On the Plausibility of Fully Homomorphic Encryption for RAMs

Ariel Hamlin[1], Justin Holmgren[2], Mor Weiss[3], and Daniel Wichs[1]

[1] Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA. {ahamlin, wichs}@ccs.neu.edu
[2] Department of Computer Science, Princeton University, Princeton, New Jersey, USA. justin.holmgren@princeton.edu
[3] Department of Computer Science, IDC Herzliya, Herzliya, Israel. mor.weiss01@post.idc.ac.il

Abstract. We initiate the study of fully homomorphic encryption for RAMs (RAM-FHE). This is a public-key encryption scheme where, given an encryption of a large database D, anybody can efficiently compute an encryption of $P(D)$ for an arbitrary RAM program P. The running time over the encrypted data should be as close as possible to the worst case running time of P, which may be sub-linear in the data size.

A central difficulty in constructing a RAM-FHE scheme is hiding the sequence of memory addresses accessed by P. This is particularly problematic because an adversary may homomorphically evaluate many programs over the same ciphertext, therefore effectively "rewinding" any mechanism for making memory accesses oblivious.

We identify a necessary prerequisite towards constructing RAM-FHE that we call *rewindable oblivious RAM* (rewindable ORAM), which provides security even in this strong adversarial setting. We show how to construct rewindable ORAM using *symmetric-key doubly efficient PIR (SK-DEPIR)* (Canetti-Holmgren-Richelson, Boyle-Ishai-Pass-Wootters: TCC '17). We then show how to use rewindable ORAM, along with virtual black-box (VBB) obfuscation for specific circuits, to construct RAM-FHE. The latter primitive can be heuristically instantiated using existing indistinguishability obfuscation candidates. Overall, we obtain a RAM-FHE scheme where the multiplicative overhead in running time is polylogarithmic in the database size N. Our basic scheme is single-hop, but we also extend it to obtain multi-hop RAM-FHE with overhead N^ϵ for arbitrarily small $\epsilon > 0$.

We view our work as the first evidence that RAM-FHE is likely to exist.

Justin Holmgren is supported in part by the Simons Collaboration on Algorithms and Geometry and by NSF grant CCF-1714779. This research was done in part while affiliated with MIT, supported in part by the NSF MACS project CNS-1413920. Mor Weiss is supported in part by ISF grants 1861/16 and 1399/17, and AFOSR Award FA9550-17-1-0069. Daniel Wichs and Ariel Hamlin are supported by NSF grants CNS-1314722, CNS-1413964, CNS-1750795 and the Alfred P. Sloan Research Fellowship.

1 Introduction

Fully Homomorphic Encryption. Fully Homomorphic Encryption (FHE), proposed by Rivest, Adleman, and Dertouzos [RAD78], is an extension of standard semantically secure encryption that supports computations "underneath" encryption. That is, given an encryption of some data D, anybody can compute an encryption of $P(D)$ for arbitrary programs P, while D remains computationally hidden. We currently have constructions of FHE schemes based on the Learning With Errors (LWE) assumption (either satisfying a relaxation called "leveled" FHE, or additionally requiring a circular security assumption) [Gen09, BV11].

FHE has proven to be an indispensable tool in the foundational study of cryptography, with wide-ranging applications including functional encryption [GKP$^+$b], program obfuscation [GGH$^+$], verifiable computation [GGP10, KRR14], cryptographic hash functions [CCH$^+$19], and more.

The most immediate use-case of FHE is to outsource private computation. A client Alice stores her sensitive database D on an untrusted server, and the server non-interactively executes computations on Alice's behalf (by computing encryptions of $P(D)$ for arbitrary programs P), but learns nothing about D. In known FHE schemes, Alice's work is asymptotically optimal: encrypting her database takes $|D| \cdot \mathsf{poly}(\lambda)$ work, and decrypting the server's ciphertexts takes $|P(D)| \cdot \mathsf{poly}(\lambda)$ work. The server's work is also optimal; however, the program P *must be represented as a circuit* C, and the server's work is then $|C| \cdot \mathsf{poly}(\lambda)$.

There has been much work towards making FHE more practical by minimizing the $\mathsf{poly}(\lambda)$ factors [BGH13, GHS12, BGV12, GSW, GHPS13], but the necessity of representing P as a circuit can lead to a much larger asymptotic loss in efficiency. Indeed, we typically think of programs and their efficiency in the *Random-Access Memory (RAM)* model of computation. Although any RAM program can be converted into a circuit, this may result in a large efficiency loss: in general, a RAM program that runs in time T over a database of size N can be converted into a circuit of size $\widetilde{O}(N + T^2)$ [CR72, PF79]. As a result, for RAM computations running in time $T \ll N$ (e.g., binary search, whose RAM running time is $O(\log N)$), the circuit conversion can incur an exponential efficiency loss. Even for RAM computations with longer running times $T > N$, circuit conversion incurs a quadratic overhead, which asymptotically will be more significant than any $\mathsf{poly}(\lambda)$ multiplicative factor. Therefore, it is crucial to ask the question: Can an FHE scheme "natively" support RAM computations?

1.1 Our Results

RAM-FHE. We define and construct two notions of RAM-FHE. In both notions, given an encryption $\widehat{D}$ of an N-bit database D, a RAM program P, and a bound T on the running time of P, anyone can obtain an encryption $\widehat{y}$ of $P(D)$ in time roughly T. We note that the bound T on evaluation runtime is necessary for semantic security: if homomorphic evaluation preserved the input-specific running time of P, then one could completely learn D by measuring the time to homomorphically evaluate several carefully chosen programs.

Our basic notion is *single-hop*, in which the output ciphertext $\widehat{y}$ and any changes made to D by P, cannot be meaningfully used by future homomorphic computations. We also consider a *multi-hop* variant, in which one can homomorphically evaluate a sequence of RAM programs, which may read *and* write to D, with the changes made by each program execution visible to the next.

We give the first evidence that these notions are possible by constructing (single- and multi-hop) RAM-FHE schemes using extremely strong but plausible assumptions. Specifically, we rely on a recent primitive called *Secret Key Doubly-Efficient Private Information Retrieval (SK-DEPIR)*, as well as *Virtual Black-Box (VBB) obfuscation* for specific circuits. We have candidate SK-DEPIR constructions based on non-standard assumptions related to permuted and noisy Reed-Muller codes [BIPW17, CHR17]. While VBB obfuscation for general circuits is impossible [BGI$^+$01], it appears reasonable to assume that it can be done for most specific circuits and, indeed, any of the candidate constructions of indistinguishability obfuscation (iO) [GMM$^+$16, BMSZ16, MZ18, CVW18, BGMZ18, Agr, LM18, AJS18] can be used to heuristically instantiate it. We view such use of VBB obfuscation as analogous to the random-oracle heuristic: although it is known to be unsound in general, all examples where it fails tend to be contrived, and natural uses of it appear to be sound.[1]

Our constructions have the following efficiency guarantees:

- In the single-hop setting, encryptions of an N-bit database have size $\mathsf{poly}(\lambda, N)$, and the cost of homomorphically evaluating a program P with description size $|P|$ and run-time T is $(T + |P|) \cdot \mathsf{poly}(\lambda, \log N)$.
- In the multi-hop setting, for any constant $\epsilon > 0$, ciphertext sizes are $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$ and homomorphic evaluation takes time $(T + |P|) \cdot N^{\epsilon} \cdot \mathsf{poly}(\lambda)$.

Rewindable Oblivious RAM. As explained in Section 1.2 below, the main difficulty in constructing RAM-FHE is hiding the memory access pattern when the evaluator *repeatedly runs different programs on the same initial ciphertext*. We abstract this as a strengthening of Oblivious RAM (ORAM) [Gol87, Ost90, GO96] that we call *rewindable* ORAM, which we believe may be of interest beyond its applications to RAM-FHE. Recall that a standard ORAM scheme allows a client with a small local state k to privately access his own database whose encoding $\widetilde{D}$ is stored on a remote untrusted server. Informally, rewindable ORAM extends this notion to guarantee privacy even when the server can reset the client's state to a previous value.

We construct rewindable ORAM schemes based on any SK-DEPIR scheme. We do not assume the existence of any type of obfuscator and obtain different tradeoffs between efficiency and the types of rewinding attacks, specifically:

- If the server is only allowed to rewind the client to his initial state, then following a $\mathsf{poly}(\lambda, N)$-time setup, accessing the database costs $\mathsf{poly}(\lambda, \log N)$.

[1] Furthermore, it is possible to replace VBB obfuscation by a small stateless hardware token, resulting in a RAM-FHE scheme where ciphertexts contain such tokens, which appears to still be non-trivial. We note that VBB was similarly used to construct a public-key DEPIR scheme [BIPW17].

– If the server is allowed to rewind the client to *any* previous state, then following an $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$-time setup, accessing the database costs $N^{\epsilon} \cdot \mathsf{poly}(\lambda)$, for any $\epsilon > 0$.

1.2 Our Techniques

As alluded to above, the main difficulty in constructing RAM-FHE arises from the fact that the *memory access pattern* induced by evaluating P on D may be highly dependent on the database D, whereas the access pattern of the *homomorphic* evaluation of P must hide everything about D. One natural approach towards hiding the access pattern is to force the evaluator to emulate P via an ORAM. However, the RAM-FHE evaluator should be able to evaluate arbitrarily many different programs on *the same* ciphertex $\widehat{D}$, and is *not* required to update his state between executions. This raises the concern that (even a semi-honest) evaluator evaluating two different programs P_1, P_2 on $\widehat{D}$ may potentially deduce non-trivial information about the database D from the *correlations* between the two memory access patterns during these evaluations. This strategy corresponds to a "rewinding" attack on the underlying ORAM, and is not just a theoretical concern - all known ORAM constructions are *indeed insecure* in this case. (For example, if an ORAM client accesses an address a_0, fails to update his state, and then accesses a_1, the server's view will reveal whether or not $a_0 = a_1$.)

Main Component: Rewindable ORAM. We consider (Section 3.1) two flavors of *rewindable ORAM*, which provide security against this type of attack. The weaker flavor, called *Initial-State Rewindable ORAM* (ISR-ORAM) allows the adversary to observe the ORAM access patterns of various programs $P_1, P_2, \ldots$ executed on D, where between executions the client/server states are reset to their initial values $k, \widetilde{D}$. The adversary should learn nothing about the underlying access patterns of the programs.

We also define a stronger flavor called *Any-State Rewindable ORAM* (ASR-ORAM) where the adversary can rewind the client/server states to *any point in time*.[2] The ORAM access patterns that the adversary observes should reveal nothing about the underlying access patterns of the programs.

Rewindable ORAM Constructions. Constructing rewindable (even ISR-) ORAM appears to be difficult, and none of the standard ORAM constructions suffice. Indeed, all standard ORAM constructions follow the "balls and bins" model in which each data block is represented as a "ball" and stored on the server in some "bin". Such structures cannot guarantee even ISR-ORAM security since, as noted above, if the client state is reset between accesses then the server can distinguish whether the client is accessing the same data block or not (when accessing the

[2] For example, the adversary can observe the sequential ORAM execution of programs P_1, P_2, P_3, then rewind the client/server state to the point immediately after P_1's execution and observe the execution of a different program P_2', etc.

same block, the client will access the same "ball" on the server). Thus, we need fundamentally different techniques than prior ORAM constructions.

Our new approach to rewindable ORAM leverages a powerful recent tool called SK-DEPIR [BIPW17, CHR17], which can be viewed as a *stateless read-only* ORAM. Informally, following a setup phase in which the client receives a secret key k and the server receives an encoding $\widetilde{D}$ of the database D, the client can privately read arbitrary locations i of D by reading a few positions in $\widetilde{D}$, without having to update the client/server state during the process. The server should learn nothing about the underlying locations i being read. In particular, we can think of SK-DEPIR as a very restricted form of ISR-ORAM for the class of RAM program $P_i(D)$ that read and output the i'th location of D.

The works of [BIPW17, CHR17] constructed SK-DEPIR schemes under non-standard assumptions relating to permuted and noisy Reed-Muller codes. Note that such SK-DEPIR cannot exist in the "balls and bins" model, and must encode the data in some complex way that intertwines many data locations together. Indeed, repeatedly accessing the same data location i in a SK-DEPIR should be indistinguishable from accessing completely random and unrelated data locations, so there must be many different, and seemingly unrelated, tuples of locations in $\widetilde{D}$ that contain information about data location i. We use SK-DEPIR to construct both ISR- and ASR-ORAM schemes.

ISR-ORAM from SK-DEPIR and standard ORAM. The ISR-ORAM scheme is simple. Recall that SK-DEPIR is read-only, while ISR-ORAM supports arbitrary RAM programs that can both read and write to the database. In both cases, we can rewind the state to its initial value after an execution while maintaining privacy of the underlying access pattern. The high-level idea is to use the SK-DEPIR to support reads, and use a *standard* ORAM scheme to support writes.

Specifically, the initial states in our ISR-ORAM are the client and server states $k, \widetilde{D}$ of the SK-DEPIR. To execute a RAM program P, the client initializes a fresh copy of a standard, non-rewindable ORAM O, which is initially empty. (We provide an explicit construction of an ORAM scheme for initially empty databases in the full version [HHWW].) Writes are executed using the ORAM scheme O. To read some location i, the client reads i from both the ORAM O and the SK-DEPIR. If location i was found in O, the client uses that value, otherwise he uses the SK-DEPIR value. Thus, the client always gets the freshest copy of the value in any location. Note that rewinding the ISR-ORAM client/server to their initial states erases all information about O (which was initialized only in the first access), so we do not require rewindable security from O: the next access will instantiate a completely fresh ORAM scheme O for the execution. The scheme is described in the full version [HHWW].

ASR-ORAM from SK-DEPIR via a hierarchical structure. The ASR-ORAM construction is more complex. ASR-ORAM should support repeated sequential execution of different programs, and remain secure when the adversary can rewind to any intermediate state from which it starts a new sequence of program executions. Unfortunately, this precludes our previous solution of storing

intermediate values written during the execution in a standard, non-rewindable ORAM: rewinding to an intermediate point will rewind the ORAM.

We solve this problem by combining SK-DEPIR with techniques from hierarchical ORAM [Ost90, GO96]. In particular, our ASR-ORAM consists of a hierarchy of SK-DEPIR schemes of exponentially increasing size, where the top-most scheme has size 1 and the bottom-most scheme has size N. Initially, the data is entirely contained in the bottom-most scheme. To read a location i we try to read it using the SK-DEPIR schemes at all levels, and use the value found in the top-most scheme that contains i. To write a location i, we write it to the top level (which requires re-generating its SK-DEPIR scheme). As in Hierarchical ORAM this requires "reshuffles": every pre-determined number of writes, we need to merge sufficiently many of the top levels to ensure that their combined size is large enough to hold the database. Since levels are implemented using SK-DEPIR, this requires reading and re-writing the levels in their entirety. However, as levels get larger, they are "reshuffled" with decreasing frequency so the overall amortized[3] complexity is low. Notice that reshuffles reveal no information, even under arbitrary rewinding, because they occur at pre-determined times (independent of the access history), and reads are secure by the security of the (stateless) SK-DEPIR even under arbitrary rewinding.

We note that the actual construction (Section 3.2) is somewhat more involved. One issue arises because SK-DEPIR schemes are designed for array structures (i.e., reading a data block requires knowing its location in D), whereas the hierarchical construction imposes a map structure at each level because it contains a subset of (not necessarily consecutive) data blocks. To resolve this we use the standard data-structures trick of pseudorandomly mapping data blocks into buckets, thus guaranteeing that the block's location in each level in which it appears is independent of the history of accesses.

RAM-FHE from Rewindable ORAM. We construct RAM-FHE from rewindable ORAM using VBB obfuscation. At a high level, to encrypt some database D, we first construct the rewindable ORAM client/server states $k, \widetilde{D}$ for D. We then obfuscate the ORAM client program, with k hard-wired into it, and output the ciphertext consisting of $\widetilde{D}$ and the obfuscated program. The evaluator can then use the obfuscated ORAM client to execute an arbitrary RAM program over the encrypted database $\widetilde{D}$ and derive an encrypted output. During the execution, the evaluator emulates the ORAM server using $\widetilde{D}$ (performing `read`/`write` operations as instructed by the client).

Formalizing the above approach is challenging, and requires some adaptations. The final construction is obtained through the following steps.

Step (1): emulating statefulness. We cannot directly use a circuit obfuscator to obfuscate the rewindable ORAM client, because the client is *stateful*, and state is needed even for correctness. Instead, we obfuscate the circuit emulating a *single* client step in the ORAM scheme. This circuit takes the client

[3] We note that as in [OS97], reshuffles can be "spread-out" over many operations to achieve low *worst-case* complexity.

state as input, and returns the updated client state as part of its output. We note that representing the client as a circuit in this way is fundamentally different (and significantly more efficient) than representing an entire RAM program as a circuit. Indeed, the circuit performs *a single execution step*, thus the overhead is independent of the database size or the worst-case runtime of the program.

For simplicity of the exposition, we assume for now that the program's description is short (of size $p(\lambda)$ for some a-priori fixed polynomial p), and can therefore be given in its entirety to the obfuscated circuit in the first execution step. We explain below how to remove this restriction.

Step (2): hiding client state. (Standard/rewindable) ORAM security assumes the adversary does not see the client state, but in our construction the evaluator sees the client's internal states throughout the execution (since the obfuscated circuit outputs them). To hide the client states, we have the obfuscated circuit encrypt the state, using a hard-wired (symmetric) encryption key.

Step (3): forcing honest behavior. The rewindable ORAM is secure only as long as the ORAM client behaves honestly, and the ORAM server behaves semi-honestly. However, RAM-FHE should guarantee semantic security of the encrypted database against arbitrary (possibly malicious) evaluators. A malicious evaluator may deviate from a semi-honest emulation of the rewindable ORAM scheme in two ways.

First, the evaluator may emulate a malicious server whose answers to `read` requests are inconsistent with the database, and who fails to perform requested `write` operations. Such attacks can be prevented using the standard approach of maintaining a Merkle Hash Tree (MHT) of the server state. More specifically, we hard-code the initial MHT root into the obfuscated circuit. Answers to `read` requests include also the MHT path proving consistency of the answer (which is verified by the obfuscated circuit using the MHT root). Answers to `write` requests outputted by the obfuscated circuit additionally include an updated MHT path proving that the root was updated correctly.

Second, the evaluator may emulate a malicious client, by providing incorrect/inconsistent client states to the obfuscated circuit. We prevent such attacks by hard-wiring a Message-Authentication Code (MAC) key into the obfuscated circuit, and having it verify the input state and MAC the output state.

Step (4): hiding the output. Recall from Step (2) that the internal ORAM client state is encrypted using a "temporary" symmetric encryption key that is chosen at encryption time. Consequently, this key cannot be used to encrypt the computation output (which should be encrypted using a persistent public key that is chosen during key generation). We encrypt the output using a standard PKE scheme, where the public key is hard-wired into the obfuscated circuit.

Step (5): generating randomness for the execution. Even if the emulated RAM program is deterministic, the obfuscated circuit described above needs random coins for encryption, and to emulate the ORAM client. We use a PRF (applied to the MHT root, and the entire execution history) to derive the needed randomness, where the PRF key is hard-wired into the circuit.

An additional point that needs to be handled is the fact that a RAM program P has a volatile tape (a "scratch tape") which is used only during P's execution, after which it is erased. We use a *standard ORAM* to instantiate the scratch tape at the onset of the execution. Notice that standard ORAM security suffices here, since each execution instantiates a fresh ORAM for the scratch tape.[4]

The construction described above gives a single-hop RAM-FHE scheme when the underlying ORAM is an ISR-ORAM (see Section 6). The multi-hop RAM-FHE scheme is obtained by instantiating the ORAM with an ASR-ORAM, with some modifications to allow the evaluator to perform sequential computations on the database. (For example, this requires MAC-ing the initial state of the ASR-ORAM client together with the MHT root of the updated database, see the full version [HHWW] for more details.)

Generalizing to programs of any length. The construction described above assumed the entire program description was given as input to the obfuscated circuit (this requires an a-priori fixed bound on the description size). To support longer programs, we first copy the program description into the scratch tape at the onset of the computation. More specifically, the evaluator provides a MHT root for the program description as input to the obfuscated circuit, and the circuit then copies the program bit-by-bit into the scratch-tape, verifying consistency with the MHT root in each step. See the full version for details.

On the necessity of rewindable ORAM and DEPIR. As a final note, we informally argue that rewindable ORAM is inherent to the construction of RAM-FHE, by explaining how to construct ISR/ASR-ORAM from single-hop/multi-hop RAM-FHE. To initialize the ORAM with a database D, the client generates a random encryption-decryption key pair, encrypts D using the encryption key, and stores the ciphertext $\widehat{D}$ on the server. To execute a RAM program P on D, the client homomorphically evaluates P on $\widehat{D}$ by accessing all relevant bits of $\widehat{D}$ remotely on the server. Finally, the client decrypts the computation output using the decryption key. These ORAM access patterns reveal nothing about the database because the RAM-FHE scheme is semantically secure.[5] If we use multi-hop RAM-FHE then we can sequentially execute many programs and rewind to any intermediate state; semantic security still ensures that the access patterns reveal nothing about the underlying database, so we obtain ASR-ORAM. If we use a single-hop RAM-FHE, the ORAM only allows for the execution of a single program before rewinding to the initial state, so we only get ISR-ORAM. As

[4] We note that if an a-priori bound on the scratch tape size is known during encryption, then in the single-hop setting the scratch tape can be included as part of the encrypted database, since any updates to the database during execution are anyway lost when the execution ends.

[5] More formally, there is a discrepancy since the access pattern of homomorphic evaluation, though revealing nothing about D, may reveal something about P. To prevent this, we can append an encryption secret key sk to the database D, and execute a program $\widetilde{P}$ in which P's code is encrypted under sk, where $\widetilde{P}$ first decrypts P and then executes it over D. This way, the access pattern of the FHE evaluation cannot reveal anything about neither P nor D.

discussed above, SK-DEPIR can be thought of as a read-only ISR-ORAM, so RAM-FHE also implies SK-DEPIR.

1.3 Related Work

Supporting RAM computations directly, without first representing the RAM program as a circuit, has been considered for several cryptographic primitives.

Similar to RAM-FHE, Garbled RAM [LO13, GHL+14] (also known as private RAM delegation) allows a user to garble a database D, following which an evaluator can run RAM computations on the garbled D. (There are also works on non-private RAM delegation, e.g., [KP16].) However, in garbled RAM the evaluator can only compute specific RAM programs P which the garbler generated. Similar to RAM-FHE, the size of the garbled program, and the garbling and evaluation times, are proportional to P's running time. There has been a large body of works on garbled RAM, improving its efficiency, underlying assumptions, properties, and applications [GLOS, CHJV15, CH16, CCHR16, ACC+16, BCP, CCC+, Mia16, GGMP16, HY16, LO17, GOS18]. Succinct garbled RAMs together with iO for circuits also imply indistinguishability Obfuscation (iO) for RAMs [CHJV15, BCG+18].

Functional Encryption (FE) for RAMs, namely an FE scheme in which the master secret key can be used to generate function keys for *RAM programs*, was studied in [AIT16, GHRW, BCG+18]. These constructions are not function-private, and [AIT16] additionally do not hide the access pattern of the RAM program (which, as discussed in Section 1.2, seems to be a central difficulty in constructing RAM-FHE).

The notion of FHE for Turing machines was considered in [GKP+a], who construct FHE schemes with *input-specific* running time during evaluation. However, the runtime is still at least linear in the database size, whereas RAM-FHE evaluation time may be sublinear in the database size (if the original RAM program runs in sublinear time). Moreover, their model is somewhat restricted in that the Turing machine and its input are encrypted together (so one cannot execute arbitrary Turing machines on the input).

2 Preliminaries

Throughout this paper, λ denotes a security parameter. We use $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ to denote unspecified functions that are polynomial and negligible in λ, respectively. We use standard cryptographic definitions of one-way functions (OWFs), pseudorandom functions (PRFs), collision-resistant hash functions (CRHFs), and message authentication codes (MACs) (see, e.g., [Gol01, Gol04]). For a randomized algorithm A with n inputs, we use $A(x_1, \ldots, x_n; r)$ to denote the output of A on inputs $x_1, \ldots, x_n$ when it uses randomness r. We use $\approx$ to denote computational indistinguishability.

We use PPT to refer to probabilistic polynomial-time algorithms, and non-uniform PPT to refer to (ensembles of) polynomial-sized probabilistic circuits.

We use the notion of Virtual Black Box (VBB) obfuscation with auxiliary input (see the full version [HHWW]).

2.1 Doubly-Efficient Private Information Retreival (DEPIR)

Definition 1 (Secret-Key Doubly-Efficient PIR (SK-DEPIR) [CHR17, BIPW17]). *A* secret-key doubly-efficient PIR (SK-DEPIR) *scheme consists of procedures* $(\mathsf{KeyGen}, \mathsf{Process}, \mathsf{Query}, \mathsf{Decode})$ *where* $\mathsf{KeyGen}, \mathsf{Process}, \mathsf{Query}$ *are randomized and* Decode *is deterministic, with the following syntax:*

- $\mathsf{KeyGen}\left(1^\lambda\right)$ *takes as input a security parameter* λ*, and outputs a client secret-key* sk*.*
- $\mathsf{Process}\left(\mathsf{sk}, \mathsf{DB}\right)$ *takes as input a client secret-key* sk *and a database* $\mathsf{DB} \in \{0,1\}^N$*, and outputs a processed database* $\widetilde{\mathsf{DB}} \in \{0,1\}^{\tilde{N}}$*.*
- $\mathsf{Query}\left(\mathsf{sk}, \mathsf{addr}\right)$ *takes as input a client secret-key* sk *and an address* $\mathsf{addr} \in [N]$*, and outputs a set* $\mathcal{Q} \subseteq \left[\tilde{N}\right]$ *of queries, and a temporary state* st*.*
- $\mathsf{Decode}\left(\mathsf{sk}, \mathsf{st}, \left\{\widetilde{\mathsf{DB}}_i \ : \ i \in \mathcal{Q}\right\}\right)$ *takes as input a secret key* sk*, a temporary state* st*, and a set of values from the processed database* $\left\{\widetilde{\mathsf{DB}}_i \ : \ i \in \mathcal{Q}\right\}$*, and outputs a value* val*.*

We require that the scheme satisfies the following properties:

- ***Correctness:*** *for every* $N \in \mathbb{N}$*, every* $\mathsf{DB} \in \{0,1\}^N$*, and every* $\mathsf{addr} \in [N]$*, it holds that:*

$$\Pr\left[\mathsf{Decode}\left(\mathsf{sk}, \mathsf{st}, \left\{\widetilde{\mathsf{DB}}_i \ : \ i \in \mathcal{Q}\right\}\right) = \mathsf{DB}_i \ : \ \begin{array}{r} \mathsf{sk} \leftarrow \mathsf{KeyGen}\left(1^\lambda\right) \\ \widetilde{\mathsf{DB}} \leftarrow \mathsf{Process}\left(\mathsf{sk}, \mathsf{DB}\right) \\ \left(\mathcal{Q}, \mathsf{st}\right) \leftarrow \mathsf{Query}\left(\mathsf{sk}, \mathsf{addr}\right) \end{array}\right] = 1$$

- ***Security:*** *Any non-uniform PPT adversary* $\mathcal{A}$ *has only* $\mathsf{negl}\left(\lambda\right)$ *advantage in the following security game with a challenger* $\mathcal{C}$*:*
  1. $\mathcal{A}$ *sends to* $\mathcal{C}$ *a database* $\mathsf{DB} \in \{0,1\}^N$*.*
  2. $\mathcal{C}$ *picks a random bit* $b \leftarrow \{0,1\}$*, and runs* $\mathsf{sk} \leftarrow \mathsf{KeyGen}\left(1^\lambda\right)$ *to obtain a client secret-key* sk*, and then runs* $\widetilde{\mathsf{DB}} \leftarrow \mathsf{Process}\left(\mathsf{sk}, \mathsf{DB}\right)$ *to obtain a processed database* $\widetilde{\mathsf{DB}}$*, which it sends to* $\mathcal{A}$*.*
  3. $\mathcal{A}$ *selects two addresses* $\mathsf{addr}^0, \mathsf{addr}^1 \in [N]$*, and sends* $(\mathsf{addr}^0, \mathsf{addr}^1)$ *to* $\mathcal{C}$*.*
  4. $\mathcal{C}$ *samples* $(Q, \mathsf{st}) \leftarrow \mathsf{Query}(\mathsf{sk}, \mathsf{addr}^b)$*, and sends* Q *to* $\mathcal{A}$*.*
  5. *Steps 3 and 4 are repeated an arbitrary (polynomial) number of times.*
  6. $\mathcal{A}$ *outputs a bit* b'*, and his* $\mathsf{advantage}$ *in the game is defined to be* $\Pr[b = b'] - \frac{1}{2}$*.*
- ***Efficiency.*** *The runtime of* KeyGen *is* $\mathsf{poly}\left(\lambda\right)$*, the runtime of* $\mathsf{Process}$ *is* $\mathsf{poly}\left(N, \lambda\right)$*, and the runtime of* $\mathsf{Query}, \mathsf{Decode}$ *is* $o\left(N\right) \cdot \mathsf{poly}\left(\lambda\right)$*, where* N *is the database size.*

We will need a SK-DEPIR scheme with the additional guarantee that preprocessing is oblivious of the database contents. We note that both the SK-DEPIR constructions of [CHR17, BIPW17] satisfy this guarantee.

Definition 2 (Security with oblivious preprocessing). *We say that a SK-DEPIR scheme is* secure with oblivious preprocessing *if the security property of Definition 1 holds even when in Step 2 above, the adversary is given the sequence of memory accesses (including which address was accessed, whether it was read or written, and what value was written) performed during the execution of* $\mathsf{Process}(\mathsf{sk}, \mathsf{DB})$.

Remark on existence of SK-DEPIR schemes with specific parameters and oblivious preprocessing. The works [BIPW17, CHR17] prove that under a new assumption on noisy Reed-Muller codes, there exist SK-DEPIR schemes with either of the following parameters for databases of size N and security parameter λ:

- **Sublinear SK-DEPIR:** For any $\epsilon > 0$, the running time of $\mathsf{Process}$ can be $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$, and the running time of Query and Decode can be $N^{\epsilon} \cdot \mathsf{poly}(\lambda)$.
- **Polylog SK-DEPIR:** The running time of $\mathsf{Process}$ can be $\mathsf{poly}(\lambda, N)$, and the running time of Query and Decode can be $\mathsf{poly}(\lambda, \log N)$.

We note that both of these schemes have oblivious preprocessing. Indeed, in these constructions $\mathsf{Process}$ randomly permutes a (noisy) Reed-Muller encoding of an encryption of the database. The encoding is data-oblivious since it is applied to ciphertexts, and using oblivious sorting algorithms the permuting operation can also be done obliviously.

3 Rewindable Oblivious RAM

We define two ORAM variants which guarantee security against rewinding attacks. The two notions differ in the type of attacks they can handle. We first recall the notion of an access pattern, and the standard ORAM definition [Gol87, Ost90, GO96].

Notation 1 (Access pattern)**.** A length-q *access pattern* Q consists of a list $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)_{1 \le l \le q}$ of instructions, where instruction $(\mathsf{op}_l, \mathsf{val}_l, \mathsf{addr}_l)$ denotes that the client performs operation $\mathsf{op}_l \in \{\mathtt{read}, \mathtt{write}\}$ at address addr_l with value val_l (which, if $\mathsf{op}_l = \mathtt{read}$, is $\perp$).

Informally, an ORAM scheme allows a client to store his database, or "logical memory", remotely on a server, or "physical memory". Following a Setup procedure which generates client and server states, reads and writes to logical memory are performed through an interactive protocol Access between the client and server, where in each round the client generates a read request and an update request for the server. The access pattern to physical memory during the Access protocol completely hides from the server the database contents and access pattern to logical memory (see the full version for the formal definition).

3.1 Rewindable ORAM Security

We now describe a game that formalizes the security of our ORAM variants. The adversarial server in the game chooses a pair of initial databases, and (as in standard ORAM) two sequences of access patterns, with the goal of distinguishing between the executions of these sequences on the two databases. Unlike standard ORAM, the adversarial server in our security game can also *rewind* the execution to a previous state, and continue the execution from that state.

Definition 3 (Rewindable ORAM security game). *The ORAM security game is run between an adversary $\mathcal{A}$, and a challenger $\mathcal{C}$.*

1. *$\mathcal{A}$ sends to $\mathcal{C}$ two databases $\mathsf{DB}^0, \mathsf{DB}^1 \in \{0,1\}^N$.*
2. *$\mathcal{C}$ picks a random bit $b \leftarrow \{0,1\}$, and runs $\mathsf{Setup}\left(1^\lambda, \mathsf{DB}^b\right)$ to obtain client and server states ck, st. $\mathcal{C}$ sends st to $\mathcal{A}$.*
3. *Let $\mathsf{st}_0 = \mathsf{st}$ and $\mathsf{ck}_0 = \mathsf{ck}$. Repeat the following $\mathsf{poly}(\lambda)$ times, where in the i'th iteration:*
   (a) *$\mathcal{A}$ sends to $\mathcal{C}$ an index $j_i \in \{0, 1, \ldots, i-1\}$, as well as two sequences of instructions $Q_i^0 = \left(\mathsf{op}_{i,l}, \mathsf{addr}_{i,l}^0, \mathsf{val}_{i,l}^0\right)_{l \in [q_i]}$, and $Q_i^1 = \left(\mathsf{op}_{i,l}, \mathsf{addr}_{i,l}^1, \mathsf{val}_{i,l}^1\right)_{l \in [q_i]}$, where $q_i \leq \mathsf{poly}(\lambda)$, $\mathsf{op}_{i,l} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr}_{i,l}^0, \mathsf{addr}_{i,l}^1 \in [N]$, and $\mathsf{val}_{i,l}^0, \mathsf{val}_{i,l}^1 \in \{0,1\}$.*
   (b) *Starting from server state st_{j_i} and client state ck_{j_i}, $\mathcal{C}$ executes $\mathsf{Access}\left(\mathsf{op}_{i,l}, \mathsf{addr}_{i,l}^b, \mathsf{val}_{i,l}^b\right)$ for $1 \leq l \leq q_i$. Let $\mathsf{ck}_i, \mathsf{st}_i$ denote the updated client and server states (respectively) at the end of this sequence of executions. Let $\mathcal{ACC}_i$ denote the access pattern to physical memory during this sequence of* Access *executions.*
   (c) *$\mathcal{C}$ sends $\mathcal{ACC}_i$ to $\mathcal{A}$.*
4. *$\mathcal{A}$ outputs a bit b', and his advantage in the game is defined as $\Pr[b = b'] - \frac{1}{2}$.*

Discussion. The rewindable ORAM security game of Definition 3 captures several security variants, depending on the permissible choice of j_i. First, notice that the security game with $\mathsf{poly}(\lambda)$ iterations in the security game, when the adversary is restricted to choose $j_i = i - 1$ in each iteration, and $\mathsf{DB}^0 = \mathsf{DB}^1$, yields the standard ORAM security definition without rewinds. Second, restricting the adversary to choose $j_i = \{0, i-1\}$ in every iteration i means the adversary can only rewind the execution to the initial state, but can adaptively decide to "extend" a previous execution. Restricting the adversary to choose $j_i = 0$ in every iteration corresponds to an adversary that can only rewind the execution to the initial state, where any rewind "finalizes" the current branch of the execution, and the adversary cannot later extend it. In the most general form, when j_i can take any value in $\{0, 1, \ldots, i-1\}$, we can assume without loss of generality that the adversary chooses a length-1 sequence in each iteration of the security game. This corresponds to an adversary that can rewind the ORAM to any intermediate state. The security game of Definition 3 can be used to capture various other security variants; we choose to focus on the latter two notions. Formally,

Definition 4 (Any-State Rewindable ORAM (ASR-ORAM)). *We say that an ORAM scheme is* Any-State Rewindable (ASR) *if any PPT adversary $\mathcal{A}$ has a* $\mathsf{negl}(\lambda)$ *advantage in the rewindable ORAM security game of Definition 3.*

Definition 5 (Initial-State Rewindable ORAM (ISR-ORAM)). *We say that an adversary $\mathcal{A}$ is* initial-state restricted *if in every iteration i of the rewindable ORAM security game of Definition 3, it chooses $j_i = 0$. We say that an ORAM scheme is* Initial-State Rewindable (ISR) *if any initial-state restricted PPT adversary $\mathcal{A}$ has a* $\mathsf{negl}(\lambda)$ *advantage in the rewindable ORAM security game of Definition 3.*

3.2 Rewindable ORAM Constructions

In this section we construct ISR- and ASR-ORAM schemes from SK-DEPIR and standard ORAM schemes. Our ISR-ORAM scheme, despite having a weaker security guarantee than ASR-ORAM, has the advantage of being simpler and more efficient. In the full version [HHWW], we construct an ISR-ORAM scheme from a SK-DEPIR scheme along with an ORAM scheme for initially-empty databases, proving the following:

Theorem 2 (ISR-ORAM). *Assume there exist OWFs and SK-DEPIR. Then there exists an ISR-ORAM scheme.*

Moreover, if the Query *and* Decode *algorithms of the SK-DEPIR scheme have* $\mathsf{poly}(\lambda)$ *complexity for databases of size N and security parameter λ, and the client (resp., server) state has size* $\mathsf{poly}(\lambda)$ *(resp.,* $\mathsf{poly}(\lambda, N)$*), then the* Access *complexity of the ISR-ORAM is* $\mathsf{poly}(\lambda)$*, and the client (resp., server) state has size* $\mathsf{poly}(\lambda)$ *(*$\mathsf{poly}(\lambda, N)$*).*

We now construct an ASR-ORAM scheme from SK-DEPIR and PRFs, proving the following (the proof appears in the full version):

Theorem 3 (ASR-ORAM). *Assume the existence of OWFs and SK-DEPIR, then there exists an ASR-ORAM scheme. Moreover, if for $\epsilon > 0$ the* Query *and* Decode *algorithms of the SK-DEPIR scheme have $N^{\epsilon} \cdot \mathsf{poly}(\lambda)$ complexity, and* Process *has $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$ complexity for databases of size N and security parameter λ, then:*

- *The complexity of* Access *is $N^{\epsilon} \cdot \mathsf{poly}(\lambda)$.*
- *The client state has size* $\mathsf{poly}(\lambda)$*, and the server state has size $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$.*

The construction. Recall from Section 1.2 that we use a hierarchical structure whose levels contain SK-DEPIR schemes. Since a SK-DEPIR scheme is designed for array structures, we use PRFs to map the data blocks of the level into buckets, thus guaranteeing that a block's location in each level (if it appears in the level) is independent of the access history. To allow for more efficient reshuffles, each level i also contains the (encrypted, unprocessed) database stored in the SK-DEPIR of the level. We note that whenever a level is initialized as part of a

reshuffle, we pick new PRF and SK-DEPIR keys for the level. This guarantees security even under rewinds. Indeed, though a SK-DEPIR is rewind-secure, by rewinding the ORAM the adversary may rewind a reshuffle. However, this will result in a completely fresh SK-DEPIR scheme, and therefore doesn't violate security. In the following, we use $\mathsf{B} = \lambda$ to denote the bucket size.

Construction 1 (ASR-ORAM from SK-DEPIR and PRFs)**.** The scheme uses:

- A PRF F.
- A SK-DEPIR scheme ($\mathsf{DEPIR.KeyGen}, \mathsf{Process}, \mathsf{Query}, \mathsf{Decode}$) with oblivious preprocessing (Definition 2).
- A CPA-secure symmetric encryption scheme ($\mathsf{SE.KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}$).

The scheme consists of the following procedures.

$\mathsf{Setup}(1^\lambda, \mathsf{DB})$**:** Recall that λ denotes the security parameter, and $\mathsf{DB} \in \{0,1\}^N$. Let DB' be the database obtained from DB by concatenating the address to each bit, i.e., entries of DB' have the form $(\mathsf{addr}, \mathsf{DB}_{\mathsf{addr}})$. (This will be needed when blocks are mapped to buckets.) Let $\ell = \log N$, and proceed as follows.

- Counter initialization: initialize a counter count_W to 0. (count_W counts the total number of writes performed so far.)
- Encryption initialization: run $\mathsf{sk} \leftarrow \mathsf{SE.KeyGen}\left(1^\lambda\right)$ to generate a secret-key sk for the encryption scheme.
- PRF and SK-DEPIR key initialization for all levels: for every level $1 \le i \le \ell$, set $\widetilde{K}^i = \widetilde{\mathsf{sk}}^i = \perp$. (Later, $\widetilde{K}^i, \widetilde{\mathsf{sk}}^i$ will contain encryptions of level-specific PRF and SK-DEPIR keys, respectively.)
- Initializing level ℓ: encrypt the database by running $\mathsf{DB}'' \leftarrow \mathsf{Encrypt}\left(\mathsf{sk}, \mathsf{DB}'\right)$. Run $\left(\mathsf{DB}'', \widetilde{\mathsf{DB}}, \widetilde{K}^{\ell\prime}, \widetilde{\mathsf{sk}}^{\ell\prime}\right) \leftarrow \mathsf{InitLevel}\left(\ell, \mathsf{DB}''\right)$ (Figure 1 on page 16) to obtain the processed SK-DEPIR database $\widetilde{\mathsf{DB}}$, and the PRF and SK-DEPIR keys for level ℓ. Initialize level ℓ to be $\mathsf{L}^\ell = \left(\mathsf{DB}'', \widetilde{\mathsf{DB}}\right)$, and all other levels L^i to be empty. Replace $\widetilde{K}^\ell, \widetilde{\mathsf{sk}}^\ell$ with $\widetilde{K}^{\ell\prime}, \widetilde{\mathsf{sk}}^{\ell\prime}$, respectively.
- Output: the client state $\mathsf{ck} = \mathsf{sk}$ consists of the encryption key. The server state $\mathsf{st} = \left(\mathsf{count}_W, \left(\mathsf{L}^i, \widetilde{K}^i, \widetilde{\mathsf{sk}}^i\right)_{i\in[\ell]}\right)$ consist of the counter, the contents of all levels, and the (encrypted) PRF and SK-DEPIR keys for all levels (which are currently empty, except for the keys of level ℓ).

The Access **protocol.** To perform the operation op on location $\mathsf{addr} \in [N]$ in the database with value val, the client C with state $\mathsf{ck} = \mathsf{sk}$, and the server with state $\mathsf{st} = \left(\mathsf{count}_W, \left(\mathsf{L}^i, \widetilde{K}^i, \widetilde{\mathsf{sk}}^i\right)_{i\in[\ell]}\right)$ operate as follows.

- **If** $\mathsf{op} = \mathtt{read}$**:**
  - Initialize an output value val' to $\perp$.
  - For every non-empty level i from 1 to ℓ, do:

* Computing bucket index: read $\widetilde{K}^i, \widetilde{\mathsf{sk}}^i$ from the server, and decrypt $K^i = \mathsf{Decrypt}\left(\mathsf{sk}, \widetilde{K}^i\right), \mathsf{sk}^i = \mathsf{Decrypt}\left(\mathsf{sk}, \widetilde{\mathsf{sk}}^i\right)$. Compute $l = F\left(K^i, \mathsf{addr}\right)$. (If addr appears in level i, it will be in the l'th bucket.)
* Looking for data block addr in level i: look for block addr in the l'th bucket by running the procedure $\mathsf{ReadBucket}\left(l, i, \mathsf{sk}^i, \mathsf{addr}\right)$ of Figure 2 to obtain a value val^i. If $\mathsf{val}' \neq \perp$ then set $\mathsf{val}' := \mathsf{val}^i$.

- Output: output val' to the client.

If op = `write`**:**

- Encrypt the data block as $c \leftarrow \mathsf{Encrypt}\left(\mathsf{sk}, (\mathsf{addr}, \mathsf{val})\right)$, and generate a "dummy" level 0 database which contains a single (encrypted) data block c.
- Update the server state as follows:
  - $\mathsf{count}_W := \mathsf{count}_W + 1$.
  - For $i = 0, 1, \ldots, \ell$ such that 2^i divides count_W, reshuffle level i into level $i+1$ using the ReShuffle procedure of Figure 3, namely executes $\mathsf{ReShuffle}\left(i, \mathsf{L}^i, \mathsf{L}^{i+1}\right)$.[6]

4 Definition of RAM-FHE

We first informally describe the RAM model we work with, which is a simple model of RAM computation that captures their essential efficiency advantage over Turing machines. Specifically, we define RAM machines via a transition circuit δ, with the following functionality. The circuit δ is designed to be evaluated repeatedly in a prescribed way, such that the main output of the i'th evaluation is an operation on one of the RAM machine's tapes, which is either the "persistent" tape containing the database, a volatile work tape which we call the "scratch tape", or the input and output tapes. The main input to δ is the result of the previously outputted operation. Additionally, the circuit δ simulates statefulness by taking as input and producing as output an internal state. We now define single-hop RAM-FHE. (See full version [HHWW] for the formal definition of RAM model and the multi-hop version.)

Definition 6 (Single-hop RAM FHE). *A* public-key (single-hop) RAM FHE scheme *is a tuple of PPT*[7] *algorithms (*KeyGen, Enc, Dec, Eval*) such that:*

- ***Syntax.***
  - $\mathsf{KeyGen}\left(1^\lambda\right)$ *takes as input a security parameter* λ*, and outputs public and secret keys* pk, sk.

[6] Using a technique of Ostrovsky and Shoup [OS97], these operations can be be spread-out over multiple `write` operations. We analyze the scheme below assuming the reshuffle operations are indeed spread-out across all `write` operations.

[7] In fact, in our construction Eval and Dec are deterministic.

The InitLevel procedure

Constant: the encryption key sk, and the security parameter λ.
Inputs:

i**:** the index of the level to initialize.
DB^i**:** a size-2^i database DB^i encrypted using $\mathsf{Encrypt}(\mathsf{sk}, \cdot)$.

Operation:

1. Pick a (fresh) random PRF key $K^{i\prime}$ for level i, generate a (fresh) SK-DEPIR key $\mathsf{sk}^{i\prime} \leftarrow \mathsf{DEPIR.KeyGen}\left(1^\lambda\right)$ for level i, and encrypt the keys by running $\widetilde{K}^{i\prime} \leftarrow \mathsf{Encrypt}\left(\mathsf{sk}, K^{i\prime}\right), \widetilde{\mathsf{sk}}^{i\prime} \leftarrow \mathsf{Encrypt}\left(\mathsf{sk}, \mathsf{sk}^{i\prime}\right)$.
2. Generate 2^i buckets, each with B "empty" blocks,[a] and encrypt the bucket contents using $\mathsf{Encrypt}$.
3. Randomly and obliviously permute DB^i using the Fisher-Yates shuffle, to obtain a permuted database $\widehat{\mathsf{DB}}^i$. In each step of the shuffle, the blocks touched during that step are re-encrypted. (That is, if a step of the shuffle touches blocks i, j then these blocks are downloaded from the server, decrypted, encrypted with fresh randomness, and then uploaded to the server again, in the correct order as determined by the shuffle.)
4. Insert $\widehat{\mathsf{DB}}^i$ into the buckets as follows. For every $1 \leq j \leq 2^i$, compute the index l of the bucket into which block j is mapped, as follows:
   - If block j is "empty", then pick l at random from 2^i.
   - Otherwise, let addr be the logical address of block j (recall that each block contains its logical address). Set $l = F\left(K^{i\prime}, \mathsf{addr}\right)$.

   Insert block j into bucket l by downloading the entire bucket l from the server, decrypting all blocks in the bucket, replacing the first "empty" block with block j, encrypting each block in the bucket, and reloading the bucket to the server.[b]
5. Run $\mathsf{Process}\left(\mathsf{sk}^{i\prime}, L\right)$ to obtain a processed database $\widetilde{\mathsf{DB}}^i$, and output $\left(\mathsf{DB}^i, \widetilde{\mathsf{DB}}^i, \widetilde{K}^{i\prime}, \widetilde{\mathsf{sk}}^{i\prime}\right)$.

[a] See remark on physical memory block contents in full version for a discussion of empty blocks.

[b] To obtain perfect correctness, if a bucket overflows then the contents of the level are stored "in the clear" (i.e., the block encryptions are stored in an array). As we show in the full version, this happens with negligible probability.

Fig. 1: The InitLevel procedure used in Construction 1

The ReadBucket procedure

Input:

l**:** the index of the bucket to read.
i**:** the index of the level in which the bucket resides.
sk^i**:** the secret key of the SK-DEPIR of level i.
addr: the address of the block to read.

Operation: recall that B denotes the bucket size.

- Initialize an output value val to $\perp$.
- For every $(l-1)\cdot\mathsf{B}+1 \le m \le l\cdot\mathsf{B}$:
  - Run $\mathsf{Query}\left(\mathsf{sk}^i, m\right)$ to obtain queries Q and a short-term client state st_C, send Q to S, and obtains answers $\{a_j\}_{j\in Q}$.
  - Run $\mathsf{Decode}\left(\mathsf{sk}^i, \mathsf{st}_C, \{a_j\}_{j\in Q}\right)$ to obtain value $(\mathsf{addr}^m, \mathsf{val}^m)$.
  - If $\mathsf{addr}^m = \mathsf{addr}$ then set $\mathsf{val} := \mathsf{val}^m$.
- Output val.

Fig. 2: The ReadBucket procedure used in Construction 1

- $\mathsf{Enc}\left(\mathsf{pk}, D, 1^B\right)$ *takes as input a pubic key* pk*, a database* D*, and a bound* B *on the description size of RAM machines. It outputs a database-ciphertext* $\hat{D}$*. For improved efficiency, it may also take as input a bound* s *(in unary) on the space usage of the RAM machines for which homomorphic evaluation will be supported.*
- $\mathsf{Eval}\left(M, x, 1^T\right)$ *takes as input a description* M *of a RAM machine, an input* x*, and a running time bound* T*, and is given read/write random-access to a database-ciphertext* $\hat{D}$*.* Eval *outputs an output-ciphertext* $\hat{y}$*, and may also change the contents of* $\hat{D}$ *to some new value* $\hat{D}'$*. We write* $(\hat{y}, \hat{D}') = \mathsf{Eval}^{\hat{D}}(M, x, 1^T)$*.*
- $\mathsf{Dec}\left(\mathsf{sk}, \hat{y}\right)$ *takes as input a secret key* sk *and an output-ciphertext* $\hat{y}$*, and outputs a plaintext message* y*.*

– ***Correctness.*** *For any security parameter* λ*, any size bound* B*, any RAM machine* M *satisfying* $|M| \le B$*, any database* $D \in \{0,1\}^*$*, any input* x*, and any* $T \in \mathbb{Z}^+$ *with* $\mathsf{Time}(M, x, D) \le T$*, in the probability space defined by sampling*

$$\begin{aligned} &(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ &\hat{D} \leftarrow \mathsf{Enc}(\mathsf{pk}, D, 1^B) \\ &\left(\hat{y}, \hat{D}'\right) := \mathsf{Eval}^{\hat{D}}(M, x, 1^T) \\ &(y, D') := M^D(x) \\ &y' := \mathsf{Dec}(\mathsf{sk}, \hat{y}), \end{aligned} \tag{1}$$

it holds that $y = y'$ *except with* $\mathsf{negl}\left(\lambda\right)$ *probability.*

The ReShuffle procedure

Constant: the encryption key sk.

Inputs:

i**:** the index of a level to reshuffle.

$\left(\mathsf{DB}^j, \widetilde{\mathsf{DB}}^j\right), j \in \{i, i+1\}$**:** the databases DB^j (encrypted with $\mathsf{Encrypt}(\mathsf{sk}, \cdot)$), and the processed databases $\widetilde{\mathsf{DB}}^j$, of levels $i, i+1$.

Operation:

1. For $j \in \{i, i+1\}$, if DB^j is empty (because it was not initialized yet, or following a previous reshuffle), instantiate DB^j with 2^j "empty" blocks, encrypted with $\mathsf{Encrypt}(\mathsf{sk}, \cdot)$. (See full version for a discussion of empty blocks.)
2. For $j \in \{i, i+1\}$, perform a linear scan of DB^j, concatenating encryptions of the label "$j - i$" to all blocks. (That is, level-i blocks are given label 0, and blocks from level $i+1$ are given label 1.)
3. Let A be the array of size $\left(2^i + 2^{i+1}\right)$ obtained by concatenating $\mathsf{DB}^i, \mathsf{DB}^{i+1}$.
4. Obliviously sort A according to block addresses, breaking ties using the labels created in Step 2. Each touched block is re-encrypted before being uploaded to the server. (After this step, duplicate block copies appear consecutively, and the copy from level i appears first.)
5. Perform a linear scan over A, replacing all duplicate blocks with "empty" blocks, and updating the labels (created in Step 2) of all non-duplicate blocks to 0. This is done as follows: the client locally stores the address of the previous block in A (initialized to 0). When traversing the current block, if its address is the same as the previous block, then replace the block with an "empty" block with label 1, otherwise update the block label to 0. Each block is re-encrypted before being uploaded to the server.
6. Obliviously sort A according to the labels, breaking ties according to block addresses. Each touched block is re-encrypted before being uploaded to the server. (After this step, real blocks appear before "empty" blocks.)
7. Perform a linear scan over A, removing the labels. Truncate A to size 2^{i+1}. (Notice that the truncated A still contains the freshest version of all blocks from $\mathsf{DB}^i, \mathsf{DB}^{i+1}$.)
8. Run the procedure $\left(\mathsf{DB}^{i+1\prime}, \widetilde{\mathsf{DB}}^{i+1\prime}, \widetilde{K}^{i+1\prime}\widetilde{\mathsf{sk}}^{i+1\prime}\right) \leftarrow \mathsf{InitLevel}(i+1, A)$ of Figure 1 to obtain the processed database $\widetilde{\mathsf{DB}}^{i+1\prime}$ of level $i+1$, and fresh (encrypted) PRF and SK-DEPIR keys $\widetilde{K}^{i+1\prime}, \widetilde{\mathsf{sk}}^{i+1\prime}$ (respectively). Replace $\widetilde{K}^{i+1}, \widetilde{\mathsf{sk}}^{i+1}$ with $\widetilde{K}^{i+1\prime}, \widetilde{\mathsf{sk}}^{i+1\prime}$ (respectively). Update level i to be empty $\mathsf{L}^i = \perp$, and level $i+1$ to $\mathsf{L}_{i+1} = \left(\mathsf{DB}^{i+1\prime}, \widetilde{\mathsf{DB}}^{i+1\prime}\right)$.

Fig. 3: The ReShuffle protocol used in Construction 1

– ***IND-CPA Security.*** *For all non-uniform PPT* $\mathcal{A}_0$ *and* $\mathcal{A}_1$*, there is a negligible function* negl *such that for every security parameter* λ*,*

$$\Pr\left[b' = b \ : \begin{array}{r}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ (\mathsf{st}, D_0, D_1, 1^B) := \mathcal{A}_0(\mathsf{pk}) \\ b \leftarrow \{0,1\} \\ \hat{D} \leftarrow \mathsf{Enc}(\mathsf{pk}, D_b, B) \\ b' := \mathcal{A}_1(\mathsf{st}, \hat{D})\end{array}\right] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

– $\eta\left(|D|\right)$***-Efficiency.*** *With probability* 1*, the running time of* Eval *in the experiment described in Eq.* (1) *is at most* $T \cdot \eta\left(|D|\right) \cdot \mathsf{poly}\left(B, \lambda\right)$*.*
– ***Compactness.*** *In the experiment described in Eq.* (1)*,* $|\hat{y}| \leq \mathsf{poly}(\log|\mathcal{Y}|, \lambda)$*.*

Remark 1. We note that when Enc is executed with the additional space-bound parameter s, then correctness holds for every RAM machine M whose volatile tape throughout the execution has size at most s, and the adversary in the security game is also allowed to choose s.

5 Road Map Towards Constructing RAM-FHE

As described in Section 1.2, the encryption of a database D consists of the server state in a rewindable ORAM for D, together with a VBB obfuscation of the circuit that emulates a single execution step of the rewindable ORAM client. Formalizing this idea requires two steps. First, we need to emulate a *consistent* client state throughout the execution (because the ORAM client is stateful, while the obfuscated circuit is not), as well as guarantee semi-honest emulation of the ORAM server. This covers steps (1) and (3) from Section 1.2. Second, we need to hide the ORAM client state from the evaluator, using pseudorandom bits for encryption, which was described as steps (2) and (5) in Section 1.2. We obtain both of these using a new abstraction which we call a database-dependent RAM-VBB obfuscator (Section 5.1) in which, informally, the obfuscator takes as input not only a database D, but also a *specific* RAM machine M, and the evaluator can run M on different inputs x with RAM access to (the mutable) D. We provide two constructions (Section 5.2) to handle each of the issues described above. We obtain the RAM-FHE by applying the RAM-VBB obfuscator to the universal RAM machine (which takes as input a description M of a RAM machine, and an input x for it, and outputs $M^D\left(x\right)$, where D is the database), that additionally encrypts its output using a PKE scheme (step (4) in Section 1.2).

5.1 Database-Dependent RAM-VBB Obfuscation

We define two notions of RAM-VBB obfuscation, in which the RAM machine is obfuscated with relation to a specific database. These notions, which we call *database-dependent* RAM-VBB, provide weaker security than RAM-FHE, and incomparable correctness. We note that though such obfuscation is unlikely to exist in general, similar to circuit-VBB obfuscation it might exist for restricted

ensembles of RAM machines, and in particular might exist for the *specific* ensemble we consider in this work.

Informally, the obfuscator $\mathcal{O}$ is parameterized by an ensemble $\mathcal{M} = \{\mathcal{M}_N\}_N$ of classes of RAM programs. It takes as input not only a database $D_0 \in \{0,1\}^N$, but also a RAM machine $M \in \mathcal{M}_N$. The evaluator is able to compute $M^D(x)$ for any input x and any database D that is either D_0 or was obtained by a previous execution of M. Formally,

Definition 7 (Database-dependent RAM-VBB obfuscator). *Let $n \in \mathbb{N}$ be an input length, $N \leq 2^\lambda$ be a database size, and $\mathcal{M} = \{\mathcal{M}_N\}_N$ be an ensemble of classes of RAM programs. A* database-dependent RAM-VBB obfuscator *for $\mathcal{M}$ is an algorithm $\mathcal{O}$ that takes as input a security parameter 1^λ, a database $D_0 \in \{0,1\}^N$, and a RAM machine $M \in \mathcal{M}_N$. It outputs a database $\widetilde{D}_0$, a RAM machine $\widetilde{M}$, and some auxiliary input $\mathcal{I}_0$ for $\widetilde{M}$. We require that $\mathcal{O}$ satisfies the following requirements:*

- ***Correctness.*** *For every $n, k, N \in \mathbb{N}$, every $M \in \mathcal{M}_N$, every database $D_0 \in \{0,1\}^N$, and every inputs $x_1, \ldots, x_m \in \{0,1\}^n$, the following two experiments yield the same values of $(y_1, \ldots, y_m) \in \left(\{0,1\}^k\right)^m$ except with* $\mathsf{negl}(\lambda)$ *probability.*

$$\begin{aligned}
&(\widetilde{D}_0, \widetilde{M}, \mathcal{I}_0) \leftarrow \mathcal{O}(1^\lambda, D_0, M)\\
&(y_1, \widetilde{D}_1, \mathcal{I}_1) \leftarrow \widetilde{M}^{\widetilde{D}_0}(x_1, \mathcal{I}_0)\\
&\ldots\\
&(y_m, \widetilde{D}_m, \mathcal{I}_m) \leftarrow \widetilde{M}^{\widetilde{D}_{m-1}}(x_m, \mathcal{I}_{m-1})
\end{aligned}$$

  and

$$\begin{aligned}
&(y_1, D_1) \leftarrow M^{D_0}(x_1)\\
&\ldots\\
&(y_m, D_m) \leftarrow M^{D_{m-1}}(x_m)
\end{aligned} \tag{2}$$

- ***Efficiency.*** *In the above experiments, it holds that*

$$\mathsf{Time}(\widetilde{M}, (x_i, \mathcal{I}_{i-1}), \widetilde{D}_{i-1}) \leq \mathsf{Time}(M, x_i, D_{i-1}) \cdot \mathsf{poly}(|M|, \lambda)$$

  where $|M|$ denotes the combined length of the internal state and the description of M.

We define two security notions for database-dependent RAM-VBB obfuscation. The first, which we call transcript-simulable, is roughly that any adversary (with single-bit output) given an obfuscation of (D_0, M) is simulatable given only the execution trace (see full version [HHWW]for definition), namely given oracle access to the function that takes a sequence of inputs $x_1, \ldots, x_d$, and returns the operations performed by M when sequentially executed (i.e., with a mutable database D that is initially D_0 but persists across executions) on the inputs $x_1, \ldots, x_d$. The second security property, which we call address simulatable, is stronger since it gives the simulator less information. Specifically, the simulator no longer sees the entire computation transcripts but instead sees only

the *addresses* of the physical memory which are operated on, the *type* (read or write) of memory operation, and the outcome of the computation. The simulator does *not* see the *values* read from / written to memory, or the contents D_0 of the initial database, but instead sees only its size $|D_0|$. These definitions appear in the full version [HHWW]. We abbreviate transcript/address-simulatable database-dependent RAM-VBB as transcript/address-simulatable RAM-VBB.

5.2 Database-Dependent RAM-VBB Obfuscation: Constructions

In this section we construct (single-hop) transcript-simulatable and address-simulatable RAM-VBB obfuscators. These will be used in Section 6 to construct a RAM-FHE scheme.

In the single hop setting, we can assume without loss of generality that the database is read-only, since database updates can be emulated in the scratch tape, causing a multiplicative factor-2 increase in the scratch tape size, and the number of `read` accesses. Therefore, we can (by performing dummy accesses if needed) assume without impact that every execution step performs a single `read` from the database and scratch tape, and a single `write` to the scratch tape.

We now construct a single-hop transcript-simulatable RAM-VBB obfuscator (see full version for a multi-hop variant). The high level idea is to use MACs and Merkle hash trees to enforce consistent execution, and to obfuscate the transition circuit (computing the transition function δ of the RAM machine) which has the MAC key hard-wired into it. This intuition is formalized in the next construction. In the full version [HHWW], we prove the following construction is a transcript Simulatable RAM-VBB obfuscator.

Construction 2 (Transcript-simulatable RAM-VBB obfuscation)**.** The transcript-simulatable RAM-VBB obfuscator $\mathcal{O}_{\mathsf{trans}}$ uses:

- A family $\mathcal{H}$ of hash functions.
- A MAC scheme $(\mathsf{KeyGen}, \mathsf{Tag}, \mathsf{Verify})$, in which $\mathsf{Tag}, \mathsf{Verify}$ are deterministic (this assumption is without loss of generality).
- A circuit obfuscator $\mathcal{O}$.

Given a security parameter λ, a database D_0, and a RAM machine M, $\mathcal{O}_{\mathsf{trans}}$:

- Generates a random MAC key $K_{\mathsf{MAC}} \leftarrow \mathsf{KeyGen}\left(1^{\lambda}\right)$, and picks a description of a hash function $h \leftarrow \mathcal{H}$.
- Generate a MHT MT for D_0, and let Rt denote its root.
- Let st_M denote the initial state of the RAM machine M, set $\mathsf{st} = (\mathsf{true}, \mathsf{st}_M, \mathsf{Rt})$, and pad st with zeros to have the same size as st in Figure 4. (The boolean value true in st indicates that the execution hasn't started yet.) $\mathcal{O}_{\mathsf{trans}}$ generates a tag $\sigma = \mathsf{Tag}\left(K_{\mathsf{MAC}}, (\mathsf{false}, \mathsf{st})\right)$. (The signature is on the state st, as well as a boolean variable b_{fin} indicating whether the execution has already terminated.)
- Runs the obfuscator $\widetilde{C} \leftarrow \mathcal{O}\left(1^{\lambda}, C_{\mathsf{Exec}}\right)$ to obfuscate the circuit C_{Exec} described in Figure 5, with the constants described in Figure 4 hard-wired into it.

– Outputs $\left(\mathsf{MT}, M_{\mathsf{wrap}}, \mathcal{I} = \left(\mathsf{st}, \sigma, \widetilde{C}\right)\right)$, where M_{wrap} is the RAM machine described in Figure 6.

The constants and inputs used in the circuit C_{Exec} of Figure 5

Constants: a description h of a CRHF, and a key K_{MAC} for a MAC scheme.

Inputs:

$x \in \{0,1\}^n$**:** an input for the RAM machine M.

b_{fin}**:** a boolean variable indicating whether the computation has already terminated.

$\mathsf{st} = (b_{\mathsf{first}}, \mathsf{st}_M, \mathsf{Rt}, \mathsf{Rt}_{\mathsf{hist}}, \mathcal{P}_{\mathsf{hist}}, \mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}, \mathsf{addr}_w, \mathsf{val}_w, x')$**:** an internal state st, consisting of: a boolean variable b_{first} indicating whether this is the first operation, the internal state st_M of a RAM machine, the root Rt for a MHT MT for a database, the root $\mathsf{Rt}_{\mathsf{hist}}$ of a MHT $\mathsf{MT}_{\mathsf{hist}}$ of the history of accesses performed so far, and the path $\mathcal{P}_{\mathsf{hist}}$ to the right-most (i.e., last) node in $\mathsf{MT}_{\mathsf{hist}}$, addresses $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}$ read from the database and scratch tape (respectively) in the previous execution step, the value val_w written in the last execution step to address addr_w of the scratch tape, and an input x' for M.

σ**:** a MAC tag for $(b_{\mathsf{fin}}, \mathsf{st})$.

$\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$**:** the values at locations $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}$ (respectively) in the database and scratch tape, respectively.

$\mathcal{P}_{\mathsf{DB}}, \mathcal{P}_{\mathsf{stape}}, \mathcal{P}_w$**:** the paths of nodes $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}, \mathsf{addr}_w$ (respectively) in MT.

Fig. 4: Description of the constants and inputs of C_{Exec}

In full version [HHWW], we construct an address-simulatable RAM-VBB obfuscator from a transcript-simulatable RAM-VBB obfuscator. The high level idea is to apply the transcript-simulatable VBB obfuscator to a RAM program M that has a hard-wired encryption key, which the transition circuit uses to encrypt the internal state. One issue that arises is how to generate randomness for encryption, when M cannot toss coins. This is done by applying a PRF to the current execution state. We also include a counter in the internal state to guarantee that the states are unique throughout the execution.

6 A RAM-FHE Scheme

In this section we describe our single-hop RAM-FHE scheme, which uses an address-simulatable RAM-VBB as a building block. We assume that (polynomial) a-priori bounds on the input, output, and description lengths of the RAM machine are known. In the full version [HHWW], we discuss extensions to the general setting (in which no such bounds are a-priori known) and how we upgrade the scheme to a multi-hop scheme. Concretely, we prove (in the full version) the following theorem:

The execution circuit C_{Exec}

Constants and inputs: as described in Figure 4.

1. **Verify input consistency:** verify that $\mathsf{Verify}\left(K_{\mathsf{MAC}}, \left(b_{\mathsf{fin}}, \mathsf{st}\right), \sigma\right) = 1$. If $b_{\mathsf{first}} = \mathsf{false}$ then verify additionally that:
   (a) $\mathcal{P}_{\mathsf{DB}}, \mathcal{P}_{\mathsf{stape}}, \mathcal{P}_w$ are paths to the nodes $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}, \mathsf{addr}_w$ in the MHT whose root is Rt, and the values at $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}$ are $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ (respectively).
   (b) $\mathcal{P}_{\mathsf{hist}}$ is the path to the right-most node in the MHT whose root is $\mathsf{Rt}_{\mathsf{hist}}$.
   (c) $x' = x$. (Verifying the same input is used throughout the execution.)
   If any of these checks fail, output abort.
2. **Updating database MHT:** if $b_{\mathsf{first}} = \mathsf{false}$, use $\mathcal{P}_w$ to compute the root Rt' of the MHT obtained from MT by replacing the value of the node addr_w with val_w.
3. **Emulating next transition step:**
   (a) Execute the next command of M, as described in st_M (which indicates which command is next), by applying the transition function δ using $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ as the values obtained from the last command executed. (If $b_{\mathsf{first}} = \mathsf{true}$ then $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ are not needed and are therefore ignored.) The execution results in an updated internal state st'_M of M (i.e., the state outputted by the transition circuit).
   (b) If M terminated in the current step with output y, then set $b_{\mathsf{fin}} = \mathsf{true}$ and $\mathsf{out} = (b_{\mathsf{fin}}, y)$, and go to Step 5.
   (c) Otherwise, the execution step results in accesses $\mathsf{addr}'_{\mathsf{DB}}, \mathsf{addr}'_{\mathsf{stape}}, (\mathsf{addr}'_w, \mathsf{val}'_w)$ reading addresses $\mathsf{addr}'_{\mathsf{DB}}, \mathsf{addr}'_{\mathsf{stape}}$ from the database and scratch tape (respectively), and writing value val'_w to address addr'_w in the scratch tape. Set $b_{\mathsf{fin}} = \mathsf{false}$.
4. **Updating history and state:** set $\mathsf{leaves} = \left(\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}, \mathsf{addr}'_{\mathsf{DB}}, \mathsf{addr}'_{\mathsf{stape}}, \mathsf{addr}'_w, \mathsf{val}'_w\right)$. If $b_{\mathsf{first}} = \mathsf{false}$, use $\mathcal{P}_{\mathsf{hist}}$ to compute the root $\mathsf{Rt}'_{\mathsf{hist}}$ of the MHT $\mathsf{MT}'_{\mathsf{hist}}$ obtained from $\mathsf{MT}_{\mathsf{hist}}$ by adding the leaves leaves. If $b_{\mathsf{first}} = \mathsf{true}$, set $b_{\mathsf{first}} = \mathsf{false}$ and generate a (new) MHT $\mathsf{MT}'_{\mathsf{hist}}$ for leaves (see remark on page 26). In either case, let $\mathcal{P}'_{\mathsf{hist}}$ be the path to the right-most node in $\mathsf{MT}'_{\mathsf{hist}}$, and set $\mathsf{st}' = \left(b_{\mathsf{first}}, \mathsf{st}'_M, \mathsf{Rt}', \mathsf{Rt}'_{\mathsf{hist}}, \mathcal{P}'_{\mathsf{hist}}, \mathsf{addr}'_{\mathsf{DB}}, \mathsf{addr}'_{\mathsf{stape}}, \mathsf{addr}'_w, \mathsf{val}'_w, x\right)$, compute $\sigma' = \mathsf{Tag}\left(K_{\mathsf{MAC}}, (b_{\mathsf{fin}}, \mathsf{st}')\right)$, and set $\mathsf{out} = (b_{\mathsf{fin}}, \mathsf{st}', \sigma')$.
5. **Output:** return out.

Fig. 5: Description of the circuit used to emulate a single transition of the RAM machine

The wrapper program M_{wrap}

RAM access to: a MHT MT for a database D (which contains an initial database, concatenated with a scratch tape).

Inputs:

$x \in \{0,1\}^n$**:** an input for a RAM machine M.
st**:** the internal state used in the execution of C_{Exec}.
σ**:** a MAC tag for st.
$\widetilde{C}$**:** an obfuscation of the circuit C_{Exec} of Figure 5.

Operation:

1. **Initialization:** Set $\mathcal{P}_{\mathsf{hist}}$ to be empty (this is a "place holder" for the right-most path in a MHT $\mathsf{MT}_{\mathsf{hist}}$ of the access history so far), and $z = (x, \mathsf{false}, \mathsf{st}, \sigma)$, padded with zeros to have the size of inputs to $\widetilde{C}$.
2. Repeat:
   - **Emulate an execution step:** execute $(b_{\mathsf{fin}}, y) = \widetilde{C}(z)$ to obtain a bit b_{fin} indicating whether the execution has terminated, and an additional output y.
   - **Generate output when the execution ends:** if $b_{\mathsf{fin}} = \mathsf{true}$ then output y and halt.
   - **Emulate database and scratch tape accesses:**
     - Interpret y as (st', σ'), where σ' is a MAC tag for $(b_{\mathsf{fin}}, \mathsf{st}')$, and st' is the current internal execution state, consists of: a bit b, the current internal state st_M of M (as outputted by the transition circuit), the root Rt of an updated MHT for D, a root $\mathsf{Rt}_{\mathsf{hist}}$ for a MHT $\mathsf{MT}_{\mathsf{hist}}$, the path $\mathcal{P}_{\mathsf{hist}}$ to the right-most node in $\mathsf{MT}_{\mathsf{hist}}$, addresses $\mathsf{addr}_{\mathsf{DB}}$, $\mathsf{addr}_{\mathsf{stape}}$ to read from D, a value val_w to write to address addr_w of D, and an input x' for M.
     - Read the path $\mathcal{P}_{\mathsf{DB}}$ to the node $\mathsf{addr}_{\mathsf{DB}}$ in MT, and let $\mathsf{val}_{\mathsf{DB}}$ be the value of the node.
     - Read the path $\mathcal{P}_{\mathsf{stape}}$ to the node $\mathsf{addr}_{\mathsf{stape}}$ in MT, and let $\mathsf{val}_{\mathsf{stape}}$ be the value of the node.
     - Read the path $\mathcal{P}_w$ to the node addr_w in MT. Replace the value in addr_w with val_w, and update its path in MT.
   - **Compute input for next execution step:** Set $z = (x, b_{\mathsf{fin}}, \mathsf{st}', \sigma', \mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}, \mathcal{P}_{\mathsf{DB}}, \mathcal{P}_{\mathsf{stape}}, \mathcal{P}_w)$.

Fig. 6: Description of the wrapper RAM machine

Theorem 4 (Single-hop RAM-FHE). *Assume the existence of OWFs, CRHFs, PKE schemes, and SK-DEPIR which for size-N databases has* $\mathsf{poly}(\lambda, \log N)$ Query *and* Decode *complexity, where λ denotes the security parameter. Then for every $d = \mathsf{poly}(\lambda)$ there exists a* $\mathsf{poly} \log N$*-efficient single-hop RAM-FHE scheme in the circuit-VBB hybrid model for RAM machines with input length, output length, description size, and space usage at most d.*

The Construction. The high level idea is to combine the address-simulatable RAM-VBB for the universal RAM machine, with an ISR-ORAM (which is replaced with an ASR-ORAM in the multi-hop setting). The address-simulatable RAM-VBB guarantees that the RAM machine emulation only reveals the sequence of physical memory addresses it accesses, which by ISR-ORAM security reveals no information about the access pattern to logical memory. One technical issue is that the universal machine should encrypt its output (using a persistent encryption key that is generated during KeyGen, independent of the database and any RAM machine that will be run on it) which requires generating randomness. We use a PRF to generate this randomness.

Construction 3 (Single-hop RAM-FHE)**.** The RAM-FHE scheme uses:

- An address-simulatable RAM-VBB obfuscator $\mathcal{O}$.
- An ISR-ORAM scheme $(\mathsf{ISR-ORAM.Setup}, \mathsf{ISR-ORAM.Access})$ with a deterministic client during $\mathsf{ISR-ORAM.Access}$.
- A PKE scheme $(\mathsf{PKE.KeyGen}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$.
- An unbounded-input PRF F.

It consists of the following algorithms:

- $\mathsf{KeyGen}\left(1^\lambda\right)$ generates a public-secret key pair $\left(\mathsf{pk}', \mathsf{sk}'\right) \leftarrow \mathsf{PKE.KeyGen}\left(1^\lambda\right)$, and outputs $\left(\mathsf{pk} = \left(1^\lambda, \mathsf{pk}'\right), \mathsf{sk} = \mathsf{sk}'\right)$.
- $\mathsf{Encrypt}\left(\mathsf{pk} = \left(1^\lambda, \mathsf{pk}'\right), \mathsf{DB}, 1^d, 1^s\right)$ takes as input a public key pk, a database DB, and bounds d, s on the description size and space usage of RAM machines (respectively). It operates as follows:
  - Set DB' to be the database of size $|\mathsf{DB}| + s$ obtained by concatenating s empty blocks to DB.(Intuitively, these blocks are "place holders" for the contents of the scratch tape of a RAM machine; see remark on physical memory block contents in the full version for a discussion of empty blocks.)
  - Initialize an ISR-ORAM with DB', by running $\mathsf{ISR-ORAM.Setup}\left(1^\lambda, \mathsf{DB}'\right)$, to obtain a client state $\mathsf{ck}_{\mathsf{ISR}}$ and a server state $\mathsf{st}_{\mathsf{ISR}}$.
  - Pick a random PRF key $K \leftarrow \{0,1\}^\lambda$.
  - Run $\left(\widetilde{\mathsf{DB}}, \widetilde{M_{\mathcal{U}}}, \mathcal{I}\right) \leftarrow \mathcal{O}\left(1^\lambda, \mathsf{st}_{\mathsf{ISR}}, M_{\mathcal{U}}\right)$, where $M_{\mathcal{U}}$ is the RAM machine described in Figure 7, with hard-wired values $|\mathsf{DB}|, \mathsf{pk}', K$, and internal variable $\mathsf{ck}_{\mathsf{ISR}}$.
  - Output the ciphertext $c_{\mathsf{DB}} = \left(\widetilde{\mathsf{DB}}, \widetilde{M_{\mathcal{U}}}, \mathcal{I}\right)$.

– $\mathsf{Eval}^{c_{\mathsf{DB}}}\left(M, x, 1^T\right)$ takes as input a description M of size at most d of a RAM machine, an input x for M, and a bound T on the runtime of M. It also has RAM access to a database-ciphertext $c_{\mathsf{DB}} = \left(\widetilde{\mathsf{DB}}, \widetilde{M_{\mathcal{U}}}, \mathcal{I}\right)$. It runs $\widetilde{M_{\mathcal{U}}}^{\widetilde{\mathsf{DB}}}\left(M, 1^T, x, \mathcal{I}\right)$, and outputs whatever it outputs.
– $\mathsf{Decrypt}\left(\mathsf{sk}, c\right)$ takes as input a secret key sk, and an output-ciphertext c. It outputs $\mathsf{PKE.Decrypt}\left(\mathsf{sk}, c\right)$.

The RAM machine $M_{\mathcal{U}}$ with RAM access to $\widetilde{\mathsf{DB}}$

Hard-wired value: a database size N, a public key pk for a PKE scheme, and a PRF key K.
Internal variables:

ck: a client state in an ISR-ORAM.
y: the output of a RAM machine (initialized to 0).
fin: a boolean variable indicating whether the execution has terminated or not (initialized to false).
count: a counter of the number of operations performed so far (initialized to 0).

Inputs:

M: a description (of length at most d) of a RAM machine.
T: a bound on the runtime of M.
$x \in \{0,1\}^*$: the input for the RAM machine M.

Operation:

1. **Initialize the run:** set $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ to be empty. (This is the first operation, no values were previously read from the database and scratch tape.)
2. **Execute M for T steps:** for $i = 1, \ldots, T$, do:
   – **Emulate a transition step:** execute the procedure from Figure 8 with $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ as the values read from the database and the scratch tape, respectively.
   – **Access DB and scratch tape:** if $\mathsf{count} \leq T$ then for $j = 1, 2, 3$: execute the procedure from Figure 9.
3. **Output:** set $r = F\left(K, (M, T, x)\right)$, encrypt $c = \mathsf{PKE.Encrypt}\left(\mathsf{pk}, y; r\right)$ and output c.

Fig. 7: RAM machine used in Construction 3

Remark on growing Merkel Hash Trees. Our construction (in particular, the circuit C_{Exec} of Figure 5 on page 23) generate and grow MHTs. The hash trees use an underlying hash function $H : \{0,1\}^{2n} \to \{0,1\}^n$ for some $n \in \mathbb{N}$. Generating a MHT T for a string s is done in the standard way by hashing adjacent pairs of nodes repeatedly, and we say that the resultant tree T *represents* s. Growing

Emulating a single transition of M in $M_{\mathcal{U}}$

Hard-wired values, internal variables, M: as in Figure 7.

1. **If $\mathsf{fin} = \mathsf{true}$, perform a dummy step:** perform a no-op operation,[a] and set $\mathsf{addr}_{\mathsf{DB}} = 0$, $\mathsf{addr}_{\mathsf{stape}} = \mathsf{addr}_w = N$, and $\mathsf{val}_w = 0$ (these are dummy accesses which read address 0 from the database and scratch tape, and write 0 to address 0 of the scratch tape).
2. **If $\mathsf{fin} = \mathsf{false}$, perform the next transition step:**
   (a) Execute the next command of M using $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ as the values read from the database and the scratch tapes, respectively.
   (b) If M terminated in the current step with output out, then set $y = \mathsf{out}$ and $\mathsf{fin} = \mathsf{true}$.
   (c) Otherwise, if $\mathsf{count} < T$ then this step results in accesses $\mathsf{addr}_{\mathsf{DB}}$, $\mathsf{addr}_{\mathsf{stape}}$, $(\mathsf{addr}_w, \mathsf{val}_w)$ reading addresses $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}$ (respectively) from the database and scratch tape, and writing value val_w to address addr_w of the scratch tape. Set $\mathsf{addr}_{\mathsf{stape}} := \mathsf{addr}_{\mathsf{stape}} + N$, $\mathsf{addr}_w := \mathsf{addr}_w + N$. (This "translates" the addresses in the scratch tape to addresses in $\widetilde{\mathsf{DB}}$, since M's scratch tape appears in $\widetilde{\mathsf{DB}}$ after the size-N database.)
3. **Update the counter:** set $\mathsf{count} = \mathsf{count} + 1$.

[a] This is needed to hide whether a command of M was executed or not, which would reveal information about the actual runtime of M.

Fig. 8: Emulating a single transition of M

Emulating a database or scratch tape access of M in $M_{\mathcal{U}}$

Hard-wired value, internal variables, j: as in Figure 7.
Memory accesses $\mathsf{addr}_{\mathsf{DB}}$, $\mathsf{addr}_{\mathsf{stape}}$, $(\mathsf{addr}_w, \mathsf{val}_w)$: as in Figure 8.

1. **Determine ORAM client input:** if $j = 1$ (i.e., `read` from database) set $v = (\mathsf{addr}_{\mathsf{DB}}, \perp)$. If $j = 2$ (i.e., `read` from scratch tape) set $v = (\mathsf{addr}_{\mathsf{stape}}, \perp)$. If $j = 3$ (i.e., `write` to scratch tape), set $v = (\mathsf{addr}_w, \mathsf{val}_w)$.
2. **Initiate ORAM access:** run the ISR-ORAM client from state ck with input v, to obtain a query q to the physical memory, and an update instruction update to the physical memory. (This results also in an updated client state which is updated in $M_{\mathcal{U}}$'s internal state).
3. **Emulate ORAM access:** until the ORAM client halts, do:
   – Read the value val written in block q of $\widetilde{\mathsf{DB}}$, and perform update on $\widetilde{\mathsf{DB}}$.
   – Run the ORAM client from state ck, given val as the server's answer to the last query q. The client outputs either the next query q and an update instruction update to the physical memory, or an output value $\mathsf{val}_{\mathsf{out}}$ (in this case, the ORAM client halts; in either case, this also results in an updated ORAM client state).
4. **Output:** if $j = 1$ (i.e., a value was read from the database), set $\mathsf{val}_{\mathsf{DB}} = \mathsf{val}_{\mathsf{out}}$, and if $j = 2$ (i.e., a value was read from the scratch tape), set $\mathsf{val}_{\mathsf{stape}} = \mathsf{val}_{\mathsf{out}}$.

Fig. 9: Emulating a database or scratch tape access in M

an existing MHT T which represents a string s is done as follows. Assume T has hight h growing from the leaves to the root, and let $v_1, \ldots, v_h$ be the right-most nodes in each level of T, i.e., v_1 is a suffix of s, and v_h is the root. To generate a MHT representing the string $s \circ s'$ for some $s' \in \{0,1\}^n$, concatenate s' to level 1 of the tree as the new right-most node, and let $v'_1 := s'$. Compute a new right-most path in the tree by generating, for every $1 < i \leq h$ the node $v'_i = H\left(v_{i-1}, v'_{i-1}\right)$ and concatenating v'_i to the right of node v_i in level i. Finally, generate a new root at level $h+1$ by computing $H\left(v_h, v'_h\right)$. To grow T be a string of length $> n$, partition the string into length-n substrings, and apply this procedure sequentially on each of the substrings.

References

ACC+16. Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating RAM computations with adaptive soundness and privacy. In *TCC 2016-B, Proceedings, Part II*, pages 3–30, 2016.

Agr. Shweta Agrawal. New methods for indistinguishability obfuscation: Bootstrapping and instantiation. *IACR Cryptology ePrint Archive*, 2018:633.

AIT16. Afonso Arriaga, Vincenzo Iovino, and Qiang Tang. Updatable functional encryption. *IACR Cryptology ePrint Archive*, 2016:1179, 2016.

AJS18. Prabhanjan Ananth, Aayush Jain, and Amit Sahai. Indistinguishability obfuscation without multilinear maps: iO from LWE, bilinear maps, and weak pseudorandomness. *IACR Cryptology ePrint Archive*, 2018:615, 2018.

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 J. Comput.*, 47(3):1123–1210, 2018.

BCP. Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC 2016-A, Proceedings, Part II*, pages 175–204.

BGH13. Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in LWE-based homomorphic encryption. In *PKC 2013, Proceedings*, pages 1–13, 2013.

BGI+01. Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO 2001, Proceedings*, pages 1–18, 2001.

BGMZ18. James Bartusek, Jiaxin Guan, Fermi Ma, and Mark Zhandry. Return of GGH15: provable security against zeroizing attacks. In *TCC 2018, Proceedings, Part II*, pages 544–574, 2018.

BGV12. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS 2012, Proceedings*, pages 309–325. ACM, 2012.

BIPW17. Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *TCC 2017, Proceedings, Part II*, pages 662–693, 2017.

BMSZ16. Saikrishna Badrinarayanan, Eric Miles, Amit Sahai, and Mark Zhandry. Post-zeroizing obfuscation: New mathematical tools, and the case of evasive circuits. In *EUROCRYPT 2016, Proceedings, Part II*, pages 764–791, 2016.

BV11. Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *ECCC 2011*, 18:109, 2011.

CCC+. Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In *ITCS 2016, Proceedings*, pages 179–190.

CCH+19. Ran Canetti, Yilei Chen, Justin Holmgren, Alex Lombardi, Guy N. Rothblum, Ron D. Rothblum, and Daniel Wichs. Fiat-Shamir: From practice to theory. 2019.

CCHR16. Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled RAM or: How to delegate your database. In *TCC 2016-B, Proceedings, Part II*, pages 61–90, 2016.

CH16. Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. In *ITCS 2016, Proceedings*, pages 169–178, 2016.

CHJV15. Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In *STOC 2015, Proceedings*, pages 429–437, 2015.

CHR17. Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC 2017, Proceedings, Part II*, pages 694–726, 2017.

CR72. Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *STOC 1972, Proceedings*, pages 73–80, 1972.

CVW18. Yilei Chen, Vinod Vaikuntanathan, and Hoeteck Wee. GGH15 beyond permutation branching programs: Proofs, attacks, and candidates. In *CRYPTO 2018, Proceedings, Part II*, pages 577–607, 2018.

Gen09. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC 2009, Proceedings*, pages 169–178. ACM, 2009.

GGH+. Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *STOC 2013, Proceedings*, pages 40–49.

GGMP16. Sanjam Garg, Divya Gupta, Peihan Miao, and Omkant Pandey. Secure multiparty RAM computation in constant rounds. In *TCC 2016-B, Proceedings, Part I*, pages 491–520, 2016.

GGP10. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO 2010, Proceedings*, pages 465–482. Springer, 2010.

GHL+14. Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT 2014, Proceedings*, pages 405–422, 2014.

GHPS13. Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Field switching in BGV-style homomorphic encryption. *Journal of Computer Security*, 21(5):663–684, 2013.

GHRW. Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *FOCS 2014, Proceedings*, pages 404–413.

GHS12. Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT 2012, Proceedings*, pages 465–482. Springer, 2012.

GKP+a. Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run Turing machines on encrypted data. In *CRYPTO 2013, Proceedings, Part II*, pages 536–553.

GKP+b. Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *STOC 2013, Proceedings*, pages 555–564. ACM.

GLOS. Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *STOC 2015, Proceedings*, pages 449–458.

GMM+16. Sanjam Garg, Eric Miles, Pratyay Mukherjee, Amit Sahai, Akshayaram Srinivasan, and Mark Zhandry. Secure obfuscation in a weak multilinear map model. In *TCC 2016-B, Proceedings, Part II*, pages 241–268, 2016.

GO96. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

Gol87. Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC 1987, Proceedings*, pages 182–194, 1987.

Gol01. Oded Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

Gol04. Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

GOS18. Sanjam Garg, Rafail Ostrovsky, and Akshayaram Srinivasan. Adaptive garbled RAM from laconic oblivious transfer. In *CRYPTO 2018, Proceedings, Part III*, pages 515–544, 2018.

GSW. Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO 2013, Proceedings, Part I*, pages 75–92.

HHWW. Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. On the plausibility of fully homomorphic encryption for RAMs. *IACR Cryptology ePrint Archive*, 2019.

HY16. Carmit Hazay and Avishay Yanai. Constant-round maliciously secure two-party computation in the RAM model. In *TCC 2016-B, Proceedings, Part I*, pages 521–553, 2016.

KP16. Yael Tauman Kalai and Omer Paneth. Delegating RAM computations. In *TCC 2016-B, Proceedings, Part II*, pages 91–118, 2016.

KRR14. Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. How to delegate computations: the power of no-signaling proofs. In *STOC 2014, Proceedings*, pages 485–494. ACM, 2014.

LM18. Huijia Lin and Christian Matt. Pseudo flawed-smudging generators and their application to indistinguishability obfuscation. *IACR Cryptology ePrint Archive*, 2018:646, 2018.

LO13. Steve Lu and Rafail Ostrovsky. How to garble RAM programs. In *EUROCRYPT 2013, Proceedings*, pages 719–734, 2013.

LO17. Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In *CRYPTO 2017, Proceedings, Part II*, pages 66–92, 2017.

Mia16. Peihan Miao. Cut-and-choose for garbled RAM. *IACR Cryptology ePrint Archive*, 2016:907, 2016.

MZ18. Fermi Ma and Mark Zhandry. The MMap strikes back: Obfuscation and new multilinear maps immune to CLT13 zeroizing attacks. In *TCC 2018, Proceedings, Part II*, pages 513–543, 2018.

OS97. Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC 1997, Proceedings*, pages 294–303, 1997.

Ost90. Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC 1990, Proceedings*, pages 514–523, 1990.

PF79. Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

RAD78. Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation, Academia Press*, 1978.