Laconic Function Evaluation, Functional Encryption and Obfuscation for RAMs with Sublinear Computation

Fangqi Dong[*]
IIIS, Tsinghua University

Zihan Hao[†]
IIIS, Tsinghua University

Ethan Mook[‡]
Northeastern University

Daniel Wichs[§]
Northeastern University and NTT Research

June 5, 2024

Abstract

Laconic function evaluation (LFE) is a "flipped" version of fully homomorphic encryption, where the server performing the computation gets the output. The server commits itself to a function f by outputting a small digest. Clients can later efficiently encrypt inputs x with respect to the digest in much less time than computing f, and ensure that the server only decrypts $f(x)$, but does not learn anything else about x. Prior works constructed LFE for *circuits* under LWE, and for *Turing Machines (TMs)* from indistinguishability obfuscation (iO). In this work we introduce LFE for *Random-Access Machines* (RAM-LFE). The server commits itself to a potentially huge database y via a short digest. Clients can later efficiently encrypt inputs x with respect to the digest and the server decrypts $f(x, y)$ for some specified RAM program f (e.g., a universal RAM), without learning anything else about x. The main advantage of RAM-LFE is that the server's decryption run-time only scales with the RAM run-time T of the computation $f(x, y)$, which can be sublinear in both $|x|$ and $|y|$. We consider a *weakly efficient* variant, where the client's run-time is also allowed to scale linearly with T, but not $|y|$, and a *fully efficient* variant, where the client's run-time must be sublinear in both T and $|y|$. We construct the former from doubly efficient private information retrieval (DEPIR) and laconic OT (LOT), both of which are known from RingLWE, and the latter from an additional use of iO. We then show how to leverage fully efficient RAM-LFE to also get (many-key) *functional encryption for RAMs (RAM-FE)* where secret keys are associate with big databases y and the decryption time is sublinear in $|y|$, as well as *iO for RAMs* where the obfuscated program contains a big database y and the evaluation time is sublinear in $|y|$.

[*]`dfq20@mails.tsinghua.edu.cn`

[†]`haozh20@mails.tsinghua.edu.cn`

[‡]`mook.e@northeastern.edu`

[§]`wichs@ccs.neu.edu` Research supported by NSF grant CNS-1750795, CNS-2055510 and the JP Morgan Faculty Research Award.

Contents

1 Introduction

Laconic Function Evaluation. Laconic function evaluation (LFE), introduced by Quach, Wee and Wichs [QWW18], can be seen as a "flipped" version of fully homomorphic encryption (FHE). Consider a powerful server who has some function f and a weak client who holds an input x, but does not have the computational power to run f. They want to run a 2-round protocol to compute $f(x)$ without revealing x to the server. FHE provides a solution where the client learns $f(x)$, while LFE provides one where the server learns $f(x)$.

In more detail, in an LFE scheme, the server computes a short digest dig_f that commits it to the function f. The client encrypts an input x using the digest dig_f, and the server decrypts the ciphertext to recover $f(x)$. Security ensures that the server does not learn anything about x beyond $f(x)$. This is formalized by requiring the ciphertext to be simulatable given only $f(x)$. Efficiency requires that the encryption time is smaller than simply computing f.[1]

For example, consider a scenario where the FBI has a database of most-wanted suspects. There are many security cameras in public spaces, and we want to allow the FBI to learn when one of the suspects passes in front of one of the cameras, but not to learn anything else about what the cameras are observing. LFE gives a simple non-interactive solution to this problem. The FBI publishes a digest dig_f for the function f that contains a hard-coded database of suspects and, on input an image x, it outputs whether x has a match in the database. The cameras have the digest dig_f, periodically capture images x of their surroundings, encrypt them using this digest, and send the ciphertexts to the FBI.

LFE has many applications, such as MPC where the online run-time of the parties is sublinear in the complexity of the function being computed. Furthermore, as shown in [QWW18], it implies 1-key succinct functional encryption, which in turn implies succinct reusable garbled circuits [GKP+13] as a special case.

Prior Work on LFE. The prior work of [QWW18] constructed *leveled LFE for circuits* under the LWE assumption. In particular, if the function f is represented by a boolean circuit of size $|f|$ and depth d, then the server's run-time (to generate the digest and to decrypt) is $|f| \cdot \mathrm{poly}(d)$, the size of the digest is $O(1)$, and the client's run-time (to encrypt) is $|x| \cdot \mathrm{poly}(d)$. A recent work of [DGM23] constructed LFE for Turing Machines assuming indistinguishability obfuscation and somewhere statistically binding hash functions. The main advantage is that computing the digest is more efficient than computing the function. In particular, for a function f represented as a Turing Machine of size $|f|$ and run-time T, the server's run-time to generate a digest is $O(|f|)$, the size of the digest is $O(1)$, the client's run-time (to encrypt) is $O(|x|)$, and the server's run-time to decrypt is $O(T)$.[2]

[1]It is known that LFE schemes require a common reference/random string (CRS); for simplicity we largely omit it from the discussion in the introduction. Also, by default we require that the digest is derived deterministically given f, which is a crucial feature for some applications. In this case, the digest cannot fully hide f, and we do not require any "function hiding" security. However, [QWW18] showed a generic transformation to achieve function hiding at the cost of having a randomized procedure to generate the digest. The security guarantee implicitly assumes a semi-honest server that computes dig_f correctly. If the function f is public, anyone can audit the digest by re-computing it to check that it is correct, or the server can provide a SNARG (for P) that it was computed correctly. If f is secret, the server can still provide a SNARK (for NP) that f belongs to some restricted class of functions deemed safe.

[2]Throughout the introduction we omit fixed polynomial factors in the security parameter or polyogarithmic terms. We also restrict to boolean functions with 1-bit output.

RAM Computation and RAM-FHE. A major limitation, affecting prior LFE and FHE constructions, is that the server's run-time scales with the circuit size of the computation, which may potentially be much larger than the natural run-time of the computation in the random-access machine (RAM) model. In particular, consider a function $f_y(x) = f(x, y)$ that depends on some large database y held by the server. Then the server's run-time in FHE/LFE for f_y will be at least linear in y (and in x), even if the function can be computed in vastly smaller sublinear time in the RAM model. For example, in the FBI scenario mentioned above, the FBI's computation to decrypt each LFE ciphertext would be linear in the database of suspects, even though more efficient sublinear time algorithms for image comparison are possible [Gra10]. The recent work of Lin, Mook and Wichs [LMW22] (see also [HHWW19]) showed how to get around this limitation in the context of FHE, by constructing a RAM-FHE scheme where the server preprocesses the database y once ahead of time, but can then homomorphically evaluate $f(x, y)$ over various encrypted inputs x in time that only scales with the (worst-case) RAM run-time of f. The main tool in their construction is *doubly efficient private information retrieval* (DEPIR), which allows the server to deterministically preprocess a database y into some data structure $\widetilde{y}$ and later run (2-round) PIR protocols with various clients, where both the server and client run-times are only polylogarithmic in the database size. Constructions of both RAM-FHE and DEPIR were given under the RingLWE assumption.

RAM-LFE and Our Results. In this work we extend LFE to the RAM setting, and define the new notion of RAM-LFE, analogously to RAM-FHE. In a RAM-LFE scheme, the server holds some large database y and produces a short digest dig_y that commits it to y, along with a large data-structure $\widetilde{y}$ that it holds locally and uses during decryption. The client encrypts x using the digest dig_y and the server decrypts the ciphertext using the data-structure $\widetilde{y}$ to recover $f(x, y)$, where f is some fixed RAM program (e.g., a universal RAM) with boolean output.[3] As with standard LFE, security ensures that the ciphertext hides everything about x beyond the output $f(x, y)$, as formalized via the simulation paradigm.

For efficiency, we assume the client knows some upper-bound on the RAM run-time T of the computation at encryption time, and we denote the actual RAM run-time of $f(x, y)$ by $t \leq T$. We consider two notions of efficiency that we call *weak efficiency* and *full efficiency*. Roughly speaking weak efficiency allows the client's run time to scale with T but not with $|y|$, while full efficiency doesn't allow it to scale with either T or $|y|$. Our results are as follows:

- *Weak Efficiency:* Assuming DEPIR and laconic OT [CDG+17] (both of which follow from RingLWE), we construct a RAM-LFE scheme where, for any constant $\varepsilon > 0$:
  - The size of the common random string is $O(1)$.
  - The server's run-time to generate $\mathsf{dig}_y, \widetilde{y}$ is $O(|y|^{1+\varepsilon})$. The size of dig_y is $O(1)$ and the size of $\widetilde{y}$ is $O(|y|^{1+\varepsilon})$.
  - The client's encryption run-time and the ciphertext size is $O(|x| + T)$.
  - The server's decryption run-time given random-access to $\widetilde{y}$ and to the ciphertext is just $O(t)$.

[3] We rely on the above formulation for simplicity in the intro, but it is equivalent to formulations where the client or the server chooses the program f since we can always embed the code of the actual program to be executed inside either x or y. Our technical definition allows the client to choose the program, but this is just for notational convenience. We also require the procedure that maps y to $(\mathsf{dig}_y, \widetilde{y})$ to be deterministic and do not require dig_y to fully hide y. However, we can generically apply the transformation of [QWW18] to fully hide y at the cost of having a randomized procedure.

- *Full Efficiency:* Assuming DEPIR, laconic OT, and iO, we construct a RAM-LFE scheme with the same parameters as above, except that the client's encryption run-time and the ciphertext size is just $O(|x|)$.

We remark that we assume the RAM program f can have random-access to both x and y as well as to any additional read/write random access memory. In particular, T, t can both be sublinear in $|x|$ and $|y|$. Note that, in the full efficiency setting, we can remove the restriction that the client knows an upper bound on the computation run-time T at encryption time, and instead have the client try all possible powers of 2: $T = 2, 4, \dots, 2^{\omega(\log \lambda)}$ up to some super-polynomial upper bound, and the server tries each one in order until the first that completes. However, this comes at the cost of needing to assume slightly super-polynomial security level of the underlying assumptions since our reduction has a security loss of $O(T)$. Also, the above results hold for boolean functions f that output 1 bit. We can generically extend this to arbitrary output size m by incurring a multiplicative factor in m for the encryption time and the decryption time. (We also sketch at how to optimize this to only incur an additive factor in m with some additional effort.)

Remarks on RAM-LFE. We note that RAM-LFE implies 3-round DEPIR (in the CRS model), and therefore it is not surprising that we need to rely on DEPIR as a building block.[4]

RAM-LFE with weak efficiency is conceptually similar to garbled RAM [LO13, GHL+14]. In a garbled RAM scheme, the client garbles a database y and sends the garbled version $\widetilde{y}$ to a server. Later the client garbles some program $f(x, \cdot)$ and sends the garbled program to the server who can evaluate the garbled program over the garbled database to recover $f(x, y)$, but cannot learn anything else about x, y. In both garbled RAM and RAM-LFE with weak efficiency, the client's run time to garble $f(x, \cdot)$ (resp. encrypt x) and the server's run-time to evaluate the garbled program (resp. decrypt) scale with the RAM run-time of the computation T. The main difference is that, in garbled RAM, the database y belongs to a single client and only that designated client can create garbled programs $f(x, \cdot)$ that the server can execute over this database to learn $f(x, y)$, while in RAM-LFE the database y belongs to the server and once the server publishes dig_y, any client can encrypt inputs x to allow the server to learn $f(x, y)$. One can think of the difference between garbled RAM and RAM-LFE with weak efficiency as the difference between ORAM and DEPIR; the former only allows a single designated client to privately access her own remotely stored data, while the latter allows any arbitrary client to privately access the server's data. Note that RAM-LFE with weak efficiency is a generalization that implies garbled RAM as a special case.[5]

Analogously, RAM-LFE with full efficiency is conceptually similar to *succinct garbled RAM* [GHRW14, CHJV15, CH16, BCG+18, JLL23], which requires the client's run-time to garble $f(x, \cdot)$ to

[4]In particular, consider the RAM program $f(x, y)$ that interprets $x = (i, b)$ as an index i and a bit b and outputs $y[i] \oplus b$ denoting the i'th location of y one-time padded with b. This program runs in time $T = O(1)$. Using a RAM-LFE for f we can construct a 3-round DEPIR as follows. The server preprocesses y to derive $\mathsf{dig}_y, \widetilde{y}$. To privately retrieve $y[i]$, the server/client run the following 3 round protocol: In the first round, the server sends dig_y to the client, in the second round the client encrypts $x = (i, b)$ with $b \leftarrow \{0, 1\}$ chosen uniformly at random under the LFE and sends the ciphertext, and in the third round the server decrypts the ciphertext and sends $y[i] \oplus b$ to the client who removes the pad to recover $y[i]$. LFE security ensures that the servers view can be simulated given $y[i] \oplus b$, which is uniformly random and therefore reveals no information about the client's index i.

[5]Garbled RAM has weaker functionality in that the database y must be preprocessed by the client, but then provides stronger security by ensuring that y is hidden from the server. However, it is easy to also use RAM-LFE to achieve the stronger security guarantee by having the client encrypt the database y via one-time pad derived from a PRF, and then include the PRF key as part of the encrypted input x and have the program execution use the PRF to decrypt each bit it reads from the database.

be independent of the RAM run-time T. The main difference is that RAM-LFE allows for computations over the server's data y, while succinct garbled RAM only allows for computations over the designated client's previously garbled data y. It is again easy to show that RAM-LFE with full efficiency is a generalization that implies succinct garbled RAM as a special case. All known construction of succinct garbled RAM rely on iO and therefore it is not surprising that we need to rely on iO as a building block.

Application: Functional Encryption for RAMs. LFE is also very related to (1-key) functional encryption (FE). The main difference is that FE requires some trusted third party to create a master public key mpk for the client and a secret key sk_f for the server: the client encrypts x under mpk and the server decrypts with sk_f to recover $f(x)$. In contrast, LFE allows the server itself to create the digest dig_f that the client uses to encrypt x; the server does not need any secret key to decrypt the ciphertext and can recover $f(x)$ using only knowledge of f. The work of [QWW18] showed that LFE generically implies succinct 1-key FE and the construction there directly extends to the RAM setting and preserves weak/full efficiency.

What about multi-key FE? Two recent works [ACFQ22, JLL23] constructed variants of multi-key FE for RAMs from functional encryption for circuits (which is equivalent to iO up to a sub-exponential security loss) and, in the case of [ACFQ22], DEPIR. This may appear highly related to our notion. However, beyond the distinction of FE vs LFE, there is an even more crucial distinction between our work and [ACFQ22, JLL23] in the type of RAM setting considered. The latter thinks of the server (decryptor) as having a short RAM program f and only the client has some large data x that is encrypted such that the server can decrypt $f(x)$, where f has random access to x. In particular, the notion in [ACFQ22] does not allow the program f to have random-access to a large server input y, and in [JLL23] the decryption time is at least linear in the description size of $f(\cdot, y)$ and hence linear in y itself. In contrast, our RAM-LFE setting envisions the server as also having some potentially large y and the computation $f(x, y)$ has random-access to both x and y. Arguably, allowing the decryption run-time to be sublinear in the server's input y is more crucial than making it sublinear in the client's input x, since anyway the server needs to spend linear time in x just to download the ciphertext from the client. We define a notion of FE for RAMs (RAM-FE), analogous to our notion of RAM-LFE, where the server gets a secret key sk_y tied to some potentially large input y, the client encrypts an input x, and the server decrypts $f(x, y)$ where f is some fixed RAM program (e.g., a universal RAM) with random-access to both x and y. We then show how to use RAM-LFE with full efficiency together with FE for circuits to construct multi-key RAM-FE.

Application: iO for RAMs. We also explore the notion of indistinguishability obfuscation for RAM programs with huge data (RAM-iO). In this setting, we consider obfuscating a RAM program $f(\cdot, y)$ containing some potentially huge data y and having worst-case run-time T for some fixed input length n. The obfuscated program $\widetilde{f}(\cdot, \widetilde{y})$ contains obfuscated data $\widetilde{y}$.[6] On any input $x \in \{0,1\}^n$ it should be possible to evaluate $\widetilde{f}(x, \widetilde{y}) = f(x, y)$ in time that scales proportionally to T but can be sublinear in $|y|$.[7] Indistinguishability security requires that for any two functionally equivalent program/data tuples $f(\cdot, y), f'(\cdot, y')$ such that $f(x, y) = f'(x, y')$ for all $x \in \{0,1\}^n$, the

[6]We can simply think of f as just being a universal RAM and all the actual code as being contained in y.

[7]Note that we cannot achieve sublinear run-time in $|x|$ since it is not preprocessed in this setting. Hence if evaluation reads only a subset of the positions of x that would reveal additional information about the computation.

obfuscation of $f(\cdot, y)$ is computationally indistinguishable from that of $f'(\cdot, y')$. We show how to generically construct iO for RAMs using iO for circuits and RAM-LFE with full efficiency (both with sub-exponential security).

We compare to the prior work of [BCG+18] on iO for RAMs. In that work, the obfuscated program f does not contain any large data y and there is no attempt to achieve sublinear run-time. Instead, the goal is just to (1) minimize the size of the obfuscated program to be proportional to the RAM/TM description length rather than circuit size of f and (2) to achieve the potential quadratic savings of evaluating $f(x)$ in the RAM model with random access to read/write memory over thinking of f as TM or circuit. While our work achieves (1) and (2), our main focus is sublinear run-time.

1.1 Our Techniques

As our starting point, we want to take prior (succinct / not succinct) garbled RAM schemes and transform them into (resp. fully efficient / weakly efficient) RAM-LFE. The main difficulty is how to incorporate the server's input y into the computation. The constructions of garbled RAM consist of 2 steps: (1) construct a scheme with unprotected memory and access (UMA), where the memory data and the access pattern of the computation are revealed to the server, but the program remains hidden, (2) hide the data and the access pattern using oblivious RAM (ORAM) [GO96]. The first issue is that, already in step (1), even though the data and the access pattern are not protected, many garbled RAM constructions still need to garble all the data using a secret key held by the client. Therefore these constructions don't allow us to incorporate the server's input y, which is preprocessed publicly by the server, into the computations. However, it turns out that some of the more recent constructions of garbled RAM based on laconic OT do achieve UMA security while allowing the data to be preprocessed publicly. In fact, the preprocessing simply computes a "laconic OT digest" of the data. Therefore such schemes give us RAM-LFE with unprotected memory and access. The second issue comes from step (2), which protects the memory access pattern via ORAM. Unfortunately, ORAM can only be used to protect client data that is encoded using a secret key held by the client, and therefore cannot be used to protect the access pattern to the server's data y. Instead, we replace ORAM by DEPIR to protect the access pattern to y. While this is the high-level template, instantiating it correctly is quite subtle. One of the bigger difficulties that comes up is that, in the case of succinct garbled RAM needed for fully efficient RAM-LFE, part (1) only achieves an indistinguishability-based notion of UMA security, where the server cannot distinguish between two garbled programs with the same output and the same access pattern. Upgrading such schemes into simulation-secure ones with full security requires a complex hybrid argument using a special type of ORAM with "localized randomness" [CH16]. We show how to adapt this type of argument using DEPIR as well. Our techniques borrow heavily from recent prior works on garbled circuits, garbled RAMs, succinct garbled TMs, FE for RAMs and succinct garbled RAMs; see e.g., [CDG+17, GS18a, GOS18, GS18b, ACFQ22, DGM23]. These schemes are quite involved and combine many different elements (laconic OT, garbled circuits, pebbling games, ORAM, and iO) via a subtle and brittle security proofs. Essentially, we show how to correctly combine these techniques together with DEPIR to upgrade the prior results into RAM-LFE schemes. Part of our contribution is to find the right abstractions for how to encapsulate prior results in a way that allows us to extend them to our setting. We sketch some of our techniques for RAM-LFE with weak efficiency and full efficiency respectively. Then we discuss our applications to FE and iO.

RAM-LFE with Weak Efficiency. This case turns out to be relatively simple. The scheme of [CDG+17] for "non-interactive secure computation in RAM Setting (RAM-NISC)" can already almost be thought of as a RAM-LFE with weak efficiency and with unprotected memory access (UMA) security. In particular, it allows the server to publish a short digest dig_y of y, the client to efficiently encrypt x using this digest, and the server to decrypt $f(x, y)$. The efficiency requirements are similar to RAM-LFE with weak-efficiency (except for one caveat mentioned below). Security states that the server learns nothing more than the output $f(x, y)$ *and* the memory access pattern of the computation. The construction of [CDG+17] works by combining updatable laconic oblivious transfer (LOT) and garbled circuits. At a high level the digest is an LOT digest of the data y. For a computation of f with run-time T, the client's ciphertext is a sequence of $O(T)$ garbled circuits that have x hard-coded and perform the computation of f while using LOT for random-access to the data y as well as any additional read/write memory. The main differences between RAM-NISC and our notion of RAM-LFE with UMA security are as follows:

- A minor difference is that in the RAM-NISC scheme the computation of the digest is randomized and the digest hides y. As discussed earlier, we chose to make our default definition require a deterministic digest generation and not require hiding security (and rely on the fact that the latter can be generically upgraded to the former).

  To bridge this gap, we simply remove the part of their scheme that implicitly performs this upgrade.

- While the RAM-NISC scheme allows f to have random-access to y, it thinks of x as a short input and does not allow random access to it. In particular, x is hard-coded in the garbled circuits and therefore decryption scales linearly in $|x|$.

  To bridge this gap, we can think of the client's input x as being treated similarly to y. The client encrypts x via a symmetric-key encryption and computes an LOT digest of it. The garbled circuits use LOT to make random-access queries to the encrypted x and decrypt the resulting bits.

With the above modifications, we get RAM-LFE with UMA security, where the server's view can be simulated given the output and the memory access pattern of the computation. To upgrade to full security, we simply use DEPIR to hide the access pattern to y and ORAM to hide the access pattern to x and any read/write memory. In particular the server initially applies DEPIR preprocessing on its input y to get $\widetilde{y}$ and computes the digest of $\widetilde{y}$ using the UMA scheme. The client initially puts its input x inside an ORAM $\widetilde{x}$ and we then run a UMA scheme for the function $\widetilde{f}(\widetilde{x}, \widetilde{y})$ that runs f but uses ORAM queries to $\widetilde{x}$ to privately access x and any read/write memory, and DEPIR queries on $\widetilde{y}$ to privately access y.

RAM-LFE with Full Efficiency. As in the previous case, we start by constructing RAM-LFE with UMA security and then upgrade to full security. Our idea is to start with a construction of succinct garbled RAM that has a very similar structure to the above RAM-NISC scheme, but gets a succinct ciphertext size and efficient encryption using iO: instead of the client computing the sequence of $O(T)$ garbled circuits, the client creates an obfuscated program that on input i outputs the i'th garbled circuit. Analyzing security using iO is however quite involved and requires a careful sequence of hybrids making small local changes to the garbled circuits. While the ideas to do so appeared in several prior works, (e.g.,) [BCG+18, GS18a, GOS18, GS18b, ACFQ22], we cannot use these results as a black box. Instead, we start with the work of [ACFQ22], which gets the closest to what we need, and modify it. In particular, that work constructs a succinct garbled RAM, where a client holds a database x and RAM program P. The client first processes the database x into some garbled database $\widetilde{x}$ and stores some associated secret key sk. Then the client can use sk to garble the program. A server can then recover $P(x)$ by evaluating the garbled program on the garbled database. The work of [ACFQ22] first constructs this in the UMA setting and then upgrades to full security. In the UMA model of security, the server learns nothing about P other than its access pattern and output when evaluated on x. We observe that in the UMA setting, the prepocessing of x is actually public/deterministic and the "secret key" sk is just an LOT digest. Therefore, the server can apply the same preprocessing on its input y and and we can think of the program $P(x, y)$ as having random-access to both the client's value and the server's value as an input.

To upgrade from UMA security to full security is conceptually similar to the weakly efficient case: we encrypt the client's database and hide the access patterns to x and y under an ORAM and DEPIR respectively. However, analyzing security is significantly more complicated. The main reason is that the fully efficient RAM LFE with UMA security can only achieve an indistinguishability-based notion of UMA, where the server cannot distinguish between ciphertexts that yield the same output and the exact same access pattern. This is in contrast to the weakly efficient case, which allowed for simulation security where the ciphertext can be simulated given the output and the access pattern. Using the indistinguishability based notion of UMA is more difficult. Essentially we need a sequence of hybrids that make only local changes to the access pattern. We use the techniques of [CH16] using "ORAM with localized randomness" along with punctured programming. We do so in two steps. In the first step we upgrade to a security notion where we hide the content of the client's database x and the contents of read/write memory, but where the pattern of addresses accessed are still unprotected. Then we show how to go from this intermediate notion to full (simulation-based) RAM-LFE security. The main observation is that the use of DEPIR works well with these techniques – since every access uses fresh/independent random coins, it essentially already satisfies a notion akin to the "localized randomness" needed in ORAM.

Application to FE. We show how to use fully efficient RAM-LFE to also construct RAM-FE where secret keys are associated with large databases y and the decryption time is sublinear in y. In particular, a key generation authority generates a master public key mpk and a master secret key msk. Using msk it can generate secret keys sk_y corresponding to some potentially large databases y for various servers. A client uses the master public-key to encrypt an input x and any server with secret key sk_y should only decrypt the output $f(x, y)$ for some fixed RAM program f (e.g., the universal RAM). We consider security against an adversary that can see many keys $\mathsf{sk}_{y_1}, \ldots, \mathsf{sk}_{y_q}$. In this setting simulation-based security is unachievable [AGVW13], and instead we must opt for indistinguishability based security where the adversary cannot distinguish between encryptions of x vs x' as long as $f(x, y_i) = f(x', y_i)$ for all $i \in [q]$. Assuming standard functional encryption for circuits and fully efficient RAM-LFE (with some augmentations supported by our construction, as discussed below) we get a RAM-FE scheme where, for any constant $\varepsilon > 0$, generating $(\mathsf{mpk}, \mathsf{msk})$ takes $O(1)$ time, generating sk_y takes time $|y|^{1+\varepsilon}$ encrypting x takes time $|x|^{1+\varepsilon}$ and decrypting $f(x, y)$ takes time $O(t)$ where t is the RAM run-time of f. The high-level idea is to use FE for circuits for the circuit that, on input x, generates the RAM-LFE encryption of x under the digest of y. One issue is that the circuit FE only satisfies indistinguishability based security and we overcome this via relatively standard (but non-trivial) output-programming tricks. This gives a scheme that achieve sublinear efficiency in y but not in x. To get sublinear efficiency in x we also observe that in our fully efficient RAM-LFE, the encryption of x can be split into two components: a digest-independent component that's at least linear in x, and a short digest-dependent component that's sublienar in x. We therefore have the client compute the digest-independent component as part of the RAM-FE ciphertext, and only use the circuit FE to compute the digest-dependent component.

Application to iO. We show how to use fully efficient RAM-LFE to also construct iO for RAMs (RAM-iO). Here we want to obfuscate a RAM program $f(\cdot, y)$ containing some large database y. We assume the program has some fixed input sizes and run-time T. The obfuscation procedure should run in time linear in y, but sublinear in T, while evaluating the obfuscated program should run in time linear in T but sublinear in y. We want standard indistinguishability security guaranteeing that the obfuscations of any functionally equivalent $f(\cdot, y) \equiv f'(\cdot, y')$ are computationally indistinguishable. We construct such RAM-iO using iO for circuits and RAM-LFE. The high-level idea is to encrypt y using a symmetric-key encryption with key k, and preprocess the ciphertext using DEPIR to get $\mathsf{dig}_y, \widetilde{y}$. The we use iO for circuits that on input x generates a RAM-LFE encryption of (k, x) for the modified function $f'((k, x), \mathsf{Enc}_k(y))$ that computes $f(x, y)$ by making random-access to the bits of the ciphertext and decrypting the answers. The obfuscated RAM program consists of $\widetilde{y}$ and the obfuscated circuit. Again, this high-level idea does not work as is since the iO for circuits only provides indistinguishability based security. We show to how make it work by using puncturing/programming tricks. Unfortunately, we need a sequence of $2^{|x|}$ hybrids, where we reprogram the output of the obfuscated circuit for every input x one by one. This comes at the cost of relying on sub-exponential security of the underlying components. However, iO constructions from falsifiable assumptions anyway already rely on sub-exponential assumptions [JLS21, JLS22] and hence this cost was needed already anyway.

2 Preliminaries

Define $\mathbb{N} = \{0, 1, 2, \ldots\}$ to be the set of natural numbers. For any integer $n \geq 1$, define $[n] = \{1, \ldots, n\}$. For an array $A \in \{0, 1\}^n$, we index the array from 1, and $A[i]$ denotes the bit in position $i \in [n]$. By default, all our logarithms are base 2 and $\log n$ stands for $\log_2 n$. A function $\nu : \mathbb{N} \to \mathbb{N}$ is said to be negligible, denoted $\nu(n) = \mathrm{negl}(n)$, 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 λ. For two distributions X, Y parameterized by λ we say that they are computationally indistinguishable, denoted by $X \approx_c Y$ if for every PPT distinguisher D we have $|\Pr[D(X) = 1] - \Pr[D(Y) = 1]| = \mathrm{negl}(\lambda)$.

2.1 Oblivious RAM

In the setting of ORAM [GO96], a client has a database $D \in \{0, 1\}^N$ that it wants to store on an untrusted server. The client wishes to preserve (read/write) random access to D while hiding its access pattern from the server. To do so, the client encodes D into an oblivious database D^* that is sent to the server. Then each "logical accesses" the client may want to make to D can be emulated by making $\text{poly}\log(N)$ "physical accesses" to D^*. The goal is that the sequence of physical accesses reveals nothing about the client's logical access pattern on D.

For our use cases, we wish to consider the slightly non-standard scenario where the client wishes to allocate an oblivious database of a large size (fixed in advance), but only has some smaller amount of initial data. We allow the client to initialize a database D which can be truncated into two parts: the former contains the initial memory content we want to store and the latter consists of empty cells. Specifically, for $D \in \{0,1\}^N$, let $D_0 \in \{0,1\}^n$ for some $n \leq N$ be its former part that stores the initial memory content, and let cells with address larger than n be initially empty.

Definition 2.1 (ORAM). *An Oblivious RAM scheme* ORAM *consists of procedures* $(\mathsf{ORAM.Setup}, \mathsf{ORAM.Access})$*, with following syntax:*

- $(\mathsf{ck}_0, D^*) \leftarrow \mathsf{ORAM.Setup}(1^\lambda, D_0, N)$*: Given a security parameter λ, an initial memory content $D_0 \in \{0,1\}^n$ and size N, it outputs an initial client secret key* ck_0 *and oblivious database D^*.*
- $(\mathsf{ck}', \mathsf{val}') \leftarrow \mathsf{ORAM.Access}^{D^*}(\mathsf{ck}, \mathsf{op}, \mathsf{addr}, \mathsf{val})$*: It takes as input the current client secret key* ck*, an operation* $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$*, an address* $\mathsf{addr} \in [N]$ *and a value* val*(if* $\mathsf{op} = \mathsf{read}$*, then* val *is ignored). With random access to oblivious database D^*, it emulates the logical access* $(\mathsf{op}, \mathsf{addr}, \mathsf{val})$ *that the client wants to make, and outputs the updated client secret key* ck' *and the result* val' *of the logical access(if* $\mathsf{op} = \mathsf{write}$*, then* $\mathsf{val}' = \perp$*).*

Correctness. *Let $\lambda \in \mathbb{N}$, $D \in \{0,1\}^N$ with initial content $D_0 \in \{0,1\}^n$, $t \in \mathbb{N}$, and* $\{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)\}_{i=1}^t$ *be t sequential logical accesses. Let* $(\mathsf{ck}_0, D^*) \leftarrow \mathsf{ORAM.Setup}(1^\lambda, D_0, N)$*, and let* $(\mathsf{ck}_i, \mathsf{val}'_i) \leftarrow \mathsf{ORAM.Access}(\mathsf{ck}_{i-1}, \mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)$ *for each $i = 1$ to t. On the other hand, if the user directly makes the logical accesses* $\{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)\}_{i=1}^t$ *sequentially on D, it obtains* $(\mathsf{val}^*_1, \cdots, \mathsf{val}^*_t)$ *as results. Then, with probability* 1, $(\mathsf{val}'_1, \cdots, \mathsf{val}'_t) = (\mathsf{val}^*_1, \cdots, \mathsf{val}^*_t)$.

Efficiency. *Given security parameter λ, initial content size n and entire database size N, we require that:*

- $\mathsf{ORAM.Setup}$ *runs in time* $n \cdot \text{poly}(\lambda, \log N)$.
- *There is a function $\eta : \mathbb{N} \to \mathbb{N}$ satisfying $\eta(N) = \text{poly}\log(N)$, such that* $\mathsf{ORAM.Access}$ *accesses at most $\eta(N)$ physical addresses each time.*

Security. *There exists a PPT algorithm* $\mathsf{ORAMSim}$ *such that for any stateful PPT adversary $\mathcal{A}$, we have*

$$\left|\Pr\left[\mathsf{Real}_{\mathsf{ORAM}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Ideal}_{\mathsf{ORAM}}(1^\lambda) = 1\right]\right| = \text{negl}(\lambda),$$

where the experiments $\mathsf{Real}_{\mathsf{ORAM}}$ *and* $\mathsf{Ideal}_{\mathsf{ORAM}}$ *are defined as follows:*

$\mathsf{Real}_{\mathsf{ORAM}}(1^\lambda)$:

1. $N, D_0 \leftarrow \mathcal{A}(1^\lambda)$
2. $(\mathsf{ck}_0, D^*) \leftarrow \mathsf{ORAM.Setup}(1^\lambda, D_0, N)$
3. *for* $i = 1$ *to* t*:*
   $(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i) \leftarrow \mathcal{A}(\mathsf{Addrs}_{i-1}, \mathsf{val}'_{i-1})$,
   $(\mathsf{ck}_i, \mathsf{val}'_i) \leftarrow \mathsf{ORAM.Access}^{D^*}(\mathsf{ck}_{i-1}, \mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)$,
   with $\mathsf{Addrs}_i \subset \{\mathsf{read}, \mathsf{write}\} \times [|D^*|]$ *being the set of physical accesses that* $\mathsf{ORAM.Access}$ *makes.*
4. *Output* $\mathcal{A}(\mathsf{Addrs}_t, \mathsf{val}'_t)$

$\mathsf{Ideal}_{\mathsf{ORAM}}(1^\lambda)$:

1. $N, D_0 \leftarrow \mathcal{A}(1^\lambda)$
2. $(\mathsf{ck}_0, D^*) \leftarrow \mathsf{ORAM.Setup}(1^\lambda, D_0, N)$
3. *for* $i = 1$ *to* t*:*
   $(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i) \leftarrow \mathcal{A}(\mathsf{Addrs}_{i-1}, \mathsf{val}'_{i-1})$,
   $\begin{cases} \mathsf{val}'_i = D[\mathsf{addr}_i] & \mathsf{op}_i = \mathsf{read} \\ \mathsf{val}'_i = \perp, D[\mathsf{addr}_i] = \mathsf{val}_i & \mathsf{op}_i = \mathsf{write} \end{cases}$
   $\mathsf{Addrs}_i \leftarrow \mathsf{ORAMSim}(1^\lambda, N)$
4. *Output* $\mathcal{A}(\mathsf{Addrs}_t, \mathsf{val}'_t)$

Remark 2.1 (Setup efficiency)**.** We note that our requirement of the running time of ORAM.Setup is linear in the initial content size n, which is stronger than normal definition which only requires the time to be linear in the entire database size N. However, this property can be achieved by simply combining two normal ORAM schemes. Basically, for $D \in \{0,1\}^N$ with initial content $D_0 \in \{0,1\}^n$ and the latter part $E_0 \in \{0,1\}^{N-n}$, we use two separate ORAM schemes for D_0 and E_0. We initialize one ORAM with D_0 and we initialize a separate empty ORAM of size $N - n$ to hold E_0. We assume initializing an empty ORAM can be done in logarithmic time in N (as in, e.g., [CP13]). Thus with $D^* = D_0^*||E_0^*$, we obtain a $n \cdot \mathrm{poly}(\lambda, \log N)$ total setup time. For each access $q = (\mathsf{op}, \mathsf{addr}, \mathsf{val})$ on D, no matter which part of database it is accessing, we emulate both an access to D_0^* and an access to E_0^*. The one on the database corresponding to q is a valid access that achieves functionality of q, and the other one is a dummy access. Therefore, the security and localized randomness (later defined in Definition 2.3) of our scheme directly follows the same property of two component ORAM schemes.

Remark 2.2. The ORAM scheme we defined above only aims to make sure that physical addresses does not reveal any information, i.e. only guarantee that the adversary cannot extract any information if it only sees physical addresses. Therefore, the type of physical accesses, the written value of physical accesses and oblivious database may still divulge information about the logical accesses.

Theorem 2.2 ([GO96])**.** *Assuming only the existence of one way functions, there exists a standard* ORAM *scheme with* $\mathrm{poly}\log(N)$ *overhead (as in Definition 2.1).*

2.1.1 ORAM with Localized Randomness

The notion of localized randomness was introduced by [CH16], which describes the property of an ORAM scheme to emulate each logical access in an independent way, so that the physical addresses correlated to this step is determined only by a relatively small subset of the entire randomness. We also require that the subsets of randomness corresponding to different steps should be disjoint, so that the physical addresses of different steps are independent random variables.

Let $D \in \{0,1\}^N$, $\eta = \eta(N)$, $\{q_i\}_{i=1}^t$ be t logical accesses where $q_i = (\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)$. Under an ORAM scheme, consider the deterministic function AddrGen:

$$\mathsf{AddrGen}_{D,\{q_i\}} : \{0,1\}^M \to ([N] \times \{\mathsf{read}, \mathsf{write}\})^{\eta t}$$
$$\mathsf{AddrGen}_{D,\{q_i\}}(R) = \{\mathsf{Addrs}_i\}_{i=1}^t$$

where $M = \text{poly}(\lambda, t, \log N)$ is an upper bound on the number of random bits used in ORAM.Setup and a sequence of t calls to ORAM.Access, $R \in \{0,1\}^M$ is the entire random tape used in the calls to ORAM.Setup and ORAM.Access, and $\mathsf{Addrs}_i = \{\mathsf{Addr}_{i,1}, \cdots, \mathsf{Addr}_{i,\eta}\}$ is the set of physical accesses made by ORAM.Access when emulating $(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)$, the i-th logical access, given random tape R.

Definition 2.3 (Localized Randomness). *An ORAM scheme* ORAM = (ORAM.Setup, ORAM.Access) *is said to have localized randomness, if there is a deterministic simulator* LRSim *such that for any λ, N, t, for any $D \in \{0,1\}^N$ and t sequential logical accesses $\{q_i\}_{i=1}^t = \{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{val}_i)\}_{i=1}^t$, there exists sets $S_1, \cdots, S_t \subseteq [M]$ such that,*

- *For each i, $|S_i| \le \text{poly}\log(N) \cdot \text{poly}(\lambda)$.*
- *For each $i \ne j$, $S_i \cap S_j = \varnothing$.*
- *For each i,*

$$\Pr\left[\mathsf{Addrs}_i = \mathsf{LRSim}(R_{S_i}) \;\middle|\; \begin{array}{c} R \leftarrow \{0,1\}^M \\ \{\mathsf{Addrs}_j\}_{j=1}^t \leftarrow \mathsf{AddrGen}_{D,\{q_j\}}(R) \end{array}\right] \ge 1 - \text{negl}(\lambda).$$

Theorem 2.4 ([CH16]). *There exists an ORAM scheme with localized randomness, without relying on any cryptographic hardness assumptions.*

2.2 DEPIR

In this subsection, we define doubly efficient private information retrieval (DEPIR). In the setting of DEPIR, a server holds a large database $\mathsf{DB} \in \{0,1\}^N$ and a client holds an index $i \in [N]$. The goal is for the client to learn $\mathsf{DB}[i]$ while hiding i from the server. The server performs a one-time offline preprocessing on its database, and then later the client can make queries using independent randomness and without maintaining any state. We give the formal definition below.

Definition 2.5 (DEPIR). *A doubly efficient private information retrieval scheme (DEPIR) is a tuple of algorithms* (Prep, Query, Dec) *with the following syntax and correctness and security properties.*

- $\widetilde{\mathsf{DB}} := \mathsf{Prep}(1^\lambda, \mathsf{DB})$*: Given the security parameter and a database $\mathsf{DB} \in \{0,1\}^N$, it deterministically outputs a preprocessed database $\widetilde{\mathsf{DB}} \in \{0,1\}^{\widetilde{N}}$.*
- $(Q, s) \leftarrow \mathsf{Query}(1^\lambda, N, i)$*: Given the security parameter λ, database size N and a location $i \in [N]$, it outputs a query given by a small set of indices $Q \subseteq [\widetilde{N}]$ to locations in $\widetilde{\mathsf{DB}}$ and a query-specific decryption key s. Without loss of generality, we assume that all queries Q are the same size. For a set of locations $I \subseteq [N]$ we also write $\mathsf{Query}(1^\lambda, I)$ to indicate running $\mathsf{Query}(1^\lambda, i)$ on each $i \in I$ and aggregating the results.*
- $b := \mathsf{Dec}(s, V)$*: Given the decryption key s and a small set V of values from $\widetilde{\mathsf{DB}}$, it outputs a bit $b \in \{0,1\}$.*

Correctness. *Let $\lambda \in \mathbb{N}$ and $\mathsf{DB} \in \{0,1\}^N$. Let $\widetilde{\mathsf{DB}} := \mathsf{Prep}(1^\lambda, \mathsf{DB})$. Then for any $i \in [N]$, if we sample a query $(Q, s) \leftarrow \mathsf{Query}(1^\lambda, i)$, then $\mathsf{Dec}(s, \widetilde{\mathsf{DB}}[Q]) = \mathsf{DB}[i]$ with probability 1.*

Security. *For any $i_0, i_1 \in [N]$, the queries Q_0 and Q_1 output by* Query *on indices i_0 and i_1 are computationally indistinguishable.*

The recent work of [LMW22] shows the existence of DEPIR assuming only the hardness of RingLWE.

Theorem 2.6 ([LMW22]). *Under the Ring LWE assumption, for any $\varepsilon > 0$, there is a DEPIR in which* Prep *runs in time $N^{1+\varepsilon} \cdot \mathrm{poly}(\lambda, \log N)$ and* Query *and* Dec *each run in time $\mathrm{poly}(\lambda, \log N)$.*

We remark that the definition we give above is slightly modified from the one that appears in [LMW22] in that it requires that the DEPIR queries are themselves sets of locations in the preprocessed database, and the server can respond to a query simply by looking up those locations. This implicitly requires that a query can be answered by reading locations from $\widetilde{\mathsf{DB}}$ non-adaptively. However, the DEPIR construction in [LMW22] satisfies this requirement and thus fits our definition.

2.3 Puncturable PRFs

Definition 2.7. *A* puncturable PRF *is a PRF family $\mathcal{F} = \{F : \{0,1\}^n \to \{0,1\}^m\}_\lambda$ of functions satisfying the following additional conditions.*

Puncturability. *There exists an efficient procedure that, given any $F \in \mathcal{F}$ and a set $S \subset \{0,1\}^n$ of inputs, produces a punctured function that we denote $F' = F\{S\}$. The function F' is efficiently computable and has description size $|F'| = \mathrm{poly}(|S|, |F|)$.*

Functionality. *For any PPT adversary $\mathcal{A}(1^\lambda)$ that outputs $S \subseteq \{0,1\}^n$ and for all inputs $x \notin S$, we have $F'(x) = F(x)$ with probability 1, where the probability is over $F \leftarrow \mathcal{F}$ and $F' = F\{S\}$.*

Pseudorandomness at punctured points. *For any PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ such that $\mathcal{A}_1(1^\lambda)$ outputs a set $S \subset \{0,1\}^n$ and a state* st*, consider the experiment where we sample $F \leftarrow \mathcal{F}$ and $F' = F\{S\}$. Then*

$$\left|\Pr[\mathcal{A}_2(\mathsf{st}, F', S, F(S)) = 1] - \Pr[\mathcal{A}_2(\mathsf{st}, F', S, U)]\right| = \mathrm{negl}(\lambda),$$

where here we abuse notation to write $F(S) = \{F(s) : s \in S\}$ (sorted in some fixed order, e.g., lexicographical) and $U = \{u_s : s \in S\}$ where each $u_s \leftarrow \{0,1\}^m$ is uniformly and independently sampled.

2.4 Laconic OT

In this subsection, we define laconic oblivious transfer [CDG+17] (laconic OT, also LOT). It allows a server to commit to a large database D via a short digest. Then the client can respond with a single short message to the server depending on dynamically chosen two messages m_0, m_1 and a location L in D, so that the server can only recover $m_{D[L]}$, without learning anything about $m_{1-D[L]}$. We give the formal definition below.

Definition 2.8 (Laconic OT, [CDG+17]). *A laconic oblivious transfer scheme (Laconic OT) is a tuple of algorithms* $\mathsf{LOT} = (\mathsf{LOT.crsGen}, \mathsf{LOT.Hash}, \mathsf{LOT.Send}, \mathsf{LOT.Receive})$ *with the following sytax and correctness, security and efficiency properties.*

- $\mathsf{crs} \leftarrow \mathsf{LOT.crsGen}(1^\lambda)$: *Given the security parameter, it samples a common reference string* crs.
- $(\mathsf{dig}, \tilde{D}) := \mathsf{LOT.Hash}(\mathsf{crs}, D)$: *Given the database* $D \in \{0,1\}^*$ *and the common reference string* crs, *it outputs a digest* dig *for the database and a state* $\tilde{D}$.
- $e \leftarrow \mathsf{LOT.Send}(\mathsf{crs}, \mathsf{dig}, L, m_0, m_1)$: *Given the* crs, *the digest* dig, *a database location* L, *two messages* m_0, m_1 *of length* L, *it outputs a ciphertext* e.
- $m := \mathsf{LOT.Receive}^{\tilde{D}}(\mathsf{crs}, e, L)$: *Given the* crs, *a ciphertext* e, *a database location* L, *and with random access to* $\tilde{D}$, *it outputs a message* m.

Correctness. *Let* $\lambda \in \mathbb{N}$ *and* $D \in \{0,1\}^N$. *For any location* $L \in [N]$ *and any pair of messages* $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$,

$$\Pr\left[m = m_{D[L]} \;\middle|\; \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{LOT.crsGen}(1^\lambda) \\ (\mathsf{dig}, \tilde{D}) = \mathsf{LOT.Hash}(\mathsf{crs}, D) \\ e \leftarrow \mathsf{LOT.Send}(\mathsf{crs}, \mathsf{dig}, L, m_0, m_1) \\ m = \mathsf{LOT.Receive}^{\tilde{D}}(\mathsf{crs}, e, L) \end{array} \right] = 1.$$

Security. *(Against Semi-honest receiver, simulation-based security) There exists a PPT algorithm* LOTSim *such that for any* $D \in \{0,1\}^N, L \in [N]$ *and any pair of messages* $(m_0, m_1) \in \{0,1\}^\lambda \times \{0,1\}^\lambda$, *let* $\mathsf{crs} \leftarrow \mathsf{LOT.crsGen}(1^\lambda)$, $(\mathsf{dig}, \tilde{D}) := \mathsf{LOT.Hash}(\mathsf{crs}, D)$, *there is*

$$(\mathsf{crs}, \mathsf{LOT.Send}(\mathsf{crs}, \mathsf{dig}, L, m_0, m_1)) \approx_c (\mathsf{crs}, \mathsf{LOTSim}(\mathsf{crs}, D, L, m_{D[L]})).$$

In the proof of security in [CDG+17], it reaches adaptive security. That is, a PPT adversary $\mathcal{A}$ *can choose* $(D, L, m_0, m_1) \leftarrow \mathcal{A}(\mathsf{crs})$, *while* $\mathcal{A}$ *still cannot distinguish the two distributions above.*

Efficiency. *The length of the digest* dig *is a fixed polynomial of* λ *and is independent of the size of the database* N. *The algorithm* $\mathsf{LOT.Hash}$ *runs in time* $N \cdot \mathrm{poly}(\log N, \lambda)$, *and* $\mathsf{LOT.Send}$, $\mathsf{LOT.Receive}$ *run in time* $\mathrm{poly}(\log N, \lambda)$.

In [CDG+17], they also handle with writes to the database D, and they put forward a version of laconic OT that can update the digest quickly, which is called updatable laconic OT. At a high level, updatable laconic OT has two additional algorithms $\mathsf{LOT.SendWrite}$, $\mathsf{LOT.ReceiveWrite}$, which send keys for an updated digest dig^* to the receiver.

Definition 2.9 (Updatable Laconic OT, [CDG+17]). *An updatable laconic oblivious transfer scheme (updatable* LOT*) consists of a tuple of algorithms of laconic OT, and additionally has two algorithms with the following syntax, correctness, security and efficiency properties.*

- $e_w \leftarrow \mathsf{LOT.SendWrite}(\mathsf{crs}, \mathsf{dig}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{dig}|})$: *Given the* crs, *the digest* dig, *the written location* $L \in [N]$, *a bit* $b \in \{0,1\}$ *to be written, and* $|\mathsf{dig}|$ *pairs of messages* $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{dig}|}$), *it outputs a ciphertext* e_w.
- $\{m_j\}_{j=1}^{|\mathsf{dig}|} := \mathsf{LOT.ReceiveWrite}^{\tilde{D}}(\mathsf{crs}, L, b, e_w)$: *Given the* crs, *the written location* L, *a written bit* b, *a ciphertext* e_w, *and with random read/write access to* $\tilde{D}$, *it outputs messages* $\{m_j\}_{j=1}^{|\mathsf{dig}|}$.

Correctness. *Let $\lambda \in \mathbb{N}$ and $D \in \{0,1\}^N$. For any location $L \in [N]$ and $b \in \{0,1\}$, define D^* to be identical to D except $D^*[L] = b$. For all pairs of messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{dig}|}$,*

$$\Pr\left[\begin{array}{l|r} & \mathsf{crs} \leftarrow \mathsf{LOT.crsGen}(1^\lambda) \\ & (\mathsf{dig}, \tilde{D}) = \mathsf{LOT.Hash}(\mathsf{crs}, D) \\ m'_j = m_{j,\mathsf{dig}^*_j} & (\mathsf{dig}^*, \tilde{D^*}) = \mathsf{LOT.Hash}(\mathsf{crs}, D^*) \\ \forall j \in [|\mathsf{dig}|] & e_w \leftarrow \mathsf{LOT.SendWrite}(\mathsf{crs}, \mathsf{dig}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{dig}|}) \\ & \{m'_j\}_{j=1}^{|\mathsf{dig}|} = \mathsf{LOT.ReceiveWrite}^{\tilde{D}}(\mathsf{crs}, L, b, e_w) \end{array}\right] = 1.$$

It is also required that $\mathsf{LOT.ReceiveWrite}^{\tilde{D}}$ will update $\tilde{D}$ to $\tilde{D^*}$.

Security. *(With regard to write, simulation-based security) There exists a PPT algorithm $\mathsf{LOTSimWrite}$ such that for any $D \in \{0,1\}^N, L \in [N], b \in \{0,1\}$ and all pairs of messages $\{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{dig}|}$, let $\mathsf{crs} \leftarrow \mathsf{LOT.crsGen}(1^\lambda)$, $(\mathsf{dig}, \tilde{D}) := \mathsf{LOT.Hash}(\mathsf{crs}, D)$, and $(\mathsf{dig}^*, \tilde{D^*}) := \mathsf{LOT.Hash}(\mathsf{crs}, D^*)$, where D^* is identical to D except $D^*[L] = b$, there is*

$$\begin{aligned} &(\mathsf{crs}, \mathsf{LOT.SendWrite}(\mathsf{crs}, \mathsf{dig}, L, b, \{m_{j,0}, m_{j,1}\}_{j=1}^{|\mathsf{dig}|})) \\ \approx_c\ &(\mathsf{crs}, \mathsf{LOTSimWrite}(\mathsf{crs}, D, L, b, \{m_{j,\mathsf{dig}^*_j}\}_{j=1}^{|\mathsf{dig}|})). \end{aligned}$$

Efficiency. *Algorithms $\mathsf{LOT.SendWrite}$ and $\mathsf{LOT.ReceiveWrite}$ run in time $\mathrm{poly}(\log N, \lambda)$.*

2.5 Functional Encryption

Definition 2.10 (Functional Encryption for Circuits). *A public-key functional encryption for circuits (circuit-FE) over message space $\mathcal{M}$ and circuit space $\mathcal{C}$ is a tuple of algorithms $\mathsf{cFE} = (\mathsf{cFE.Setup}, \mathsf{cFE.Keygen}, \mathsf{cFE.Enc}, \mathsf{cFE.Dec})$ with the following syntax and correctness, security and efficiency properties:*

- $(\mathsf{msk}, \mathsf{pk}) \leftarrow \mathsf{cFE.Setup}(1^\lambda)$*: Given the security parameter 1^λ, output a pair of a master secret key and a public key.*
- $\mathsf{sk}_C \leftarrow \mathsf{cFE.Keygen}(\mathsf{msk}, C)$*: Given the master secret key and a circuit $C \in \mathcal{C}$, output a functional key sk_C for circuit C.*
- $\mathsf{ct}_x \leftarrow \mathsf{cFE.Enc}(\mathsf{pk}, x)$*: Given the public key pk and a message $x \in \mathcal{M}$, output a ciphertext ct_x for x.*
- $z \leftarrow \mathsf{cFE.Dec}(\mathsf{ct}_x, \mathsf{sk}_C)$*: Given a ciphertext ct_x and a functional key sk_C, output the result z.*

Correctness. *For any $x \in \mathcal{M}$ and any $C \in \mathcal{C}$, it holds that*

$$\Pr[\mathsf{cFE.Dec}(\mathsf{cFE.Enc}(\mathsf{pk}, x), \mathsf{cFE.Keygen}(\mathsf{msk}, C)) = C(x)] = 1 - \mathrm{negl}(\lambda),$$

where $(\mathsf{pk}, \mathsf{msk}) \leftarrow \mathsf{cFE.Setup}(1^\lambda)$ and the probability is taken over the randomness of all of the algorithms.

Security. *A functional encryption scheme satisfies the* indistinguishability-based selective security, *if for all PPT adversaries* $\mathcal{A}$,

$$\left|\Pr[\mathrm{Expt}^{\mathcal{A}}_{\mathsf{cFE}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Expt}^{\mathcal{A}}_{\mathsf{cFE}}(1^\lambda, 1) = 1]\right| \leq \mathrm{negl}(\lambda),$$

where the experiment $\mathrm{Expt}^{\mathcal{A}}_{\mathsf{cFE}}(1^\lambda, b)$ *is defined as follows:*

1. *The challenger computes* $(\mathsf{msk}, \mathsf{pk}) \leftarrow \mathsf{cFE.Setup}(1^\lambda)$.
2. $\mathcal{A}$ *chooses two messages* x_0, x_1 *of the same size and sends them to the challenger. The challenger replies with* pk *and* $\mathsf{ct}_b \leftarrow \mathsf{cFE.Enc}(\mathsf{pk}, x_b)$.
3. $\mathcal{A}$ *may adaptively request arbitrarily many keys in the following way:* $\mathcal{A}$ *chooses a circuit* $C \in \mathcal{C}$ *and sends it to the challenger. The challenger checks that* $C(x_0) = C(x_1)$. *If the check fails, the challenger aborts; otherwise it sends* $\mathsf{sk}_C \leftarrow \mathsf{cFE.Keygen}(\mathsf{msk}, C)$ *to* $\mathcal{A}$.
4. *Finally,* $\mathcal{A}$ *outputs a bit* b', *and the output of the experiment is* b'.

Efficiency ([GGH+13]). *We require that the algorithms satisfy the following efficiency requirements.*

- $\mathsf{cFE.Setup}(1^\lambda)$ *runs in time* $\mathrm{poly}(\lambda)$.
- $\mathsf{cFE.Keygen}(\mathsf{msk}, C)$ *runs in time* $\mathrm{poly}(\lambda, |C|)$.
- $\mathsf{cFE.Enc}(\mathsf{pk}, m)$ *runs in time* $\mathrm{poly}(\lambda, s, |m|)$, *where* s *is the maximum size of the circuit.*
- $\mathsf{cFE.Dec}(\mathsf{ct}_m, \mathsf{sk}_C)$ *runs in time* $s + \mathrm{poly}(\lambda)$.

3 Laconic Function Evaluation for RAM Programs

In this section we define Laconic Function Evaluation for RAM Programs (RAM-LFE). At a high level, a RAM-LFE is a protocol between a server who holds a public database y and a client who holds a private database x and a private RAM program P. The protocol begins with the server committing itself to y by compressing it down to some short digest that it sends to the client. Then, the client is able to use this digest to encrypt its inputs P and x and produce a ciphertext ct, in a process that doesn't depend on the size of y. Finally using its database and the ciphertext, the server can decrypt the ciphertext to recover the output of the RAM program $P(x, y)$ without learning anything else about P or x. Moreover, this decryption process should run in time that is roughly linear in the RAM run time of evaluating $P(x, y)$ in the clear. We note that during the protocol we do not allow P to make updates to the server's database that would persist to later executions of the protocol.

3.1 RAM Model

Before we can formally define RAM-LFE, we first specify the model we use for RAM computation in this work. We consider RAM programs P that take two inputs x and y. Because we will ultimately model y as the server's database and we don't permit persistent updates to y, we think of y as being stored in read-only memory, and x is stored in read/write enabled memory which has a larger size to include both the data content and some additional empty cells at the end portion of database. We note that we make this choice without loss of generality because any program that makes updates to y can be converted into one that stores its edits in a working memory that is located in the end portion of x with only little overhead. Evaluation of a RAM program consists of a sequence of steps, where in each step the program reads one memory cell each from x and y and writes one cell to x.

Formally, a RAM program P is defined by a tuple $(C, Q, T, n_x, n_y, \Sigma, Z)$, where C is a binary circuit called the *step circuit* that defines a transition function $C : Q \times \Sigma^2 \to (Q \times [n_x] \times [n_y] \times (\Sigma \times [n_x])) \cup Z$, Q is a set of states, $T \in \mathbb{N}$ is an upper bound on the number of steps the computation will take, n_x and n_y are upper bounds on the sizes of the inputs x and y respectively, Σ is a set defining the alphabet of symbols that can appear in a memory cell, and Z is the set of possible outputs. Throughout this work we exclusively consider $Z = \{0, 1\}$. Unless otherwise specified, we identify Q with a set of binary strings having length polynomial in the security parameter, and we assume that $\Sigma = \{0, 1\}$.

Evaluation of a RAM program $P = (C, Q, T, n_x, n_y, \Sigma, Z)$ on inputs x and y proceeds iteratively as follows. We begin by initializing $\mathsf{st}^0 \in Q$ to the all zeros string and arbitrarily initializing $v_x^0, v_y^0 \in \Sigma$. For each $t \in [T]$:

1. Evaluate the step circuit to obtain $\mathsf{out} = C(\mathsf{st}^{t-1}, v_x^{t-1}, v_y^{t-1})$.
2. If $\mathsf{out} = z \in Z$, halt and output z. Otherwise, parse $\mathsf{out} = (\mathsf{st}^t, i_x^t, i_y^t, j_x^t, w^t)$ and continue.
3. Write $x[j_x^t] = w^t$ to x.
4. Read values $v_x^t = x[i_x^t]$ and $v_y^t = y[i_y^t]$ from x and y respectively.

If the computation halts in step 2 after t steps, then we write $P(x, y) = z$ and say its run time is t, otherwise we write $P(x, y) = \perp$ and the program has run time T.

For a RAM program $P = (C, Q, T, n_x, n_y, \Sigma, Z)$ we define $\mathsf{md}(P) = (|C|, Q, T, n_x, n_y, \Sigma, Z)$ to denote all of the metadata associate with a RAM program. We write $|P| = |C|$, and unless otherwise specified, we assume that $|P| = \mathrm{poly}(\lambda, \log T, \log n_x, \log n_y)$.

Remark 3.1 (Uniform vs non-uniform computation)**.** We note that the above is a non-uniform model of computation, where the size and description of the step circuit necessarily depend on the lengths $|x|$ and $|y|$. We can implicitly assume that the step circuit C and time upper bound T are efficiently computable by a Turing machine that is given the database sizes as input. This fits our setting as we can assume without loss of generality that the digest of the servers input contains the length $|y|$ and thus the client knows both $|x|$ and $|y|$ when it needs to specify the description of the RAM program.

3.2 Definition

Definition 3.1 (RAM-LFE)**.** *A* laconic function evaluation for RAM programs *(RAM-LFE) is a tuple of algorithms* ($\mathsf{LFE.Gen}$, $\mathsf{LFE.Hash}$, $\mathsf{LFE.Enc}$, $\mathsf{LFE.Dec}$) *that has the following syntax:*

- $\mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda)$ *: Given the security parameter* 1^λ *, the generation algorithm returns a common reference string* crs.

- $(\mathsf{dig}, \widetilde{y}) := \mathsf{LFE.Hash}(\mathsf{crs}, y)$: *Given the common reference string* crs *and the database* y*, the compression algorithm deterministically outputs a short digest* dig *and a preprocessed database* $\widetilde{y}$.
- $\mathsf{ct} \leftarrow \mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$: *Given the common reference string* crs*, the digest* dig*, a RAM program* P*, and a secret input database* x*, the encoding algorithm returns a ciphertext* ct.
- $z := \mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$: *Given the common reference string* crs*, the preprocessed database* $\widetilde{y}$*, and a ciphertext* ct*, the decoding algorithm returns a RAM program output* $z \in \{0, 1\}$.

We require the algorithms to satisfy the following correctness and security properties.

Correctness: *We require that for all* $\lambda \in \mathbb{N}$*, for databases* $D \in \{0,1\}^N$*, and all messages input* $x \in \{0,1\}^n$ *it holds that*

$$\Pr\left[z = P(x,y) \;\middle|\; \begin{array}{r} \mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda) \\ (\mathsf{dig}, \widetilde{y}) := \mathsf{LFE.Hash}(\mathsf{crs}, y) \\ \mathsf{ct} \leftarrow \mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x) \\ z := \mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct}) \end{array} \right] = 1.$$

Security: *There exists a PPT algorithm* LFESim *such that for any stateful PPT adversary* $\mathcal{A}$*, we have*

$$\left| \Pr\left[\mathsf{Real}_{\mathsf{LFE}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Ideal}_{\mathsf{LFE}}(1^\lambda) = 1\right] \right| = \mathrm{negl}(\lambda),$$

where the experiments $\mathsf{Real}_{\mathsf{LFE}}$ *and* $\mathsf{Ideal}_{\mathsf{LFE}}$ *are defined as follows:*

$\mathsf{Real}_{\mathsf{LFE}}(1^\lambda)$:

1. *Sample* $\mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda)$
2. $(P, x, y) \leftarrow \mathcal{A}(\mathsf{crs})$
3. $(\mathsf{dig}, \widetilde{y}) := \mathsf{LFE.Hash}(\mathsf{crs}, y)$;
   $\mathsf{ct} \leftarrow \mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$
4. *Output* $\mathcal{A}(\mathsf{ct})$

$\mathsf{Ideal}_{\mathsf{LFE}}(1^\lambda)$:

1. *Sample* $\mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda)$
2. $(P, x, y) \leftarrow \mathcal{A}(\mathsf{crs})$
3. $\mathsf{ct} \leftarrow \mathsf{LFESim}(\mathsf{crs}, y, P(x,y), t, \mathsf{md})$ *where* t *is the run time of* $P(x,y)$ *and* $\mathsf{md} = \mathsf{md}(P)$
4. *Output* $\mathcal{A}(\mathsf{ct})$

Efficiency. In the above definition, we don't explicitly define desired efficiency properties for a RAM-LFE. Thus the trivial scheme where the server simply sends the entire database y to the client who then computes $P(x,y)$ and sends it back to the server does satisfy the definition. However, our goal will be to achieve a RAM-LFE where the decryption run time is nearly linear in the RAM run time of $P(x,y)$ and the encryption run time is independent of the size of y.

Additionally, we consider two broad classes of efficiency within this goal. If a RAM-LFE satisfies the above efficiency requirements and has encryption run time that is at least linear in the worst case complexity T of $P(x,\cdot)$, then we say it is *weakly efficient*. On the other hand, if the encryption run time is sublinear in T (or, ideally, polylogarithmic) we say it is *fully efficient*.

Remark 3.2 (Universal programs)**.** In the above definition, we adopt the convention that the client chooses the RAM program P that is to be evaluated. We make this choice merely for notational convenience and conceptual clarity throughout security arguments. This definition is equivalent to an alternate version where the server chooses the program or where the program is fixed in advance. These equivalences can be seen by viewing the program P as a universal RAM program that reads instructions from either the server's database y or the client's private database x (or both).

Remark 3.3 (Known upper bound on P)**.** Implicit in our formulation of the RAM model, we assume that the client knows an a priori upper bound T on the run time of P. In the weakly efficient setting, this is unavoidable because the client's run time is proportional to T. In the strongly efficient setting, we can set $T = 2^{\omega(\log \lambda)}$ to be super-polynomial without affecting the efficiency of our scheme. However, simply setting T this way requires us to assume slightly super-polynomial security of the underlying assumptions because the security reduction has a security loss linear in T.

It is possible to generically transform the scheme to avoid this super-polynomial security loss using a trick from [AL18]. The client samples a sequence of keys $\mathsf{sk}_4, \mathsf{sk}_8, \ldots, \mathsf{sk}_{2^{\omega(\log \lambda)}}$ for a symmetric-key encryption scheme, then it produces a sequence of LFE ciphertexts $\mathsf{ct}_2, \mathsf{ct}_4, \ldots, \mathsf{ct}_{2^{\omega(\log \lambda)}}$ respectively encrypting the pairs $(P_2, x), (P_4, x), \ldots, (P_{2^i}, x), \ldots, (P_{2^{\omega(\log \lambda)}}, x)$, where the program P_{2^i} runs P for 2^i steps. If P halts in the first 2^i steps, P_{2^i} halts and returns the output of P, otherwise P_{2^i} returns $\mathsf{sk}_{2^{i+1}}$. The final ciphertext consists of symmetric-key encryptions of each of the LFE ciphertexts after the first under corresponding keys, that is, $(\mathsf{ct}_2, \mathsf{Enc}_{\mathsf{sk}_4}(\mathsf{ct}_4), \ldots, \mathsf{Enc}_{\mathsf{sk}_{2^i}}(\mathsf{ct}_{2^i}), \ldots, \mathsf{Enc}_{\mathsf{sk}_{2^{\omega(\log \lambda)}}}($
The server can then work its way up the chain using LFE decryption to learn exactly the secret keys necessary to recover the ciphertext that can fully evaluate P.

Remark 3.4 (Larger output lengths)**.** The above definition restricts the client's program to having single bit output. We can somewhat ease this restriction generically by permitting RAM programs P with longer output lengths. However, in that case, the step circuit (and hence the description size $|P|$) must necessarily have size that is at least linear in $|z|$. Therefore we incur an additional cost of $|z| \cdot T$ in the server's decryption time. Later in Remark 5.1 we sketch how to achieve only additive cost in $|z|$, but this transformation will require going under the hood of our construction.

4 RAM-LFE with Unprotected Memory and Access

As a stepping stone toward RAM-LFE, we define and construct a notion of RAM-LFE with a weaker security guarantee which we call RAM-LFE with unprotected memory and access. In this notion, the client's long input x is not hidden, and neither is the memory access pattern the program makes to x and y throughout the computation. Instead we're only attempting to hide the short RAM program description P. We give the formal definition below.

Definition 4.1 (RAM-LFE with Unprotected Memory and Access)**.** *A RAM-LFE with unprotected memory and access (UMA RAM-LFE) is a tuple of algorithms* $(\mathsf{LFE.Gen}, \mathsf{LFE.Hash}, \mathsf{LFE.Enc}, \mathsf{LFE.Dec})$ *that satisfies the same syntax and correctness properties as in Definition 3.1, but only satisfies the following weaker notion of security: There exists a PPT algorithm* $\mathsf{Sim}_{\mathsf{UMA}}$ *such that for any PPT adversary* $\mathcal{A}$*, we have*

$$\left|\Pr\left[\mathsf{Real}^{\mathsf{UMA}}_{\mathsf{LFE}}(1^\lambda) = 1\right] - \Pr\left[\mathsf{Ideal}^{\mathsf{UMA}}_{\mathsf{LFE}}(1^\lambda) = 1\right]\right| = \mathrm{negl}(\lambda),$$

where the experiment $\mathsf{Real}^{\mathsf{UMA}}_{\mathsf{LFE}}$ *is the same as* $\mathsf{Real}_{\mathsf{LFE}}$*, and* $\mathsf{Ideal}^{\mathsf{UMA}}_{\mathsf{LFE}}$ *is the same as* $\mathsf{Ideal}_{\mathsf{LFE}}$ *except for the third step, which is changed to* $3'$ *as follows:*

3′. $\mathsf{ct} \leftarrow \mathsf{Sim}_{\mathsf{UMA}}(\mathsf{crs}, x, y, P(x,y), t, \mathsf{md}, \mathsf{MemAccess})$ *where* t *is the run time of* $P(x,y)$, $\mathsf{md} = \mathsf{md}(P)$ *and* $\mathsf{MemAccess} = (i_x^t, v_x^t, i_y^t, v_y^t, j_x^t, w^t)_{t\in[T]}$ *is the memory access pattern of* $P(x,y)$.

It turns out that full simulation based UMA security is not achievable in the full efficiency setting. To get around this, we define an analogous version of UMA security that is indistinguishability based as follows.

Definition 4.2 (RAM-LFE with Indistinguishability-based UMA Security)**.** *A* RAM-LFE with indistinguishability based UMA security *(indUMA RAM-LFE) is a tuple of algorithms* (LFE.Gen, LFE.Hash, LFE.Enc, LFE.Dec) *that satisfies the same syntax and correctness properties as in Definition 3.1, but only satisfies the following game-based security notion. For all PPT adversaries* $\mathcal{A}$*,*

$$\left|\Pr[\mathrm{Expt}^{\mathcal{A}}_{\mathsf{indUMA}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Expt}^{\mathcal{A}}_{\mathsf{indUMA}}(1^\lambda, 1) = 1]\right| \leq \mathrm{negl}(\lambda),$$

where the experiment $\mathrm{Expt}^{\mathcal{A}}_{\mathsf{indUMA}}(1^\lambda, b)$ *is defined as follows:*

1. *The challenger samples* $\mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda)$ *and sends it to* $\mathcal{A}$*.*

2. $\mathcal{A}$ *chooses the public database* y *and a pair of private inputs* (P_0, x_0), (P_1, x_1) *and sends them to the challenger.*

3. *The challenger checks that* $\mathsf{md}(P_0) = \mathsf{md}(P_1)$, *that* $P_0(x_0, y) = P_1(x_1, y)$ *and that each execution runs for the same number of time steps. Then it additionally checks that* $x_0 = x_1$ *and both computations access memory in an identical manner (i.e. each program would write the same bit to the same location in every step, and similar for reads).*

4. *If either of the checks fail, the challenger aborts the experiment; otherwise, it computes* $(\mathsf{dig}, \widetilde{y}) := \mathsf{LFE.Hash}(\mathsf{crs}, y)$ *and the ciphertext* $\mathsf{ct}_b \leftarrow \mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x_b)$ *and sends it to* $\mathcal{A}$*.*

5. $\mathcal{A}$ *outputs* b'*. The output of the experiment is* b'*.*

In this section, we give two constructions of RAM-LFE with unprotected memory and access. In Section 4.1, we construct a scheme that is weakly efficient and satisfies simulation-based UMA security. Then in Section 4.2, we show how combining the same techniques and additionally using indistinguishability obfuscation can yield a fully efficient RAM-LFE but only with indistinguishability-based UMA security.

4.1 UMA RAM-LFE with Weak Efficiency

In the weak efficiency setting where we allow the running time of LFE.Enc and the size of the ciphertext to scale proportionally to the running time of the RAM program, we can build UMA secure RAM-LFE based on laconic oblivious transfer and garbled circuits. Our construction is adapted from the construction of non-interactive secure computation for RAM computation (RAM-NISC) from [CDG+17], so we begin by summarizing it below.

The [CDG$^+$17] RAM-NISC. The setting of RAM-NISC is very similar to that of (UMA) RAM-LFE. A RAM NISC is a protocol between a server that holds a large database private y and a client that holds a RAM program P (potentially with short private data hardcoded into it). Like RAM-LFE, the protocol consists of two messages: first the server deterministically preprocesses the database y into a digest dig and preprocessed database $\widetilde{y}$, then the client can use dig to encrypt P into a ciphertext ct that can be decrypted by the server to recover $P(y)$. Client security states that the server learns nothing more than $P(y)$ and its memory access pattern $\mathsf{MemAccess}$. The main differences from RAM-NISC and our model of RAM-LFE are that the [CDG$^+$17] RAM-NISC does not permit random accesses to a client input database x and it additionally requires security for the server's database.

The construction of [CDG$^+$17] works by combining updatable laconic oblivious transfer (LOT), standard oblivious transfer (OT) and garbled circuits. At a high level the construction works as follows.

- The server uses the LOT hash function to compute dig and $\widetilde{y}$. For each bit in dig, the server sends a OT receiver message to the client.
- To encrypt P, the client garbles each of the T-many step circuits that may need to be evaluated to compute $P(\cdot)$. However, before each step circuit is garbled, it is modified in the following way so that it hides the intermediate states of the computation.
  - Instead of outputting the local state, it outputs labels corresponding to that state to be used as the input to the next garbled step circuit.
  - When the step circuit would read from the database in position i, it instead outputs an LOT message with respect to dig encrypting the label corresponding to the value of the bit $y[i]$.
  - When the step circuit would write to the database, it instead outputs an LOT update message along with the labels for the updated LOT digest.

  The client sends these garbled circuits along with the labels for the first step circuit and OT sender messages encrypting the labels corresponding to the bits of dig.
- The server can then evaluate $P(y)$ by evaluating each garbled step circuit in sequence. Any time the server needs to access at a location i in y, it will use the LOT message from the step circuit to recover (only) the garbled circuit labels corresponding to $y[i]$.

In order to adapt this construction to our setting, we make the following modifications. First, we observe that, since we don't require security for the server's input y, we can remove the OT and have the server simply send dig to the client. Second, we can use LOT to permit random access to the client's input x in addition to y. While this modification does reveal x to the server, this is fine for UMA security, and we note that any secret information stored in the description of P will still be hidden from the server.

Theorem 4.3 (Adapted from [CDG$^+$17], Section 6)**.** *Assuming the existence of updatable laconic oblivious transfer, there exists a UMA RAM-LFE with the following efficiency properties:*

- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ *runs in time* $|y| \cdot \mathrm{poly}(\lambda, \log |y|)$
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$ *runs in time* $T \cdot \mathrm{poly}(\lambda, |P|) + |x| \cdot \mathrm{poly}(\lambda, \log |x|)$

- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ *runs in time* $t \cdot \mathrm{poly}(\lambda, |P|)$

where T is the maximum running time of P, and t is the actual running time of $P(x, y)$.

Proof sketch. After performing the modifications to the [CDG+17] construction as described above, we obtain the following construction of UMA RAM-LFE.

- $\mathsf{LFE.Gen}(1^\lambda)$: Sample an LOT CRS corresponding to each database x and y and output $\mathsf{crs} = (\mathsf{crs}_x, \mathsf{crs}_y)$.
- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$: Run the LOT hash on the server's database y, and output the resultant digest dig_y and preprocessed database $\widetilde{y}$.
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}_y, P, x)$: First run the LOT hash on the client's database x, obtaining the digest dig_x and preprocessed state $\hat{x}$. Then garble the step circuits of P as in [CDG+17], handling accesses to x in an analogous way as accesses to y. Output a ciphertext ct containing the garbled step circuits and preprocessed state $\hat{x}$.
- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$: Evaluate the garbled circuits step by step with the labels from decrypting the LOT messages (for both x and y) that are output by the previous step circuit. Ultimately recover the RAM program output z from the final step circuit.

It is easy to see that the modifications we made to the [CDG+17] construction do not affect the hiding property on P. We provide a sketch of the proof of security here and refer the reader to [CDG+17] for the full details.

It suffices to define a simulator that when given the output of $P(x, y)$ along with the databases x, y and the memory access pattern $\mathsf{MemAccess}$, outputs a simulated sequence of garbled step circuits. Since the simulator is give the databases and the access pattern, it can use the LOT send and update simulators to simulate the LOT messages output by each garbled step circuit over the course of the computation. Then, using the garbled circuit simulator and the final output of the computation, we can produce a sequence of simulated garbled step circuit that output the simulated LOT messages and labels that eventually lead to the final circuit outputting $P(x, y)$.

We argue that this simulated ciphertext distribution is indistinguishable from the real world ciphertext distribution over a sequence of hybrid experiments where we gradually switch each garbled step circuit to a simulated one, one at a time beginning with the first. The indistinguishability of consecutive hybrids follows from the garble circuit security and LOT sender security. $\square$

4.2 UMA RAM-LFE with Full Efficiency

In this section we show how to build on the techniques of the previous section to build UMA RAM-LFE with full efficiency where the running time of $\mathsf{LFE.Enc}$ scales only polylogarithmically with the worst case run time of the RAM program P. However, we recall that simulation based security is not achievable in this setting, so we instead construct UMA RAM-LFE with indistinguishability based security (see Definition 4.2). Our construction relies on updatable laconic OT and garbled circuits as before, along with the additional assumption of indistinguishability obfuscation (iO). Our construction is adapted from the construction of UMA-secure garbled RAM (UMA GRAM) from [ACFQ22], which we summarize below.

The [ACFQ22] UMA GRAM. The construction of UMA GRAM from [ACFQ22] is in the model of garbled RAM, which is slightly different than our model of RAM-LFE. In the GRAM setting, the client holds a database x and RAM program P. The client first processes the database into some garbled database $\hat{x}$ and stores some associated secret key sk. Then the client can use sk to garble the program. A server can then recover $P(x)$ by evaluating the garbled program on the garbled database. In the UMA model of security, the server learns nothing about P other than its access pattern and output when evaluated on x.

At its core, the [ACFQ22] GRAM construction follows the same rough template as the [CDG+17] RAM-NISC (see Section 4.1): The database is compressed into a digest using the LOT hash, then the RAM program is encrypted and evaluated as a sequence of garbled step circuits that pass along labels for database accesses through LOT send and update messages. The main conceptual difference here is the use of iO to reduce the encryption time and compress the ciphertext. Instead of garbling each step circuit individually, the encryptor outputs an obfuscated program that, when evaluated on a step t, outputs the t-th garbled step circuit. At a high level the construction works as follows.

- To garble the database x, the client samples an LOT CRS crs and uses the LOT hash function to compute a digest dig and preprocessed database $\hat{x}$. It then outputs $\hat{x}$ as the garbled database and $\mathsf{sk} = (\mathsf{crs}, \mathsf{dig})$ as the secret key.
- To garble the program P, the client generates a "program generator" circuit PG that, on input a step $t \in [T]$, outputs the t-th garbled step circuit that has been modified in the same way as in [CDG+17] (see Section 4.1). The program generator PG uses a (puncturable) PRF evaluated on its input as the source for the randomness of the circuit garbling. The client outputs the obfuscation $\widetilde{\mathsf{PG}} = i\mathcal{O}(\mathsf{PG})$ as the garbled program.
- Finally, to evaluate the garbled program, in each step the server runs the obfuscated program $\widetilde{\mathsf{PG}}$ to recover the next garbled step circuit to evaluate, and uses the LOT to handle database accesses.

In order to cast the [ACFQ22] UMA GRAM as a UMA RAM-LFE, we need only make a few modifications. First, we observe that in the UMA setting, all of the information stored in the garbled database secret key can actually be made public. Indeed, LOT security holds when the CRS is public, and the digest dig can be computed deterministically from the CRS and x. For this reason, we can have the server also produce a "garbled" database and send its digest to the client so that it can garble the program. This leads naturally into our second modification that, as in the previous section, we can modify the protocol to handle both client and server databases x and y.

Theorem 4.4 (Adapted from [ACFQ22], Theorem 5.6)**.** *Assuming the existence of updatable laconic oblivious transfer and* $i\mathcal{O}$ *with log-sized inputs, there exists a UMA RAM-LFE scheme* $\mathsf{LFE} = (\mathsf{LFE.Gen}, \mathsf{LFE.Hash}, \mathsf{LFE.Enc}, \mathsf{LFE.Dec})$ *with the following efficiency:*

- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ *runs in time* $|y| \cdot \mathrm{poly}(\lambda, \log|y|)$
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$ *runs in time* $\mathrm{poly}(\lambda, |P|) + |x| \cdot \mathrm{poly}(\lambda, \log|x|)$
- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ *runs in time* $t \cdot \mathrm{poly}(\lambda, |P|)$

where t is the running time of $P(x, y)$.

Proof Sketch. After performing the modifications to the [ACFQ22] UMA GRAM construction as described above, we obtain the following construction of UMA RAM-LFE with full efficiency.

- $\mathsf{LFE.Gen}(1^\lambda)$: Sample an LOT CRS corresponding to each database x and y and output $\mathsf{crs} = (\mathsf{crs}_x, \mathsf{crs}_y)$.
- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$: Run the LOT hash on the server's database y using crs_y, and output the resultant digest dig_y and preprocessed database $\widetilde{y}$.
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{d}_y, P, x)$: First, run the LOT hash on the client's database x using crs_x, obtaining digest dig_x and state $\hat{x}$. Then construct the program generator circuit PG as in [ACFQ22], hardcoding inside of it both digests dig_x and dig_y, as well as a puncturable PRF key. The program generator PG handles accesses to y in an analogous manner to accesses to x. Finally, output the preprocessed client database $\hat{x}$, the obfuscation of the program generator $\widetilde{\mathsf{PG}} = \mathsf{i}\mathcal{O}(\mathsf{PG})$ as well as labels for the first step circuit.
- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{c})$: For each $t \in [T]$, evaluate the obfuscated the program generator to recover the t-th garbled step circuit. Then evaluate the step circuit and use the LOT to recover the labels corresponding to the memory accesses to x and y. Ultimately recover the RAM program output z from the final step circuit.

It is clear that the modifications we described above do not affect the security of the client's chosen RAM program P. We modified the protocol so that the LOT CRS for the client's input x is revealed to the adversary rather than being hidden in the garbled database secret key. However, that CRS is only used in performing LOT operations, and LOT sender and update security still holds when the CRS is made public. Therefore the security argument from [ACFQ22], simply adjusted to account for the two independent LOT schemes for x and y, will still apply to prove security of our construction. We briefly outline the proof of security below, and we refer the reader to [ACFQ22] for the full details.

To show security of the construction it suffices to show that an encryption of (P_0, x) under the digest of some y is indistinguishable from the encryption of (P_1, x) under the same digest, so long as $P_0(x, y)$ and $P_1(x, y)$ have the same output, running time and memory access pattern. We begin by switching to a hybrid world where the obfuscated program generator has both P_0 and P_1 hardcoded into it, however it always outputs garbled step circuits for P_0. Then we will gradually switch the program generator to output garbled step circuits for P_1 by hardcoding randomness in the generator for specific steps of the computation using the puncturable PRF, and appealing to the security of the garbled circuit and LOT schemes. This strategy of hardcoding at specific steps of the computation must be done carefully via a pebbling argument so as to avoid hardcoding too much data at any one time. □

5 Upgrading to Full Security

In this section we upgrade the UMA RAM-LFE constructions of the previous section to achieve full security. We consider the weak efficiency and full efficiency setting separately.

5.1 The Weak Efficiency Case

In this section we show how to upgrade the weakly efficient UMA RAM-LFE from Section 4.1 to full security. Recall that in the weak efficiency setting we allow the client's run time to be proportional to the running time of P. In this case we can upgrade from UMA RAM-LFE by hiding the program's memory and access pattern by additionally using symmetric-key encryption, DEPIR and ORAM, all of which follow from RingLWE.

At a high level, we will use a symmetric-key encryption scheme to encrypt the client's database together with the memory contents, and use ORAM and DEPIR to hide the memory address in the access pattern. Simulation security of the construction will follow from CPA-security of the encryption scheme and the simulation security of UMA RAM-LFE and ORAM and security of the DEPIR.

Theorem 5.1. *Assuming the existence of a (simulation-secure) UMA RAM-LFE and DEPIR, there exists a fully-secure RAM-LFE scheme. In particular, assuming the RingLWE assumption holds there exists a weakly efficient RAM-LFE scheme with the following efficiency properties: for any constant* $\epsilon > 0$

- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ *runs in time* $|y|^{1+\epsilon} \cdot \mathrm{poly}(\lambda)$
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$ *runs in time* $T \cdot \mathrm{poly}(\lambda, |P|) + |x| \cdot \mathrm{poly}(\lambda, \log|x|, \log T)$
- $\mathsf{LFE.Dec}(\mathsf{crs}, y, \mathsf{ct})$ *runs in time* $t \cdot \mathrm{poly}(\lambda, |P|)$

where T *is the maximum running time of* P*, and* t *is the actual running time of* $P(x, y)$*.*

Construction. We construct a fully-secure RAM-LFE scheme with weak efficiency. We use the following building blocks

- A CPA-secure symmetric-key encryption scheme $\mathsf{SKE} = (\mathsf{SKE.Gen}, \mathsf{SKE.Enc}, \mathsf{SKE.Dec})$
- A DEPIR scheme: $\mathsf{DEPIR} = (\mathsf{Prep}, \mathsf{Query}, \mathsf{Dec})$
- An ORAM scheme: $\mathsf{ORAM} = (\mathsf{ORAM.Setup}, \mathsf{ORAM.Access})$
- A UMA RAM-LFE scheme uLFE = (uLFE.Gen, uLFE.Hash, uLFE.Enc, uLFE.Dec)

At a high level our construction proceeds as follows. We use the encryption scheme to encrypt the client's input x bit by bit. Then we compile P into an oblivious program P^* that accesses x under the encryption and hides the access pattern of P by accessing x through ORAM and y through DEPIR. Because making one logical access via DEPIR or ORAM takes multiple physical accesses, each logical step in evaluating P will be compiled into η steps in P^* where η is the number of physical accesses it takes to make an ORAM or DEPIR query. For convenience we assume that the value η is the same for the ORAM and DEPIR schemes. This is without loss of generality because we can simply pad the shorter one with dummy accesses without affecting security. We additionally note that the encryption scheme, the ORAM and the DEPIR all necessarily require randomness for security. We provide this by hardcoding randomness into P^* to serve as the random tapes. We formally define our $\mathsf{RAM\text{-}LFE} = (\mathsf{LFE.Gen}, \mathsf{LFE.Hash}, \mathsf{LFE.Enc}, \mathsf{LFE.Dec})$ as follows:

- $\mathsf{LFE.Gen}(1^\lambda)$ does the same as $\mathsf{uLFE.Gen}$, generating $\mathsf{crs} = (\mathsf{crs}_x, \mathsf{crs}_y)$ for the two LOT scheme.

- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ outputs the digest of DEPIR preprocessed database.
  1. Run DEPIR preprocessing $\hat{y} := \mathsf{Prep}(1^\lambda, y)$.
  2. Compute $(\mathsf{dig}, \widetilde{y}) := \mathsf{uLFE.Hash}(\mathsf{crs}_y, \hat{y})$ and output it.
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$:
  1. Sample a secret key $\mathsf{sk} \leftarrow \mathsf{SKE.Gen}(1^\lambda)$ and encrypt the database x to get $x' \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, x)$. Then run $(\mathsf{ck}_0, x^*) \leftarrow \mathsf{ORAM.Setup}(1^\lambda, x', |x|+T)$. Here T is the maximum running time of $P(x, \cdot)$.
  2. Sample PRFs F_O, F_D, F_E. Construct the oblivious program P^* as described in Algorithm 5.2, hardcoding the program P, the encryption key sk, the ORAM client key ck along with the PRFs.
  3. Run $\mathsf{ct} \leftarrow \mathsf{uLFE.Enc}(\mathsf{crs}, \mathsf{dig}, P^*, x^*)$ and output ct to the server.
- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ simply evaluates $\mathsf{uLFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ and outputs the result.

Algorithm 5.2: The oblivious program P^*

Hardcoded: The step circuit C for P, secret key sk for symmetric-key encryption, an ORAM client secret key ck_0, pseudorandom functions F_O, F_E, F_D.

Inputs: The encrypted database x^* and the preprocessed database $\widetilde{y}$

Algorithm: Initialize $\mathsf{st}^{*,0} = (\mathsf{st}_0, \mathsf{ck}_0), v_x^0 = 0, v_y^0 = 0$, where st_0 is the initial state for the first step circuit C of P. For $\tau \in [T]$, run the following:

1. Parse $\mathsf{st}^{*,\tau-1} = (\mathsf{st}^{\tau-1}, \mathsf{ck}^{\tau-1})$. Run $(\mathsf{st}^\tau, i_x^\tau, i_y^\tau, j_x^\tau, w^\tau) := C(\mathsf{st}^{\tau-1}, v_x^{\tau-1}, v_y^{\tau-1})$.
2. Encrypt $w_\mathsf{e}^\tau \leftarrow \mathsf{SKE.Enc}(\mathsf{sk}, w^\tau)$, and $F_E(\tau)$ is the randomness used for encryption. Then run the following two ORAM access in order as a subroutine:
   (a) Run $(\mathsf{ck}_1^{\tau-1}, \perp) \leftarrow \mathsf{ORAM.Access}^{x^*}(\mathsf{ck}^{\tau-1}, \mathsf{write}, j_x^\tau, w_\mathsf{e}^\tau)$.
   (b) Run $(\mathsf{ck}^\tau, v_{x^*}^\tau) \leftarrow \mathsf{ORAM.Access}^{x^*}(\mathsf{ck}_1^{\tau-1}, \mathsf{read}, i_x^\tau, \perp)$.

   Where here we use $F_O(\tau)$ as the random tape used in the $\mathsf{ORAM.Access}$ function.
3. Run DEPIR encryption $(Q^\tau, s^\tau) \leftarrow \mathsf{Query}(1^\lambda, |y|, i_y^\tau)$, and make read access to $\hat{y}$ at the locations in set Q^τ, obtaining $V_{\hat{y}}^\tau = \{\hat{y}[q^\tau] : q^\tau \in Q^\tau\}$. Here we use $F_D(\tau)$ as the randomness used for DEPIR query.
4. Decrypt $v_x^\tau := \mathsf{SKE.Dec}(\mathsf{sk}, v_{x^*})$ and $v_y^\tau := \mathsf{Dec}(s^\tau, V_{\hat{y}}^\tau)$. Update $\mathsf{st}^{*,\tau} = (\mathsf{st}^\tau, \mathsf{ck}^\tau)$

Proof. We prove that the above construction gives us a fully-secure RAM-LFE scheme with weak efficiency. We give the following proofs of correctness, efficiency, and full security.

Correctness. The proof of correctness follows from the correctness of the UMA RAM-LFE and the correctness of constructing P^*, and, in turn, the correctness of P^* follows from the correctness of the DEPIR, ORAM and SKE schemes.

Efficiency. The running time of $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ is dominated by the DEPIR preprocessing which runs in time $|y|^{1+\epsilon} \cdot \mathrm{poly}(\lambda, \log |y|)$

The running time of $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$ is dominated by the time it takes to initialize an ORAM of size $|x| + T$ and run $\mathsf{uLFE.Enc}$ for P^*. Thus the encryption time is given by $|x| \cdot \mathrm{poly}(\lambda, \log(|x| + T)) + \eta \cdot T \cdot \mathrm{poly}(\lambda, \eta, \log |x|, \log |\hat{y}|, \log T)$. Since $\eta = \mathrm{poly}\log(|x|, |y|)$, we have encryption time $|x| \cdot \mathrm{poly}(\lambda, \log |x|, \log T) + T \cdot \mathrm{poly}(\lambda, \log |x|, \log |y|, \log T)$.

The $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ decryption time follows from the efficiency in UMA RAM-LFE, which is given by $t \cdot \mathrm{poly}(\lambda, \eta, \log |x|, \log |\hat{y}|) = t \cdot \mathrm{poly}(\lambda, \log |x|, \log |y|, \log T)$, where t is the running time of $P(x, y)$.

Security. To argue security, we begin by defining a sequence of hybrids.

- $\mathsf{Hyb}_0 = \mathsf{Real}_{\mathsf{LFE}}(1^\lambda)$ corresponds to the real world experiment as defined in Definition 3.1.
- Hyb_1 is the same as $\mathsf{Real}_{\mathsf{LFE}}(1^\lambda)$ except for the third step. In Hyb_1, ct will be generated by

  $$\mathsf{ct} \leftarrow \mathsf{Sim}_{\mathsf{UMA}}(\mathsf{crs}, x^*, \hat{y}, z, t', \mathsf{md}, \mathsf{MemAccess})$$

  where $z = P(x, y) = P^*(x^*, \hat{y})$ according to correctness, t' is the run time of $P^*(x^*, \hat{y})$, $\mathsf{md} = \mathsf{md}(P^*)$, and according to our definition, $t' = \eta \cdot T'$, where T' is the running time of $P(x, y)$. $\mathsf{MemAccess} = (i^t_{x^*}, v^t_{x^*}, i^t_{\hat{y}}, v^t_{\hat{y}}, j^t_{x^*}, w^t_{*})_{t \in [\eta T']}$ is the memory access pattern of $P^*(x^*, \hat{y})$.
- Hyb_2 is the same as Hyb_1 except we replaces the memory accesses $\mathsf{MemAccess}$ to the memory accesses that would be made by $P^*(x^*, \hat{y})$ if we replace the hardcoded PRFs F_O, F_E, F_D with truly random function R_O, R_E, R_D.
- Hyb_3 is the same as Hyb_2 except that we replace $(x^*, \mathsf{MemAccess})$ with

  $$(\mathsf{Sim}_{x^*}(1^{|x^*|}, 1^{\eta T}, \{i^t_{x^*}, j^t_{x^*}\}_{t \in [\eta T']}), (i^t_{\hat{y}}, v^t_{\hat{y}})_{t \in [\eta T']})$$

  where $\mathsf{Sim}_{x^*}(1^{|x^*|}, 1^{\eta T'}, \{i^t_{x^*}, j^t_{x^*}\}_{t \in [\eta T']})$ is defined as follows:

  1. Run uniformly random sampling $x^* \leftarrow [0,1]^{|x^*|}$ and $\{w^t\}_{t \in [\eta T']} \leftarrow [0,1]^{\eta T'}$
  2. Run memory access on x^* by the modified memory access pattern $(i^t_{x^*}, j^t_{x^*}, w^t)_{t \in [\eta T']}$, updating the database x^* to x^*_t after step t. Define $v^t_{x^*}$ as the $i^t_{x^*}$ entry of database x^*_t. This will make the memory access pattern consistent.
  3. Output $(x^*, (i^t_{x^*}, v^t_{x^*}, j^t_{x^*}, w^t)_{t \in [\eta T']})$.

  That is, we simulate the memory contents of x^* and ORAM memory.
- Hyb_4 is the same as Hyb_3 except that we replace $\{(i^t_{\hat{y}}, v^t_{\hat{y}})\}_{t \in [\eta\tau - \eta + 1, \eta\tau]}$ with $\{(i^\tau, \hat{y}[i^\tau]) : i^\tau \in Q^\tau_0\}$, $\forall \tau \in [T']$. Here Q^τ_0 is a set that is generated by $(Q^\tau_0, s^\tau) \leftarrow \mathsf{Query}(1^\lambda, |y|, 1)$, which is a dummy DEPIR query set on a fixed index.
- Hyb_5 is the same as Hyb_4 except that we replace $(i^t_{x^*}, j^t_{x^*})_{\tau \in [\eta\tau - \eta + 1, \eta\tau]}$ with $\mathsf{ORAMSim}(1^\lambda, |x|)$, for all $\tau \in [T']$. We can see that Hyb_5 is a fully simulated world, so we define $\mathsf{LFESim}(\mathsf{crs}, y, P(x, y), t, \mathsf{md}(P))$ as the following, and observe $\mathsf{Hyb}_5 = \mathsf{Ideal}_{\mathsf{LFE}}(1^\lambda)$.

  1. Run $\hat{y} := \mathsf{Prep}(1^\lambda, y)$.

2. Run $(i^t_{x^*}, j^t_{x^*})_{t\in[\eta\tau-\eta+1,\eta\tau]} \leftarrow \mathsf{ORAMSim}(1^\lambda, |x|)$ for all $\tau \in [T']$. Then generate Q^τ_0 and $\{(i^\tau, \hat{y}[i^\tau]) : i^\tau \in Q^\tau_0\}, \forall \tau \in [T']$ as in Hyb_3. Parse them as $(i^t_{\hat{y}}, v^t_{\hat{y}})_{t\in[\eta T']}$.
3. Run $(x^*, (i^t_{x^*}, v^t_{x^*}, j^t_{x^*}, w^t)_{t\in[\eta T']}) \leftarrow \mathsf{Sim}_{x^*}(1^{|x^*|}, 1^{\eta T'}, \{i^t_{x^*}, j^t_{x^*}\}_{t\in[\eta T']})$.
4. Output $\mathsf{ct}' \leftarrow \mathsf{Sim}_{\mathsf{UMA}}(\mathsf{crs}, x^*, \hat{y}, P(x,y), t', \mathsf{md}(P^*), \mathsf{MemAccess}_{\mathsf{Sim}})$, where $\mathsf{MemAccess}_{\mathsf{Sim}} = (i^t_{x^*}, v^t_{x^*}, i^t_{\hat{y}}, v^t_{\hat{y}}, j^t_{x^*}, w^t)_{t\in[\eta T']}$.

We show the indistinguishability between the hybrid experiments.

- $\mathsf{Hyb}_0 \approx_c \mathsf{Hyb}_1$ follows from the simulation security of UMA RAM-LFE, as defined in 4.1.
- $\mathsf{Hyb}_1 \approx_c \mathsf{Hyb}_2$ follows from the pseudorandomness of the PRFs F_O, F_E and F_D.
- $\mathsf{Hyb}_2 \approx_c \mathsf{Hyb}_3$ follows from the security of CPA-secure private-key encryption scheme, which gives that $\mathsf{SKE.Enc}(\mathsf{sk}, m)$ is indistinguishable from a uniformly random string that has the same length as $\mathsf{SKE.Enc}(\mathsf{sk}, m)$.
- $\mathsf{Hyb}_3 \approx_c \mathsf{Hyb}_4$ follows from the security of DEPIR, which gives that queries on indices (which can also be the same) are computationally indistinguishable.
- $\mathsf{Hyb}_4 \approx_c \mathsf{Hyb}_5$ follows from the standard simulation-based security of ORAM.

□

Remark 5.1 (Larger outputs with additive cost)**.** The RAM-LFE we constructed above only handles RAM programs that have one-bit output. Here we briefly sketch how to remove this constraint while only incurring an additive cost in the output length, however we omit the precise details for the sake of brevity. We notice that our construction can actually be modified to allow the client to send a batch of multiple encrypted programs for the server to evaluate *in sequence* on the same inputs (and hence consistent working memory). This can be done by having the final garbled step circuit output one bit in the clear in addition to the labels that would be required to begin evaluating another garbled circuit on the updated database. Using this observed property, the client can construct a program P that writes its desired output to working memory and then follow that program with a sequence of programs that simply read one bit each from memory and output it. In this way, the server can decrypt the output z in time proportional to $T + |z|$.

5.2 The Full Efficiency Case

In this section we construct fully efficient RAM-LFE by upgrading the fully efficient UMA RAM-LFE from Section 4.2. Ultimately the construction we use will be conceptually very similar to that of the weakly efficient case in the previous section: we encrypt the client's database and hide the access patterns to x and y under an ORAM and DEPIR respectively. However, because Theorem 4.4 only gives indistinguishability-based UMA security in the full efficiency setting, the security proof for this construction requires significantly more care. We use the techniques of punctured programming and follow the outline of [CH16], working in two steps. In the first step we upgrade to a security notion where we hide the content of the client's database and memory accesses, but where the addresses accessed are still unprotected. Then we show how to go from this intermediate notion to full (simulation-based) RAM-LFE security.

5.2.1 Hiding Database Content

Definition 5.3 (RAM-LFE with Unprotected Access)**.** *A* RAM-LFE with unprotected access *(UA RAM-LFE) is a tuple of algorithms* ($\mathsf{LFE.Gen}$, $\mathsf{LFE.Hash}$, $\mathsf{LFE.Enc}$, $\mathsf{LFE.Dec}$) *that satisfies the same syntax and correctness properties as in Definition 3.1, but only satisfies the following weaker notion of security. For all PPT adversaries $\mathcal{A}$,*

$$\left|\Pr[\mathrm{Expt}^{\mathcal{A}}_{\mathsf{UA}}(1^\lambda, 0) = 1] - \Pr[\mathrm{Expt}^{\mathcal{A}}_{\mathsf{UA}}(1^\lambda, 1) = 1]\right| \leq \mathrm{negl}(\lambda),$$

where the experiment $\mathrm{Expt}^{\mathcal{A}}_{\mathsf{UA}}(1^\lambda, b)$ is defined as follows:

1. *The challenger samples* $\mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda)$ *and sends it to $\mathcal{A}$.*
2. *$\mathcal{A}$ chooses the public database y and a pair of private inputs (P_0, x_0), (P_1, x_1) and sends them to the challenger.*
3. *The challenger checks that $\mathsf{md}(P_0) = \mathsf{md}(P_1)$, that $P_0(x_0, y) = P_1(x_1, y)$ and that each execution runs for the same number of time steps. Then it only checks that both computations access the same sequence of memory* addresses *(i.e. the locations read from and written to, not the content).*
4. *If either of the checks fail, the challenger aborts the experiment; otherwise, it computes $(\mathsf{dig}, \widetilde{y}) := \mathsf{LFE.Hash}(\mathsf{crs}, y)$ and the ciphertext $\mathsf{ct}_b \leftarrow \mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P_b, x_b)$ and sends it to $\mathcal{A}$.*
5. *$\mathcal{A}$ outputs b'. The output of the experiment is b'.*

Theorem 5.4. *Assuming the existence of an indistinguishability-based UMA RAM-LFE, there exists an indistinguishability-based UA RAM-LFE. In particular, assuming the existence of updatable laconic oblivious transfer and indistinguishability obfuscation, there exists a UA RAM-LFE with the following efficiency:*

- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ *runs in time* $|y| \cdot \mathrm{poly}(\lambda, \log|y|)$
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$ *runs in time* $\mathrm{poly}(\lambda, |P|) + |x| \cdot \mathrm{poly}(\lambda, \log|x|, \log T)$
- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ *runs in time* $t \cdot \mathrm{poly}(\lambda, |P|)$,

where T is the maximum running time of P and t is the actual running time of $P(x, y)$.

Construction. Our two building blocks are an indistinguishability-based UMA RAM-LFE uLFE and a puncturable PRF family $\mathcal{F}$. At a high level, the construction works by compiling the clients input (P, x) into an encrypted database x' and a transformed program P' that operates over the encrypted x', but leaves all of its accesses to y unchanged. Then we simply use $\mathsf{uLFE.Enc}$ on (P', x') to produce the final ciphertext. We make the following modifications to compile P into P'. Without loss of generality, we assume that P has memory cell alphabet $\Sigma = \{0, 1\}$. First, P' executes two copies of P in parallel, on two independent "tracks" of read/write memory which we call the left and right tracks; at the end it will only output the result of the left-track computation, meaning the right track is only used as a "dummy" computation. Second, each time P' would write to memory, it additionally writes metadata containing the time step this memory cell was last written to. We model these changes by setting the alphabet of P' to $\Sigma' = [T] \times \{0, 1\}^2$, so each memory cell of x contains a tuple (t, u, v) where $t \in [T]$ is a time step and u and v are the values in the left and right track of memory respectively at that location. Finally, we also modify P' to use a pair of (puncturable) PRFs to mask each bit it writes to memory. We formally define the encryption algorithm below; all other algorithms in the scheme simply invoke the corresponding algorithms in the underlying UMA RAM-LFE.

- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$:
  1. Sample two puncturable PRFs $F, G \leftarrow \mathcal{F}$.
  2. Construct the modified program P' as defined in Algorithm 5.5 by hardcoding $P_L = P_R = P$ and the two PRFs F and G.
  3. Compute the encrypted database x' using F and G to mask each bit of x and writing it to both tracks of memory: $x'[i] = (0, x[i] \oplus F(0, i), x[i] \oplus G(0, i))$.
  4. Compute ciphertext $\mathsf{ct} \leftarrow \mathsf{uLFE.Enc}(\mathsf{crs}, \mathsf{dig}, P', x')$ and output it.

Algorithm 5.5: The transformed RAM Program P'

Hardcoded: Two RAM programs P_L, P_R and two puncturable PRFs F, G.

Algorithm: Takes as input the encrypted database x' and the unaltered public database y. Let C_L, C_R be the step circuits of P_L and P_R respectively. Evaluation of P' is defined by the following loop. Initialize $\mathsf{st} = (\mathsf{st}_L, \mathsf{st}_R)$, $v_x = (t', v_{x,L}, v_{x,R})$ and v_y with zeroes. For each time step $t \in [T]$:

1. Evaluate both step circuits $\mathsf{out}_L = C_L(\mathsf{st}_L, v_{x,L}, v_y)$ and $\mathsf{out}_R = C_R(\mathsf{st}_R, v_{x,R}, v_y)$. If the left track is in an output state (i.e. $\mathsf{out}_L \in Z$), halt and output out_L. Otherwise parse $\mathsf{out}_L = (\mathsf{st}_L, i_x, i_y, j_x, w_L)$ and $\mathsf{out}_R = (\mathsf{st}_R, i_x, i_y, j_x, w_R)$. If the memory addresses accessed by the left and right tracks differ, abort and output $\perp$.
2. Write each of w_L and w_R to their respective tracks, masked under independent PRFs, along with metadata of the time step this write occurred: $x'[j_x] = (t, w_L \oplus F(t, j_x), w_R \oplus G(t, j_x))$.
3. Read $(t', c_L, c_R) = x'[i_x]$ from the encrypted memory, and use the PRFs to unmask the data: $v_{x,L} = c_L \oplus F(t', i_x)$ and $v_{x,R} = c_R \oplus G(t', i_x)$.
4. Read from y to get $v_y = y[i_y]$ and update the state $\mathsf{st} = (\mathsf{st}_L, = \mathsf{st}_R)$ to carry forward to the next step.

Proof of Theorem 5.4. First we argue correctness. During encrytion, the program P' is constructed with hardcoded the same RAM program on each track, $P_L = P_R = P$, therefore the addresses accessed in each step will agree between the left and right tracks, and thus P' will not abort early. Then, by inspection, we have $P'(x', y) = P(x, y)$ and thus correctness follows from the correctness of the underlying UMA RAM-LFE scheme.

Next we argue efficiency. We note that transformed program P' runs in the same amount of time as the client's original program P and has description size $|P'| = \text{poly}(\lambda, |P|)$.[8] The encrypted database x' is only larger than x by a $\log T$ factor. Applying the UMA RAM-LFE efficiency properties to (P', x') yields the desired efficiency.

Next we prove security. Let (P_0, x_0) and (P_1, x_1) be the two client inputs chosen by the adversary in the security game. Assume without loss of generality, these two inputs satisfy the conditions such that the UA security game will not abort. That is, we have $P_0(x_0, y) = P_1(x_1, y)$, both computations run in t^* steps for some $t^* \in [T]$ and both computations access the same sequence of addresses in memory. We define the following sequence of hybrid experiments wherein we change the way the challenger constructs the transformed program P' and initializes the encrypted database x'.

- Hyb_0: This is the distribution on the LFE ciphertext ct in the real world UA-security experiment $\text{Expt}_{\mathsf{UA}}(1^\lambda, 0)$. That is, the challenger constructs P' by hardcoding $P_L = P_R = P_0$ and initializes x' by setting $x'[i] = (0, x_0[i] \oplus F(0, i), x_1[i] \oplus G(0, i))$ for each $i \in [N]$.
- Hyb_1: This is the same as Hyb_0, except we modify the way the challenger initializes x'. Instead of writing x_0 to both tracks of memory masked under the PRFs, the challenger writes x_0 to the main track and x_1 to the dummy track (still masking each track under the independent PRF keys). More formally, the challenger computes x' by setting $x'[i] = (0, x_0[i] \oplus F(0, i), x_1[i] \oplus G(0, i))$ for each $i \in [N]$.
- Hyb_2: This is the same as Hyb_1, except now we modify the way the challenger constructs the transformed program P'. Now the challenger hardcodes $P_L = P_0$ as the main program in P' and $P_R = P_1$ as the dummy program.
- Hyb_3: This is the same as Hyb_2, except we now modify P' so that it outputs the result of the computation done on the right track of memory (thereby making the left track the dummy track).
- Hyb_4: Now we change the program P' so that $P_L = P_R = P_1$ is evaluated on both tracks of memory, but x' is initialized with both x_0 and x_1 (as in Hyb_1).
- Hyb_5: This is the real world experiment $\text{Expt}_{\mathsf{UA}}(1^\lambda, 1)$.

The fact that $\mathsf{Hyb}_2 \approx_c \mathsf{Hyb}_3$ follows immediately from security of the underlying RAM-LFE because we simply swap the output of the two tracks and it holds that $P_0(x_0, y) = P_1(x_1, y)$. We focus on proving $\mathsf{Hyb}_0 \approx_c \mathsf{Hyb}_1$ and $\mathsf{Hyb}_1 \approx_c \mathsf{Hyb}_2$ in the following claims. The indistinguishability $\mathsf{Hyb}_3 \approx_c \mathsf{Hyb}_4$ and $\mathsf{Hyb}_4 \approx_c \mathsf{Hyb}_5$ follow in a symmetrical manner.

Claim 5.5.1. $\mathsf{Hyb}_0 \approx_c \mathsf{Hyb}_1$.

Proof. We prove the indistinguishability of Hyb_0 from Hyb_1 through a further series of indistinguishable hybrids. Intuitively, we change the dummy track of the initial encrypted database to contain an encryption of x_1. We work one cell at a time by puncturing the PRF G at the appropriate place. For each $i \in [N]$ define the following hybrids.

[8] We note that P' must be padded to have description size equal to the largest program we encrypt as an intermediate hybrid in the security argument below. The largest program that appears in the security argument is P'' defined in Algorithm 5.6, and it only ever contains constantly many hardcoded bits and PRF keys that are punctured in only one location. Thus we have $|P''| = \text{poly}(\lambda, |P|)$ as well.

- $\mathbf{H}_0^i$: This is the same as Hyb_0 except we modify x' so that the first i memory cells contain data from x_1 on the dummy (i.e. right) track instead of x_0. That is, the challenger computes x' as follows: for all $j \in [N]$

$$x'[j] = \begin{cases} (0, x_0[j] \oplus F(0,j), x_1[j] \oplus G(0,j)) & j < i \\ (0, x_0[j] \oplus F(0,j), x_0[j] \oplus G(0,j)) & j \geq i. \end{cases}$$

- $\mathbf{H}_1^i$: This is the same as $\mathbf{H}_0^i$ except we replace the PRF G that is hardcoded into P' by the PRF G' which is the result of puncturing G at input $(0, i)$. However, we still use the unpunctured value $G(0, i)$ as a mask in initializing the dummy track of $x'[i]$.
- $\mathbf{H}_2^i$: This is the same as $\mathbf{H}_1^i$ except we initialize $x'[i] = (0, x_0[i] \oplus F(0,i), r)$ where $r \leftarrow \{0,1\}$ is a uniformly random bit.
- $\mathbf{H}_3^i$: This is the same as $\mathbf{H}_2^i$ except we use data from x_1 to initialize the dummy track of $x'[i]$ (masked with the unpunctured value $G(0,i)$). That is, we set $x'[i] = (0, x_0[i] \oplus F(0,i), x_1[i] \oplus G(0,i))$.

It is clear that $\mathsf{Hyb}_0 \equiv \mathbf{H}_0^1$ and $\mathsf{Hyb}_1 \equiv \mathbf{H}_0^{N+1}$. We claim that for all $i \in [N]$, $\mathbf{H}_0^i \approx_c \mathbf{H}_1^i$ by UMA security of the UMA RAM-LFE because puncturing G at $(0, i)$ without changing how we initialize x' doesn't affect the access pattern of P'. Then by the pseudorandomness of the puncturable PRF family at the punctured location, it follows that for all $i \in [N]$ $\mathbf{H}_1^i \approx_c \mathbf{H}_2^i$ and similarly $\mathbf{H}_2^i \approx_c \mathbf{H}_3^i$. Finally, we note that the only thing that changes between $\mathbf{H}_3^i$ and $\mathbf{H}_0^{i+1}$ is that we unpuncture G in P'. But this doesn't change memory accesses of P', so $\mathbf{H}_3^i \approx_c \mathbf{H}_0^{i+1}$ follows again by UMA security. □

Claim 5.5.2. $\mathsf{Hyb}_1 \approx_c \mathsf{Hyb}_2$.

Proof. We prove the indistinguishability of Hyb_0 from Hyb_1 through a further series of indistinguishable hybrids. Here, the intuition is that we change the computation done on the dummy track to be an execution of P_1 rather than P_0. Here we work one step at a time by puncturing the PRF G at the appropriate place. For each $\tau \in \{0, \dots, T\}$ define the following hybrids.

- $\mathbf{H}_0^\tau$: This is the same as Hyb_1 except we replace P' with the program P'' defined in Algorithm 5.6 with $P_L = P_0$, $P_R = P_1$ and the time step τ and ciphertext $c = \perp$ hardcoded into it. Intuitively, P'' will run P_1 on the dummy track for the first t steps of the computation, and afterwards it will only execute the main track which contains P_0, and copy everything it writes to the dummy track.
- $\mathbf{H}_1^\tau$: This is the same as $\mathbf{H}_0^\tau$, except we change P'' to only contain the PRF G', the result of puncturing G on (τ, j_x) where j_x is the address that is written to in step τ. Then we additionally hardcode a ciphertext $c = w_0 \oplus G(\tau, j_x)$ that P'' will write to the dummy track in step τ, where w_0 is the value that P_0 would write to x in step τ.
- $\mathbf{H}_2^\tau$: This is the same as $\mathbf{H}_1^\tau$, except we change the hardcoded ciphertext to $c = w_1 \oplus G(\tau, j_x)$.

First observe that we have $\mathsf{Hyb}_1 \approx_c \mathbf{H}_0^0$ by UMA security of the underlying RAM-LFE because P'' with $\tau = 0$ hardcoded runs P_0 on each track in parallel. Additionally, we have $\mathbf{H}_0^{t^*} \equiv \mathsf{Hyb}_2$

by inspection. Fix $\tau \in \{0, \dots, T\}$. We have $\mathbf{H}_0^\tau \approx_c \mathbf{H}_1^\tau$ again by UMA security because the hardcoded ciphertext ensures that the same value is written to memory in both experiments. Then the indistinguishability $\mathbf{H}_1^\tau \approx_c \mathbf{H}_2^\tau$ follows from the pseudorandomness of the puncturable PRF family at the punctured location. Finally UMA security implies $\mathbf{H}_2^\tau \approx_c \mathbf{H}_0^{\tau+1}$ because we merely unpuncture the PRF. □

Algorithm 5.6: The hybrid RAM Program P''

Hardcoded: Two RAM programs P_L, P_R and two puncturable PRFs F, G. And additionally a time step τ and a hardcoded ciphertext c.

Algorithm: Takes the same inputs and initializes variables the same as in Algorithm 5.5. We describe the behavior of P'' in cases depending on the value of $t \in [T]$:

- If $t < \tau$: Behave exactly as in Algorithm 5.5, that is evaluate both the left and write step circuits and writes their updates independently to the two tracks of memory.
- If $t = \tau$: Evaluate both step circuits as in Algorithm 5.5, halting as it would if $\mathsf{out}_L \in Z$. Parse $\mathsf{out}_L = (\mathsf{st}_L, i_x, i_y, j_x, w_L)$ and $\mathsf{out}_R = (\mathsf{st}_R, i_x, i_y, j_x, w_R)$. Write
$$x'[j_x] = \begin{cases} (t, w_L \oplus F(t, j_x), c) & c \neq \bot \\ (t, w_L \oplus F(t, j_x), w_R \oplus G(t, j_x)) & \text{otherwise.} \end{cases}$$
Read from memory as in Algorithm 5.5.
- If $t > \tau$: Evaluate only the left step circuit $\mathsf{out}_L = C_L(\mathsf{st}_L, v_{x,L}, v_y)$. If it is in an output state (i.e. $\mathsf{out}_L \in Z$), halt and output out_L. Otherwise parse $\mathsf{out}_L = (\mathsf{st}_L, i_x, i_y, j_x, w_L)$. Write w_L to both tracks of memory: $x'[j_x] = (t, w_L \oplus F(t, j_x), w_L \oplus G(t, j_x))$. Read from memory exactly as Algorithm 5.5 does.

This completes the proof of Theorem 5.4.

□

5.2.2 Hiding the Access Pattern

In this section we show how to upgrade the indistinguishability-based UA RAM-LFE from the previous section into a RAM-LFE with full (simulation-based) security using DEPIR and ORAM to hide the addresses that are accessed.

Theorem 5.7. *Assuming the existence of a UA RAM-LFE, an ORAM with localized randomness and DEPIR, there exists a (fully simulation secure) RAM-LFE with full efficiency. In particular, assuming RingLWE holds and the existence of indistinguishability obfuscation, there exists a RAM-LFE with the following efficiency properties: for any constant $\epsilon > 0$*

- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$ *runs in time* $|y|^{1+\epsilon} \cdot \mathrm{poly}(\lambda)$.
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$ *runs in time* $\mathrm{poly}(\lambda, |P|) + |x| \cdot \mathrm{poly}(\lambda, \log |x|, \log T)$
- $\mathsf{LFE.Dec}(\mathsf{crs}, y, \mathsf{ct})$ *runs in time* $t \cdot \mathrm{poly}(\lambda, |P|)$.

where T is the maximum running time of P, and t is the actual running time of $P(x, y)$.

Construction. The ingredients to our construction are as follows:

- An indistinguishability-based UA RAM-LFE $\mathsf{uLFE} = (\mathsf{uLFE.Gen}, \mathsf{uLFE.Hash}, \mathsf{uLFE.Enc}, \mathsf{uLFE.Dec})$
- An ORAM with localized randomness: $\mathsf{ORAM} = (\mathsf{ORAM.Setup}, \mathsf{ORAM.Access})$
- A DEPIR: $\mathsf{DEPIR} = (\mathsf{Prep}, \mathsf{Query}, \mathsf{Dec})$
- Puncturable PRFs with various input and output sizes

At a high level, our construction here is analogous to the weakly efficient construction from Section 5.1: Given a client's input (P, x), we compile the program P into a new program P^* that hides the access pattern of P by accessing x via the ORAM and y via DEPIR. Because making one logical access via DEPIR or ORAM takes multiple physical accesses, each logical step in evaluating P will be compiled into η steps in P^* where η is the number of physical accesses it takes to make an ORAM or DEPIR query.[9] Since accessing memory via ORAM and DEPIR are necessarily randomized processes, we use (puncturable) PRFs as the source of the randomness for each access. In DEPIR, each query uses independent contiguous blocks of randomness, so we model its random tape as a PRF $F_D : [T] \to \{0,1\}^\lambda$. However in the ORAM with localized randomness, we think of the access algorithm as having a global random tape that is consistent throughout all ORAM accesses, thus we model the ORAM random tape as a PRF $F_O : [M] \to \{0,1\}$, where M is an upper bound on the length of the random tape.

- $\mathsf{LFE.Gen}(1^\lambda)$: Sample $\mathsf{crs} \leftarrow \mathsf{uLFE.Gen}(1^\lambda)$ as in the UA RAM-LFE.
- $\mathsf{LFE.Hash}(\mathsf{crs}, y)$: Use the DEPIR to preprocess $\hat{y} := \mathsf{Prep}(1^\lambda, y)$, and use the UA RAM-LFE to hash the preprocessed database: $(\mathsf{dig}, \widetilde{y}) := \mathsf{uLFE.Hash}(\mathsf{crs}, \hat{y})$.
- $\mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P, x)$:
  1. Sample two puncturable PRFs $F_O : [M] \to \{0,1\}$ and $F_D : [T] \to \{0,1\}^\lambda$.
  2. Initialize an ORAM containing x, by computing $(\mathsf{ck}, x^*) \leftarrow \mathsf{ORAM.Setup}(1^\lambda, x, N + T)$ using F_O as the random tape of the computation. Here $N = |x|$ and T is the upper bound on the running time of P.
  3. Compile P into the oblivious program P^* as defined in Algorithm 5.8 by hardcoding into it the PRFs F_O and F_D as well as the ORAM client key ck.
  4. Compute the ciphertext $\mathsf{ct} \leftarrow \mathsf{uLFE.Enc}(\mathsf{crs}, \mathsf{dig}, P^*, x^*)$.
- $\mathsf{LFE.Dec}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$: Simply evaluate $z := \mathsf{uLFE}(\mathsf{crs}, \widetilde{y}, \mathsf{ct})$ and output the result.

Algorithm 5.8: The oblivious program P^*

Hardcoded: A RAM program P, an ORAM client key ck and two puncturable PRFs $F_O : [M] \to \{0,1\}$ and $F_D : [T] \to \{0,1\}^\lambda$.

[9] As in Section 5.1, we assume without loss of generality that the value η is the same for both the ORAM and DEPIR schemes, potentially padding the one that needs fewer accesses with arbitrary dummy accesses.

Algorithm: Takes as input the ORAM database x^* and DEPIR preprocessed public database $\hat{y}$. Let C be the step circuit of P. Initialize $\mathsf{st}^* = (\mathsf{st}, \mathsf{ck}), v_x = 0, v_y = 0$, where st is the initial state for the step circuit C of P. For $t \in [T]$, run the following:

1. Parse $\mathsf{st}^* = (\mathsf{st}, \mathsf{ck})$, and evaluate the step circuit yielding $\mathsf{out} := C(\mathsf{st}, v_x, v_y)$. If out is an output state (i.e. $\mathsf{out} \in Z$), halt and output it. Otherwise parse $\mathsf{out} = (\mathsf{st}', i_x, i_y, j_x, w)$
2. Run the following two ORAM accesses in order as subroutines to handle the accesses to x. We use the PRF $F_O : [M] \to \{0, 1\}$ to function as the (global) random tape for all ORAM accesses throughout the computation. Recall M is an upper bound on the number of random coins the ORAM will use in making at most $2T$ logical accesses.
   (a) Run $(\mathsf{ck}', \perp) \leftarrow \mathsf{ORAM.Access}^{x^*}(\mathsf{ck}, \mathsf{write}, j_x, w)$.
   (b) Run $(\mathsf{ck}'', v_x) \leftarrow \mathsf{ORAM.Access}^{x^*}(\mathsf{ck}', \mathsf{read}, i_x, \perp)$.
3. Sample a DEPIR query $(Q, s) \leftarrow \mathsf{Query}(1^\lambda, |y|, i_y)$ using randomness given by $F_D(t)$, and read $V_{\hat{y}} = \hat{y}[Q] = \{\hat{y}[q] : q \in Q\}$. Decrypt to recover $v_y := \mathsf{Dec}(s, V_{\hat{y}})$.
4. Update $\mathsf{st}^* = (\mathsf{st}', \mathsf{ck}'')$ with the newly updated step circuit state and ORAM client key, and carry st^*, v_x and v_y to the next step.

Proof of Theorem 5.7. First we argue correctness by observing that correctness of the ORAM and DEPIR schemes implies that $P^*(x^*, \hat{y}) = P(x, y)$. Thus the overall correctness of the scheme follows from the correctness of the UA RAM-LFE.

Next we prove security. For a fixed $\mathsf{crs} \leftarrow \mathsf{LFE.Gen}(1^\lambda)$, let (P, x) be the client input and let y be the server input chosen by the adversary in the RAM-LFE security experiment. Let T be the upper bound on the run time of P, let t^* be the actual run time of $P(x, y)$, and let $N = |x|$. For the fixed sequence of $2t^*$ logical memory accesses P^* makes using the ORAM and fixed PRF F_O, let $\{S_i\}_{i=1}^{2t^*}$ be the disjoint subsets of $[M]$ that give the randomness used in each ORAM access. Additionally let $\ell = \max_i |S_i| = \mathrm{poly}(\lambda, \log N)$ denote the "locality" of the localized randomness ORAM.

Our argument proceeds by the technique of punctured programming. We crucially rely on the localized randomness of the ORAM and the property that DEPIR queries use independent randomness. The unprotected access security of the UA RAM-LFE allows us to ingore the content being written to/read from memory and focus on the addresses being accessed. We define the following sequence of hybrids where we change the distribution of the ciphertext ct by gradually simulating steps of the computation, starting from the end and working toward the beginning.

- Real: This is the distribution on ct that is sampled in the real security experiment $\mathsf{Real}_{\mathsf{LFE}}(1^\lambda)$. That is ct is sampled as $\mathsf{ct} \leftarrow \mathsf{uLFE.Enc}(\mathsf{crs}, \mathsf{dig}, P^*, x^*)$, where P^* and x^* are defined as in LFE.Enc above.

- For each $\tau \in \{t^*, \ldots, 1\}$, define Hyb_τ: This is the distribution on ct where it is sampled by $\mathsf{ct} \leftarrow \mathsf{LFE.Enc}(\mathsf{crs}, \mathsf{dig}, P^{**}_\tau, x^*)$. Here P^{**}_τ is the hybrid program defined in Algorithm 5.10 that has the output $z = P(x, y)$, and runtime t^* hardcoded. Intuitively, in P^{**}_τ we execute P for the first τ steps, making accesses via ORAM and DEPIR as in P^*, but then for the latter steps, P^{**}_τ makes only simulated ORAM and DEPIR accesses before finally outputting the hardcoded output z after t^* steps.

- Ideal: This is the distribution on ct where it is sampled by ct $\leftarrow$ LFE.Enc(crs, dig, $S^*, 0^{|x^*|}$). Here S^* is the fully simulated program defined in Algorithm 5.9 that has the output $z = P(x, y)$ and run-time t^* hardcoded, but otherwise contains no information about P.

Algorithm 5.9: The fully simulated program S^*

Hardcoded: The output $z = P(x, y)$ and runtime t^*, as well as a puncturable PRF $G : [T] \to \{0,1\}^\ell$ where ℓ is the locality of the ORAM and a puncturable PRF $F_D : [T] \to \{0,1\}^\lambda$.

Algorithm: Takes as input the ORAM database x^* and DEPIR preprocessed public database $\hat{y}$. For $t \in [t^*]$, run the following:

1. Simulate the two logical ORAM accesses using the local simulator LRSim$(G(t))$ given randomness from the PRF G.
2. Sample a dummy DEPIR query to an arbitrary fixed address $(Q, s) \leftarrow$ Query$(1^\lambda, |y|, 1)$ using randomness given by $F_D(t)$, and read $V_{\hat{y}} = \hat{y}[Q] = \{\hat{y}[q] : q \in Q\}$.

After t^* steps, halt and output the hardcoded value z.

Algorithm 5.10: The hybrid oblivious program P^{**}_τ

Hardcoded: A time step $\tau \in [t^*]$. It has all the same hardcoded values as in P^* (Algorithm 5.8): A RAM program P, an ORAM client key ck, two puncturable pseudorandom functions $F_O : [M] \to \{0,1\}$ and $F_D : [T] \to \{0,1\}^\lambda$.

Additionally, it has all the values as hardcoded in S^* (Algorithm 5.9): the output $z = P(x, y)$ and runtime t^*, as well as another puncturable PRF $G : [T] \to \{0,1\}^\ell$ where ℓ is the locality of the ORAM.

Algorithm: Takes the same input as in Algorithm 5.8 and initializes the state the same way. We describe the behavior of P^{**}_τ in two cases depending on the time step $t \in [t^*]$.

- If $t \leq \tau$: Behave exactly as in the oblivious algorithm P^*, that is, evaluate the step circuit of P and make memory accesses via the ORAM and DEPIR.
- If $t > \tau$: Behave as in the simulated algorithm S^*, that is, don't evaluate the step circuit, make dummy accesses to x^* at the addresses computed using the ORAM local simulator LRSim$(G(t))$, and read addresses from $\hat{y}$ corresponding to a dummy DEPIR query $(Q, s) \leftarrow$ Query$(1^\lambda, |y|, 1)$ sampled using randomness $F_D(t)$.

After t^* steps, halt and output the hardcoded value z.

It is easy to see that the distribution on ct defined in Ideal can be sampled given only the crs, the public database y, the client database size $N = |x|$, the output $z = P(x, y)$, and the run time t^*. Security of the UA RAM-LFE implies that Real $\approx_c$ Hyb$_{t^*}$ because $P^{*}{*}_{t^*}$ has the same access pattern as P^*. Similarly security of the UA RAM-LFE also implies that Ideal $\approx_c$ Hyb$_0$. It remains to show the following claim.

Claim 5.10.1. *For all* $\tau \in \{t^*, \ldots, 1\}$, $\mathsf{Hyb}_\tau \approx_c \mathsf{Hyb}_{\tau-1}$.

Proof of Claim 5.10.1. At a high level, we will swap the DEPIR query in step τ with a dummy DEPIR query to a fixed address and the ORAM access with a simulated access pattern that is simulated with independent randomness (we use the PRF G as the independent source of randomness for LRSim). Let $S_\tau \subset [M]$ be the subset of the ORAM random tape that is used by the ORAM.Access algorithm in making the two logical accesses in step τ. We prove the claim by making a sequence of indistinguishable changes to the way we sample ct by changing the definition of P^{**}_τ.

- $\mathbf{H}_0$: This is the same as Hyb_τ except we instead construct P^{**}_τ using $F'_O = F_O\{S_\tau\}$ that has been punctured at the points in S_τ as the ORAM's random tape. However, we additionally hardcode the values $F(S_\tau)$ into $P^{*}*_\tau$, and, in step τ, we compute the ORAM accesses using the localized randomness simulator: $\mathsf{LRSim}(F_O(S_\tau))$.[10]
- $\mathbf{H}_1$: This is the same as $\mathbf{H}_0$ except instead of hardcoding the values $F(S_\tau)$ into $P^{*}*_\tau$, we replace those values with uniformly sampled $\{b_s\}_{s\in S_\tau}$.
- $\mathbf{H}_2$: This is the same as $\mathbf{H}_1$ except we now unpuncture the ORAM tape F_O and instead puncture the PRF $G' = G\{\tau\}$.
- $\mathbf{H}_3$: This is the same as $\mathbf{H}_2$ except we replace the hardcoded values $\{b_s\}_{s\in S_\tau}$ with randomness from the independent PRF $G(\tau)$.
- $\mathbf{H}_4$: This is the same as $\mathbf{H}_3$ except we now unpuncture G and instead puncture the DEPIR random tape at input τ to get $F'_D = F_D\{\tau\}$. And we additionally hardcode the value $F_D(\tau)$ to be used as the randomness for sampling the DEPIR query in step τ.
- $\mathbf{H}_5$: This is the same as $\mathbf{H}_4$ except we replace the hardcoded value $F_D(\tau)$ with a uniformly random string $r \leftarrow \{0,1\}^\lambda$ for use as the randomness of the DEPIR query in step τ.
- $\mathbf{H}_6$: This is the same as $\mathbf{H}_5$ except instead of sampling the DEPIR query $\mathsf{Query}(1^\lambda, |y|, i_y)$ using the read address output by the step circuit, we sample a dummy DEPIR query $\mathsf{Query}(1^\lambda, |y|, 1)$ to the fixed address 1.
- $\mathbf{H}_7$: This is the same as $\mathbf{H}_6$ except we replace the hardcoded random string r with the PRF evaluation $F_D(\tau)$.
- $\mathbf{H}_8$: This is the same as $\mathbf{H}_7$ except we now replace the punctured F'_D with the original unpunctured PRF F_D.

We argue the indistinguishability of the above hybrids as follows:

- $\mathsf{Hyb}_\tau \approx_c \mathbf{H}_0$ follows by security of the UA RAM-LFE together with the localized randomness property of the ORAM. This is because the simulated accesses output by $\mathsf{LRSim}(F_O(S_\tau))$ are equal to real ORAM accesses in Hyb_τ with all but negligible probability. Thus P^{**}_τ has the same access pattern in the two experiments.
- $\mathbf{H}_0 \approx_c \mathbf{H}_1$ follows by pseudorandomness of the puncturable PRF F_O at the punctured locations.

[10]Here we abuse notation to write $F_O(S_\tau)$ to indicate the set $\{F_O(s) : s \in S_\tau\}$.

- $\mathbf{H}_1 \approx_c \mathbf{H}_2$ follows by security of the UA RAM-LFE because the value $G(\tau)$ is never used elsewhere in P^{**}_τ, thus puncturing G at τ doesn't affect the access pattern of P^{**}_τ.
- $\mathbf{H}_2 \approx_c \mathbf{H}_3$ follows by pseudorandomness of the puncturable PRF G at the punctured location.
- $\mathbf{H}_3 \approx \mathbf{H}_4$ follows by security of the UA RAM-LFE because in both experiments P^{**}_τ uses the same string as the randomness for the DEPIR query, and therefore it has the same access pattern in both experiments.
- $\mathbf{H}_4 \approx_c \mathbf{H}_5$ follows by pseudorandomness of the puncturable PRF F_D at the punctured location.
- $\mathbf{H}_5 \approx_c \mathbf{H}_6$ follows by security of the DEPIR scheme.
- $\mathbf{H}_6 \approx_c \mathbf{H}_7$ follows by pseudorandomness of the puncturable PRF F_D at the punctured location.
- $\mathbf{H}_7 \approx_c \mathbf{H}_8$ follows by security of the UA RAM-LFE because the access pattern does not change.
- $\mathbf{H}_8 \approx_c \mathsf{Hyb}_{\tau-1}$ follows by security of the UA RAM-LFE because the only difference is that in $\mathbf{H}_8$ P^{**}_τ evaluates the step circuit in the τth step, but otherwise they have the same access pattern because they both fully simulate the accesses in the τth step.

$\square$

This completes the proof of security.

Finally we argue efficiency. Assuming RingLWE holds and the existence of indistinguishability obfuscation, we instantiate the DEPIR from Theorem 2.6, the localized randomness ORAM from Theorem 2.4 and the UA RAM-LFE from Theorem 5.4. The efficiency of LFE.Hash is implied by the efficiency of the DEPIR Prep algorithm. It remains to argue that compiling P into the oblivious program P^* doesn't blow up the run time or description size by more than a $\mathrm{poly}(\lambda, \log|x|, \log T)$ factor. Observe that the program P^* executes a single logical step of P in η steps where η is the (max of the) query overhead of the ORAM and DEPIR schemes. However, both the ORAM and DEPIR have $\eta = \mathrm{poly}(\lambda, \log|x|, \log|y|)$. Finally we note that in the security proof, we only ever hard code one logical step's worth of data at a time, so the total (padded) description size of $|P^*| = \eta \cdot |P| \cdot \mathrm{poly}(\lambda)$.

This completes the proof of Theorem 5.7. $\square$

Acknowledgements

We thank Ji Luo for helpful comments.

References

[ACFQ22] Prabhanjan Ananth, Kai-Min Chung, Xiong Fan, and Luowen Qian. Collusion-resistant functional encryption for RAMs. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022, Part I*, volume 13791 of *Lecture Notes in Computer Science*, pages 160–194. Springer, Heidelberg, December 2022. 4, 5, 6, 7, 21, 22, 23

[AGVW13] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 500–518. Springer, Heidelberg, August 2013. 7

[AL18] Prabhanjan Ananth and Alex Lombardi. Succinct garbling schemes from functional encryption through a local simulation paradigm. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 455–472. Springer, Heidelberg, November 2018. 18

[BCG$^+$18] Nir Bitansky, Ran Canetti, Sanjam Garg, Justin Holmgren, Abhishek Jain, Huijia Lin, Rafael Pass, Sidharth Telang, and Vinod Vaikuntanathan. Indistinguishability obfuscation for ram programs and succinct randomized encodings. *SIAM Journal on Computing*, 47(3):1123–1210, 2018. 3, 5, 6

[CDG$^+$17] Chongwon Cho, Nico Döttling, Sanjam Garg, Divya Gupta, Peihan Miao, and Antigoni Polychroniadou. Laconic oblivious transfer and its applications. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part II*, volume 10402 of *Lecture Notes in Computer Science*, pages 33–65. Springer, Heidelberg, August 2017. 2, 5, 6, 12, 13, 19, 20, 21, 22

[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. 3, 5, 7, 10, 11, 27

[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. 3

[CP13] Kai-Min Chung and Rafael Pass. A simple oram. Cryptology ePrint Archive, Paper 2013/243, 2013. https://eprint.iacr.org/2013/243. 10

[DGM23] Nico Döttling, Phillip Gajland, and Giulio Malavolta. Laconic function evaluation for turing machines. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023: 26th International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 13941 of *Lecture Notes in Computer Science*, pages 606–634. Springer, Heidelberg, May 2023. 1, 5

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th Annual Symposium on Foundations of Computer Science*, pages 40–49. IEEE Computer Society Press, October 2013. 15

[GHL$^+$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. 3

[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. 3

[GKP$^+$13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th Annual ACM Symposium on Theory of Computing*, pages 555–564. ACM Press, June 2013. 1

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM*, 43(3):431–473, May 1996. 5, 8, 10

[GOS18] Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. Adaptive garbled RAM from laconic oblivious transfer. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 515–544. Springer, Heidelberg, August 2018. 5, 6

[Gra10] Kristen Grauman. Efficiently searching for similar images. *Commun. ACM*, 53(6):84–94, jun 2010. 2

[GS18a] Sanjam Garg and Akshayaram Srinivasan. Adaptively secure garbling with near optimal online complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 535–565. Springer, Heidelberg, April / May 2018. 5, 6

[GS18b] Sanjam Garg and Akshayaram Srinivasan. A simple construction of iO for turing machines. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018: 16th Theory of Cryptography Conference, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 425–454. Springer, Heidelberg, November 2018. 5, 6

[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. 2

[JLL23] Aayush Jain, Huijia Lin, and Ji Luo. On the optimal succinctness and efficiency of functional encryption and attribute-based encryption. In Carmit Hazay and Martijn Stam, editors, *Advances in Cryptology – EUROCRYPT 2023, Part III*, volume 14006 of *Lecture Notes in Computer Science*, pages 479–510. Springer, Heidelberg, April 2023. 3, 4

[JLS21] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In Samir Khuller and Virginia Vassilevska Williams, editors, *53rd Annual ACM Symposium on Theory of Computing*, pages 60–73. ACM Press, June 2021. 8

[JLS22] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from LPN over $\mathbb{F}_p$, DLIN, and PRGs in NC^0. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 670–699. Springer, Heidelberg, May / June 2022. 8

[LMW22] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. Cryptology ePrint Archive, Paper 2022/1703, 2022. https://eprint.iacr.org/2022/1703. 2, 12

[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. 3

[QWW18] Willy Quach, Hoeteck Wee, and Daniel Wichs. Laconic function evaluation and applications. In Mikkel Thorup, editor, *59th Annual Symposium on Foundations of Computer Science*, pages 859–870. IEEE Computer Society Press, October 2018. 1, 2, 4