is defined in the natural way as ensuring that  $\text{Dec}(\text{sk}, \text{ct}_{\text{out}}) = \text{out}$  and  $\text{GoodMem}(\tilde{z}_k^T, z_k^T) = 1$ .

The construction is the same as before, and we use the respective methods for reading from an encrypted input  $\tilde{\text{ct}}_{x_i}$ , a preprocessed plaintext input  $\tilde{y}_j$ , and reading/writing to the homomorphic memory  $\tilde{z}$  on each component separately. The run-time of  $\text{Eval}$  for a program  $P = (C, T, \{S_k\}_{k \in \ell_z}, M)$  is now bounded by

$$T \cdot \left( \sum_{k \in [\ell_z]} S_k^\epsilon + |C| \right) \cdot \text{poly}(\lambda, \log N).$$

Note that  $|C| \geq \ell_x + \ell_y + \ell_z$  since the step circuit  $C$  must output  $\ell_x + \ell_y + \ell_z$  indices.

**Remark on Memory as Output.** Note that the output ciphertext  $\text{ct}_{\text{out}}$  does not allow for future homomorphic evaluations with random-access to its contents. However, using multi-hop/multi-input RAM-FHE, we can instead use one of the homomorphic memory data structures  $\tilde{z}_k$  as the encrypted output and ignore  $\text{ct}_{\text{out}}$  entirely. In particular, if we want to evaluate some RAM program  $P$  with  $S$ -bit read/write memory and  $M$  bit output, we can think of it as a program with two memories: an “internal memory”  $z_1$  of size  $S$  and an “output memory”  $z_2$  of size  $M$ , where the final output will be written. The homomorphic evaluation will then have two homomorphic memory data structures  $\tilde{z}_1, \tilde{z}_2$  corresponding to  $z_1, z_2$ , and we can think of  $\tilde{z}_2$  as the encrypted output of the computation, which can then be used as an input to future homomorphic evaluations.

### 8.4.3 Additional Extensions

**Parallel/Distributed RAM computation.** Using multi-hop and multi-input RAM-FHE, we can perform a homomorphic computation in a parallelized/distributed setting. Suppose we have  $n$  processors, each of which starts out holding a pair of inputs  $x_i, y_i$ . We have some distributed protocol that the processors can execute over the plaintexts, where the protocol runs in many rounds, and each round the processors perform some local RAM computations over their current states in parallel, and then exchange messages with each other. Further, assume that the communication pattern (which processor talks with which other in each round), the sizes of the exchanged messages, the run-time of each processor in each step, and the memory size of each processor are fixed non-adaptively. Then we can execute this distributed protocol homomorphically over encrypted data under RAM-FHE, when each processor starts off with a ciphertext  $ct_{x_i}$  encrypting  $x_i$  (all under the same key) and a preprocessed version  $\tilde{y}_i$  of  $y_i$ . The processors don't know the secret key, which may be held by an external party. The processors simply run the local computation of each round homomorphically using RAM-FHE and exchange the encrypted outputs at the end of each round. Here we are using the "memory as output" trick discussed previously to be able to perform future computations with random-access to the outputs of previous computations.

In terms of efficiency, the round complexity of the homomorphic protocol execution is the same as that of the plaintext execution. The communication complexity of the homomorphic execution is bounded by  $C^{1+\epsilon} \text{poly}(\lambda, \log N)$  where  $C$  is the communication complexity of the original protocol. This comes from the fact that the encrypted communication between the processors in each round is represented as a homomorphic memory data structure which blows up the message size from  $S$  to  $S^{1+\epsilon} \text{poly}(\lambda, \log N)$ . The run-time of each processor  $i$  in the homomorphic protocol execution is  $T_i \cdot S_i^{1+\epsilon} \cdot \text{poly}(\lambda, \log N)$  where  $T_i$  is the processor's run-time in the plaintext execution and  $S_i$  is its space complexity in the plaintext execution.

**Scaling the size of homomorphic memory.** In a multi-hop homomorphic evaluation, where we execute multiple programs that operate over some persistent memory  $z$ , we assumed the memory size stays fixed throughout. However, this is not necessary. We can easily take some homomorphic memory data structure  $\tilde{z}$  corresponding to a memory  $z$  of size  $S$  and *resize* it: we can either scale it up to  $S' > S$  or down to  $S' < S$  by taking the first  $S'$  locations. An easy way to do this is to initialize a fresh homomorphic memory data structure  $\tilde{z}'$  for a memory  $z'$  of size  $S'$ . We then homomorphically execute a multi-input RAM program  $P$  that operates over the original memory  $z$  of size  $S$  as well as a new memory  $z'$  of size  $S'$  and simply copies the first  $\min\{S, S'\}$  bits of  $z$  to  $z'$ ; the result of the homomorphic execution will be a correctly filled homomorphic memory  $\tilde{z}'$ . The run-time of this process will be  $\min\{S, S'\} \cdot (\max\{S, S'\})^\epsilon \cdot \text{poly}(\lambda, \log N)$

**Large Output / Small-space Programs.** So far, we assumed that the output size  $M$  of the program is bounded by the space  $S$ , i.e.,  $M \leq S$ . In particular, our RAM model defined the output to be written in the first  $M$  memory locations at the end of the execution. However, it also makes sense to define a RAM model where programs can have larger output size than space, i.e.,  $M > S$ . The program is given access to an append-only output tape, and for each step  $t \in [T]$ , the program writes at most one bit to the output tape. We further assume that the steps  $t$  in which the program writes to the output tape are fixed a-priori and non-adaptively. We can homomorphically evaluate such programs by simply taking the encrypted output bits produced during the homomorphic

evaluation of the circuit  $C$  (see Step 2a of Algorithm 7.3) in the special steps  $t$  and appending them together to form the output ciphertext. The run-time of homomorphic evaluation is the same as that claimed in our single-hop RAM-FHE, that is, Theorems 7.7, 8.1, 8.2, but without the restriction of  $M \leq S$ . In particular, the overhead of the homomorphic evaluation depends on  $S^\epsilon$  rather than the potentially much larger  $M^\epsilon$ .

**Circuit Privacy.** The output ciphertext  $\text{ct}_{\text{out}}$  of a RAM-FHE computation may reveal something about the program  $P$  and the server's input  $y$  beyond the output  $f(x, y)$ , especially to a client who knows  $\tilde{\text{ct}}_x$  and  $\text{sk}$ . Indeed, if the homomorphic evaluation is deterministic then this is inherently so. However, we can achieve *circuit privacy* and ensure that  $\text{ct}_{\text{out}}$  does not reveal anything beyond the output in the same way as in standard FHE. See [OPP14, DS16, BdMW16, DD22].

## References

- [ACFQ22] Prabhanjan Ananth, Kai-Min Chung, Xiong Fan, and Luowen Qian. Collusion-resistant functional encryption for RAMs. Cryptology ePrint Archive, Paper 2022/1269, 2022. <https://eprint.iacr.org/2022/1269>, to appear in Asiacrypt 2022. 12
- [ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 595–618. Springer, Heidelberg, August 2009. 15
- [AKL<sup>+</sup>20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMA: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 403–432. Springer, Heidelberg, May 2020. 11
- [AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An  $O(n \log n)$  sorting network. In *15th Annual ACM Symposium on Theory of Computing*, pages 1–9. ACM Press, April 1983. 48
- [Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring)*, page 307–314. Association for Computing Machinery, 1968. 48
- [BdMW16] Florian Bourse, Rafaël del Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016, Part II*, volume 9815 of *Lecture Notes in Computer Science*, pages 62–89. Springer, Heidelberg, August 2016. 59
- [BGG<sup>+</sup>22] Vishwas Bhargava, Sumanta Ghosh, Zeyu Guo, Mrinal Kumar, and Chris Umans. Fast multivariate multipoint evaluation over all finite fields, 2022. <https://arxiv.org/abs/2205.00342>, to appear in FOCS 2022. 6

- [BGKM22] Vishwas Bhargava, Sumanta Ghosh, Mrinal Kumar, and Chandra Kanta Mohapatra. Fast, algebraic multivariate multipoint evaluation in small characteristic and applications. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 403–415. Association for Computing Machinery, 2022. [6](#)
- [BGL<sup>+</sup>15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 439–448. ACM Press, June 2015. [12](#)
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012: 3rd Innovations in Theoretical Computer Science*, pages 309–325. Association for Computing Machinery, January 2012. [4](#), [5](#), [8](#), [9](#), [10](#), [32](#), [33](#), [34](#), [35](#), [75](#), [77](#)
- [BHMW21] Elette Boyle, Justin Holmgren, Fermi Ma, and Mor Weiss. On the security of doubly efficient PIR. Cryptology ePrint Archive, Report 2021/1113, 2021. <https://eprint.iacr.org/2021/1113>. [2](#)
- [BHW19] Elette Boyle, Justin Holmgren, and Mor Weiss. Permuted puzzles and cryptographic hardness. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019: 17th Theory of Cryptography Conference, Part II*, volume 11892 of *Lecture Notes in Computer Science*, pages 465–493. Springer, Heidelberg, December 2019. [2](#)
- [BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 55–73. Springer, Heidelberg, August 2000. [1](#), [2](#), [12](#)
- [BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part II*, volume 10678 of *Lecture Notes in Computer Science*, pages 662–693. Springer, Heidelberg, November 2017. [1](#), [2](#), [4](#), [5](#), [18](#), [19](#)
- [BJRW20] Katharina Boudgoust, Corentin Jeudy, Adeline Roux-Langlois, and Weiqiang Wen. Towards classical hardness of module-LWE: The linear rank case. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part II*, volume 12492 of *Lecture Notes in Computer Science*, pages 289–317. Springer, Heidelberg, December 2020. [75](#)
- [BLP<sup>+</sup>13] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 575–584. ACM Press, June 2013. [14](#)
- [BV11a] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In Rafail Ostrovsky, editor, *52nd Annual Symposium on Foundations of Computer Science*, pages 97–106. IEEE Computer Society Press, October 2011. [2](#)

- [BV11b] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, Heidelberg, August 2011. [2](#), [5](#), [6](#), [8](#), [9](#), [10](#), [15](#), [16](#), [33](#)
- [CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B: 14th Theory of Cryptography Conference, Part II*, volume 9986 of *Lecture Notes in Computer Science*, pages 61–90. Springer, Heidelberg, October / November 2016. [12](#)
- [CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science*, pages 41–50. IEEE Computer Society Press, October 1995. [1](#)
- [CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In Madhu Sudan, editor, *ITCS 2016: 7th Conference on Innovations in Theoretical Computer Science*, pages 169–178. Association for Computing Machinery, January 2016. [12](#)
- [CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th Annual ACM Symposium on Theory of Computing*, pages 429–437. ACM Press, June 2015. [12](#)
- [CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 3–33. Springer, Heidelberg, May / June 2022. [11](#)
- [CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017: 15th Theory of Cryptography Conference, Part II*, volume 10678 of *Lecture Notes in Computer Science*, pages 694–726. Springer, Heidelberg, November 2017. [1](#), [2](#), [4](#), [5](#), [18](#)
- [CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sub-linear online time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 44–75. Springer, Heidelberg, May 2020. [11](#)
- [CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg, editors, *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, pages 73–80. ACM, 1972. [3](#)
- [CS15] Jung Hee Cheon and Damien Stehlé. Fully homomorphic encryption over the integers revisited. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 513–536. Springer, Heidelberg, April 2015. [75](#)

- [DD22] Nico Döttling and Jesko Dujmovic. Maliciously circuit-private FHE from information-theoretic principles. In Dana Dachman-Soled, editor, *3rd Conference on Information-Theoretic Cryptography, ITC 2022, July 5-7, 2022, Cambridge, MA, USA*, volume 230 of *LIPICs*, pages 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. [59](#)
- [DS16] Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016, Part I*, volume 9665 of *Lecture Notes in Computer Science*, pages 294–310. Springer, Heidelberg, May 2016. [59](#)
- [Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st Annual ACM Symposium on Theory of Computing*, pages 169–178. ACM Press, May / June 2009. [2](#)
- [GG13] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, England, 3rd edition, 2013. [73](#)
- [GHL<sup>+</sup>14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 405–422. Springer, Heidelberg, May 2014. [12](#)
- [GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *55th Annual Symposium on Foundations of Computer Science*, pages 404–413. IEEE Computer Society Press, October 2014. [12](#)
- [GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled RAM. In Venkatesan Guruswami, editor, *56th Annual Symposium on Foundations of Computer Science*, pages 210–229. IEEE Computer Society Press, October 2015. [12](#)
- [GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, may 1996. [8](#), [11](#), [25](#)
- [Goo14] Michael T. Goodrich. Zig-zag sort: a simple deterministic data-oblivious sorting algorithm running in  $O(n \log n)$  time. In David B. Shmoys, editor, *46th Annual ACM Symposium on Theory of Computing*, pages 684–693. ACM Press, May / June 2014. [48](#)
- [GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, Heidelberg, August 2013. [8](#)
- [HHWW19] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. On the plausibility of fully homomorphic encryption for RAMs. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 589–619. Springer, Heidelberg, August 2019. [3](#), [4](#), [8](#), [25](#), [31](#), [38](#)

- [HOWW19] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 244–273. Springer, Heidelberg, May 2019. 8, 12, 25, 31
- [HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, volume 1423 of *Lecture Notes in Computer Science*, pages 267–288. Springer, 1998. 4, 14
- [JLL22] Aayush Jain, Huijia Lin, and Ji Luo. On the optimal succinctness and efficiency of functional encryption and attribute-based encryption. Cryptology ePrint Archive, Paper 2022/1317, 2022. <https://eprint.iacr.org/2022/1317>. 12
- [KL21] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious RAM (for all parameters). In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021, Part IV*, volume 12828 of *Lecture Notes in Computer Science*, pages 579–609, Virtual Event, August 2021. Springer, Heidelberg. 11
- [KO00] Eyal Kushilevitz and Rafail Ostrovsky. One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 104–121. Springer, Heidelberg, May 2000. 1
- [KU08] Kiran S. Kedlaya and Christopher Umans. Fast modular composition in any characteristic. In *49th Annual Symposium on Foundations of Computer Science*, pages 146–155. IEEE Computer Society Press, October 2008. 5
- [KU11] Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. *SIAM Journal on Computing*, 40(6):1767–1802, 2011. 5, 6, 7, 13, 65, 68, 71
- [LM06] Vadim Lyubashevsky and Daniele Micciancio. Generalized compact Knapsacks are collision resistant. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP 2006: 33rd International Colloquium on Automata, Languages and Programming, Part II*, volume 4052 of *Lecture Notes in Computer Science*, pages 144–155. Springer, Heidelberg, July 2006. 14
- [LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 523–542. Springer, Heidelberg, August 2018. 11
- [LO13] Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 719–734. Springer, Heidelberg, May 2013. 12

- [LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EURO-CRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, Heidelberg, May / June 2010. 4, 14, 15
- [LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptography*, 75(3):565–599, jun 2015. 4, 75
- [LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In Howard J. Karloff and Toniann Pitassi, editors, *44th Annual ACM Symposium on Theory of Computing*, pages 1219–1234. ACM Press, May 2012. 4, 8
- [Mic02] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions. In *43rd Annual Symposium on Foundations of Computer Science*, pages 356–365. IEEE Computer Society Press, November 2002. 14
- [NIS] Post-quantum cryptography: Selected algorithms 2022. <https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022>. Accessed: 2022-10-28. 4
- [OPP14] Rafail Ostrovsky, Anat Paskin-Cherniavsky, and Beni Paskin-Cherniavsky. Maliciously circuit-private FHE. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 536–553. Springer, Heidelberg, August 2014. 59
- [OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th Annual ACM Symposium on Theory of Computing*, pages 294–303. ACM Press, May 1997. 9, 29, 56
- [PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979. 3
- [PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMA: Oblivious RAM with logarithmic overhead. In Mikkel Thorup, editor, *59th Annual Symposium on Foundations of Computer Science*, pages 871–882. IEEE Computer Society Press, October 2018. 11
- [PR06] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In Shai Halevi and Tal Rabin, editors, *TCC 2006: 3rd Theory of Cryptography Conference*, volume 3876 of *Lecture Notes in Computer Science*, pages 145–166. Springer, Heidelberg, March 2006. 14
- [PRS17] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. Pseudorandomness of ring-LWE for any ring and modulus. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *49th Annual ACM Symposium on Theory of Computing*, pages 461–473. ACM Press, June 2017. 14

- [RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation, Academia Press*, 1978. [2](#)
- [vGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, Heidelberg, May / June 2010. [4](#), [8](#), [74](#), [75](#)
- [ZLTS22] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. *Cryptology ePrint Archive*, Report 2022/609, 2022. <https://eprint.iacr.org/2022/609>. [11](#)

## A Fast Polynomial Evaluation with Preprocessing

We prove Theorem [2.1](#), restated below.

**Theorem 2.1** (Preprocessing Polynomials [[KU11](#)]). *Let  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for some  $q \in \mathbb{N}$  and arbitrary monic polynomials  $E_1$  over  $Y$  and  $E_2$  over  $Z$  with degrees  $e_1, e_2 > 0$  respectively. Let  $f \in R[X_1, X_2, \dots, X_m]$  be a polynomial of individual degree  $< d$  in every variable. Then, there is some preprocessing algorithm that takes the coefficients of  $f$  as an input, runs in time*

$$S = d^m \cdot \text{poly}(m, d, \log |R|) \cdot O(m(\log m + \log d + \log \log |R|))^m$$

*and outputs a data structure of size at most  $S$ , and there is some evaluation algorithm with random access to the data structure, that is given an evaluation point  $\alpha \in R^m$  and computes  $f(\alpha)$  in time*

$$\text{poly}(d, m, \log |R|).$$

We now give a full proof of the theorem, using the techniques extracted from Kedlaya and Umans [[KU11](#), Section 4]. The proof relies on the following two auxiliary lemmas.

**Lemma A.1** (Product of small primes [[KU11](#), Lemma 2.4]). *For all integers  $M \geq 2$ , the product of the primes less than or equal to  $16 \log M$  is greater than  $M$  (where the log is base 2). That is,*

$$M < \prod_{\substack{p \leq 16 \log M, \\ p \text{ prime}}} p.$$

**Lemma A.2** (Multidimensional FFT [[KU11](#), Theorem 4.1]). *Let  $\mathbb{F}_p$  be a field of prime order  $p \in \mathbb{N}$ . Given  $m$ -variate polynomial  $f(X_1, \dots, X_m) \in \mathbb{F}_p[X_1, \dots, X_m]$  of individual degree  $< d$  in each variable, there exists a deterministic algorithm that outputs  $f(\alpha)$  for all  $\alpha \in \mathbb{F}_p^m$  in time  $O(m(d^m + p^m) \cdot \text{poly}(\log p))$ .*

We prove Theorem [2.1](#) in stages. First we prove it for the special case where the ring is  $R = \mathbb{Z}_q$ . Then we prove it for  $R = \mathbb{Z}_q[Y]/(E(Y))$  by reducing this problem to the case where the ring is  $\mathbb{Z}_{q'}$  for some  $q'$ . Finally, we prove it for  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  by reducing this problem to the case where the ring is  $\mathbb{Z}_{q'}[Y]/(E'(Y))$  for some  $q', E'$ .

## A.1 Polynomials Over $\mathbb{Z}_q$

We begin by proving a special case of Theorem 2.1 for the ring  $R = \mathbb{Z}_q$  for any  $q \in \mathbb{N}$ . There are two parts of the construction: we first use the Chinese Remainder Theorem (CRT) and modular reduction to obtain a “large data structure” of size proportional to  $(\log q)^m$ , and then we recursively use this construction to obtain a “small data structure” of size proportional to  $(\log \log q)^m$ .

**Note:** Throughout this section we identify elements of  $\mathbb{Z}_q$  with their integer representatives in the range  $\{0, \dots, q-1\}$ . This allows us to “lift” elements of  $\mathbb{Z}_q$  to  $\mathbb{Z}$  by taking the appropriate representative.<sup>27</sup>

**Theorem A.3** (Preprocessing with large data structure over  $\mathbb{Z}_q$ ). *Let  $q \in \mathbb{N}$  and let  $f \in \mathbb{Z}_q[X_1, X_2, \dots, X_m]$  be a polynomial of individual degree  $< d$  in every variable. Then, there is some preprocessing algorithm that takes the coefficients of  $f$  as an input, runs in time*

$$S = O(md \log q)^m \cdot \text{poly}(m, d, \log q)$$

*and outputs a data structure of size at most  $S$ , and there is some evaluation algorithm with random access to the data structure, that is given an evaluation point  $\alpha \in \mathbb{Z}_q^m$  and computes  $f(\alpha)$  in time*

$$\text{poly}(d, m, \log q).$$

*Proof.* The following algorithm preprocesses the coefficients of a polynomial  $f$  into a data structure:

1. Let  $M := d^m q^{m(d-1)+1}$ , and let  $p_1, \dots, p_h$  be the distinct primes less than or equal to  $16 \log M$  for some  $h \in \mathbb{N}$ .
2. Lift  $f \in \mathbb{Z}_q[X_1, \dots, X_m]$  to  $\bar{f} \in \mathbb{Z}[X_1, \dots, X_m]$ , i.e., the coefficients of  $\bar{f}$  are obtained by lifting the corresponding coefficients of  $f$  from  $\mathbb{Z}_q$  to  $\mathbb{Z}$ , where the coefficients are represented by  $\{0, 1, \dots, q-1\}$  in both  $\mathbb{Z}_q$  and  $\mathbb{Z}$ .
3. For each  $i \in [h]$ , let  $\bar{f}_i \in \mathbb{Z}_{p_i}[X_1, \dots, X_m]$  be the polynomial  $\bar{f}_i = \bar{f}$  modulo  $p_i$  (i.e, take every coefficient of  $\bar{f}$  modulo  $p_i$ ). Moreover, compute the following for each  $i$ :
  - (a) Using FFT on  $\mathbb{Z}_{p_i}^m$  (Lemma A.2), evaluate  $\bar{f}_i(a)$  on all points  $a \in \mathbb{Z}_{p_i}^m$ .
  - (b) Let  $T_i$  be the table of evaluations, that is,  $T_i := (\bar{f}_i(a) : a \in \mathbb{Z}_{p_i}^m)$ .
4. The data structure consists of  $p_1, p_2, \dots, p_h, T_1, T_2, \dots, T_h$ .

Given any point  $\alpha \in \mathbb{Z}_q^m$ , the following algorithm computes the evaluation  $f(\alpha)$  using the above data structure:

1. Lift  $\alpha$  to  $\bar{\alpha} \in \mathbb{Z}^m$ , i.e., each coordinate of  $\bar{\alpha}$  is obtained by lifting the corresponding coordinate of  $\alpha$  from  $\mathbb{Z}_q$  to  $\mathbb{Z}$ .
2. For each  $i \in [h]$ , let  $\bar{\alpha}_i \in \mathbb{Z}_{p_i}^m$  be the point obtained by  $\bar{\alpha}$  modulo  $p_i$  coordinate-wise. Moreover, for each  $i$ , the evaluation  $\bar{f}_i(\bar{\alpha}_i) \in \mathbb{Z}_{p_i}$  is obtained by looking it up in table  $T_i$ .

<sup>27</sup>This is in contrast to Sections 3.1, 6.1 where it is more convenient to use the representatives in the range  $(-q/2, q/2]$ . These choices are internal to their respective domains and there is no need for consistency between them.

3. Use Chinese Remainder Theorem (CRT) to find the smallest  $z \in \mathbb{Z}$  such that

$$z \equiv \bar{f}_i(\bar{\alpha}_i) \pmod{p_i} \quad \text{for all } i \in [h].$$

4. Output  $z$  modulo  $q$  as the evaluation  $f(\alpha)$ .

The correctness of the above algorithm is argued as follows. Because the lifted polynomial  $\bar{f}$  and the lifted point  $\bar{\alpha}$  each have coefficients/component in  $\{0, \dots, q-1\}$ , and  $\bar{f}$  has individual degree  $< d$ , we can bound the lifted evaluation over  $\mathbb{Z}$  by:  $0 \leq \bar{f}(\bar{\alpha}) < M$ . By Lemma A.1, we then have  $\bar{f}(\bar{\alpha}) < M < \prod_{i \in [h]} p_i$ . Notice that  $\bar{f}(\bar{\alpha}) \equiv \bar{f}_i(\bar{\alpha}_i) \pmod{p_i}$  for all  $i \in [h]$ . Also, by CRT, there is a unique solution for  $z < \prod_{i \in [h]} p_i$  such that  $z \equiv \bar{f}_i(\bar{\alpha}_i) \pmod{p_i}$  for all  $i \in [h]$ . Recalling that  $\bar{f}_i(\bar{\alpha}_i)$  are correctly computed by the FFT evaluation (Lemma A.2), we have  $z = \bar{f}(\bar{\alpha})$  by the uniqueness. Correctness follows since  $z = \bar{f}(\bar{\alpha}) \pmod{q} = f(\alpha)$ .

We next calculate the computation time of the data structure. As  $h < 16 \log M = O(md \log q)$ , it takes time  $\text{poly}(m, d, \log q)$  to compute  $M, p_1, p_2, \dots, p_h$ . Also, it takes time  $d^m \cdot \text{poly}(m, d, \log q)$  to compute  $\bar{f}_i$  for all  $i \in [h]$  since  $f$  consists of  $d^m$  coefficients. We then apply Lemma A.2 to obtain  $T_i$ 's, which takes time  $(d^m + (16 \log M)^m) \cdot \text{poly}(m, d, \log q)$ . We thus have

$$S = O(md \log q)^m \cdot \text{poly}(m, d, \log q).$$

The evaluation time (given  $\alpha$ ) is straightforward. It takes time  $\text{poly}(m, d, \log q)$  to perform the modular reduction (from  $\alpha$  to  $\bar{\alpha}_i$ ) as well as the looking up for  $\bar{f}_i(\bar{\alpha}_i)$  for all  $i \in [h]$ . Performing the CRT and taking the result modulo  $q$  also take time  $\text{poly}(m, d, \log q)$ . □

We now apply the above algorithm recursively to replace the  $(\log q)^m$  factor in the preprocessing run-time by a  $(\log \log q)^m$  factor. This gives us the following theorem, which is a special case of Theorem 2.1 for  $R = \mathbb{Z}_q$ .

**Theorem A.4** (Theorem 2.1 restricted to  $R = \mathbb{Z}_q$ ). *Let  $q \in \mathbb{N}$ . Let  $f \in \mathbb{Z}_q[X_1, X_2, \dots, X_m]$  be a polynomial of individual degree  $< d$  in every variable. Then, there is some preprocessing algorithm that takes the coefficients of  $f$  as an input, runs in time*

$$S = d^m \cdot \text{poly}(m, d, \log q) \cdot O(m(\log m + \log d + \log \log q))^m$$

*and outputs a data structure of size at most  $S$ , and there is some evaluation algorithm with random access to the data structure, that is given an evaluation point  $\alpha \in R^m$  and computes  $f(\alpha)$  in time*

$$\text{poly}(d, m, \log q).$$

*Proof.* The algorithm that computes the data structure from (the coefficients of)  $f$  is similar to that of Theorem A.3 where the only difference (highlighted in blue below) is that we introduce a level of recursion, applying Theorem A.3 instead of invoking FFT directly. Notice that this means CRT is applied recursively in order to further reduce the problem size in  $q$ .

1. Let  $M := d^m q^{m(d-1)+1}$ , and let  $p_1, \dots, p_h$  be the distinct primes less than or equal to  $16 \log M$  for some  $h \in \mathbb{N}$ .

2. Lift  $f \in \mathbb{Z}_q[X_1, \dots, X_m]$  to  $\bar{f} \in \mathbb{Z}[X_1, \dots, X_m]$ , i.e., the coefficients of  $\bar{f}$  are obtained by lifting the corresponding coefficients of  $f$  from  $\mathbb{Z}_q$  to  $\mathbb{Z}$ , where the coefficients are represented by  $\{0, 1, \dots, q-1\}$  in both  $\mathbb{Z}_q$  and  $\mathbb{Z}$ .
3. For each  $i \in [h]$ , let  $\bar{f}_i \in \mathbb{Z}_{p_i}[X_1, \dots, X_m]$  be the polynomial obtained by  $\bar{f}$  modulo  $p_i$  (i.e., modulo  $p_i$  on every coefficient of  $\bar{f}$ ). Moreover, compute the following for each  $i$ :
  - (a) Apply Theorem A.3 to compute the data structure for the polynomial  $\bar{f}_i$  over  $\mathbb{Z}_{p_i}$ ; let  $T_i$  be the resulting data structure.
4. The data structure consists of  $p_1, p_2, \dots, p_h, T_1, T_2, \dots, T_h$ .

Given any point  $\alpha \in \mathbb{Z}_q^m$ , the evaluation algorithm of  $f(\alpha)$  is almost identical to that of Theorem A.3, where the only difference is that we use Theorem A.3 instead of looking up the table directly (as highlighted in blue).

1. Lift  $\alpha$  to  $\bar{\alpha} \in \mathbb{Z}^m$ , i.e., each coordinate of  $\bar{\alpha}$  is obtained by lifting the corresponding coordinate of  $\alpha$  from  $\mathbb{Z}_q$  to  $\mathbb{Z}$ .
2. For each  $i \in [h]$ , let  $\bar{\alpha}_i \in \mathbb{Z}_{p_i}^m$  be the point obtained by  $\bar{\alpha}$  modulo  $p_i$ . Moreover, for each  $i$ , the evaluation  $\bar{f}_i(\bar{\alpha}_i) \in \mathbb{Z}_{p_i}$  is obtained by invoking the evaluation algorithm of Theorem A.3 using the data structure  $T_i$ .
3. Use Chinese Remainder Theorem (CRT) to find the smallest  $z \in \mathbb{Z}$  such that

$$z \equiv \bar{f}_i(\bar{\alpha}_i) \pmod{p_i} \quad \text{for all } i \in [h].$$

4. Output  $z$  modulo  $q$  as the evaluation  $f(\alpha)$ .

The correctness of the recursion follows by the same argument as Theorem A.3 and the correctness of the base case then follows from Theorem A.3 directly.

We next calculate the efficiency of the data structure. For each  $i \in [h]$ , by Theorem A.3, the data structure  $T_i$  can be computed in time  $S_i = O(md \log p_i)^m \cdot \text{poly}(m, d, \log p_i)$ , and takes at most  $S_i$  space. Since  $p_i \leq 16 \log M = O(md \log q)$  for all  $i$ , the total space of all  $h$  data structures is

$$\begin{aligned} S &= O(md \log(md \log q))^m \cdot \text{poly}(m, d, \log q) \\ &= d^m \cdot \text{poly}(m, d, \log q) \cdot O(m(\log m + \log d + \log \log q))^m. \end{aligned}$$

Finally, given  $\alpha \in \mathbb{Z}_q^m$ , the evaluation time is at most  $\text{poly}(\log M) \cdot \text{poly}(d, m, \log \log M) = \text{poly}(d, m, \log q)$ , where  $\text{poly}(\log M)$  comes from the CRT and  $h < 16 \log M$ , and  $\text{poly}(d, m, \log \log M)$  comes from each evaluation of  $\bar{f}_i(\bar{\alpha}_i)$  using Theorem A.3.  $\square$

Note that the recursion decreases the problem size in  $q$  but at the cost of extra factors in  $(\log m + \log d)^m$ , and that the two-level recursion suffices for our purpose; see [KU11, Theorem 4.2] for a version that allows for more levels of recursion.

## A.2 Polynomials Over Extension Rings

We now extend the results of the previous section to work over a class of extensions rings of  $\mathbb{Z}_q$ . We first show how to construct a data structure for evaluating polynomials over univariate extension rings of the form  $R = \mathbb{Z}_q[Y]/(E(Y))$  where  $E$  is an arbitrary (non-constant) monic polynomial, and then we show how the same techniques allow us to prove an almost identical statement for evaluating polynomials over bivariate extension rings of the form  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for arbitrary monic polynomials  $E_1$  over  $Y$  and  $E_2$  over  $Z$ .

**Overview.** The main idea for  $R = \mathbb{Z}_q[Y]/(E(Y))$  is as follows. First, instead of evaluating  $f(\alpha)$  over  $R$ , let us “lift” the evaluation to  $\mathbb{Z}[Y]$  without reducing modulo  $q$  or  $E(Y)$ . In that case, we can show that the output  $\beta = f(\alpha) \in \mathbb{Z}[Y]$  is a polynomial  $\beta = \sum_{i=0}^D \beta_i Y^i$  of degree  $\leq D$  with non-negative coefficients  $\beta_i < M$  for some integer bounds  $D, M$  that we compute below. But this means that we can recover the entire polynomial  $\beta$  given its evaluation  $\beta(M) \in \mathbb{Z}$  on input  $Y = M$ , since the  $D + 1$  coefficients of  $\beta$  are just the  $D + 1$  digits of  $\beta(M)$  when written in base- $M$ . Furthermore,  $\beta(M) < r$  for  $r := M^{D+1}$ . Therefore, instead of evaluating  $f(\alpha)$  over  $\mathbb{Z}[Y]$  it suffices to evaluate it over  $\mathbb{Z}_r$  by reducing modulo  $(Y - M)$  and modulo  $r$ ; the reduction modulo  $r$  does not have any affect and the the base- $M$  digits of the output  $\beta(M)$  as an integer are the coefficients of  $\beta$  as a polynomial over  $\mathbb{Z}[Y]$ . We can now simply rely on the results of the previous section to preprocess the reduced version of the polynomial  $f$  over  $\mathbb{Z}_r$  for fast evaluation.

The idea for  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  is similar. We first lift the evaluation to  $\mathbb{Z}[Y, Z]$  and then reduce it to an evaluation over  $\mathbb{Z}_{r'}[Y]/(Y^{D'} + 1)$  for an appropriate  $r', D'$ , which we then solve using the algorithm from the previous paragraph.

**Theorem A.5** (Theorem 2.1 restricted to  $R = \mathbb{Z}_q[Y]/(E(Y))$ ). *Let  $R = \mathbb{Z}_q[Y]/(E(Y))$  for some  $q \in \mathbb{N}$  and some arbitrary monic polynomial  $E$  with degree  $e > 0$ , and let  $f \in R[X_1, X_2, \dots, X_m]$  be a polynomial of individual degree  $< d$  in every variable. Then, there is some preprocessing algorithm that takes the coefficients of  $f$  as an input, runs in time*

$$S = d^m \cdot \text{poly}(m, d, \log |R|) \cdot O(m(\log m + \log d + \log \log |R|))^m$$

*and outputs a data structure of size at most  $S$ , and there is some evaluation algorithm with random access to the data structure, that is given an evaluation point  $\alpha \in R^m$  and computes  $f(\alpha)$  in time*

$$\text{poly}(d, m, \log |R|).$$

*Proof.* We can naturally lift elements  $a \in R = \mathbb{Z}_q[Y]/(E(Y))$  into elements  $\tilde{a} \in \mathbb{Z}[Y]$  where  $\tilde{a}(Y)$  is a polynomial of degree  $< e$  with coefficients in  $\{0, \dots, q - 1\}$ .

The following algorithm preprocesses the coefficients of a polynomial  $f$  into a data structure:

1. Let  $M := d^m(e(q - 1))^{(d-1)m+1} + 1$ ,  $D := (e - 1)((d - 1)m + 1)$  and let  $r := M^{D+1}$ .
2. Lift  $f \in R[X_1, \dots, X_m]$  to  $\tilde{f} \in \mathbb{Z}[Y][X_1, \dots, X_m]$  by lifting each coefficient of  $f$  from  $R$  to  $\mathbb{Z}[Y]$ .
3. Compute  $\bar{f} \in \mathbb{Z}_r[X_1, \dots, X_m]$  by reducing  $\tilde{f}$  modulo the ideal  $(r, Y - M)$ .<sup>28</sup>

<sup>28</sup>The value of  $r$  is chosen to be large enough that no “wrapping” over the modulus occurs, so computationally reducing modulo  $r$  does nothing; it is just formally necessary to apply Theorem A.4

4. Apply Theorem A.4 to compute the data structure  $T$  for  $\bar{f}$ .
5. The data structure consists of  $M, r, T$ .

Given any  $\alpha \in R^m$ , the algorithm to compute  $f(\alpha)$  is as follows:

1. Lift  $\alpha$  to  $\tilde{\alpha} \in (\mathbb{Z}[Y])^m$  by lifting each coordinate.
2. Compute  $\bar{\alpha} \in \mathbb{Z}_r^m$  from  $\tilde{\alpha}$  by reducing each coordinate modulo the ideal  $(r, Y - M)$ .
3. Use the data structure  $T$  to compute  $\bar{\beta} = \bar{f}(\bar{\alpha}) \in \mathbb{Z}_r$ , and lift to  $\tilde{\beta} \in \mathbb{Z}$ .
4. Let  $\tilde{\beta}_0, \dots, \tilde{\beta}_D \in \llbracket M \rrbracket$  be the digits of  $\tilde{\beta}$  written in base  $M$  so that  $\tilde{\beta} = \sum_{i=0}^D \tilde{\beta}_i M^i$ .
5. Construct the polynomial  $Q(Y) = \sum_{i=0}^D \tilde{\beta}_i Y^i \in \mathbb{Z}[Y]$ , and output  $Q(Y) \bmod (q, E(Y))$ .

We begin by showing correctness. For  $f \in R[X_1, \dots, X_m]$  and  $\alpha \in R^m$ , let  $\gamma = \tilde{f}(\tilde{\alpha}) \in \mathbb{Z}[Y]$  denote the evaluation of the lifted polynomial on the lifted input. Observe that  $\gamma(Y) = \sum_{i=0}^D \gamma_i Y^i$  is a polynomial in formal variable  $Y$  with non-negative coefficients  $\gamma_i \geq 0$  and degree  $\leq D = (e-1)((d-1)m+1)$ . This holds because each coordinate of  $\tilde{\alpha}$  and each coefficient of  $\tilde{f}$  is itself a polynomial in  $Y$  of degree  $\leq (e-1)$  with coefficients in  $\llbracket q \rrbracket$ . Also observe that we can bound the coefficients  $\gamma_i$  by

$$\gamma_i \leq \sum_{i=0}^D \gamma_i \leq \gamma(1) < \underbrace{d^m (e(q-1))^{(d-1)m+1}}_{=M} + 1.$$

The last inequality follows by substituting  $Y = 1$  in all coefficients of  $\tilde{f}$  and coordinates of  $\tilde{\alpha}$ , in which case each of them becomes an integer  $\leq e(q-1)$ ; each monomial term then evaluates to an integer  $\leq (e(q-1))^{(d-1)m+1}$ , and we sum  $d^m$  of them. Next note that reducing  $\gamma = \tilde{f}(\tilde{\alpha})$  modulo  $Y - M$  is equivalent to evaluating  $\gamma(M)$ . Also  $\gamma(M) < M^{D+1} = r$  so reducing  $\gamma(M)$  modulo  $r$  and then lifting the answer to  $\mathbb{Z}$  is the same as computing  $\gamma(M)$  in  $\mathbb{Z}$ . Therefore, by the correctness of Theorem A.4, the value  $\tilde{\beta} = \tilde{f}(\tilde{\alpha}) \bmod (r, Y - M)$  computed in step 3 of evaluation is just  $\gamma(M)$ . Furthermore, for the values  $\tilde{\beta}_i$  computed in step 4, we have:

$$\tilde{\beta} = \sum_{i=0}^D \tilde{\beta}_i M^i = \gamma(M) = \sum_{i=0}^D \gamma_i M^i$$

with  $\tilde{\beta}_i, \gamma_i \in \llbracket M \rrbracket$ , which implies  $\tilde{\beta}_i = \gamma_i$ . Therefore, for the polynomial  $Q$  computed in step 5, we have  $Q = \gamma = \tilde{f}(\tilde{\alpha}) \in \mathbb{Z}[Y]$  and hence  $Q(Y) \bmod (q, E(Y)) = f(\alpha)$ .

The efficiency of the data structure and evaluation algorithm follow almost immediately from the efficiency properties of Theorem A.4 applied to the ring  $\mathbb{Z}_r$ . This is because constructing and using the data structure for  $\bar{f}$  contributes the dominant term in the run time and data structure space. Thus the claimed efficiency holds since  $\log r = \text{poly}(m, d, \log |R|)$  and hence  $\log \log r = O(\log m + \log d + \log \log |R|)$ .  $\square$

Finally, we are ready to prove Theorem 2.1, restated again below.

**Theorem 2.1** (Preprocessing Polynomials [KU11]). *Let  $R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  for some  $q \in \mathbb{N}$  and arbitrary monic polynomials  $E_1$  over  $Y$  and  $E_2$  over  $Z$  with degrees  $e_1, e_2 > 0$  respectively. Let  $f \in R[X_1, X_2, \dots, X_m]$  be a polynomial of individual degree  $< d$  in every variable. Then, there is some preprocessing algorithm that takes the coefficients of  $f$  as an input, runs in time*

$$S = d^m \cdot \text{poly}(m, d, \log |R|) \cdot O(m(\log m + \log d + \log \log |R|))^m$$

*and outputs a data structure of size at most  $S$ , and there is some evaluation algorithm with random access to the data structure, that is given an evaluation point  $\alpha \in R^m$  and computes  $f(\alpha)$  in time*

$$\text{poly}(d, m, \log |R|).$$

*Proof.* The algorithms for computing the data structure and using it for evaluation are very similar to those from Theorem A.5. We use the same technique of lifting the evaluation to  $\mathbb{Z}[Y, Z]$  and then reducing it to an evaluation over  $\mathbb{Z}_r[Y]/(Y^{D_1+1} + 1)$  for an appropriate  $r$  and  $D_1$ , which we solve by applying Theorem A.5. We can naturally lift elements  $a \in R = \mathbb{Z}_q[Y, Z]/(E_1(Y), E_2(Z))$  into elements  $\tilde{a} \in \mathbb{Z}[Y, Z]$  where  $\tilde{a}(Y, Z)$  is a polynomial of degree  $< e_1$  in  $Y$  and degree  $< e_2$  in  $Z$  with coefficients in  $\{0, \dots, q-1\}$ .

The following algorithm preprocesses the coefficients of a polynomial  $f$  into a data structure:

1. Let  $M := d^m(e_1 e_2 (q-1))^{(d-1)m+1} + 1$ ,  $D_1 := (e_1 - 1)((d-1)m + 1)$ ,  $D_2 := (e_2 - 1)((d-1)m + 1)$  and let  $r := M^{D_2+1}$ . Also let  $R' = \mathbb{Z}_r[Y]/(Y^{D_1+1} + 1)$ .
2. Lift  $f \in R[X_1, \dots, X_m]$  to  $\tilde{f} \in \mathbb{Z}[Y, Z][X_1, \dots, X_m]$  by lifting each coefficient of  $f$  from  $R$  to  $\mathbb{Z}[Y, Z]$ .
3. Compute  $\bar{f} \in R'[X_1, \dots, X_m]$  by reducing  $\tilde{f}$  modulo the ideal  $(r, Z - M, Y^{D_1+1} + 1)$ .<sup>29</sup>
4. Apply Theorem A.5 to compute the data structure  $T$  for  $\bar{f}$ .
5. The data structure consists of  $M, r, T$

Given any  $\alpha \in R^m$ , the algorithm to compute  $f(\alpha)$  is as follows:

1. Lift  $\alpha$  to  $\tilde{\alpha} \in (\mathbb{Z}[Y, Z])^m$  by lifting each coordinate.
2. Compute  $\bar{\alpha} \in (R')^m$  from  $\tilde{\alpha}$  by reducing modulo the ideal  $(r, Z - M, Y^{D_1+1} + 1)$ .
3. Use the data structure  $T$  to compute  $\bar{\beta} = \bar{f}(\bar{\alpha}) \in R'$ , and lift to  $\tilde{\beta} = \sum_{i=0}^{D_1} c_i Y^i \in \mathbb{Z}[Y]$ .
4. For each  $i \in \llbracket D_1 \rrbracket$ , let  $c_{i,0}, \dots, c_{i,D_2} \in \llbracket M \rrbracket$  be the digits of  $c_i$  written in base  $M$  so that  $c_i = \sum_{j=0}^{D_2} c_{i,j} M^j$ .
5. Construct the polynomial  $Q(Y, Z) = \sum_{i=0}^{D_1} \sum_{j=0}^{D_2} c_{i,j} Y^i Z^j \in \mathbb{Z}[Y, Z]$ , and output  $Q(Y, Z) \bmod (q, E_1(Y), E_2(Z))$ .

---

<sup>29</sup>The value of  $r, D_1$  are chosen to be large enough that no “wrapping” occurs and the reduction does not change anything; it is just formally necessary to apply Theorem A.5.

We begin by showing correctness. For  $f \in R[X_1, \dots, X_m]$  and  $\alpha \in R^m$ , let  $\gamma = \tilde{f}(\tilde{\alpha}) \in \mathbb{Z}[Y, Z]$  denote the evaluation of the lifted polynomial on the lifted input. Observe that  $\gamma(Y, Z) = \sum_{i=0}^{D_1} \sum_{j=0}^{D_2} \gamma_{i,j} Y^i Z^j$  is a polynomial in formal variables  $Y, Z$  with non-negative coefficients  $\gamma_{i,j} \geq 0$ , degree  $\leq D_1 = (e_1 - 1)((d - 1)m + 1)$  in  $Y$  and degree  $\leq D_2 = (e_2 - 1)((d - 1)m + 1)$  in  $Z$ . This holds because each coordinate of  $\tilde{\alpha}$  and each coefficient of  $\tilde{f}$  is itself a polynomial of degree  $\leq (e_1 - 1)$  in  $Y$  and degree  $\leq (e_2 - 1)$  in  $Z$  with coefficients in  $\llbracket q \rrbracket$ . Also observe that we can bound the coefficients  $\gamma_{i,j}$  by

$$\gamma_{i,j} \leq \sum_{i=0}^{D_1} \sum_{j=0}^{D_2} \gamma_{i,j} \leq \gamma(1, 1) < \underbrace{d^m (e_1 e_2 (q - 1))^{(d-1)m+1}}_{=M} + 1.$$

The last inequality follows by substituting  $Y = 1, Z = 1$  in all coefficients of  $\tilde{f}$  and coordinates of  $\tilde{\alpha}$ , in which case each of them becomes an integer  $\leq e_1 e_2 (q - 1)$ ; each monomial term then evaluates to an integer  $\leq (e_1 e_2 (q - 1))^{(d-1)m+1}$ , and we sum  $d^m$  of them. Next note that reducing  $\gamma = \tilde{f}(\tilde{\alpha})$  modulo  $Z - M$  is equivalent to computing  $\gamma_{(Z=M)}(Y) = \sum_{i=0}^{D_1} (\sum_{j=0}^{D_2} \gamma_{i,j} M^j) Y^i \in \mathbb{Z}[Y]$ . Also  $\sum_{j=0}^{D_2} \gamma_{i,j} M^j < M^{D_2+1} = r$  so reducing  $\gamma_{(Z=M)}(Y)$  modulo  $(r, Y^{D_1+1} + 1)$  and then lifting the answer to  $\mathbb{Z}[Y]$  is the same as computing  $\gamma_{(Z=M)}(Y)$  in  $\mathbb{Z}[Y]$ . Therefore, by the correctness of Theorem A.5, the value  $\tilde{\beta} = \tilde{f}(\tilde{\alpha}) \bmod (r, Z - M, Y^{D_1+1} + 1)$  computed in step 3 of evaluation is just  $\gamma_{(Z=M)}(Y)$ . Furthermore, for the values  $c_{i,j}$  computed in step 4, we have:

$$\tilde{\beta} = \sum_{i=0}^{D_1} \left( \sum_{j=0}^{D_2} c_{i,j} M^j \right) Y^i = \gamma_{(Z=M)}(Y) = \sum_{i=0}^{D_1} \left( \sum_{j=0}^{D_2} \gamma_{i,j} M^j \right) Y^i$$

with  $c_{i,j}, \gamma_{i,j} \in \llbracket M \rrbracket$ , which implies  $c_{i,j} = \gamma_{i,j}$ . Therefore, for the polynomial  $Q$  computed in step 5, we have  $Q = \gamma = \tilde{f}(\tilde{\alpha}) \in \mathbb{Z}[Y, Z]$  and hence  $Q \bmod (q, E_1(Y), E_2(Z)) = f(\alpha)$ .

Notice that the invocations of Theorem A.5 comprise the dominant terms in the run time and space of the above algorithms and data structure. Thus the claimed efficiency properties follow immediately from Theorem A.5 by observing that  $\log |R'| = \text{poly}(d, m, \log |R|)$  and hence  $\log \log |R'| = O(\log m + \log d + \log \log |R|)$ .  $\square$

**Remark A.1** (Extension rings over more variables). Following very similar techniques to those in the proof of Theorem 2.1, we can actually extend the result to handle evaluating polynomials over extension rings of  $t$  variables  $R = \mathbb{Z}_q[Y_1, \dots, Y_t] / (E_1(Y_1), \dots, E_t(Y_t))$  for arbitrary  $q$  and arbitrary non-constant monic polynomials  $E_i$ , as long as  $t = O(1)$ . That is, given a ring  $R$  of the above form and a polynomial  $f \in R[X_1, \dots, X_m]$ , we can reduce the problem of preprocessing/evaluating the polynomial over  $R$  to the case of preprocessing/evaluating the polynomial over the ring  $R' = \mathbb{Z}_{r'}[Y_1, \dots, Y_{t-1}] / (E'_1(Y_1), \dots, E'_{t-1}(Y_{t-1}))$  for some  $r_1 > q$  and polynomials  $E'_1, \dots, E'_{t-1}$  of greater degree, as in the proof of Theorem 2.1. This is done by (partially) evaluating at  $Y_t = M_t$  for some large enough  $M_t \in \mathbb{N}$ , and reducing modulo  $r'$  and the polynomials  $E'_1, \dots, E'_{t-1}$  which are chosen large enough to avoid any “wrapping” over the modulus. Repeating this process iteratively, we ultimately reduce to invoking Theorem A.4 for the ring  $\mathbb{Z}_{r^*}$  for some  $r^* \gg q$ . Because  $t$  is constant, the size of the final ring satisfies  $\log r^* = \text{poly}(d, m, \log |R|)$  so we achieve the same efficiency properties as in Theorem 2.1.

## B Multivariate Polynomial Interpolation

Lemma 2.2 is restated and proved below.

**Lemma B.1** (Multi-variate polynomial interpolation). *Let  $\mathbb{F}_d$  be a field of prime order  $d$ , and let  $m \in \mathbb{N}$  an integer. Let  $\{y_{(x_1, \dots, x_m)} \in \mathbb{F}_d\}_{(x_1, \dots, x_m) \in \mathbb{F}_d^m}$  be any set of  $d^m$  values. Then there is an algorithm that runs in quasi-linear time  $O(d^m \cdot m \cdot \text{poly log } d)$  and recovers the coefficients of a polynomial  $f(X_1, \dots, X_m) \in \mathbb{F}_d[X_1, \dots, X_m]$  with individual degree  $< d$  in each variable such that  $f(x_1, \dots, x_m) = y_{(x_1, \dots, x_m)}$  for all  $(x_1, \dots, x_m) \in \mathbb{F}_d^m$ .*

*Proof.* We will use the fast interpolation algorithm for univariate polynomials over  $\mathbb{F}_d$ .

**Fact B.2** (Univariate polynomial interpolation [GG13, Corollary 10.12]). *Let  $\mathbb{F}_d$  be a prime field of order  $d$ . Then, there is an interpolation algorithm such that takes as input any distinct  $a_1, \dots, a_n \in \mathbb{F}_d$  and any  $b_1, \dots, b_n \in \mathbb{F}_d$  for  $n \leq d$  outputs the  $n$  coefficients of a univariate polynomial  $g$  of degree  $< n$ , such that  $g(a_i) = b_i$  for all  $i \in [n]$  in time  $n \cdot \text{poly log}(n, d)$ .*

We then use the following algorithm to interpolate  $f$  given values  $\{y_{(x_1, \dots, x_m)} \in \mathbb{F}_d\}_{(x_1, \dots, x_m) \in \mathbb{F}_d^m}$ :

1. Base case: if  $m = 1$ , run the univariate polynomial interpolation (Fact B.2), to recover the coefficients of the univariate polynomial  $f(X)$  of degree  $< d$  such that  $f(x_1) = y_{x_1}$  for all  $x_1 \in \mathbb{F}_d$ .
2. Otherwise, if  $m > 1$ :
  - (a) For each  $(x_1, \dots, x_{m-1}) \in \mathbb{F}_d^{m-1}$ :

Run the univariate polynomial interpolation (Fact B.2) to recover the coefficients of a univariate polynomial  $g_{x_1, \dots, x_{m-1}}(X) = \sum_{i=0}^{d-1} c_{x_1, \dots, x_{m-1}, i} X^i$  of degree  $< d$  such that  $g_{x_1, \dots, x_{m-1}}(x_m) = y_{(x_1, \dots, x_m)}$  for all  $x_m \in \mathbb{F}_d$
  - (b) For each  $i \in [d]$ :

Recursively call this algorithm with the input  $(c_{x_1, \dots, x_{m-1}, i})_{(x_1, \dots, x_{m-1}) \in \mathbb{F}_d^{m-1}}$  to interpolate a polynomial  $f_i(X_1, \dots, X_{m-1})$  of individual degree  $< d$  such that  $f_i(x_1, \dots, x_{m-1}) = c_{x_1, \dots, x_{m-1}, i}$  for all  $(x_1, \dots, x_{m-1}) \in \mathbb{F}_d^{m-1}$ .
  - (c) Output  $f$  as

$$f(X_1, \dots, X_m) = \sum_{i \in [d]} f_i(X_1, \dots, X_{m-1}) \cdot X_m^i.$$

This is just concatenating the coefficients of  $f_i$  for all  $i$ .

**Analysis.** The correctness is argued by induction on the number of variables. For the base case  $m = 1$ , the output is correct by Fact B.2. Suppose that correctness holds for  $(m - 1)$ -variate polynomials. Then, for any  $(x_1, \dots, x_m) \in \mathbb{F}_d^m$ , we have:

$$f(x_1, \dots, x_m) = \sum_{i \in [d]} f_i(x_1, \dots, x_{m-1}) \cdot x_m^i = \sum_{i \in [d]} c_{x_1, \dots, x_{m-1}, i} \cdot x_m^i = g_{x_1, \dots, x_{m-1}}(x_m) = y_{x_1, \dots, x_m}$$

where the 1st equality follows by Step 2c, the 2nd equality follows by the induction hypothesis for  $m - 1$  variables in Step 2b, the 3rd and 4th equality's follow by the definition of  $g_{x_1, \dots, x_{m-1}}$  and then Fact B.2 in Step 2a. Therefore, correctness holds for  $m$  variables.

To see the efficiency, let  $T(m)$  be the running time for  $m$ -variate polynomials. We have  $T(1) = d \cdot \text{poly log } d$  by Fact B.2. For  $m \geq 1$ , we have

$$T(m) = \begin{cases} d \cdot \text{poly log } d & m = 1 \\ d^m \cdot \text{poly log } d + d \cdot T(m-1) & m > 1, \end{cases}$$

where Step 2a is repeated for  $d^{m-1}$  times for a cost of  $d^{m-1} \cdot T(1) = d^m \cdot \text{poly log } d$ , Step 2b is repeated  $d$  times for a cost of  $d \cdot T(m-1)$ , and Step 2c is just concatenation of coefficients, whose run-time can certainly be bounded by  $d^m \cdot \text{poly log } d$ . Solving the recursion, we get  $T(m) = d^m \cdot m \cdot \text{poly log } d$  as claimed.  $\square$

## C Constructions from Other Assumptions

In this section we sketch out a very simple construction of ASHE from the Approximate GCD assumption. We also sketch out an extension of our main construction from RingLWE to the more general Module LWE assumption with constant rank. In both cases, the resulting ASHE constructions automatically yield DEPIR from these assumptions with the same asymptotic parameters as our main construction from RingLWE. The constructions can also be adapted to achieve ASHE-FHE and therefore also yield RAM-FHE schemes under these assumptions plus a corresponding circular security assumption.

### C.1 DEPIR from Approximate GCD

We can construct a very simple ASHE scheme (Definition 3.1) over the ring  $\mathbb{Z}_q$  for some  $q$ , using the SHE/FHE of van Dijk, Gentry, Halevi and Vaikuntanathan [vGHV10], whose security relies on the approximate GCD problem defined there. We can plug this ASHE to get a DEPIR scheme using our construction in Section 4.2.

**Construction.** We lightly modify the most basic symmetric-key somewhat homomorphic encryption scheme of [vGHV10] to handle a large plaintext space  $\mathbb{F}_d$  instead of  $\mathbb{Z}_2$ . Throughout this section, we represent values in  $\mathbb{Z}_d$  as integers in the range  $(-d/2, d/2]$  which allows us to lift values from  $\mathbb{Z}_d$  to  $\mathbb{Z}_q$  and vice versa.

- $\text{params} := \text{Setup}(1^\lambda, 1^d, 1^D, N)$ : Choose parameters  $\eta, \gamma, \rho, q$  as described below and let  $R = \mathbb{Z}_q$  be the ring of the ASHE scheme.
- $s \leftarrow \text{Gen}(\text{params})$ : Choose  $s \leftarrow [2^{\eta-1}, 2^\eta) \cap (d\mathbb{Z} + 1)$  to be a random  $\eta$ -bit integer which is 1 mod  $d$  and output  $s$  as the secret key.
- $\text{ct} \leftarrow \text{Enc}(s, \mu)$ : Choose random integers  $a \leftarrow [0, 2^\gamma/s)$  and  $e \leftarrow (-2^\rho, 2^\rho)$ . Output  $a \cdot s + d \cdot e + \mu$ .
- $\mu := \text{Dec}(s, \text{ct})$ : To decrypt, output  $(\text{ct} \bmod s) \bmod d$ .
- $\bar{\mu} := \text{Lift}(\mu)$ : Interpret  $\mu \in \mathbb{Z}_d$  as an element of  $\mathbb{Z}_q$ .

We choose the parameter  $\eta, \gamma, \rho, q$  so that  $2^\eta \geq 42N \cdot d \cdot (d(2^\rho + 1))^D + 1$  and the approximate GCD assumption holds with  $(\eta, \gamma, \rho)$ ; See [vGHV10]. For example, we can set  $\rho = \lambda$ ,  $\eta$  to be the smallest value subject to the above, and  $\gamma = \eta^2 \lambda$ . Set  $q = 2 \cdot N \cdot d \cdot (2^\gamma + d \cdot (2^\rho + 1))^D + 1$ .

**Analysis.** To argue correctness, note that fresh ciphertexts outputted by the encryption algorithm are integers of size  $c \leq 2^\gamma + d \cdot (2^\rho + 1)$ . Let  $f(X_1, \dots, X_m)$  be any polynomial of total degree  $< D$  with  $N$  terms over  $\mathbb{Z}_d$ . Then the output of lifting  $f$  to the integers and evaluating it over fresh ciphertexts  $\text{ct}_i$  is an integer of norm at most  $|f(\text{ct}_1, \dots, \text{ct}_m)| \leq N \cdot d \cdot c^D < q/2$ . Therefore, there is no difference between evaluating the polynomial over  $\mathbb{Z}_q$  versus the integers and so we can just assume we do the latter. Correctness holds since, if  $\text{ct}_i = a_i \cdot s + d \cdot e_i + \mu_i$  then:

$$f(\text{ct}_1, \dots, \text{ct}_m) = f(de_1 + \mu_1, \dots, de_m + \mu_m) \pmod{s}$$

Furthermore, since  $|de_i + \mu_i| \leq d(2^\rho + 1)$  we have  $|f(de_1 + \mu_1, \dots, de_m + \mu_m)| \leq N \cdot d \cdot (d(2^\rho + 1))^D \leq s/2$  and therefore  $f(de_1 + \mu_1, \dots, de_m + \mu_m)$  is the same mod  $s$  as over the integers. Finally  $f(de_1 + \mu_1, \dots, de_m + \mu_m) = f(\mu_1, \dots, \mu_m) \pmod{d}$ .

For efficiency, note that  $\log q = \text{poly}(\lambda, D, \log d, \log N)$ .

Security follows directly from the approximate GCD assumption as defined in [vGHV10] with parameter  $(\eta, \gamma, \rho)$ .

**Extension to ASHE-FHE and RAM-FHE.** Using the ideas of [vGHV10, CS15], we can also extend the above ASHE scheme to an ASHE-FHE. The first work does this via “decryption squashing” at the cost of needing additional assumptions as discussed there, while the latter work shows how to get rid of it. In both cases, we need circular security if we want to go beyond a leveled scheme. We can plug this ASHE-FHE into our constructions to get a RAM-FHE from approximate GCD.<sup>30</sup>

## C.2 DEPIR from Module LWE

Our DEPIR and RAM-FHE schemes can also be based on the *module LWE* assumption [BGV12, LS15, BJRW20], which is a generalization of RingLWE, as long as the *rank* of the module is constant. Below, we give a construction of ASHE from module LWE, which is very similar to the construction from RingLWE. We can then directly plug this ASHE into our DEPIR construction to get a DEPIR from module LWE.

**Module LWE Assumption.** The problem of module LWE generalizes RingLWE to a rank  $\kappa > 1$ . Recall that the RingLWE problem is parameterized by  $n = n(\lambda) \in \mathbb{N}$ ,  $q = q(\lambda) \in \mathbb{N}$ , and an error distribution  $\chi = \chi(\lambda)$  over the ring  $Q = \mathbb{Z}_q[Z]/(Z^n + 1)$ , where  $\lambda$  is the security parameter (Definition 2.3). The module LWE augments RingLWE with an additional parameter  $\kappa = \kappa(\lambda) \in \mathbb{N}$ , also called the rank.

The  $(n, q, \chi, \kappa)$ -module LWE assumption states that for any  $\ell = \text{poly}(\lambda)$ , it holds that

$$\{(\vec{a}_i, \vec{a}_i \cdot \vec{s} + e_i)\}_{i \in [\ell]} \approx_c \{(\vec{a}_i, u_i)\}_{i \in [\ell]},$$

where  $\vec{s} \leftarrow Q^\kappa$ ,  $\vec{a}_i \leftarrow Q^\kappa$ ,  $e_i \leftarrow \chi$ ,  $u_i \leftarrow Q$ , and  $\vec{a}_i \cdot \vec{s}$  denotes the inner product (over vectors of ring elements). Note that rank-1 module LWE is identical to RingLWE. Similar to RingLWE, it is equivalent to the scaled error variant: instead of adding the errors  $e_i \leftarrow \chi$ , we add  $d \cdot e_i$  for some integer  $d$  relatively prime to  $q$ .

<sup>30</sup>The work of [CS15] tantalizingly shows that approximate GCD with certain parameters that are good enough for FHE follows from just the LWE assumption. Unfortunately, such parameters do not appear to be sufficient for ASHE.

**Construction of ASHE based on Constant-Rank Module LWE.** Recall that an ASHE is parameterized by the security parameter  $\lambda$ , the plaintext space  $\mathbb{Z}_d$  for prime  $d$ , the total degree  $< D$ , and the number of terms  $N$ . The construction of ASHE from module LWE of constant rank  $\kappa$  is very similar to the construction from RingLWE (Section 3.1). The main difference is that the ciphertexts are now  $\kappa$ -variate polynomials over  $Q$ , where homomorphic addition/multiplication just corresponds to adding/multiplying the ciphertext polynomials as before. Therefore, the ring for the ASHE becomes  $R = Q[Y_1, \dots, Y_\kappa]/((Y_i^D + 1)_i)$ .

params := Setup( $1^\lambda, 1^d, 1^D, N$ ): Set the gap parameter  $t := D \log d + \log N + \log d + 1$  and choose  $n = \text{poly}(\lambda, t)$ ,  $q = \lambda^{\text{poly}(t)}$  and a  $\beta$ -bounded error distribution  $\chi$  as in Section 2.2 so that  $q > (2\beta n)^t > 2Nd(d(\beta + 1)n)^D$  and  $q$  is relatively prime to  $d$ . Define the rings

$$Q := \mathbb{Z}_q[Z]/(Z^n + 1)$$

$$R := Q[Y_1, \dots, Y_\kappa]/(Y_1^D + 1, \dots, Y_\kappa^D + 1) \cong \mathbb{Z}_q[Y_1, \dots, Y_\kappa, Z]/(Z^n + 1, Y_1^D + 1, \dots, Y_\kappa^D + 1)$$

We let  $\vec{Y}$  denote the vector of symbolic variables  $(Y_1, \dots, Y_\kappa)$ .

$s \leftarrow \text{Gen}(1^\lambda)$ : Sample  $\vec{s} \leftarrow Q^\kappa$  uniformly at random.

$\text{ct} \leftarrow \text{Enc}(\vec{s}, \mu)$ : Reinterpret  $\mu \in \mathbb{Z}_d$  as an element of  $Q$ . Sample  $\vec{a} \leftarrow Q^\kappa, e \leftarrow \chi$ . Let

$$b = \vec{a} \cdot \vec{s} + d \cdot e + \mu \in Q.$$

Define  $\text{ct} \in R$  as the linear polynomial with formal variables  $\vec{Y} = (Y_1, \dots, Y_\kappa)$  via:

$$\text{ct}(\vec{Y}) = -\vec{a} \cdot \vec{Y} + b.$$

$\mu := \text{Dec}(\vec{s}, \text{ct})$ : Interpret  $\text{ct} \in R$  as a formal polynomial  $\text{ct}(\vec{Y}) \in R$  and compute  $g = \text{ct}(\vec{s}) \in Q$  to be its evaluation on  $\vec{s} \in Q^\kappa$ . Interpret  $g \in Q$  as a formal polynomial  $g(Z) \in \mathbb{Z}_q[Z]/(Z^n + 1)$  and let  $h = g(0) \in \mathbb{Z}_q$  be its constant term. Reinterpret  $h$  as an element of  $\mathbb{Z}_d$  and output it.

$\bar{\mu} := \text{Lift}(\mu)$ : Reinterpret  $\mu \in \mathbb{Z}_d$  as an element of  $R$ .

**Analysis.** The analysis is very similar to the ASHE based on RingLWE (Theorem 3.2). We begin with *correctness*. Notice that any freshly encrypted ciphertexts  $\text{ct}_i$  are  $\kappa$ -variate polynomials of total degree 1 over  $Q$ . Thus, if  $f$  has total degree  $< D$ , then  $\text{ct}' = f(\text{ct}_1, \dots, \text{ct}_m)$  results in a ciphertext  $\text{ct}'$  that is a  $\kappa$ -variate polynomial of total degree  $< D$  over  $Q$ , and therefore ciphertext is in  $R$ . It remains to argue the noisiness of ciphertexts. Recall that at a high-level, we defined the noisiness of any ciphertext, and then we argued that the homomorphic evaluation results in ciphertexts that is  $< q/2$  noisy. Here the argument is almost identical, and the only difference is that the ciphertexts are now  $\kappa$ -variate polynomials  $\text{ct}(\vec{Y})$  (over  $Q$ ). Hence, we say a ciphertext  $\text{ct}(\vec{Y})$  is  $\gamma$  noisy if  $\|\text{ct}(\vec{s})\| \leq \gamma$ . With such modification, the remaining correctness analysis is identical to the RingLWE case.

*Security* follows directly from the scaled error variant of module LWE assumption.

*Efficiency:* We determine the parameters  $n, q, \chi$  as per RingLWE, which gives  $q = \lambda^{\text{poly}(D, \log d, \log N)}$ ,  $n = \text{poly}(\lambda, D, \log d, \log N)$ . The elements in the ring  $R$  has individual degree of each variable  $Y_j$  bounded by  $< D$ . Hence, the description length of a ring element is  $O(n \cdot D^\kappa \log q) = \text{poly}(\lambda, D^\kappa, \log d, \log N)$ . Plugging in any constant  $\kappa \in \mathbb{N}$ , the efficiency is still bounded by  $\text{poly}(\lambda, D, \log d, \log N)$ .

**Plugging the ASHE into DEPIR.** The above gives us an ASHE with the ring

$$R = \mathbb{Z}_q[Z, Y_1, \dots, Y_\kappa] / (Z^n + 1, (Y_i^D + 1)_{i=1, \dots, \kappa}).$$

While the default definition of ASHE only considered more restricted rings, we mentioned in Footnote 14, that we only need the choice of the ring in the ASHE to match the types of rings for which we have fast polynomial evaluation with preprocessing with the parameters of Theorem 2.1. As sketched in Remark A.1, this is the case of the ring  $R$ . Therefore, we can use this ASHE in the construction of DEPIR and achieve the same parameters as Theorem 4.4.

**RAM-FHE from Module LWE.** We can also construct ASHE-FHE from module LWE with a constant rank, by analogously adapting the construction in Claim 6.1.1 based on the BGV FHE scheme [BGV12]. In fact, the BGV scheme is already presented based on module LWE, and therefore we only need to make the same analogous modifications as in the Claim. We can then plug in this ASHE-FHE into our constructions of RAM-FHE to get RAM-FHE from constant-rank module LWE with the same parameters as Theorem 7.7.