Fully Homomorphic Encryption for RAMs

Ariel Hamlin* Justin Holmgren† Mor Weiss‡ Daniel Wichs§

June 17, 2019

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.

*Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA. ahamlin@ccs.neu.edu

†Department of Computer Science, Princeton University, Princeton, New Jersey, USA. justin.holmgren@princeton.edu

‡Department of Computer Science, IDC Herzliya, Herzliya, Israel. mor.weiss01@post.idc.ac.il

§Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA. wichs@ccs.neu.edu

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.

Contents

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+13b], program obfuscation [GGH+13], verifiable computation [GGP10, KRR14], cryptographic hash functions [CCH+19], and more.

The most immediate use-case of FHE is privately outsourcing 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, but with a major caveat: 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, GSW13, 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, which we now informally describe. 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*, meaning that the output ciphertext $\widehat{y}$, as well as 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, each of 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, Agr18, 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. In particular, we do not assume the existence of any type of obfuscator. We obtain different tradeoffs between efficiency and the permissibles type 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 $\mathsf{poly}(\lambda, \log N)$.
- 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

[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].

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 throughout this process 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 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 conceptually 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

[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.

ORAM O, which is initially empty. (We provide an explicit construction of an ORAM scheme for initially empty databases in Appendix A.) 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 Section 3.2.

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 faces several challenges, and requires some adaptations, as we now describe. The final construction is obtained through the following steps.

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

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 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*, and in particular 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 all the internal client 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

[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.

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 Section 6.2.)

Generalizing to programs of any length. The construction described above assumed the entire program description was given as input to the obfuscated circuit (in particular, 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 Section 7.3 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 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 [GLOS15, CHJV15, CH16, CCHR16, ACC+16, BCP16, CCC+16, 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, GHRW14, 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).

[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.

The notion of FHE for Turing machines was considered in [GKP+13a], 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.

2.1 Doubly-Efficient Private Information Retreival (DEPIR)

Definition 2.1 (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.
- $\mathsf{Process}(\mathsf{sk}, \mathsf{DB})$ takes as input a client secret-key sk and a database $\mathsf{DB} \in \{0,1\}^N$, and outputs a processed database $\widetilde{\mathsf{DB}} \in \{0,1\}^{\tilde{N}}$.
- $\mathsf{Query}(\mathsf{sk}, \mathsf{addr})$ 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:

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

- **Security:** Any non-uniform PPT adversary $\mathcal{A}$ has only $\mathsf{negl}(\lambda)$ 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 $\widetilde{\mathsf{DB}} \leftarrow \mathsf{Process}(\mathsf{sk}, \mathsf{DB})$ 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 $\mathsf{poly}(\lambda)$, the runtime of $\mathsf{Process}$ is $\mathsf{poly}(N, \lambda)$, and the runtime of $\mathsf{Query}, \mathsf{Decode}$ is $o(N) \cdot \mathsf{poly}(\lambda)$, 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 2.2 (Security with oblivious preprocessing)**.** We say that a SK-DEPIR scheme is *secure with oblivious preprocessing* if the security property of Definition 2.1 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 the 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.

2.2 Virtual Black-Box (VBB) obfuscation

We use the notion of virtual black-box obfuscation with auxiliary input [BGI+01, GK05].

Definition 2.3 (Virtual black-box obfuscation with auxiliary input)**.** Let $\mathcal{C} = \{\mathcal{C}_n\}$ be a circuit class, where circuits in $\mathcal{C}$ take inputs in $\{0,1\}^n$ and have size $\mathsf{poly}(n)$. A *(non-uniform) Virtual Black-Box obfuscator (VBB obfuscator) with auxiliary input for* $\mathcal{C}$ is a PPT algorithm $\mathcal{O}$ that takes as input a security parameter λ, an input size $n \in \mathbb{N}$, and a boolean circuit $C \in \mathcal{C}_n$, outputs a circuit $\widetilde{C}$, and satisfies the following:

- **Functionality:** for every $n \in \mathbb{N}$, every $C \in \mathcal{C}_n$, and every input $x \in \{0,1\}^n$,

$$\Pr\left[\mathcal{O}\left(1^\lambda, 1^n, C\right)(x) = C(x)\right] \geq 1 - \mathsf{negl}(\lambda)$$

  where the probability is over the coins of $\mathcal{O}$.

- **Polynomial Slowdown:** there exist a polynomial p such that for every $n \in \mathbb{N}$ and every $C \in \mathcal{C}_n$, $\left|\mathcal{O}\left(1^\lambda, 1^n, C\right)\right| \leq p(|C|, n, \lambda)$.

- **Virtual Black-Box:** for every PPT adversary $\mathcal{A}$ there exists a PPT simulator Sim such that for every $n \in \mathbb{N}$, every $C \in \mathcal{C}_n$, and every auxiliary input $z \in \{0,1\}^*$ of polynomial length:

$$\left| \Pr\left[\mathcal{A}\left(\mathcal{O}\left(1^\lambda, 1^n, C\right), z\right) = 1\right] - \Pr\left[\mathsf{Sim}^C\left(1^\lambda, 1^n, 1^{|C|}, z\right) = 1\right] \right| \leq \mathsf{negl}\left(\lambda\right)$$

  where the probabilities are over the coins of $\mathcal{O}, \mathcal{A}$ and Sim.

2.3 Oblivious RAM

In this section we formally describe the standard ORAM notion. Recall that we refer to the database (i.e., the client's data) as the "logical memory", and the server's state (which the server has RAM access to) as the "physical memory". We use the terms "database" and "logical memory" interchangeably to refer to the client's data, and "physical memory" and "server state" to refer to the remote memory stored on the server. We assume the logical memory consists of bits, and the physical memory consists of blocks of size $\mathsf{polylog}\left(\lambda\right)$. This is without loss of generality up to a multiplicative $\mathsf{polylog}\left(\lambda\right)$ increase in complexity, since we can read/write the blocks one bit at a time.

We use the standard ORAM definition, but elaborate on the structure of the Access protocol. More specifically, accessing the database in a standard ORAM requires several interaction rounds, during which the client reads and writes some blocks to the physical memory. (Updating the physical memory is needed to guarantee obliviousness, even if the database is read-only.)

Definition 2.4 (Oblivious RAM (ORAM) [Gol87, Ost90, GO96])**.** An $\mathsf{Oblivious\ RAM\ (ORAM)}$ scheme consists of procedures $(\mathsf{Setup}, \mathsf{Access})$, with the following syntax:

- $\mathsf{Setup}(1^\lambda, \mathsf{DB})$ takes as input a security parameter λ and a database $\mathsf{DB} \in \{0,1\}^N$ and outputs initial client and servers states ck, st.

- $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{val})$ is an interactive protocol executed by a client to interact with a server. The client C has state ck, and his input is an operation $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, an address $\mathsf{addr} \in [N]$, and a value val (if $\mathsf{op} = \texttt{read}$, then val is ignored). The client ultimately outputs a value val' (if $\mathsf{op} = \texttt{write}$, then $\mathsf{val}' = \perp$).

  Throughout the execution, the server S is used only as remote storage, and does not perform any computations. That is, in each round of the protocol, the client reads some physical address $\mathsf{p\_addr}_i$, and performs an update operation which replaces the block at some physical location $\mathsf{p\_addr}'_i$ with block_i. We use st' to denote the updated server state at the end of the execution.

We require that the scheme satisfies the following properties:

- **Correctness.** Consider the following interaction between a stateful client and server. The client and server initially receive ck and st (respectively), sampled as $(\mathsf{ck}, \mathsf{st}) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{DB})$. They then repeatedly execute the Access protocol, where the client's input is given by a sequence of $\texttt{read}$ and $\texttt{write}$ instructions $(\mathsf{instr}_1, \ldots, \mathsf{instr}_q)$. Then the output of the client is, with probability 1, identical to his output when these instructions are sequentially performed directly on a database whose initial contents are DB.

- **Security.** Every PPT adversary $\mathcal{A}$ wins the following security game with a challenger $\mathcal{C}$ with probability at most $1/2 + \mathsf{negl}\left(\lambda\right)$:

  1. $\mathcal{A}$ sends to $\mathcal{C}$ a database $\mathsf{DB} \in \{0,1\}^N$.
  2. $\mathcal{C}$ runs $\mathsf{Setup}\left(1^\lambda, \mathsf{DB}\right)$ to obtain client and server states ck, st. $\mathcal{C}$ sends st to $\mathcal{A}$.

3. $\mathcal{C}$ picks a random bit $b \leftarrow \{0,1\}$.
4. Repeat the following $\mathsf{poly}(\lambda)$ times:
   (a) $\mathcal{A}$ sends to $\mathcal{C}$ two instructions: $(\mathsf{op}, \mathsf{addr}^0, \mathsf{val}^0)$, and $(\mathsf{op}, \mathsf{addr}^1, \mathsf{val}^1)$, where $\mathsf{op} \in \{\texttt{read}, \texttt{write}\}$, and $\mathsf{addr}^0, \mathsf{addr}^1 \in [N]$.
   (b) $\mathcal{C}$ executes $\mathsf{Access}(\mathsf{op}, \mathsf{addr}^b, \mathsf{val}^b)$, and sends to $\mathcal{A}$ the access pattern to physical memory during this execution.
5. $\mathcal{A}$ outputs a bit b', and is said to win the game if $b = b'$.

Remark on hiding the type of operation. Definition 2.4 does not hide whether the performed operation is a `read` or a `write`, whereas an ORAM scheme is usually defined to hide this information. However, any such scheme can be generically transformed into one that hides the identity of operations by always performing both a read and a write. (Specifically, in a write operation, one first performs a dummy read; in a read operation, one writes back the value that was read.) Revealing the identity of operations allows for more fine-grained overheads.

Remark on physical memory block contents. We assume that blocks of physical memory are sufficiently large to store a pair (addr, b), where addr is a logical memory address, and b is a bit. We will sometimes additionally designate each block as "valid" or not, and include a corresponding tag valid. This will be useful when instantiating an ORAM with an database of "empty" blocks in the ISR-ORAM of Construction 1, and the multi-hop RAM-FHE scheme of Construction 7. More specifically, we define a block as "empty" if it has the tag $\mathsf{valid} = \mathsf{false}$, whereas all non-empty blocks will always have the tag $\mathsf{valid} = \mathsf{true}$. A size-$N$ database of empty blocks consists of blocks corresponding to logical memory addresses $1, \ldots, N$, with $\mathsf{valid} = \mathsf{false}$ tag. In an execution of an Access protocol, blocks with $\mathsf{valid} = \mathsf{false}$ tag are "ignored" in the sense that they are never returned as output to the client, and during update operations are treated as dummy blocks (i.e., they could be arbitrarily overwritten).

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 3.1 (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 $\mathsf{op}_l \in \{\texttt{read}, \texttt{write}\}$ at address addr_l with value val_l (which, if $\mathsf{op}_l = \texttt{read}$, 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.

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 schemes) two sequences

of access patterns, with the goal of distinguishing between the executions of these sequences on the two databases. In addition (and unlike standard ORAM), the adversarial server in our security game can rewind the execution to a previous state, and continue the execution from that state.

Definition 3.2 (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 $q_i \leq \mathsf{poly}(\lambda)$, $\mathsf{op}_{i,l} \in \{\mathtt{read}, \mathtt{write}\}$, $\mathsf{addr}_{i,l}^0, \mathsf{addr}_{i,l}^1 \in [N]$, 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 3.2 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 (Definition 2.4). 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 3.2 can be used to capture various other security variants; we choose to focus on the latter two notions. Formally,

Definition 3.3 (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 3.2.

Definition 3.4 (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 3.2, 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 3.2.

3.2 Rewindable ORAM Constructions

In this section we construct ISR- and ASR-ORAM schemes from SK-DEPIR and standard ORAM schemes. We first construct (Section 3.2.1) an ISR-ORAM scheme with polylogarithmic overhead. We then describe (Section 3.2.2) an ASR-ORAM scheme with sublinear overhead. Our ISR-ORAM scheme, despite having a weaker security guarantee than ASR-ORAM, has the advantage of being simpler and more efficient.

3.2.1 ISR-ORAM from ORAM and SK-DEPIR

In this section, we construct an ISR-ORAM scheme from a SK-DEPIR scheme along with an ORAM scheme for initially-empty databases (Definition A.1), proving the following:

Theorem 3.5 (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* $\mathsf{poly}(\lambda)$ *(*$\mathsf{poly}(\lambda, N)$*).*

The construction. The high-level idea of the construction is to initialize a SK-DEPIR scheme with the database, and initialize the ORAM (in the first operation) with size bound N. Then, `write` operations write to the ORAM, and `read` operations read from both the SK-DEPIR and the ORAM, taking the copy from the ORAM if it exists. Intuitively, the scheme is secure against initial-state rewinding because a SK-DEPIR scheme is secure against arbitrary rewinding attacks (since the SK-DEPIR server is stateless, and the SK-DEPIR client maintains only a long-term key between accesses), and the initial server state in effect contains only the processed database of the SK-DEPIR (since the ORAM is initialized only in the first operation).

Construction 1 (ISR-ORAM from SK-DEPIR and ORAM for initially-empty databases)**.** The scheme uses the following building blocks:

- An ORAM scheme for initially-empty databases $(\mathsf{ORAM.Setup}, \mathsf{ORAM.Access})$ with a deterministic client during $\mathsf{ORAM.Access}$.
- A SK-DEPIR scheme $(\mathsf{DEPIR.KeyGen}, \mathsf{DEPIR.Process}, \mathsf{DEPIR.Query}, \mathsf{DEPIR.Decode})$.

The scheme consists of the following procedures:

$\mathsf{Setup}(1^\lambda, \mathsf{DB})$**:** Recall that λ denotes a security parameter, and $\mathsf{DB} \in \{0,1\}^N$.

- Initializing the SK-DEPIR component: run $\mathsf{DEPIR.KeyGen}\left(1^\lambda\right)$ to obtain a SK-DEPIR secret-key sk. Run $\mathsf{DEPIR.Process}\left(\mathsf{sk}, \mathsf{DB}\right)$ to obtain the processed SK-DEPIR database $\widetilde{\mathsf{DB}}$.
- Initializing the ORAM component: initialize a boolean variable $\mathsf{Initialized}$ to false, and the client and server states $\mathsf{ck}^O, \mathsf{st}^O$ to $\perp$ (i.e., empty states).
- Output: the server state $\mathsf{st} = \left(\widetilde{\mathsf{DB}}, \mathsf{st}^O, \mathsf{Initialized}\right)$ contains the processed database $\widetilde{\mathsf{DB}}$ of the SK-DEPIR scheme, the (empty) ORAM server state st^O, and a boolean variable $\mathsf{Initialized}$ indicating whether the ORAM has already been initialized. The client state $\mathsf{ck} = \left(\mathsf{sk}, \mathsf{ck}^O, 1^\lambda\right)$ consists of the secret key sk of the SK-DEPIR, an (empty) client state ck^O for the ORAM, and the security parameter.

The Access protocol. To perform the operation op at location $\mathsf{addr} \in [N]$ of the database with value val, the client C with state $\mathsf{ck} = \left(\mathsf{sk}, \mathsf{ck}^O, 1^\lambda\right)$, and the server with state $\mathsf{st} = \left(\widetilde{\mathsf{DB}}, \mathsf{st}^O, \mathsf{Initialized}\right)$ operate as follows.

- Read $\mathsf{Initialized}$ from the server. If $\mathsf{Initialized} = \mathsf{false}$ then run $\mathsf{ORAM.Setup}$ between the client and server, where the client has $\left(1^\lambda, N\right)$ as input. Let $\mathsf{ck}^{O\prime}$ be the updated client state, and $\mathsf{st}^{O\prime}$ be the updated server state at the end of the execution. The client and server locally replace $\mathsf{ck}^O, \mathsf{st}^O$ with $\mathsf{ck}^{O\prime}, \mathsf{st}^{O\prime}$ (respectively).
- Set $\mathsf{Initialized} = \mathsf{true}$.
- **If** $\mathsf{op} = \mathtt{read}$**:**
  - Reading from the ORAM: the client and server run $\mathsf{ORAM.Access}$, where the client emulates the ORAM-client with key ck^O and input $(\mathtt{read}, \mathsf{addr}, \mathsf{val})$, and the server emulates the ORAM-server with state st^O. Let val^O be the output to the client, and $\mathsf{ck}^{O\prime}, \mathsf{st}^{O\prime}$ denote the updated client and server states (respectively).
  - Reading from the SK-DEPIR:
    - C runs $Q \leftarrow \mathsf{DEPIR.Query}\left(\mathsf{sk}, \mathsf{addr}\right)$ to obtain the queries Q to the server, and a short-term client state st_C, and sends Q to S.
    - Given the answers $\left\{\widetilde{\mathsf{DB}}_i \ : \ i \in Q\right\}$ from S, C computes $\mathsf{val}^P = \mathsf{DEPIR.Decode}\left(\mathsf{sk}, \mathsf{st}_C, \left\{\widetilde{\mathsf{DB}}_i \ : \ i \in Q\right\}\right)$.
  - Output. If $\mathsf{val}^O \neq \perp$ then $\mathsf{val}' = \mathsf{val}^O$. ($\mathsf{val}^O \neq \perp$ if block addr was found in the ORAM.) Otherwise, set $\mathsf{val}' = \mathsf{val}^P$. Output val' to the client, together with the updated client state $\left(\mathsf{sk}, \mathsf{ck}^{O\prime}, 1^\lambda\right)$. The updated server state is $\left(\mathsf{st}^{O\prime}, \widetilde{\mathsf{DB}}, \mathsf{initialize}\right)$.
- **If** $\mathsf{op} = \mathtt{write}$**:**
  - Writing to the ORAM: the client and server run $\mathsf{ORAM.Access}$, where the client emulates the ORAM-client with key ck^O and input $(\mathtt{write}, \mathsf{addr}, \mathsf{val})$, and the server emulates the ORAM-server with state st^O. Let val' be the output to the client, and $\mathsf{ck}^{O\prime}, \mathsf{st}^{O\prime}$ denote the updated client and server states (respectively).
  - Output. Output val' to the client, together with the updated client state $\left(\mathsf{sk}, \mathsf{ck}^{O\prime}, 1^\lambda\right)$. The updated server state is $\left(\mathsf{st}^{O\prime}, \widetilde{\mathsf{DB}}, \mathsf{initialize}\right)$.

We prove the following claims about Construction 1.

Claim 3.6 (ISR-ORAM security)**.** *Assuming the security of all of the building blocks, Construction 1 is a secure ISR-ORAM scheme.*

Claim 3.7 (ISR-ORAM complexity)**.** *Assume that:*

- *For size-N databases and security parameter λ,* $\mathsf{DEPIR.Query}, \mathsf{DEPIR.Decode}$ *performs $t_Q\left(N, \lambda\right)$ and $t_D\left(N, \lambda\right)$ operations (respectively), and the client and server states have size $n_C\left(N, \lambda\right)$, $n_S\left(N, \lambda\right)$, respectively.*
- *When initialized with size parameter t and security parameter λ,* $\mathsf{ORAM.Setup}$ *performs $t_S\left(t\right)$ operations,* $\mathsf{ORAM.Access}$ *performs $t_O\left(t\right)$ operations, and the client and server states have size $n'_C\left(t, \lambda\right)$, $n'_S\left(t, \lambda\right)$, respectively.*

Then the execution of the Access *protocol of Construction 1 performs* $t_O(N)+t_Q(N)+t_D(N)+2$ *operations if* $\mathsf{op} = \texttt{read}$*, and* $t_O(N)+2$ *if* $\mathsf{op} = \texttt{write}$*. The first execution of the* Access *protocol of Construction 1 performs* $t_S(N)$ additional *operations. Moreover, the client and server states have size* $n_C(N,\lambda)+n'_C(N,\lambda)$*,* $n_S(N,\lambda)+n'_S(N,\lambda)$*, respectively.*

Claims imply Theorem. We now use the claims to prove Theorem 3.5.

Proof of Theorem 3.5. We prove that Construction 1 has the required properties. The existence of OWFs implies the existence of a secure ORAM scheme for initially-empty databases by Claim A.2 so security of Construction 1 follows from Claim 3.6. Moreover, by Claim A.2 when the ORAM scheme is instantiated with size bound N then Setup and Access have complexity $\mathsf{poly}(\lambda)$, and the client and server state have size $\mathsf{poly}(\lambda)$, $\mathsf{poly}(\lambda, N)$, respectively. The stated complexity now follows from the complexity of the ORAM scheme, the complexity of the SK-DEPIR scheme (as stated in the theorem statement), and Claim 3.7.

To get a deterministic client, we apply the transformation of Appendix B, and use ClaimsB.3 and B.1. □

Proofs of Claims. We now prove Claims 3.6 and 3.7.

Proof of Claim 3.6. The correctness of the scheme follows directly from the correctness of the underling building blocks. Indeed, ORAM correctness is preserved when it is initialized with $\mathsf{OblSetup}$; and when a block appears in both the ORAM and the SK-DEPIR components, then the freshest copy is in the ORAM, which is what the ISR-ORAM outputs to the client. We now argue security, via a sequence of hybrids.

$\mathcal{H}_{\mathbf{0}}^{\mathbf{b}}$: Hybrid $\mathcal{H}_0^b$ is the view $(\mathsf{st}_0, (\mathcal{ACC}_i)_i)$ of the adversary $\mathcal{A}$ in the ORAM security game when the challenger chooses bit b.

$\mathcal{H}_{\mathbf{1}}^{\mathbf{b}}$: In $\mathcal{H}_1^b$, we replace DB^b with the all-0 database, and replace every execution of Query with a dummy execution that runs Query on index 1. The accesses to the ORAM component remain unchanged (and are, in both hybrids, according to the b'th access pattern). *Hybrids* $\mathcal{H}_0^b$ *and* $\mathcal{H}_1^b$ *are computationally indistinguishable by the security of the SK-DEPIR scheme.*

$\mathcal{H}_{\mathbf{2}}^{\mathbf{b}}$: In hybrid $\mathcal{H}_2^b$, we replace every execution of $\mathsf{ORAM.Access}$ of the underlying ORAM with a dummy operation that reads or writes to the first database location, according to the operation performed in $\mathcal{H}_1^b$. (That is, a `read` in $\mathcal{H}_1^b$ is replaced with a read of location 1, a `write` in $\mathcal{H}_1^b$ is replaced with a write of an arbitrary value to location 1.)

Hybrids $\mathcal{H}_1^b$ and $\mathcal{H}_2^b$ are computationally indistinguishable by the security of the underlying ORAM scheme for initially-empty databases, as we now show. We use the observation that any secure ORAM scheme for initially-empty databases is secure under the following notion of sequential composition. For any $q = \mathsf{poly}(\lambda)$, any $n_1, \ldots, n_q < 2^\lambda$, any $q_1, \ldots, q_q = \mathsf{poly}(\lambda)$, and any two lists $\mathcal{Q}^0 = \left\{Q_j^0\right\}_{j\in[q]}, \mathcal{Q}^1 = \left\{Q_j^1\right\}_{j\in[q]}$ of sequences $Q_j^0 = \left(\mathsf{op}_{j,l}, \mathsf{val}_{j,l}^0, \mathsf{addr}_{j,l}^0\right)_{1\le l\le q_j}, Q_j^1 = \left(\mathsf{op}_{j,l}, \mathsf{val}_{j,l}^1, \mathsf{addr}_{j,l}^1\right)_{1\le l\le q_j}$ of access patterns, $\left(\mathcal{ACC}_\emptyset\left(n_j, Q_j^0\right)\right)_{j\in[q]} \approx \left(\mathcal{ACC}_\emptyset\left(n_j, Q^1\right)\right)_{j\in[q]}$ ($\mathcal{ACC}_\emptyset(n, Q)$ was defined in Definition A.1), where $\approx$ denotes computational indistinguishability. This follows from the security property of ORAM schemes for initially-empty databases, using a standard hybrid argument.

Indistinguishability of $\mathcal{H}_1^b$ and $\mathcal{H}_2^b$ follows directly from the security of the underlying ORAM scheme for initially-empty databases under sequential composition, because any rewind to the initial

state resets Initialized to false, and so the next access to the database initializes the underlying ORAM by running Setup with fresh randomness.

We conclude the proof by noting that $\mathcal{H}_2^0 \equiv \mathcal{H}_2^1$ since neither depend on $\mathsf{DB}^0, \mathsf{DB}^1$ or the access patterns. □

Proof of Claim 3.7. Any execution of Access performs two operations to read and update Initialized on the server, and $t_O(N)$ operations to access the underlying ORAM. `read` operations additionally perform $t_Q(N)$ operations to run DEPIR.Query, and $t_D(N)$ operations to run DEPIR.Decode. The first execution of Access additionally performs $t_S(N)$ operations to initialize the underlying ORAM with an empty database. The client and server storage are simply the combination of the storage required by the underlying ORAM and SK-DEPIR. □

3.2.2 ASR-ORAM from SK-DEPIR and OWFs

We now construct an ASR-ORAM scheme from SK-DEPIR and PRFs, proving the following.

Theorem 3.8 (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 2 (ASR-ORAM from SK-DEPIR and PRFs)**.** The scheme uses the following building blocks:

- A PRF F.
- A SK-DEPIR scheme (DEPIR.KeyGen, Process, Query, Decode) with oblivious preprocessing (Definition 2.2).
- A CPA-secure symmetric encryption scheme (SE.KeyGen, Encrypt, 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 $1 \leq i \leq \ell$, 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 20) 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 $\mathsf{op} = \texttt{read}$:**
  - 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 $\mathsf{op} = \texttt{write}$:

- 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 $\mathsf{ReShuffle}$ procedure of Figure 3, namely executes $\mathsf{ReShuffle}\left(i, \mathsf{L}^i, \mathsf{L}^{i+1}\right)$.[6]

[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.

We prove Theorem 3.8. We first prove the following claims about Construction 2.

Claim 3.9 (ASR-ORAM security)**.** *Assuming the security of all of the building blocks, Construction 2 is a secure ASR-ORAM scheme.*

Claim 3.10 (ASR-ORAM complexity)**.** *Assume that on size-N databases,* DEPIR.Process, DEPIR.Query *and* DEPIR.Decode *perform $t_S(N), t_Q(N)$ and $t_D(N)$ operations (respectively). Then each execution of the* Access *protocol of Construction 2 with* $\mathsf{op} = \texttt{read}$ *performs*

$$\mathsf{B}\left(t_Q(N) + t_D(N) + \mathsf{poly}(\lambda)\right) \cdot \log N$$

operations, and the execution of Access *with* $\mathsf{op} = \texttt{write}$ *performs*

$$\mathsf{poly}(\lambda) \cdot \mathsf{B} \cdot \log N + \sum_{i=1}^{\log N} \left(\mathsf{poly}(\lambda) \cdot i + \frac{t_S\left(\mathsf{B} \cdot 2^{i+1}\right)}{2^i} \right)$$

operations, where B *denotes the bucket size.*

Claims imply Theorem. We now use the claims to prove Theorem 3.8.

Proof of Theorem 3.8. The existence of OWFs implies the existence of PRFs and encryption schemes as used in Construction 2, so security follows from Claim 3.9 (when $\mathsf{B} = \lambda$). As for the complexity of the scheme, by the theorem assumptions (on the existence of a SK-DEPIR scheme with $t_Q(N) = t_D(N) = N^\epsilon \cdot \mathsf{poly}(\lambda)$, and $t_S = N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$), and Claim 3.10, we get the following. Each `read` operation performs $N^\epsilon \cdot \log N \cdot \mathsf{poly}(\lambda) = N^\epsilon \cdot \mathsf{poly}(\lambda)$ operations (here, we also use the fact that $\mathsf{B} = \lambda$ and $N \leq 2^\lambda$). Moreover, each `write` operation performs

$$\mathsf{poly}(\lambda) \cdot \log N + \sum_{i=1}^{\log N} \left(\mathsf{poly}(\lambda) \cdot i + \frac{\left(\lambda \cdot 2^{i+1}\right)^{1+\epsilon} \cdot \mathsf{poly}(\lambda)}{2^i} \right)$$

$$= \mathsf{poly}(\lambda) \cdot \log^2 N + \mathsf{poly}(\lambda) \cdot \sum_{i=1}^{\log N} \left(2^i\right)^\epsilon$$

$$\leq N^\epsilon \cdot \log N \cdot \mathsf{poly}(\lambda) = N^\epsilon \cdot \mathsf{poly}(\lambda)$$

The server state consists of $\log N$ levels, where level i contains a SK-DEPIR of size $\left(\lambda \cdot 2^i\right)^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$ (and additionally $O(1)$ ciphertexts of size $\mathsf{poly}(\lambda)$), so the server state has size $N^\epsilon \cdot \log N \cdot \mathsf{poly}(\lambda) = N^\epsilon \cdot \mathsf{poly}(\lambda)$. The client only stores the SK-DEPIR secret key, of size $\mathsf{poly}(\lambda)$.

To get a deterministic client, we apply the transformation of Appendix B, and use Claims B.2 and B.3.

□

Analysis of Bucket Overflows. The analysis of Construction 2 will rely on the following lemma which states that with overwhelming probability no bucket overflows.

Lemma 3.11 (Probability of bucket overflows)**.** *Assuming the pseudorandomness of F, with overwhelming probability no bucket overflows during the execution of the ASR-ORAM of Construction 2.*

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 Section 2.3 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 Lemma 3.11 below, this happens with negligible probability.

Figure 1: The InitLevel procedure used in Construction 2

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 $(l-1)\cdot \mathsf{B}+1 \leq m \leq l \cdot \mathsf{B}$:
  - 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.

Figure 2: The ReadBucket procedure used in Construction 2

Proof. For simplicity of the analysis, we model the PRF as a random function. In particular, this means there is no difference between the mapping to buckets of real and "empty" blocks in Step 4 of the InitLevel procedure of Figure 1. Indeed, if InitLevel is called with a database DB with k real blocks, then exactly $2^i - k$ blocks are "empty". In particular, there is a way to assigning addresses in $\left[2^i\right]$ to each of the empty blocks, such that each (real or empty) block has a unique address. For unique addresses (i.e., inputs to F), choosing a random output is identical to applying a random function. Therefore, in the following we assume DB contains exactly 2^i real blocks.

Whenever InitLevel is called for some level $i \in [\ell]$, the corresponding database contains 2^i data blocks, which are then allocated to the 2^i size-B buckets of the level. Therefore, in expectation every specific bucket B will contain a single data block, and so by Chernoff's bound the probability that bucket B of size-λ overflows is at most

$$\frac{e^{\lambda-1}}{\lambda^\lambda} \leq 2^{-\lambda}$$

where the inequality holds for a large enough $\lambda \geq 2e$. Since the total number of buckets in all levels is at most $N \cdot \log N = \mathsf{poly}\left(\lambda\right)$, and each bucket is initialized at most $\mathsf{poly}\left(\lambda\right)$ times, the probability that *any* bucket overflows is negligible. □

Proofs of Claims. We now prove Claims 3.9 and 3.10.

Proof of Claim 3.9. We show that the correctness of the scheme follows from the correctness of the SK-DEPIR, and the pseudorandomness of F. First, the pseudorandomness of F guarantees that with overwhelming probability no bucket overflows (see Lemma 3.11). Conditioned on this event, the construction preserves the invariant that each level contains at most a single copy of each block, and fresher copies appear in levels with smaller indices. This is guaranteed because the ReShuffle procedure removes duplicate block copies, keeping the copy from the level with smaller index (which contains the fresher copy). Finally, since `read` operations return the copy from the smallest level

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 remark on physical memory block contents in Section 2.3 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 $\mathsf{L}_{i+1} = \left(\mathsf{DB}^{i+1\prime}, \widetilde{\mathsf{DB}}^{i+1\prime}\right)$.

Figure 3: The ReShuffle protocol used in Construction 2

that contains the block (namely, the freshest copy of the block), then correctness follows from the correctness of the underlying SK-DEPIR scheme.

We now argue security, via a sequence of hybrids. We condition all hybrids on the event that no bucket overflows. This is without loss of generality since (by Lemma 3.11) this event happens with overwhelming probability.

$\mathcal{H}_{\mathbf{0}}^{\mathbf{b}''}$: Hybrid $\mathcal{H}_0^{b''}$ is the view $(\mathsf{st}_0, (\mathcal{ACC}_i)_i)$ of the adversary $\mathcal{A}$ in the ORAM security game when the challenger chooses bit b.

$\mathcal{H}_{\mathbf{0}}^{\mathbf{b}'}$: In $\mathcal{H}_0^{b'}$, we replace DB^b with the all-0 database. Additionally, we replace the encryptions $\widetilde{K}^i, \widetilde{\mathsf{sk}}^i$ of the PRF and SK-DEPIR keys of all levels, throughout the execution of the scheme, with encryptions of the all-zero string. Moreover, we replace the ciphertexts c generated during `write` operations with encryptions of zeros. All accesses into the database are still performed correctly, by using the actual database contents and keys. (For example, in Step 4 of the InitLevel procedure, the bucket index is computed based on the actual block address and actual PRF key, even though in the encrypted database this value was replaced by 0.) *Hybrids $\mathcal{H}_0^{b''}$ and $\mathcal{H}_0^{b'}$ are computationally indistinguishable by the CPA-security of the encryption scheme.*

$\mathcal{H}_{\mathbf{0}}^{\mathbf{b}}$: In $\mathcal{H}_0^b$, we replace the PRF with a truly random function (using a different random function for every key K_i). *Hybrids $\mathcal{H}_0^{b'}$ and $\mathcal{H}_0^b$ are computationally indistinguishable by the pseudorandomness of F (using a standard hybrid argument over the different initializations of PRF keys in* InitLevel*).*

$\mathcal{H}_{\mathbf{j}}^{\mathbf{b}}, \mathbf{1} \leq \mathbf{j} \leq \ell$: In $\mathcal{H}_j^b$, we replace all calls to Query (in the execution of the ReadBucket procedure) in levels $1, \ldots, j$ with reads of address 1 (i.e., the first block in the bucket list L that was used to generate the SK-DEPIR for the level) from the SK-DEPIR of the level.

We now show that the for every $0 \leq j < \ell$, hybrids $\mathcal{H}_j^b$ and $\mathcal{H}_{j+1}^b$ are computationally indistinguishable, by the security of the SK-DEPIR scheme. The only difference between $\mathcal{H}_j^b, \mathcal{H}_{j+1}^b$ is in the accesses to the SK-DEPIR scheme of level j. However, the hybrids differ from the distributions in the SK-DEPIR security proof in two points: (1) they also include the access pattern to the physical memory during the execution of Process; and (2) they consist of a sequential composition of several *independent* instantiations of the SK-DEPIR (the instantiations are independent because a fresh SK-DEPIR key is chosen for the level whenever it is instantiated in InitLevel). Notice that in every single SK-DEPIR instance, (1) together with the access pattern during Query is exactly the adversary's view in the security with oblivious preprocessing security game. Therefore, $\mathcal{H}_j^b \approx \mathcal{H}_{j+1}^b$ by a standard hybrid argument over independent instantiations of the SK-DEPIR.

We now define a final hybrid $\mathcal{H}_{\ell+1}^b$ as follows. In $\mathcal{H}_{\ell+1}^b$, we modify InitLevel such that on input i, in Step 4 it picks randomly with repetition a list of 2^i buckets, downloads each of them (in order), re-encrypts all the blocks in the bucket, and uploads the bucket back to the server. Then hybrids $\mathcal{H}_\ell^b$ and $\mathcal{H}_{\ell+1}^b$ are identically distributed. Indeed, in $\mathcal{H}_\ell^b$ the random function is only used in InitLevel (since ReadBucket calls do not use the random function any more), so the only difference between the hybrids is that in $\mathcal{H}_\ell^b$ repeated copies of a real block would always be mapped to the same bucket, whereas in $\mathcal{H}_{\ell+1}^b$ the choice of buckets is completely random. However, Construction 2 preserves the invariant that the real blocks in every level are distinct. Since the random function is only applied to real blocks, it is never called twice with the same input. This holds also under rewinds, because each call to InitLevel is executed with a fresh random function (since each execution uses a fresh PRF key, and each such instantiation of the PRF was replaced in $\mathcal{H}_0^b$ by an independent random function).

Notice that in $\mathcal{H}_{\ell+1}^b$, the access pattern to the physical memory is completely independent of the `read` or `write` operations performed throughout the execution. Indeed, this holds for `read`

operations because the accesses there are induced by running B executions of $\mathsf{Query}\left(\mathsf{sk}^i, 1\right)$ on each level i, regardless of the read location. For `write` operations, all steps performed during reshuffle are oblivious of the database contents (either linear scans or oblivious sorting), and the execution of $\mathsf{InitLevel}$ is also oblivious of database contents (because we have replaced the mapping into buckets with a random list of buckets, and because the SK-DEPIR preprocessing is performed on ciphertexts).

We conclude the proof by noting that $\mathcal{H}^0_{\ell+1} \equiv \mathcal{H}^1_{\ell+1}$ since neither depend on $\mathsf{DB}^0, \mathsf{DB}^1$ or the access patterns. □

Proof of Claim 3.10. We first analyze the complexity of Access where $\mathsf{op} = \texttt{read}$. During a read, the client does the following for every non-empty level (there are at most $\log n$ such levels). First, he reads the level-specific PRF and SK-DEPIR keys from the server, decrypts them, and computes the bucket index, which requires $\mathsf{poly}(\lambda)$ operations. Then, for each of the B blocks in the bucket, the client performs t_Q and t_D operations to run $\mathsf{DEPIR.Query}$ and $\mathsf{DEPIR.Decode}$ (respectively). Therefore, Access performs $\mathsf{B}\left(t_Q(N) + t_D(N) + \mathsf{poly}(\lambda)\right) \cdot \log N$ operations.

When $\mathsf{op} = \texttt{write}$, each operation encrypts a single block ($\mathsf{poly}(\lambda)$ operations),, updates the counter ($\mathsf{poly}(\lambda)$ operations) and performs its "share" of the $\mathsf{ReShuffle}$ procedure for each level (i.e., a 2^{-i}-fraction of the operations of the reshuffle of level i, for every $i = 1, \ldots, \log N$). Each execution of the $\mathsf{ReShuffle}$ procedure of level i performs the following number of operation:

- $2^i \cdot \mathsf{poly}(\lambda)$ operations to initialize $\mathsf{DB}^i, \mathsf{DB}^{i+1}$ (if needed), and for a constant number of linear scans of $\mathsf{DB}^i, \mathsf{DB}^{i+1}$, and A.
- $i \cdot 2^i \cdot \mathsf{poly}(\lambda)$ operations for a constant number of oblivious sorts of A (using an oblivious sort network that has size $N \log N$ for inputs of size N).
- Initializing level $i+1$, which performs:
  - $\mathsf{poly}(\lambda)$ operations to choose the PRF and SK-DEPIR keys and encrypt them.
  - $\mathsf{B} \cdot 2^i \cdot \mathsf{poly}(\lambda)$ operations to initialize the buckets and insert A into them.
  - $i \cdot 2^i \cdot \mathsf{poly}(\lambda)$ operations to obliviously sort A before inserting it into the buckets.
  - $t_S\left(\mathsf{B} \cdot 2^i\right)$ operations to run $\mathsf{DEPIR.Process}$ on the database obtained from the concatenation of all buckets.

Therefore, reshuffling level i requires performing $\mathsf{B} \cdot 2^i \cdot \mathsf{poly}(\lambda) + i \cdot 2^i \cdot \mathsf{poly}(\lambda) + t_S\left(\mathsf{B} \cdot 2^{i+1}\right)$ operations. Therefore, each `write` operation performs the following number of operations:

$$\mathsf{poly}(\lambda) + \frac{1}{2^i} \cdot \sum_{i=1}^{\log N} \left(\mathsf{B} \cdot 2^i \cdot \mathsf{poly}(\lambda) + i \cdot 2^i \cdot \mathsf{poly}(\lambda) + t_S\left(\mathsf{B} \cdot 2^{i+1}\right)\right)$$

$$= \mathsf{poly}(\lambda) \cdot \mathsf{B} \cdot \log N + \sum_{i=1}^{\log N} \left(\mathsf{poly}(\lambda) \cdot i + \frac{t_S\left(\mathsf{B} \cdot 2^{i+1}\right)}{2^i}\right)$$

□

4 Definition of RAM-FHE

We first formally define the RAM model we work with in Section 4.1, which is a simple model of RAM computation that captures their essential efficiency advantage over Turing machines. We then define single-hop RAM-FHE in Section 4.2 and multi-hop RAM-FHE in Section 6.2.

4.1 Definition of RAM machines

We define a simple model of RAM computation that captures their essential efficiency advantage over Turing machines. We emphasize that we will not describe the main algorithms (e.g, encryption, decryption, and evaluation) of our schemes in this formalism, and will instead describe these algorithms in the usual informal style. We will use the term algorithm when we do not intend to be precise about the exact model of computation.

We reserve the more formal term RAM machine to describe the object that the FHE evaluation algorithm takes as input.

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 a an operation on one of the RAM machine's tapes. In addition to the usual input, output, and work tapes, the RAM machine also has a special "persistent" tape. The main input to δ is the result of the previously output operation. Additionally, the circuit δ simulates statefulness by taking as input and producing as output an internal state.

Definition 4.1. A RAM machine with input space $\mathcal{X}$ and output space $\mathcal{Y}$ is a tuple $M = (Q, q_0, C)$, where:

- Q is a finite set, called the state space of M,
- q_0 is an element of Q, called the initial state of M,
- C is a circuit that computes a function δ, called the transition function of M, that maps[7]

$$\delta : \mathcal{X} \times Q \times \{0, 1, \epsilon\} \to (Q \times \mathsf{Ops}) \sqcup \mathcal{Y},$$

  where Ops is tuples of the form $(\mathsf{tape}, \mathsf{instr}, \mathsf{addr}, \mathsf{val})$, where $\mathsf{tape} \in \{\texttt{persistent}, \texttt{volatile}\}$, $\mathsf{instr} \in \{\texttt{read}, \texttt{write}\}$, $\mathsf{addr} \in \mathbb{Z}^+$, and $\mathsf{val} \in \{0, 1\}$.[8] We refer to the persistent tape as the database, and to the volatile tape as the scratch tape.

4.1.1 Execution Semantics

Definition 4.2. For $D, V \in \{0,1\}^*$, and $\mathsf{op} = (\mathsf{tape}, \mathsf{instr}, \mathsf{addr}, \mathsf{val})$, we denote by $\mathsf{op}(D, V)$ the tuple (b, D', V'), where $|D'| = |D|$, $|V'| = |V|$, and

$$b = \begin{cases} D_{\mathsf{addr}} & \text{if } \mathsf{instr} = \texttt{read}, \mathsf{tape} = \texttt{persistent}, \text{ and } 1 \le \mathsf{addr} \le |D| \\ V_{\mathsf{addr}} & \text{if } \mathsf{instr} = \texttt{read}, \mathsf{tape} = \texttt{volatile}, \text{ and } 1 \le \mathsf{addr} \le |V| \\ 0 & \text{otherwise} \end{cases}$$

$$D'_i = \begin{cases} \mathsf{val} & \text{if } \mathsf{instr} = \texttt{write}, \mathsf{tape} = \texttt{persistent}, \text{ and } \mathsf{addr} = i \\ D_i & \text{otherwise} \end{cases}$$

$$V'_i = \begin{cases} \mathsf{val} & \text{if } \mathsf{instr} = \texttt{write}, \mathsf{tape} = \texttt{volatile}, \text{ and } \mathsf{addr} = i \\ V_i & \text{otherwise} \end{cases}$$

Definition 4.3. We say that $M = (Q, q_0, C)$ terminates on input x and database D with output y if there exist sequences $q_1, \ldots, q_T \in Q$, $\mathsf{op}_1, \ldots, \mathsf{op}_T \in \mathsf{Ops}$, $b_1, \ldots, b_T \in \{0,1\}$, $D_1, \ldots, D_T \in \{0,1\}^{|D|}$, and $V_0, \ldots, V_T \in \{0,1\}^*$ such that:

[7] Notice that in particular, the description of M specifies the input and output domains.

[8] Assuming $\texttt{read}, \texttt{write}$ operations are on bits is without loss of generality, since they can be used to emulate operations on larger blocks.

- $V_0 = 0^S$ for some $S \in \mathbb{Z}^+$, and $b_0 = 0$,
- For $i \in [T]$, it holds that $(q_i, \mathsf{op}_i) = C(x, q_{i-1}, b_{i-1})$ and $(b_i, D_i, V_i) = \mathsf{op}_i(D_{i-1}, V_{i-1})$.
- It holds that $y = C(x, q_T, b_T)$.

In this case, the execution trace of M on x with database D is the sequence $(q_0, \mathsf{op}_0), \ldots, (q_T, \mathsf{op}_T), y$. We write $\mathsf{Time}(M, x, D)$ to denote T, and refer to it as the running time of M on input x and database D. We refer to D_T as the resultant database after executing M on input x and database D, and write $(y, D_T) = M^D(x)$.

4.2 Single-Hop RAM FHE

Definition 4.4 (Single-hop RAM FHE)**.** A public-key (single-hop) RAM FHE scheme is a tuple of PPT[9] 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.
  - $\mathsf{Enc}\left(\mathsf{pk}, D, 1^B\right)$ 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 $|M| \leq B$, any database $D \in \{0,1\}^*$, any input x, and any $T \in \mathbb{Z}^+$ with $\mathsf{Time}(M, x, D) \leq T$, 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.
- **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,

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

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

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

Remark 4.5. 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.

4.3 Multi-Hop RAM FHE

Definition 4.6 (Multi-hop RAM-FHE)**.** A public-key *multi-hop* RAM FHE scheme is a tuple of PPT algorithms (KeyGen, Enc, Dec) along with a RAM machine Eval such that

- **Correctness.** For any security parameter $\lambda \in \mathbb{Z}^+$, any database D_0, any bound $B \in \mathbb{Z}^+$, any sequence of RAM machines $M_1, \ldots, M_t$ with input space $\mathcal{X}$ and output space $\mathcal{Y}$, any inputs $x_1, \ldots, x_t \in \mathcal{X}$, and any time bounds $T_1, \ldots, T_t$: For each $i \in [t]$, define y_i and D_i by computing $(y_i, D_i) := M_i^{D_{i-1}}(x_i)$. If for each $i \in [t]$, $|M_i| \leq B$ and $\mathsf{Time}(M_i, x_i, D_{i-1}) \leq T_i$, then in the probability space defined by sampling

$$\begin{aligned} &(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ &\hat{D}_0 \leftarrow \mathsf{Enc}(\mathsf{pk}, D_0, B) \\ &\text{For } i \in [t]: \\ &\quad (\hat{y}_i, \hat{D}_i) := \mathsf{Eval}^{\hat{D}_{i-1}}(M_i, x_i, T_i) \\ &\quad y_i' := \mathsf{Dec}(\mathsf{sk}, \hat{y}_i) \end{aligned} \tag{2}$$

  it holds with probability 1 for each $i \in [t]$ that $y_i' = y_i$.
- **IND-CPA Security.** Same as for single-hop.
- $\eta(\cdot)$**-Efficiency.** In the experiment described in (5), it holds with probability 1 for each $i \in [t]$ that the running time of Eval is at most $\mathsf{Time}(M_i, x_i, D_{i-1}) \cdot \eta(|D_{i-1}|)\, \mathsf{poly}(B, \lambda)$.
- **Compactness.** In the experiment described in (5), it holds for each $i \in [t]$ that $|\hat{y}_i| \leq \mathsf{poly}(\log |\mathcal{Y}|, \lambda)$.

Remark 4.7. We note that in Definition 6.4, we distinguish between the *output* y_i of a RAM machine, and the updated database D_i that it produces by overwriting D_{i-1}. In particular, we cannot homomorphically evaluate programs that make random-access reads to $y^{(1)}, \ldots, y^{(i-1)}$.

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 intuitive idea requires two conceptual 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 $M^D(x)$, 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 5.1 (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

$$\begin{aligned}
&(y_1, D_1) \leftarrow M^{D_0}(x_1)\\
&\ldots\\
&(y_m, D_m) \leftarrow M^{D_{m-1}}(x_m)
\end{aligned} \tag{3}$$

- **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 (Definition 4.3), 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 $|D_0|$.

To formalize this notion, we first define the notion of address pattern.

Definition 5.2 (Address pattern)**.** For a RAM machine M with input x and RAM access to a database D, the address pattern of M^D on x consists of the list of physical memory accesses which M performed, and for every access, whether it was a `read` or a `write`. It also contains the output $M^D(x)$.

Definition 5.3 (Transcript/address-simulatable)**.** We say that a database-dependent RAM-VBB $\mathcal{O}$ for $\mathcal{M} = \{\mathcal{M}_N\}_N$ is transcript-simulable if for every PPT $\mathcal{A}$ (producing a single output bit), there is a PPT Sim and negligible function α such that for all $\lambda, N, n, k \in \mathbb{N}$, all databases $D_0 \in \{0,1\}^N$, all $m = \mathsf{poly}(\lambda)$, all RAM machines $M \in \mathcal{M}_N$ with input length n and output length k, and all "auxiliary input" $z \in \{0,1\}^*$, it holds that

$$\left| \Pr\left[\mathcal{A}\big(1^\lambda, \mathcal{O}(1^\lambda, D_0, M), z\big) = 1\right] - \Pr[\mathsf{Sim}^B(1^\lambda, z', 1^{|M|}, 1^n, 1^k, z) = 1] \right| \leq \alpha(\lambda) \tag{4}$$

where $z' = D_0$, and B is an oracle that on input $(x_1, \ldots, x_m)$ runs the experiment described in Eq. (3) and outputs, for each $i \in [m]$, the execution trace of $M^{D_{i-1}}$ on x_i.

We say that a database-dependent RAM-VBB obfuscator $\mathcal{O}$ is address-simulatable if Eq. (4) holds for $z' = 1^{|D_0|}$, and the oracle B that on input $(x_1, \ldots, x_m)$ runs the experiment described in Eq. (3) and outputs, for each $i \in [m]$, the address pattern of $M^{D_{i-1}}$ on x_i.

We abbreviate transcript/address-simulatable database-dependent RAM-VBB as transcript/address-simulatable RAM-VBB.

Definition 5.4. A transcript/address-simulable single-hop database-dependent RAM-VBB obfuscator for $\mathcal{M}$ is an obfuscator which satisfies Definitions 5.1 and 5.3, but with $m = 1$ in both the correctness and the security requirement.

Discussion We note that it is possible to build a *single-hop* database-dependent RAM-VBB obfuscator from a *multi-hop* database-dependent RAM-VBB obfuscator by augmenting the RAM machine M to only allow a single sequential execution.

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.

We note that 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 loss of generality that every execution step performs a single `read` from the database and scratch tape, and a single `write` to the scratch tape.

5.2.1 Transcript-Simulatable Database-Dependent RAM-VBB

We now construct a single-hop transcript-simulatable RAM-VBB obfuscator (see Section 6.2.1 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.

Construction 3 (Transcript-simulatable RAM-VBB obfuscation)**.** The transcript-simulatable RAM-VBB obfuscator $\mathcal{O}_{\mathsf{trans}}$ uses the following building blocks:

- 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, $\mathcal{O}_{\mathsf{trans}}$ operates as follows:

- 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.

We now prove that Construction 3 is a single-hop transcript-simulatable RAM-VBB obfuscator.

Claim 5.5. *Construction 3 is a single-hop transcript-simulatable RAM-VBB obfuscator for $\mathcal{M}$, when instantiated with:*

- *a secure MAC scheme,*
- *a family of CRHFs, and*
- *a VBB circuit obfuscator for $\mathcal{M}_{\mathsf{wrap}}$, where $\mathcal{M}_{\mathsf{wrap}}$ is the ensemble of classes of RAM machines obtained by instantiating the RAM machine from Figure 6 with $M \in \mathcal{M}$.*

Proof. Single-hop correctness follows from the correctness of the MAC and the VBB obfuscator. As for efficiency, M_{wrap} simply executes $\widetilde{C}$, and reads $O(1)$ paths in MT, so its running time is $\left|\widetilde{C}\right| + \mathsf{poly}(\log N, \lambda) = \left|\widetilde{C}\right| + \mathsf{poly}(\lambda)$ (because $N \leq 2^{\lambda}$). $\widetilde{C}$ performs a single transition step of M, reads a constant number of paths from MT, generates and verifies a constant number of MACs on input whose size is $|\mathsf{st}_M| + \mathsf{poly}(|M|, \lambda)$ (here, we use the fact that $N \leq 2^{\lambda}$, and $|x| \leq |M|$), where st_M is M's internal state. So $\mathsf{Time}\left(\widetilde{M}, (x, \mathcal{I}), D\right) \leq \mathsf{Time}(M, x, D) \cdot \mathsf{poly}(|M|, \lambda)$.

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.

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

We now prove security. Let $\mathcal{A}$ be a PPT adversary in Definition 5.3. We define a simulator $\mathsf{Sim}_{\mathsf{trans}}$ which uses the simulator Sim for $\mathcal{O}$, whose existence is guaranteed from the VBB-security of $\mathcal{O}$. More specifically, one can think of $\mathcal{A}$ as a PPT adversary against the VBB-security of $\widetilde{C}$, where the rest of $\mathcal{A}$'s input (namely, the processed database $\widetilde{D_0}$, the RAM machine M_{wrap}, the initial state st and the MAC σ on it) is the auxiliary input to the adversary in Definition 2.3, and let Sim be the simulator for $\mathcal{A}$ whose existence is guaranteed by the VBB-security of $\mathcal{O}$.

$\mathsf{Sim}_{\mathsf{trans}}$ on input $\left(D_0, 1^m, 1^n, 1^k, z\right)$, where m denotes the description size of a RAM machine and n, k denote its input and output lengths (respectively), operates as follows:

1. Picks a random CRHF $h \leftarrow \mathcal{H}$, and generates a random MAC key $K \leftarrow \mathsf{KeyGen}\left(1^\lambda\right)$.
2. Generates a MHT MT for D_0 with root Rt.
3. Calls its oracle on some arbitrary input x_0 to obtain the initial state st_0^M of the RAM machine M, and sets $\mathsf{st}_0 = \left(\mathsf{true}, \mathsf{st}_M^0, \mathsf{Rt}\right)$. (The oracle replies with the entire transcript of the execution, but we are only interested in the initial state.)
4. MACs the initial state: $\sigma = \mathsf{Tag}\left(K, \mathsf{st}^0\right)$.
5. Initializes a set T to be empty. (T will contain all recorded transcripts.)
6. Runs Sim with auxiliary input $\left(\mathsf{MT}, \mathsf{st}^0, \sigma\right)$, answering Sim's oracle calls to C_{Exec} as follows:
   (a) Each oracle call contains, as part of the input to C_{Exec}, a state st that contains some internal state st_M of M.
   (b) $\mathsf{Sim}_{\mathsf{trans}}$ performs Step 1 of Figure 5 using the input Sim provided to its oracle, and if any of the checks fail, returns $\perp$ as the answer of the oracle.
   (c) If $\mathsf{st}_M \neq \mathsf{st}_M^0$ then st contains some input x. If $\mathsf{trans}_x \notin \mathsf{T}$ then $\mathsf{Sim}_{\mathsf{trans}}$ returns $\perp$ to Sim.

(d) If $\mathsf{st}_M = \mathsf{st}_M^0$, then the input to the oracle also includes an input x for M. $\mathsf{Sim}_{\mathsf{trans}}$ calls its oracle on x, obtains the entire transcript trans_x of the execution of M on x, and adds trans_x to T.

(e) Recover $\mathsf{trans}_x = (L_x, y_x)$ from T, where y_x is the output of the computation, and $L_x = \left(\mathsf{addr}_{\mathsf{DB}}^i, \mathsf{addr}_{\mathsf{stape}}^i, \mathsf{addr}_w^i, \mathsf{val}^i, \mathsf{st}_M^i\right)_i$ consists of the memory accesses and M's internal state at the onset of each execution step. Let i be such that $\mathsf{st}_M^i = \mathsf{st}_M$. Sim uses trans_x to generate the output out of C_{Exec} after the i'th execution step. (Specifically, trans_x contains the entire history of the execution, that can be used to compute MHTs with h for the history and the updated database, which are then MACed with K.) $\mathsf{Sim}_{\mathsf{addr}}$ gives out to Sim as the answer of the oracle.

7. When Sim terminates with output b, $\mathsf{Sim}_{\mathsf{trans}}$ outputs b.

We now prove that $\mathsf{Sim}_{\mathsf{trans}}$ satisfies the requirements of Definition 5.3, by showing that the distributions over the output of $\mathcal{A}$ and of $\mathsf{Sim}_{\mathsf{trans}}$ are both computationally close to the following hybrid distribution $\mathcal{H}$. In $\mathcal{H}$, we replace $\mathcal{A}$ with the simulator Sim for $\mathcal{O}$. That is, Sim receives as auxiliary input all the information available to $\mathcal{A}$ in the real execution, *except* for the obfuscated circuit $\widetilde{C}$. In particular, it obtains the MHT MT for the database, M's initial state and a MAC on it, and a description of M_{wrap}. Sim also has oracle access to C_{Exec}. $\mathcal{H}$ is the distribution over the output of Sim in this experiment. Then $\mathcal{H}$ and the distribution over the output of $\mathcal{A}$ are computationally indistinguishable due to the VBB security of $\mathcal{O}$.

It remains to prove that $\mathcal{H}$ and the distribution over the output of $\mathsf{Sim}_{\mathsf{trans}}$ are computationally indistinguishable. Intuitively, this holds because the oracle queries of Sim that do not output $\perp$ are consistent with honest executions of M on the initial database D_0 with some inputs. Indeed, the MAC guarantees that Sim can only query the oracle with valid states of M (i.e., which he received during the execution), and the MHT guarantees the database values he provides to the oracle throughout the execution are consistent with the current database. We proceed to formalize this intuition.

We claim first that except with negligible probability, the following holds for every oracle call of Sim with state st: either C_{Exec} returns $\perp$, or $\mathsf{st} = \mathsf{st}_0$, or Sim obtained st as the output of a previous oracle call. Indeed, the state given as input to C_{Exec} is MACed, and if MAC-verification fails then C_{Exec} returns $\perp$. Since verification for a new state passes only if Sim successfully forged a MAC, or found a collision of h, which happens only with negligible probability, we can condition $\mathcal{H}$ and the simulation on the event that Sim fails to forge or find a collision. Conditioned on this event, oracle calls to queries with new states (that were not observed before) are identically distributed in $\mathcal{H}$ and the simulation ($\perp$ is returned in both cases), so we can disregard queries of Sim with new states.

Therefore, we can divide the oracle queries into (*non*-disjoint) *execution chains*, where an execution chain $\left((\mathsf{st}_0, D_0, x_0), (\mathsf{st}_1, D_1, x_1), \ldots, (\mathsf{st}_l, D_l, x_l)\right)$ is a maximal sequence of triples of databases, states, and inputs such that: (1) st_0, D_0 are the initial execution state and database (D_0 is uniquely determined by st_0 which contains the root of a MHT for D_0); and (2) for every $1 \leq i \leq l$, st_i is the output of C_{Exec} on input st_{i-1}, x_i, and some values $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$, and D_i is the current database. [10]

We claim that for every execution chain, and every consecutive triples $(\mathsf{st}_i, D_i, x_i), (\mathsf{st}_{i+1}, D_{i+1}, x_{i+1})$ in the chain, the following holds: (1) $x_i = x_{i+1}$; (2) the inputs $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ provided as part of the input to the oracle call resulting in $\mathsf{st}_{i+1}, D_{i+1}$ are consistent with D_i; and (3) for every $i + 1 < l$, D_{i+1} is obtained from D_i by performing the `write` instruction

[10] For example, consider the following oracle calls of Sim ($A \to B$ denotes Sim called the oracle with input state A, and some additional inputs, and received as output the state B): $\mathsf{st}_0 \to \mathsf{st}_1$, $\mathsf{st}_1 \to \mathsf{st}_2$, $\mathsf{st}_2 \to \mathsf{st}_3$, $\mathsf{st}_1 \to \mathsf{st}_2'$, $\mathsf{st}_2 \to \mathsf{st}_3''$, $\mathsf{st}_2' \to \mathsf{st}_3'$. Then these can be divided into 3 execution chains, containing the following states: $(\mathsf{st}_0, \mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3)$, $(\mathsf{st}_0, \mathsf{st}_1, \mathsf{st}_2', \mathsf{st}_3')$, $(\mathsf{st}_0, \mathsf{st}_1, \mathsf{st}_2, \mathsf{st}_3'')$. Notice that, for example, st_0 appears in all chains, even though Sim only queried the oracle once with state st_0.

$\left(\mathsf{addr}'_w, \mathsf{val}'\right)$ specified in st_{i+1}. To see why (1) holds, recall that st_{i+1} is obtained from st_i using input x_{i+1}. In Step 1, C_{Exec} verifies that x_{i+1} is consistent with the input reported in st_i and stores x_{i+1} as part of the updated state st_{i+1}, and by induction x_i is consistent with the input reported in st_i, so $x_i = x_{i+1}$. For (2), the collision-resistance of h guarantees that if the values $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ chosen by the PPT Sim are inconsistent with D_i then they will be inconsistent with the MHT of D_i except with negligible probability, and if they are inconsistent with the MHT of D_i then C_{Exec} outputs $\perp$. The argument for (3) is similar to (2): if D_{i+1} is not the database obtained from D_i by writing val' to address addr'_w, then D_{i+1} will be inconsistent with the MHT reported in st_{i+2}, and so C_{Exec} outputs $\perp$.

Consequently, the execution chains covering Sim's queries are consistent with (possibly partial) executions of M on initial database D_0 (with some inputs $x_1, x_2, \ldots$). In particular, given h and K, the oracle answers to all queries in the chains can be computed given the transcript of M's execution. Since this is exactly the way in which $\mathsf{Sim}_{\mathsf{trans}}$ answers Sim's oracle queries, and h, K in the simulation are identically distributed to the real world, the output of $\mathsf{Sim}_{\mathsf{trans}}$ and $\mathcal{H}$ are identically distributed, conditioned on the event that Sim does not forge a MAC or finds a collision in h. Since this event occurs with overwhelming probability, the distributions are computationally close. □

5.2.2 Address-Simulatable Database-Dependent RAM-VBB

In this section, 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.

Construction 4 (Address-simulatable RAM-VBB obfuscation)**.** The address-simulatable RAM-VBB obfuscator $\mathcal{O}_{\mathsf{addr}}$ uses the following building blocks:

- A transcript-simulatable RAM-VBB obfuscator $\mathcal{O}_{\mathsf{trans}}$.
- A CPA-secure symmetric encryption scheme $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$.
- An unbounded-input PRF F.

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

- Generates an encryption key $\mathsf{sk} \leftarrow \mathsf{KeyGen}\left(1^\lambda\right)$, and a random PRF key $K \leftarrow \{0,1\}^\lambda$.
- Encrypts $c_D \leftarrow \mathsf{Encrypt}\left(\mathsf{sk}, D_0\right)$.
- Runs the transcript-simulatable RAM-VBB obfuscator $\left(\widetilde{D}, \widetilde{M}, \mathcal{I}\right) \leftarrow \mathcal{O}_{\mathsf{trans}}\left(1^\lambda, c_D, M_{\mathsf{sk},K}\right)$, where $M_{\mathsf{sk},K}$ is the RAM machine described in Figure 7 (hard-wired values are hard-wired into the transition circuit of the machine).
- Outputs $\left(\widetilde{D}, \widetilde{M}, \mathcal{I}\right)$.

We now prove that Construction 4 is an address-simulator RAM-VBB obfuscator.

Claim 5.6. *Construction 4 is a single-hop address-simulatable RAM-VBB obfuscator for $\mathcal{M}$, when instantiated with:*

- *a secure unbounded-input PRF,*

- *a perfectly-correct CPA-secure symmetric encryption scheme, and*
- *a single-hop transcript-simulatable RAM-VBB obfuscator for $\mathcal{M}_{\mathsf{sk},K}$, where $\mathcal{M}_{\mathsf{sk},K}$ is the ensemble of classes of RAM machines obtained by instantiating the RAM machine from Figure 7 with $M \in \mathcal{M}$.*

Moreover, if the underlying transcript-simulator RAM-VBB obfuscator is multi-hop, then Construction 4 is a multi-hop *address-simulatable RAM-VBB obfuscator.*

Proof. Single-hop correctness follows from the perfect correctness of the encryption scheme, and the correctness of the transcript-simulatable RAM-VBB obfuscator.

As for efficiency, by the efficiency property of the transcript-simulatable RAM-VBB obfuscator, $\mathsf{Time}\left(\widetilde{M},(x,\mathcal{I}),\widetilde{D}\right) = \mathsf{Time}\left(M_{\mathsf{sk},K},x,D\right)\cdot\mathsf{poly}\left(\left|M_{\mathsf{sk},K}\right|,\lambda\right) = \mathsf{Time}\left(M_{\mathsf{sk},K},x,D\right)\cdot\mathsf{poly}\left(|M|,\lambda\right)$ (since $\left|M_{\mathsf{sk},K}\right| = \mathsf{poly}\left(|M|,\lambda\right)$). $M_{\mathsf{sk},K}$ performs a single step of M, performs a constant number of encryptions and decryptions on messages/ciphertexts of size $\mathsf{poly}\left(|M|,\lambda\right)$ (here, we use the fact that the output and input length, as well as the size of M's internal state, is at most $|M|$), and computes a single PRF image on an input of size $|\mathsf{st}_M| + \mathsf{poly}\left(|M|,\lambda\right)$ (where st_M is M's internal state) which by the PRF and encryption efficiency take time $\mathsf{poly}\left(|M|,\lambda\right)$. Therefore, $\mathsf{Time}\left(\widetilde{M},(x,\mathcal{I}),D\right) \leq \mathsf{Time}\left(M,x,D\right)\cdot\mathsf{poly}\left(|M|,\lambda\right)$.

We now prove security. Let $\mathcal{A}$ be a PPT adversary in Definition 5.3. We define a simulator $\mathsf{Sim}_{\mathsf{addr}}$ which uses the simulator $\mathsf{Sim}_{\mathsf{trans}}$ for $\mathcal{O}_{\mathsf{trans}}$, whose existence is guaranteed because $\mathcal{O}_{\mathsf{trans}}$ is transcript-simulatable. $\mathsf{Sim}_{\mathsf{addr}}$ on input $\left(1^N,1^m,1^n,1^k,z\right)$, where N denotes a database size, m denotes the description size of a RAM machine, and n,k denote input and output lengths (respectively), operates as follows:

- Generates a random encryption key $\mathsf{sk} \leftarrow \mathsf{KeyGen}\left(1^\lambda\right)$, and generates an encryption $c_D \leftarrow \mathsf{Encrypt}\left(\mathsf{sk},D'\right)$ of the all-zero database D' of size N. Let m' denote the description size of $M_{\mathsf{sk},K}$ when instantiated with a RAM machine M whose description size is m.
- Initializes a set Q to be empty. (Q will contain the queries which $\mathsf{Sim}_{\mathsf{trans}}$ made to his oracle, and the answers.)
- Emulates $\mathsf{Sim}_{\mathsf{trans}}$ on input $\left(c_D,1^{m'},1^n,1^k,z\right)$. Whenever $\mathsf{Sim}_{\mathsf{trans}}$ makes an oracle call with input x to its oracle, $\mathsf{Sim}_{\mathsf{addr}}$:
  - If $(x,T)\in Q$ (i.e., x was queried before), $\mathsf{Sim}_{\mathsf{addr}}$ gives T to $\mathsf{Sim}_{\mathsf{trans}}$ as the oracle query.
  - Otherwise, $\mathsf{Sim}_{\mathsf{addr}}$ calls its own oracle with x, to obtain the list $L_x = \left(\mathsf{addr}^i_{\mathsf{DB}},\mathsf{addr}^i_{\mathsf{stape}},\mathsf{addr}^i_w\right)_{i\in[t]}$ of physical addresses accessed by $M_{\mathsf{sk},K}$ (here, t denotes the number of execution steps $M_{\mathsf{sk},K}$ performs on x), and its output y_x.
  - For every $1\leq i\leq t$, generates an encryption $c_i \leftarrow \mathsf{Encrypt}\left(\mathsf{sk},0\right)$ of zero, and an encryption $c'_i \leftarrow \mathsf{Encrypt}\left(\mathsf{sk},\vec{0}\right)$ of the all-zero string of the same length as $M_{\mathsf{sk},K}$'s internal state. (Here, c_i emulates the value $M_{\mathsf{sk},K}$ writes to the physical memory, and c'_i emulates its internal state, in the i'th execution step.)
  - Let $L'_x = \left(\mathsf{addr}^i_{\mathsf{DB}},\mathsf{addr}^i_{\mathsf{stape}},\mathsf{addr}^i_w,c_i,c'_i\right)_{i\in[t]}$. $\mathsf{Sim}_{\mathsf{addr}}$ gives (L'_x,y_x) to $\mathsf{Sim}_{\mathsf{trans}}$ as the oracle answer to the query x, and adds (x,L'_x,y_x) to Q.
- When $\mathsf{Sim}_{\mathsf{trans}}$ terminates with output b, $\mathsf{Sim}_{\mathsf{addr}}$ outputs b.

We now prove that $\mathsf{Sim}_{\mathsf{addr}}$ satisfies the requirements of Definition 5.3, through a sequence of hybrids. The proof will use the following observations: (1) if $x \neq x'$ then the values v used in Step 4 throughout the execution of $M_{\mathsf{sk},K}$ on x are distinct from the values used in an execution on x'; and (2) for every input x, the values v used in Step 4 throughout the execution of $M_{\mathsf{sk},K}$ on x are all distinct. Indeed, (1) holds because the input to M is included in v and so different inputs result in different v values. (2) holds because v includes the counter count which is updated in every execution step.

$\mathcal{H}_\mathbf{0}$: Hybrid $\mathcal{H}_0$ is the output of the adversary $\mathcal{A}$ in Definition 5.3.

$\mathcal{H}_\mathbf{1}$: In $\mathcal{H}_1$, we replace $\mathcal{A}$ in the real world with the simulator $\mathsf{Sim}_{\mathsf{trans}}$. We stress that this is the only change from $\mathcal{H}_0$, in particular $\mathsf{Sim}_{\mathsf{trans}}$ obtains as input the encrypted database c_D, and every oracle call is answered with the entire transcript of the run of $M_{\mathsf{sk},K}$. *Hybrids $\mathcal{H}_0$ and $\mathcal{H}_1$ are computationally indistinguishable because $\mathcal{O}_{\mathsf{trans}}$ is transcript-simulatable.*

$\mathcal{H}_\mathbf{2}$: In $\mathcal{H}_2$, we replace $F(K, \cdot)$ with a random function f. *Hybrids $\mathcal{H}_1$ and $\mathcal{H}_2$ are computationally indistinguishable by the PRF security of F.*

$\mathcal{H}_\mathbf{3}$: In $\mathcal{H}_3$, we replace all ciphertexts with encryptions of zero. That is, encryptions of database blocks are replaced with encryptions of zero (both in c_D and in the ciphertexts computed in Step 5 of $M_{\mathsf{sk},K}$), and encryptions of the internal state in Step 6 of $M_{\mathsf{sk},K}$ are replaced with encryptions of the all-zero string of the "right" length. *Hybrids $\mathcal{H}_2$ and $\mathcal{H}_3$ are computationally indistinguishable by the IND-CPA security of the encryption scheme.*

Notice that $\mathcal{H}_3$ is exactly the distribution over the output of $\mathsf{Sim}_{\mathsf{addr}}$ (because all v values used throughout the execution in $\mathcal{H}_3$ are unique), which concludes the proof.

Multi-hop correctness and security: if the underlying transcript-simulatable RAM-VBB obfuscator is *multi-hop*, then Construction 4 is also multi-hop. Correctness follows directly from the correctness of $\mathcal{O}_{\mathsf{trans}}$. The multi-hop security proof is identical to the above proof of the single-hop case, except that $\mathsf{Sim}_{\mathsf{trans}}$'s oracle calls contain a sequence of inputs $(x_1, \ldots, x_m)$, and $\mathsf{Sim}_{\mathsf{addr}}$ either finds the transcript for this sequence in Q, or calls its own oracle with $(x_1, \ldots, x_m)$. □

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 box. We assume that (polynomial) a-priori bounds on the input, output, and description lengths of the RAM machine are known, and discuss extensions to the general setting (in which no such bounds are a-priori known) in Section 7. We upgrade the scheme to a multi-hop scheme in Section 6.2.

6.1 Single-Hop RAM-FHE

In this section we construct a single-hop RAM-FHE scheme, proving the following theorem:

Theorem 6.1 (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.*

Remark on the circuit-VBB obfuscator. Though Theorem 6.1 is stated in the circuit-VBB hybrid model, namely with a general-purpose obfuscator, it suffices to have a circuit-VBB obfuscator for the circuit C_{Exec} of Figure 5, when instantiated with the machine M obtained from instantiating the RAM machine $M_{\mathsf{sk},K}$ of Figure 7 with the universal machine $M_{\mathcal{U}}$ of Figure 8.

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 5 (Single-hop RAM-FHE)**.** The RAM-FHE scheme uses the following building blocks:

- 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 Section 2.3 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 8, 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 $\mathsf{PKE.Decrypt}\left(\mathsf{sk}, c\right)$.

Remark on growing Merkel Hash Trees. Our construction (in particular, the circuit C_{Exec} of Figure 5 on page 58) 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 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 $H\left(v_h, v'_h\right)$. (Notice that the resultant tree has hight $h+1$.) 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.

Claim 6.2 (Single-hop RAM-FHE security)**.** *Construction 5 satisfies the correctness and security properties of Definition 4.4, when instantiated with:*

- *a secure unbounded-input PRF,*
- *a perfectly-correct IND-CPA secure PKE scheme,*
- *a secure ISR-ORAM scheme with a deterministic client during* $\mathsf{ISR-ORAM.Access}$*, and*
- *a secure single-hop address-simulatable RAM-VBB obfuscator for the RAM machine* $\mathcal{M}_{\mathcal{U}}$ *of Figure 8.*

Claim 6.3 (Single-hop RAM-FHE efficiency)**.** *Construction 5 satisfies the compactness property of Definition 4.4. If additionally the underlying ORAM scheme for databases of size N has a client with internal state of size* $\mathsf{poly}(\lambda)$ *and the* Access *protocol takes time* $\mathsf{poly}(\lambda)$*, then Construction 5 satisfies the* $\mathsf{poly}\log N$*-efficiency property of Definition 4.4.*

Claims imply theorem. We use the claims to prove Theorem 6.1. We now use the claims to prove Theorem 6.1.

Proof of Theorem 6.1. The assumption that OWFs exist implies the existence of secure MACs, unbounded-input PRFs, and perfectly-correct IND-CPA secure symmetric encryption schemes. Therefore, together with the assumption that CRHFs exist, Claims 5.5 and 5.6 guarantee that there exists an address-simulatable RAM-VBB obfuscator in the circuit-VBB hybrid model. The existence of OWFs and SK-DEPIR implies additionally the existence of a secure ISR-ORAM scheme with a deterministic client during Access by Theorem 3.5 and Claim B.3. Therefore, security follows from Claim 6.2 and from the assumption that PKE schemes exist. Efficiency follows from Claim 6.3 using also Theorem 3.5 and Claim B.1. □

Proof of claims. We now proceed to prove the claims.

Proof of Claim 6.2. Single-hop correctness follows from the correctness of all underlying primitives, where the perfect correctness of the PKE scheme guarantees that decryption succeeds even though the ciphertext was generated with pseudorandom coins. We proceed to prove security. Let $\mathcal{A} = \left(\mathcal{A}^0, \mathcal{A}^1\right)$ be a PPT adversary in the security property of Definition 4.4, and let $\mathsf{DB}_0, \mathsf{DB}_1$ be the database that $\mathcal{A}^0$ chooses given pk as input. Let $\mathcal{B}_0, \mathcal{B}_1$ denote the distribution over the guess b' of $\mathcal{A}^1$ when $b = 0$ and $b = 1$, respectively. We prove that $\mathcal{B}_0 \approx \mathcal{B}_1$ by a sequence of hybrids, and thus $\mathcal{A}_1$ has only negligible advantage in guessing b.

$\mathcal{H}_{\mathbf{0}}^{\mathbf{b}}, \mathbf{b} = \mathbf{0}, \mathbf{1}$: This is simply $\mathcal{B}_b$.

$\mathcal{H}_{\mathbf{1}}^{\mathbf{b}}$: In $\mathcal{H}_1^b$, we replace $\mathcal{A}^1$ with the corresponding simulator Sim for $\mathcal{O}$, whose existence is guaranteed from address-simulatability, where the oracle B of Sim provides, for every input x, the list of physical memory accesses accessed by $M_{\mathcal{U}}$ on input x, and its output. *Hybrids $\mathcal{H}_0^b$ and $\mathcal{H}_1^b$ are computationally indistinguishable because $\mathcal{O}$ is address-simulatable.*

$\mathcal{H}_{\mathbf{2}}^{\mathbf{b}}$: In $\mathcal{H}_2^b$, we replace $F(K, \cdot)$ with a truly random function f. *Hybrids $\mathcal{H}_1^b$ and $\mathcal{H}_2^b$ are computationally indistinguishable by the PRF security. (Notice that in $\mathcal{H}_1^b$, the simulator has no access to K, except through the output of $M_{\mathcal{U}}$.)*

$\mathcal{H}_{\mathbf{3}}^{\mathbf{b}}$: In $\mathcal{H}_3^b$, we replace c with an encryption of the all-zero string. *Hybrids $\mathcal{H}_2^b$ and $\mathcal{H}_3^b$ are computationally indistinguishable by the IND-CPA security of the encryption scheme, because encryption randomness is generated using a truly random function. (Here, we use the fact that encryption randomness is reused for two outputs y, y' if and only if they were obtained by applying the same RAM machine M on the same input x for the same number of transition steps T, in which case $y = y'$ since M is deterministic. Therefore, security can indeed be reduced to the IND-CPA security of the encryption scheme.)*

$\mathcal{H}_{\mathbf{4}}^{\mathbf{b}}$: In $\mathcal{H}_4^b$, we replace B with the following oracle: it replaces all the accesses to virtual memory performed by M with accesses that read address 0 from the database and the scratch tape, and write 0 to address 0 of the scratch tape. We stress that during the emulation, M is still given the actual values of the database and scratch tape that it would have read, it is just the reported memory accesses that are replaced. Notice that by the definition of $M_{\mathcal{U}}$, these virtual memory accesses result in a sequence of physical memory accesses performed by the ISR-ORAM client. Notice also that the output of $M_{\mathcal{U}}$ reported in B is simply an encryption of the all-zero string, as in $\mathcal{H}_3^b$. *Hybrids $\mathcal{H}_3^b$ and $\mathcal{H}_4^b$ are computationally indistinguishable by the rewindable security of the ISR-ORAM.*

The claim now follows because $\mathcal{H}_4^0 \equiv \mathcal{H}_4^1$ since neither contain any information about $\mathsf{DB}^0, \mathsf{DB}^1$. (Indeed, the simulator is not given an encryption of the database or the output of the computation, and the physical memory accesses provided by the oracle are generated for a fixed sequence of logical memory accesses, independent of the database.) □

Proof of Claim 6.3. For compactness, notice that the output of $M_{\mathcal{U}}$ is simply an encryption of M's output, so it has size $\mathsf{poly}(\log \mathcal{Y}, \lambda)$.

As for efficiency, by the efficiency of the address-simulatable RAM-VBB obfuscator (Claim 5.6), $\mathsf{Time}\left(\widetilde{M_{\mathcal{U}}}, x, D\right) \leq \mathsf{Time}(M_{\mathcal{U}}, x, D) \cdot \mathsf{poly}(|M_{\mathcal{U}}|, \lambda)$. $M_{\mathcal{U}}$'s internal state consists of the internal state of an ORAM client (of size $\mathsf{poly}(\log(N+s), \lambda) = \mathsf{poly}(\lambda)$), the input and output of M (each consisting of at most $|M|$ bits), and $O(1)$ additional variables of size $O(\lambda)$ (notice that storing the runtime bound T requires $\log T \leq \lambda$ bits, because $T \leq 2^{\lambda}$), as well as the input x, description M, and runtime T of the RAM machine. Therefore, $M_{\mathcal{U}}$'s internal state has size at most $\mathsf{poly}(|M|, \lambda)$. Moreover, $M_{\mathcal{U}}$ performs T steps, in each it runs a single step of M (which takes $\mathsf{poly}(|M|, \lambda)$ time), performs 2 address translations ($\log N = \mathsf{poly}(\lambda)$ time) and emulates a constant number of executions of the Access protocol (each taking $\mathsf{poly}(\log(N+s), \lambda) = \mathsf{poly}(\lambda)$ time). Therefore, $\mathsf{Time}(M_{\mathcal{U}}, x, D) \leq T \cdot \mathsf{poly}(|M|, \lambda)$. □

6.2 Upgrading to a Multi-Hop Scheme

In this section, we extend Definition 4.4 to the multi-hop setting, and describe a multi-hop RAM-FHE scheme. We first formally define the notion of multi-hop RAM-FHE.

Definition 6.4 (Multi-hop RAM-FHE)**.** A public-key *multi-hop* RAM FHE scheme is a tuple of PPT algorithms (KeyGen, Enc, Dec) along with a RAM machine Eval such that

- **Correctness.** For any security parameter $\lambda \in \mathbb{Z}^+$, any database D_0, any bound $B \in \mathbb{Z}^+$, any sequence of RAM machines $M_1, \ldots, M_t$ with input space $\mathcal{X}$ and output space $\mathcal{Y}$, any inputs $x_1, \ldots, x_t \in \mathcal{X}$, and any time bounds $T_1, \ldots, T_t$: For each $i \in [t]$, define y_i and D_i by computing $(y_i, D_i) := M_i^{D_{i-1}}(x_i)$. If for each $i \in [t]$, $|M_i| \leq B$ and $\mathsf{Time}(M_i, x_i, D_{i-1}) \leq T_i$, then in the probability space defined by sampling

$$\begin{aligned} &(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda) \\ &\hat{D}_0 \leftarrow \mathsf{Enc}(\mathsf{pk}, D_0, B) \\ &\text{For } i \in [t]: \\ &\quad (\hat{y}_i, \hat{D}_i) := \mathsf{Eval}^{\hat{D}_{i-1}}(M_i, x_i, T_i) \\ &\quad y_i' := \mathsf{Dec}(\mathsf{sk}, \hat{y}_i) \end{aligned} \tag{5}$$

  it holds with probability 1 for each $i \in [t]$ that $y_i' = y_i$.

- **IND-CPA Security.** Same as for single-hop.

- $\eta(\cdot)$**-Efficiency.** In the experiment described in (5), it holds with probability 1 for each $i \in [t]$ that the running time of Eval is at most $\mathsf{Time}(M_i, x_i, D_{i-1}) \cdot \eta(|D_{i-1}|)\,\mathsf{poly}(B, \lambda)$.

- **Compactness.** In the experiment described in (5), it holds for each $i \in [t]$ that $|\hat{y}_i| \leq \mathsf{poly}(\log|\mathcal{Y}|, \lambda)$.

Remark 6.5. We note that in Definition 6.4, we distinguish between the *output* y_i of a RAM machine, and the updated database D_i that it produces by overwriting D_{i-1}. In particular, we cannot homomorphically evaluate programs that make random-access reads to $y^{(1)}, \ldots, y^{(i-1)}$.

We now describe how to upgrade the single-hop scheme of Construction 5 to a multi-hop scheme, proving the following:

Theorem 6.6 (Multi-hop RAM-FHE)**.** *Let $\epsilon \in (0, 1)$ be a constant. Assume the existence of OWFs, CRHFs, PKE schemes, and SK-DEPIR which for size-N databases has $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$ Process complexity, and $N^\epsilon \cdot \mathsf{poly}(\lambda)$ Query and Decode complexity, where λ denotes the security parameter. Then for every $d = \mathsf{poly}(\lambda)$ there exists an N^ϵ-efficient multi-hop RAM-FHE scheme in the circuit-VBB hybrid model for RAM machines with input length, output length, and description size at most d.*

The multi-hop RAM-FHE scheme uses a multi-hop transcript-simulatable RAM-VBB obfuscator, so we first describe this modified construction.

We note that in the multi-hop case, we can no longer assume the database is read-only. Consequently, in this setting we assume (without loss of generality) that each transition step performs a single `read` to the database and scratch tape, and a single `write` to the database and scratch tape.

6.2.1 Multi-Hop Transcript-Simulatable RAM-VBB obfuscation

At a high level, the only reason Construction 3 is not multi-hop correct is that once one execution of M is completed, and the database MHT is updated, one cannot run M on the updated database, because it does not have a signature on the *initial* state of M together with the *updated* database MHT root. Therefore, to allow for multi-hop evaluation, we need to modify C_{Exec} of Figure 5 such that when M terminates, its output includes not only y, but also a signature on M's initial state and

the current MHT root. Note, however, that the current MHT should *not* include the current contents of the scratch tape, so we need to use separate MHTs for the database and scratch tape.

Specifically, the internal state st which C_{Exec} takes as input should include (in *every* call to C_{Exec}) also M's initial state. We will additionally need st to include an initial MHT root Rt which is *not* updated by C_{Exec}. This will be needed in the simulation, to allow $\mathsf{Sim}_{\mathsf{trans}}$ to associate an oracle call of the circuit-VBB simulator Sim to C_{Exec} with a sequence of executions of M on some inputs $x_1, \ldots, x_m$. This is formalized in the next construction:

Construction 6 (Multi-hop transcript-simulatable RAM-VBB obfuscator)**.** The multi-hop transcript-simulatable RAM-VBB obfuscator $\mathcal{O}^{\mathsf{mh}}_{\mathsf{trans}}$ is identical to the single-hop transcript-simulatable RAM-VBB obfuscator of Construction 3, except that it uses the circuit-VBB obfuscator $\mathcal{O}$ to obfuscate the following circuit $C^{\mathsf{mh}}_{\mathsf{Exec}}$:

- The input st of $C^{\mathsf{mh}}_{\mathsf{Exec}}$ contains also: the initial state st^0_M of M; an initial MHT root Rt_0; a root $\mathsf{Rt}_{\mathsf{stape}}$ of a MHT $\mathsf{MT}_{\mathsf{stape}}$ for the scratch tape; and a value $\mathsf{val}_{\mathsf{DB},w}$ and address $\mathsf{addr}_{\mathsf{DB},w}$ such that $\mathsf{val}_{\mathsf{DB},w}$ was written to address $\mathsf{addr}_{\mathsf{DB},w}$ of the database in the last transition step.
- $C^{\mathsf{mh}}_{\mathsf{Exec}}$ takes as input also a path $\mathcal{P}_{\mathsf{DB},w}$ to the node $\mathsf{addr}_{\mathsf{DB},w}$ in MT. The paths $\mathcal{P}_{\mathsf{stape}}, \mathcal{P}_w$ are paths in $\mathsf{MT}_{\mathsf{stape}}$ and not in MT.
- Step 1a of C_{Exec} is replaced with the following: $\mathcal{P}_{\mathsf{DB}}, \mathcal{P}_{\mathsf{DB},w}$ ($\mathcal{P}_{\mathsf{stape}}, \mathcal{P}_w$) are paths to the nodes $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{DB},w}$ ($\mathsf{addr}_{\mathsf{stape}}, \mathsf{addr}_w$) in the MHT whose root is Rt ($\mathsf{Rt}_{\mathsf{stape}}$), and the values at $\mathsf{addr}_{\mathsf{DB}}, \mathsf{addr}_{\mathsf{stape}}$ are $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$.
- The following is added to Step 1 of C_{Exec}: if $b_{\mathsf{first}} = \mathsf{true}$ then generate a MHT $\mathsf{MT}_{\mathsf{stape}}$ for the (empty) scratch tape stape, and let $\mathsf{Rt}_{\mathsf{stape}}$ be its root.
- Step 2 of C_{Exec} is replaced with the following: if $b_{\mathsf{first}} = \mathsf{false}$ then use $\mathcal{P}_{\mathsf{DB},w}$ ($\mathcal{P}_w$) to compute the root Rt' ($\mathsf{Rt}'_{\mathsf{stape}}$) of the MHT obtained from MT ($\mathsf{MT}_{\mathsf{stape}}$) by replacing the value of the node $\mathsf{addr}_{\mathsf{DB},w}$ (addr_w) with $\mathsf{val}_{\mathsf{DB},w}$ (val_w).
- Step 3b of C_{Exec} is replaced with the following: If M terminated in the current step with output y, then set $b_{\mathsf{fin}} = \mathsf{true}$, $\mathsf{st}_{\mathsf{out}} = \left(\mathsf{true}, \mathsf{st}^0_M, \mathsf{Rt}'\right)$ (where Rt' is the root of the updated MHT computed in Step 2), $\sigma_{\mathsf{out}} = \mathsf{Tag}\left(K_{\mathsf{MAC}}, \mathsf{st}_{\mathsf{out}}\right)$, and $\mathsf{out} = (b_{\mathsf{fin}}, y, \mathsf{st}_{\mathsf{out}}, \sigma_{\mathsf{out}})$, and go to Step 5.
- The execution in Step 3c of C_{Exec} results also in a `write` instruction $\left(\mathsf{addr}'_{\mathsf{DB},w}, \mathsf{val}'_{\mathsf{DB},w}\right)$ to the database.
- In Step 4 of C_{Exec}, if $b_{\mathsf{first}} = \mathsf{true}$, then set $\mathsf{Rt}_0 := \mathsf{Rt}$. (This sets Rt_0 to be the root of the MHT for the database at the onset of M's execution, otherwise it is simply copied from the previous state. Thus, throughout the execution of M, Rt_0 is consistent with the database at the *onset* of the computation.)
- In Step 4 of C_{Exec}, st' includes Rt_0, st^0_M, $\mathsf{Rt}_{\mathsf{stape}}$, $\mathsf{addr}'_{\mathsf{DB},w}$ and $\mathsf{val}'_{\mathsf{DB},w}$.

We prove that Construction 6 is a multi-hop transcript-simulatable RAM-VBB obfuscator.

Claim 6.7. *Under the assumptions of Claim 5.5, Construction 6 is a* multi-hop *transcript-simulatable RAM-VBB obfuscator for $\mathcal{M}$.*

Proof sketch. The efficiency analysis is identical to the proof of Claim 5.5. Multi-hop correctness follows from the correctness of the MAC and the circuit-VBB obfuscator, and from the description

of $C^{\mathsf{mh}}_{\mathsf{Exec}}$ that outputs a MAC on the MHT of the updated database and M's initial state, which can be used to initiate a new execution of M on the updated database.

As for security, we use the simulator described in the proof of Claim 5.5, changing only how it answers Sim's oracle calls. Specifically, we replace Step 6 with the following:

- Each oracle call contains, as part of the input to $C^{\mathsf{mh}}_{\mathsf{Exec}}$, a state st that contains some internal state st_M of M, and the root Rt_0 of a database MHT.
- $\mathsf{Sim}_{\mathsf{trans}}$ performs Step 1 of $C^{\mathsf{mh}}_{\mathsf{Exec}}$ using the input Sim provided to its oracle, and if any of the checks fail, returns $\perp$ as the answer of the oracle.
- If $\mathsf{st}_M \neq \mathsf{st}^0_M$ then st contains some input x. $\mathsf{Sim}_{\mathsf{trans}}$ checks whether there exist $x_1, \ldots, x_m$ such that $\mathsf{trans}_{x_1,\ldots,x_m,x} \in \mathsf{T}$, and additionally that the database at the end of M's sequential execution on $x_1, \ldots, x_m$ is consistent with Rt_0. If such a transcript exists, $\mathsf{Sim}_{\mathsf{trans}}$ sets $\vec{x} = (x_1, \ldots, x_m, x)$, otherwise he returns $\perp$ to Sim.
- If $\mathsf{st}_M = \mathsf{st}^0_M$, then the input to the oracle also includes an input x for M. If Rt_0 is consistent with D_0 then set $\vec{x} = (x)$. Otherwise, $\mathsf{Sim}_{\mathsf{trans}}$ checks whether there exist $x_1, \ldots, x_m$ such that $\mathsf{trans}_{x_1,\ldots,x_m} \in \mathsf{T}$, and additionally that the database at the end of M's sequential execution on $x_1, \ldots, x_m$ is consistent with Rt_0. If such a transcript exists, $\mathsf{Sim}_{\mathsf{trans}}$ sets $\vec{x} = (x_1, \ldots, x_m, x)$, otherwise he returns $\perp$ to Sim.
- $\mathsf{Sim}_{\mathsf{trans}}$ calls its oracle on $\vec{x}$, obtains the entire transcript $\mathsf{trans}_{\vec{x}}$ of the execution of M on x, and adds $\mathsf{trans}_{\vec{x}}$ to T. (We note that $\mathsf{trans}_{\vec{x}}$ may already be in T, in which case there is no need to call the oracle, since the execution is deterministic so the transcript generated by the oracle is guaranteed to be identical to the transcript already in T.)
- $\mathsf{Sim}_{\mathsf{trans}}$ Recovers $\mathsf{trans}_{\vec{x}} = (L_{\vec{x}}, y_{\vec{x}})$ from T, where $y_{\vec{x}}$ is the list of outputs of the computations on the inputs in $\vec{x}$, and $L_{\vec{x}} = \left(\mathsf{addr}^{i,j}_{\mathsf{DB}}, \mathsf{addr}^{i,j}_{\mathsf{stape}}, \mathsf{addr}^{i,j}_{\mathsf{DB},w}, \mathsf{addr}^{i,j}_{w}, \mathsf{val}^{i,j}_{\mathsf{DB}}, \mathsf{val}^{i,j}, \mathsf{st}^{i,j}_{M}\right)_{i,j}$ consists of the memory accesses and M's internal state at the onset of each transition step on each input x_j in $\vec{x}$. Let i, j be such that $\mathsf{st}^{i,j}_M = \mathsf{st}_M$. Sim uses $\mathsf{trans}_{\vec{x}}$ to generate the output out of $C^{\mathsf{mh}}_{\mathsf{Exec}}$ after the i'th transition step on input x_j. (Specifically, $\mathsf{trans}_{\vec{x}}$ contains the entire history of the execution, that can be used to compute MHTs with h for the history and the updated database and scratch tape, which are then MACed with K.) $\mathsf{Sim}_{\mathsf{trans}}$ gives out to Sim as the answer of the oracle.

The proof now proceeds similarly to the proof of Claim 5.5, and we therefore only describe the needed modifications. First, in $\mathcal{H}$ the simulator Sim is given oracle access to $C^{\mathsf{mh}}_{\mathsf{Exec}}$, not to C_{Exec}. As in the proof of Claim 5.5, we condition $\mathcal{H}$ and the simulation on the event that Sim fails to forge a MAC or find a collision of h, and can therefore assume all oracle calls Sim makes are with states it observed before. Therefore, we can divide Sim's oracle queries into (non-disjoint) *generalized execution chains*, where a generalized execution chain is $\left(\mathsf{EC}_{x_0,D^{\mathsf{in}}_0,D^{\mathsf{out}}_0}, \mathsf{EC}_{x_1,D^{\mathsf{in}}_1,D^{\mathsf{out}}_1}, \ldots, \mathsf{EC}_{x_m,D^{\mathsf{in}}_m,D^{\mathsf{out}}_m}\right)$ where $D^{\mathsf{in}}_0 = D_0$, and for every $1 \leq i \leq m$, $\mathsf{EC}_{x_i,D^{\mathsf{in}}_i,D^{\mathsf{out}}_i}$ is an execution chain (as defined in the proof of Claim 5.5) corresponding to the execution of $M^{D^{\mathsf{in}}_i}$ on x_i, and resulting in the updated database D^{out}_i.[11]

[11] In the proof of Claim 5.5, an execution chain included a single MHT for both the database and scratch tape, and the database was read-only. However, this naturally extends to the case where the database and scratch tape have different MHTs (where both roots are part of the state), and M is allowed to write to the database. As in the proof of Claim 5.5, an execution chain corresponds to an honest execution of M on x, with valid contents of the database and scratch tape throughout the execution, and the proof is similar to the proof provided in the single-hop setting, by replacing D_0 with D^{in}_i.

We claim that for every generalized execution chain, and every consecutive execution chains $\mathsf{EC}_{x_i,D_i^{\mathsf{in}},D_i^{\mathsf{out}}}, \mathsf{EC}_{x_{i+1},D_{i+1}^{\mathsf{in}},D_{i+1}^{\mathsf{out}}}$, it holds that $D_{i+1}^{\mathsf{in}} = D_i^{\mathsf{out}}$. Indeed, the first oracle call in the execution chain $\mathsf{EC}_{x_{i+1},D_{i+1}^{\mathsf{in}},D_{i+1}^{\mathsf{out}}}$ is possible only if Sim can provide a valid MAC on the initial state st_M^0 of M, together with the root of a MHT for D_{i+1}^{in}. Since we have conditioned on the event that Sim doesn't forge a MAC or find a collision in h, this is possible only if Sim obtained this MAC as the output of a previous oracle call. By the definition of $C_{\mathsf{Exec}}^{\mathsf{mh}}$, it outputs such a MAC only at the end of an execution of M which results in the updated database D_{i+1}^{in}.

Consequently, each generalized execution chain covering Sim's queries is consistent with a sequence of executions of M with a mutable database that initially equals D_0, on a sequence of inputs $(x_1, \ldots, x_m)$ (the last execution on x_m might be a partial execution). The remainder of the proof follows identically to the proof of Claim 5.5. □

6.2.2 Multi-Hop RAM FHE

We now use ASR-ORAMs and multi-hop address-simulatable RAM-VBB (whose existence follows from the existence of multi-hop transcript-simulatable RAM-VBB) to construct a multi-hop RAM-FHE scheme. At a high level, the scheme is similar to the single-hop RAM-FHE construction (Construction 5, but the underlying building blocks are replaced with their multi-hop counterparts (i.e., the ASR-ORAM and the multi-hop address-simulatable RAM-VBB). However, two additional modifications are needed, as we now describe.

First, in the single-hop setting, we could instantiate a single (ISR-)ORAM that contained both the database and scratch tape. This simplified the construction (since it used a single ORAM), and was possible because in the single-hop setting, the database and scratch tape are treated in the same way when an execution of a RAM machine M terminates: they are both restored to their initial state (the database is restored to its original content, and the scratch tape is erased). However, this is not the case in the multi-hop setting, where the scratch tape is erased when the execution ends, but the updated database should be kept (so that it can be used in future executions). Therefore, in the multi-hop construction the scratch tape and database are implemented using separate ORAMs.

Second, we need to generate randomness to encrypt the computation output, and to generate the scratch-tape ORAM during evaluation. In the single-hop setting, the randomness was generated by applying a PRF to (M, x, T) where M is the RAM machine to be evaluated, x is its input, and T are the number of transitions steps that would be performed. In the single-hop setting, this guaranteed that randomness is reused only when the entire execution transcript is identical, because it involved evaluating the (deterministic) RAM machine M on input x for T steps with RAM access to the initial database. In the multi-hop setting, this is no longer the case, because different executions might use *different databases* (that were generated in previous executions). Therefore, if the randomness is generated using only (M, x, T) then it might be reused in two *different* executions that evaluate $M^D(x)$ and $M^{D'}(x)$ for T steps, resulting in two different execution transcripts. Consequently, we must generate the randomness based also on the database contents. We do this by maintaining a MHT of the current database, and generate the randomness from (M, x, T, Rt), where Rt is the MHT root. Since such a MHT is already maintained by the transcript-simulatable RAM-VBB obfuscator which is used to instantiate the address-simulatable RAM-VBB obfuscator used to obfuscate the universal RAM machine, we avoid duplication and simply have the obfuscated machine output this as part of the auxiliary information. This is formalized in the following construction.

Construction 7 (Multi-hop RAM-FHE)**.** The RAM-FHE scheme uses the following building blocks:

- The multi-hop address-simulatable RAM-VBB obfuscator $\mathcal{O}$ of Construction 4, which is instantiated with the transcript-simulatable RAM-VBB obfuscator of Construction 6.

- An ASR-ORAM scheme $(\mathsf{ASR-ORAM.Setup}, \mathsf{ASR-ORAM.Access})$ with a deterministic client during $\mathsf{ASR-ORAM.Access}$.
- An ORAM scheme for initially-empty databases $(\mathsf{Setup}, \mathsf{Access})$ with a deterministic client during Access.
- A perfectly-correct PKE scheme $(\mathsf{PKE.KeyGen}, \mathsf{PKE.Encrypt}, \mathsf{PKE.Decrypt})$.
- An unbounded-input PRF F.

It consists of the following algorithms: KeyGen and $\mathsf{Decrypt}$ which are identical to the single-hop setting (Construction 5), and $\mathsf{Encrypt}$ and Eval which are defined as follows.

$\mathsf{Encrypt}$ on input a public key pk, a database DB, and bounds n, k, d on the input length, output length, and description size of RAM machines (respectively), operates as follows:

- Initialize an ASR-ORAM with DB, by running $(\mathsf{ck}_{\mathsf{ASR}}, \mathsf{st}_{\mathsf{ASR}}) \leftarrow \mathsf{ASR-ORAM.Setup}\left(1^\lambda, \mathsf{DB}\right)$, to obtain a client state $\mathsf{ck}_{\mathsf{ASR}}$ and a server state $\mathsf{st}_{\mathsf{ASR}}$.
- Pick a random PRF key $K \leftarrow \{0,1\}^\lambda$.
- Run $\left(\widetilde{\mathsf{DB}}, \widetilde{M^{\mathsf{mh}}_{\mathcal{U}}}, \mathcal{I}\right) \leftarrow \mathcal{O}\left(1^\lambda, \mathsf{st}_{\mathsf{ISR}}, M^{\mathsf{mh}}_{\mathcal{U}}, 1^{|M^{\mathsf{mh}}_{\mathcal{U}}|}, 1^n, 1^k\right)$, where $M^{\mathsf{mh}}_{\mathcal{U}}$ is the RAM machine described in Figure 11, with hard-wired values pk, K, and internal variable $\mathsf{ck}_{\mathsf{ASR}}$.
- Output the ciphertext $c_{\mathsf{DB}} = \left(\widetilde{\mathsf{DB}}, \widetilde{M^{\mathsf{mh}}_{\mathcal{U}}}, \mathcal{I}\right)$.

Eval takes as input a description M of size at most d of a RAM machine, an input $x \in \{0,1\}^n$ for M, and a bound T on the runtime of M. It also has RAM access to a database-ciphertext $c_{\mathsf{DB}} = \left(\widetilde{\mathsf{DB}}, \widetilde{M_{\mathcal{U}}}, \mathcal{I}\right)$ consisting of an encoding $\widetilde{\mathsf{DB}}$ of a database, an obfuscation $\widetilde{M_{\mathcal{U}}}$ of a universal RAM machine, and some input information $\mathcal{I}$ for $\widetilde{M_{\mathcal{U}}}$. Moreover, $\mathcal{I}$ contains a root Rt of a MHT for the database encoded in $\widetilde{\mathsf{DB}}$.[12] Eval runs $\widetilde{M_{\mathcal{U}}}\left(M, T, x, \mathsf{Rt}, \mathcal{I}\right)$ with RAM access to $\widetilde{\mathsf{DB}}$, and outputs whatever $\widetilde{M_{\mathcal{U}}}$ outputs.

We prove the following claims regarding Construction 7:

Claim 6.8 (Multi-hop RAM-FHE security)**.** *Construction 7 satisfies the correctness and security properties of Definition 4.4, when instantiated with:*

- *a secure unbounded-input PRF,*
- *a perfectly-correct IND-CPA secure PKE scheme,*
- *a secure ORAM scheme for initially-empty databases with a deterministic client during* Access*,*
- *a secure ASR-ORAM scheme with a deterministic client during* $\mathsf{ASR-ORAM.Access}$*, and*
- *the multi-hop address-simulatable RAM-VBB obfuscator of Construction 4 (using Construction 6 as the underlying obfuscator) for the RAM machine* $\mathcal{M}^{\mathsf{mh}}_{\mathcal{U}}$ *of Figure 11.*

Claim 6.9 (Multi-hop RAM-FHE efficiency)**.** *Construction 7 satisfies the compactness property of Definition 4.4. If additionally for size-N databases:*

- *the client state in the ORAM and ASR-ORAM schemes has size* $\mathsf{poly}\left(\lambda, \log N\right)$*,*

[12] This holds because $M^{\mathsf{mh}}_{\mathcal{U}}$ was obfuscated using Construction 4 which in turn uses Construction 6.

- $\mathsf{ASR} - \mathsf{ORAM.Access}$ *takes time* $N^\epsilon \cdot \log N \cdot \mathsf{poly}(\lambda)$*, and*
- $\mathsf{Setup}, \mathsf{Access}$ *take time* $\mathsf{poly}(\lambda, \log N)$*,*

then Construction 7 satisfies the N^ϵ-efficiency property of Definition 4.4.

Claims imply theorem. We use the claims to prove Theorem 6.6.

Proof of Theorem 6.6. The existence of OWFs implies the existence of secure unbounded-input PRFs, MACs and IND-CPA secure symmetric encryption schemes, which together with the existence of CRHFs implies, by Claim 6.7, that there exists a multi-hop transcript-simulatable RAM-VBB obfuscator in the circuit-VBB hybrid model, which by Claim 5.6 implies there exists a multi-hop address-simulatable RAM-VBB obfuscator in the circuit-VBB hybrid model. The existence of OWFs and SK-DEPIR implies additionally the existence of a secure ASR-ORAM scheme with a deterministic client by Theorem 3.8 and Claim B.3, and a secure ORAM for initially-empty databases with a deterministic client by Claim A.2, so security follows from Claim 6.8. Efficiency follows from Claim 6.9 using also Theorem 3.8 and Claims A.2 and B.2. □

Proof of claims. We now proceed to prove the claims. Since the proofs are similar to the single-hop case, we only sketch the differences.

Proof sketch for Claim 6.8. Multi-hop correctness follows from the correctness of the underlying primitives (using the perfect correctness of the encryption and ORAM schemes, which guarantees correctness holds even when using pseudorandom coins for encryption and setup), since the address-simulatable RAM-VBB obfuscator is multi-hop correct.

As for security, there are two differences from the single-hop case: (1) we now have two different ORAM schemes; and (2) we now derive the randomness based also on the database. Concretely, we define $\mathcal{H}^b_0, \mathcal{H}^b_1, \mathcal{H}^b_2$ as in the proof of Claim 6.2 (but notice that $\mathcal{H}^b_1$ uses the *multi-hop* simulator Sim for $\mathcal{O}_{\mathsf{addr}}$), and $\mathcal{H}^b_0 \approx \mathcal{H}^b_1 \approx \mathcal{H}^b_2$ follows identically to the proof of Claim 6.2.

Next, we define $\mathcal{H}^b_3$ as in the proof of Claim 6.2, and show that $\mathcal{H}^b_2 \approx \mathcal{H}^b_3$. Towards that end, we condition both hybrids on the following event E: for every M, x, T and D_0, D_1 such that Eval was executed with input (M, x, T) on both an encryption of D_0 and an encryption of D_1, the MHT roots for D_0, D_1 which were given as input to $M^{\mathsf{mh}}_{\mathcal{U}}$ are different. It suffices to prove indistinguishability conditioned on E, because E happens except with negligible probability (except if the adversary was able, through the various executions of Eval, to generate a collision of the CRHF h used in Construction 6). Conditioned on E, randomness is reused in the hybrids in two different executions of Eval, if and only if in both M^D is evaluated on some input x for T steps, in which case the entire execution transcript is identical (because M is deterministic). Therefore, $\mathcal{H}^b_2 \approx \mathcal{H}^b_3$ by the IND-CPA security of the encryption scheme.

Next, we define the hybrid $\mathcal{H}^b_4$ in which we replace the oracle B of Sim with the following oracle B': it replaces all the scratch-tape accesses performed by M with accesses that read address 0 from the scratch tape, and write 0 to address 0 of the scratch tape. (As in the proof of Claim 6.2, during the emulation M is still given the actual values of the scratch tape that it would have read, and the scratch tape is updated according to the actual values M writes to it, namely only the reported memory accesses are replaced.) Then hybrids $\mathcal{H}^b_3$ and $\mathcal{H}^b_4$ are computationally indistinguishable conditioned on E, by a standard hybrid argument over the security of the ORAM scheme for initially-empty databases. Indeed, each evaluation of a RAM machine M initializes a *new* scratch tape ORAM scheme using *fresh* randomness (because we have conditioned on E, namely randomness is reused only if the entire execution transcript is identical). Therefore, indistinguishability follows from the fact

that obliviousness of the ORAM for initially-empty databases is guaranteed even given the access pattern to physical memory during the setup of this ORAM.

Finally, we define the hybrid $\mathcal{H}_5^b$ in which we replace the oracle B' with the following oracle B'': it replaces all the database accesses performed by M with accesses that read address 0 from the database and writes 0 to address 0 of the database. Then hybrids $\mathcal{H}_4^b$ and $\mathcal{H}_5^b$ are computationally indistinguishable by the security of the ASR-ORAM.

Finally, $\mathcal{H}_5^0 \equiv \mathcal{H}_5^1$ since they do not contain any information about $\mathsf{DB}^0, \mathsf{DB}^1$. □

Proof sketch for Claim 6.9. There are three difference from the proof of Claim 6.3: (1) $M_{\mathcal{U}}^{\mathsf{mh}}$'s internal state contains also the client state in an ORAM scheme for initially-empty databases; (2) $M_{\mathcal{U}}^{\mathsf{mh}}$ initializes an ORAM scheme at the beginning of the execution, and emulates the ORAM client (in addition to the ASR-ORAM client) throughout the execution; and (3) the ASR-ORAM has different parameters than the ISR-ORAM used in Claim 6.3. (1) doesn't affect the analysis because the ORAM client state has size $\mathsf{poly}(\lambda, \log T) = \mathsf{poly}(\lambda)$ (because $T \leq 2^\lambda$). (2) doesn't affect the analysis because Setup and Access take $\mathsf{poly}(\lambda, \log T) = \mathsf{poly}(\lambda)$ time by Claim A.2. As for (3), each transition step of M performs 2 accesses into the ASR-ORAM, which take $N^\epsilon \cdot \log N \cdot \mathsf{poly}(\lambda) = N^\epsilon \cdot \mathsf{poly}(\lambda)$ time (because $N \leq 2^\lambda$). Consequently, $\mathsf{Time}\left(M_{\mathcal{U}}^{\mathsf{mh}}, x, D\right) = T \cdot N^\epsilon \cdot \mathsf{poly}(|M|, \lambda)$ using the same arguments as in the proof of Claim 6.3. □

7 Extensions

In this section we describe several extensions and generalizations of our RAM-FHE schemes. Specifically, we describe how to extend the schemes of Section 6 and Section 6.2 to support homomorphic evaluation of RAM machines with long inputs, long outputs, and long descriptions. Specifically, we mean that these quantities are not a priori bounded at encryption time by any polynomial. Moreover, we describe how our schemes can be modified to support variable-length scratch tapes and persistent databases.

7.1 Supporting Variable-Length Scratch Tapes

In our single-hop RAM-FHE construction (Construction 5 on page 36), a bound s on the scratch-tape size is provided *at encryption time*, and correctness is guaranteed only for RAM machines that do not use more that s scratch tape space. However, this can easily be generalized to any scratch tape size, by using separate ISR-ORAMs to access the database and scratch tape. While we could use any ISR-ORAM, notice that we only need a *read-only* ISR-ORAM for the database, and an ISR-ORAM for an *initially-empty* database for the scratch tape. These are easily instantiated as follows: a SK-DEPIR scheme is itself a read-only ISR-ORAM scheme, because the server is stateless, and the client only stores a long-term key (using a *short-term* state between the Query and Decode procedures). An ORAM scheme for initially-empty databases (Appendix A) is an ISR-ORAM if the setup is performed in the first access to the database, since in this case rewinding to the initial state results in an entirely new instance of the ORAM. When the ORAM for initially-empty databases is initialized with size bound 2^λ,[13] we obtain a single-hop RAM-FHE scheme as described in Theorem 6.1, but without the restriction on the scratch tape size.

We note that our multi-hop RAM-FHE scheme (Construction 7 on page 42) can support a scratch tape of any size, since its size is bounded at evaluation time by T (the bound on the number of

[13] We note that Construction 5 cannot support an exponential space bound, because in that case the ISR-ORAM would be instantiated for exponential-sized databases, so $\mathsf{ISR-ORAM.Setup}$ would require exponential time.

transition steps performed by the RAM machine), and a RAM machine running for T steps cannot use more than T scratch tape space.

7.2 Supporting Variable-Length Databases

In our RAM-FHE schemes, the database size was determined at encryption time, and remained fixed throughout iterative executions of the Eval procedure. Notice that this is only an issue in the multi-hop setting, since any writes to the database in the single-hop setting can be emulated by writes to the scratch tape (see related discussion in the second paragraph of Section 5.2).

We now show that Construction 7 can be generalized to support databases whose size *is not* fixed. More specifically, the scheme supports RAM machines that grow or shrink the database, but reveal when the database grows/shrinks. We proceed to explain how this achieved.

In Construction 7, the database is stored in an ASR-ORAM, therefore to support variable-length databases, it suffices that the underling ASR-ORAM will support this. Our ASR-ORAM (Construction 2) has a hierarchical structure which is particularly suitable for this. Specifically, new levels in the hierarchical structure can be added "on the fly" by generating a new SK-DEPIR for the level (namely, by running the $\mathsf{InitLevel}$ procedure of Figure 1 on page 20), and thus the scheme can support additions to the database.

Additionally, memory blocks can be deleted by adding a new "Delete" procedure that on input a block B emulates a `write` operation for B, but when writing B to the top level, adds a "deleted" tag to it. Reshuffle is also adapted to support deletions by removing deleted blocks (i.e., blocks with a "deleted" tag). This is done by assigning such blocks the label 1 in Step 2 of the $\mathsf{ReShuffle}$ procedure of Figure 3 (page 22). Moreover, if in Step 7 of the $\mathsf{ReShuffle}$ procedure, A has size 2^i (i.e., all blocks at addresses $> 2^i$ have label 1) then level $i+1$ is deleted, and A is used to initialize level i by running $\mathsf{InitLevel}\left(i, A\right)$ in Step 8.

We note that adding and deleting levels "on the fly" as described above reveals when an addition or deletion occurs. However, revealing additions is necessary to support a growing database when there is no a-priori bound on its size, or to guarantee the ORAM complexity at each operation depends on the *actual* database size at that point, even if such a bound is known. Similarly, revealing deletions is also necessary to avoid larger than needed overheads in accessing the ORAM.

The resultant scheme has the same efficiency guarantee as in Theorem 6.6, but N is now the *maximal* (instead of initial) database size during the execution of Eval.

7.3 Supporting Variable-Length and Long Inputs and RAM Machine Descriptions

The RAM-FHE constructions of Section 6 and Section 6.2 assume some a-priori known bound (known during encryption) on the input length and description size of RAM machines. In particular, the input and RAM machine description are given as explicit input to the universal RAM machine, and the complexity of $\mathsf{Eval}\left(M, x, 1^T\right)$ is $T \cdot \mathsf{poly}\left(\left|M\right|, \lambda\right)$ (in the single-hop setting) or $T \cdot N^{\epsilon} \cdot \mathsf{poly}\left(\left|M\right|, \lambda\right)$ (in the multi-hop setting). (Here, we also use the fact that $|x| \leq |M|$.) We now describe how to generalize our schemes to support variable input length and description size of RAM machines, where the complexity of $\mathsf{Eval}\left(M, x, 1^T\right)$ would be $\left(T + |M|\right) \cdot \mathsf{poly}\left(\log\left|M\right|, \lambda\right)$ (in the single-hop setting) or $\left(T + |M|\right) \cdot \left(N + |M|\right)^{\epsilon} \cdot \mathsf{poly}\left(\lambda\right)$ (in the multi-hop setting) for any M, x. We also show a more involved multi-hop construction with $\left(T \cdot N^{\epsilon} + |M|\right) \cdot \mathsf{poly}\left(\lambda\right)$ complexity. (Here, we also use the fact that $|M| \leq 2^{\lambda}$.) Of course, for long inputs M cannot take x as an explicit input, so we assume M has RAM access to its input x. It turns out that the multi-hop setting admits a simpler solution, so we consider the multi- and single-hop settings separately.

7.3.1 The Multi-Hop Setting.

The high-level idea of the construction is to first repeatedly run a RAM machine M_{in} that takes as input a bit-string of some fixed $\mathsf{poly}(\lambda)$ length, and concatenates it to the database (by growing the database). By repeatedly applying this program to the input x and description M of the RAM machine, we obtain at the end of the process an encryption of the database, concatenated with the input and description of the RAM machine. Now, one can run a modified universal RAM machine $\mathcal{M}_{\mathcal{U}}^{\mathsf{mh}\prime}$ which reads the needed input bits, and the next command to execute, from the encrypted database. When the execution of the universal RAM machine terminates, we again repeatedly run a RAM machine M_{out} that erases a fixed-length string from the database, thus erasing x, M from the encrypted database. We now elaborate on this construction.

First, since M_{in} grows the database, we use the RAM-FHE scheme that supports variable-length databases (Section 7.2). Fix some $p(\lambda) = \mathsf{poly}(\lambda)$, then M_{in} takes as input $z \in \{0,1\}^{p(\lambda)}$ and an address $\mathsf{addr} \in \left[2^{O(\lambda)}\right]$, has RAM access to a database DB, and writes z to DB starting in address addr. Having M_{in} take an address as input allows us to "leave room" for the original database to grow (such that the resultant evaluation can support variable-length databases). The RAM machine M_{out} takes as input a length $1^{p(\lambda)}$, and an address $\mathsf{addr} \in \left[2^{O(\lambda)}\right]$, and erases $p(\lambda)$ bits from the database, starting at address addr. The RAM machine $M_{\mathcal{U}}^{\mathsf{mh}\prime}$ operates similarly to $M_{\mathcal{U}}^{\mathsf{mh}}$ of Figure 11 (page 63), except for the following: (1) instead of taking x, M as input, $M_{\mathcal{U}}^{\mathsf{mh}\prime}$ takes $|x|, |M|$; (2) to figure out M's next command (in Step 2a in Figure 9 on page 62), $M_{\mathcal{U}}^{\mathsf{mh}\prime}$ uses the ASR-ORAM to read the command from the database (M's internal state specifies the command number, and the location of the command can be computed from the database size and input length); (3) if during the emulation of M's transition step (in Step 2a in Figure 9) M reads an input bit, $M_{\mathcal{U}}^{\mathsf{mh}\prime}$ uses the ASR-ORAM to read the input bit from the database.

Given this description, Eval operates as follows:

- Divides the input and RAM machine description into "chunks" of size $p(\lambda)$. (Eval can choose any polynomial p.)
- Copies the input and RAM machine description into the encrypted database one "chunk" at a time, by repeatedly running M_{in}, where the first execution is with input $\mathsf{addr} = N$, and in the following executions addr increases by $p(\lambda)$ in each execution. Let $\mathsf{addr}_{\mathsf{fin}}$ denote the address in the last execution.
- Runs $M_{\mathcal{U}}^{\mathsf{mh}\prime}$ on the encrypted database obtained through this processes.
- Repeatedly runs M_{out} on the database obtained from $M_{\mathcal{U}}^{\mathsf{mh}\prime}$'s execution, where the first execution is with input $\mathsf{addr} = \mathsf{addr}_{\mathsf{fin}}$, and in the following executions addr decreases by $p(\lambda)$ in each execution.
- Outputs whatever $M_{\mathcal{U}}^{\mathsf{mh}\prime}$ outputs (but uses the database obtained in the previous step as the updated database).

The scheme described above performs $O(|M|)$ operations to copy x, M into the database (each requires at most $(N + |M|)^{\epsilon} \cdot \mathsf{poly}(\lambda)$ operations, because the database now has size $N + O(|M|)$), followed by T steps, each taking $(N + |M|)^{\epsilon} \cdot \mathsf{poly}(\lambda)$ operations. We describe a more efficient construction in the next section.

7.3.2 The Single-Hop Setting.

In the single-hop setting, it is no longer possible to sequentially run multiple short-input RAM machines to write a long input (and RAM program) to memory. Instead, we must directly modify

our constructions to handle a more general model of RAM computation, in which machines have RAM access not only to their database and scratch tapes, but also to their input. The main issue in this setting can be seen already in transcript-simulatable obfuscation: we must prevent a malicious evaluator from performing "mixed input" attacks. In other words, we must guarantee that the entire execution is consistent with a fixed value of x, M.[14]

To explain how this is resolved, recall that our transcript-simulatable obfuscator produces an obfuscated circuit C that takes as input a signed Merkle commitment to the contents of its tapes. Recall that Merkle commitments are succinct and *locally openable* – if one has generated a commitment rt of an n-bit string x, it is possible to give a succinct proof that certifies an arbitrary bit of x.

To handle long inputs, we simply require that a MAC-tagged commitment to the *input tape* is also provided (under some key K_{inp}). To allow the evaluator to generate, on his own, MACed commitments for arbitrary inputs, we include in the obfuscation an additional circuit C_{inp}. This circuit is meant to be invoked $O(n)$ times in a stateful manner to produce a signed commitment. To facilitate this, C_{inp} has an independent MAC key K' to sign its own states. In the first invocation, a Merkle root rt and an input length n is provided (purportedly rt is a commitment to some n-bit string x). C_{inp} then maintains (rt, i, n) as its state, where i is initially 0 and increments by one on each "successful" invocation. In the j'th invocation, the input to C_{inp} is supposed to be the j^{th} bit of x, as well as a proof certifying the correctness of the provided bit. C_{inp} verifies the provided proof, and the invocation is said to be successful if the verification passes. If the invocation is unsuccessful, then C_{inp} aborts (i.e., outputs $\perp$). After the n^{th} execution, it outputs a MAC (under key K_{inp}) of (rt, n).

Upgrading to address-simulatable obfuscation to support arbitrary length inputs poses no additional difficulties once we have transcript-simulatable obfuscation.

The final step, constructing RAM FHE, requires the input access pattern to be oblivious. We can achieve this by applying the address-simulatable obfuscator to a machine that first copies its input to a scratch tape (instantiated with an oblivious RAM), and then accesses the input only via this copy.

We note that the same construction can also be used in the multi-hop setting to obtain better efficiency than the generic construction described in the previous section. Specifically, in this case the runtime of Eval is $(T \cdot N^{\epsilon} + |M|) \cdot \mathsf{poly}(\lambda)$, where the improved efficiency is because the input is copied into the scratch-tape ORAM (instead of the database ASR-ORAM) which is implemented by an ORAM that achieves asymptotically better parameters than ASR-ORAM.

7.4 Supporting Long Outputs

In this section we generalize our RAM-FHE constructions to support evaluation of RAM machines with long outputs. We assume that the length of the output is known at the time of FHE evaluation (otherwise the evaluator would be forced to produce a ciphertext as long as the provided runtime bound).

Machines that produce variable-length output can be modeled in several ways; the simplest way is to allow transition functions to output a bit *and also continue executing* (and therefore potentially output more bits in the future). This model (combined with a sufficiently large scratch space, e.g., as in the extension of Section 7.1) is sufficiently powerful to capture any other "reasonable" model. For example, one might alternatively imagine giving a RAM machine random write access to another "output tape", and defining the output to be the contents of this tape at the time that the machine terminates. The bit-by-bit output model can simulate this model by emulating a k-bit output tape

[14] Specifically, the transcript-simulatable RAM-VBB obfuscator of Section 6 assumes that the entire input and RAM machine description are available, and consistent, throughout the execution (since these are provided as input to the simulator). This issue does not arise in the construction outlined in Section 7.3.1, since M_{in} takes each input/RAM machine description "chunk" as *explicit* input.

(using a designated region of the scratch tape), and outputting each bit stored therein at the end of the computation. Thus we will focus on the "bit-by-bit output" model.

Before delving into any details, we first note that it is *generically* possible to upgrade a RAM-FHE scheme to support machines with k-bit outputs, incurring a multiplicative factor of k in the runtime of Eval. Specifically, one can separately homomorphically evaluate k different RAM machines where the i'th machine outputs the i'th bit of the k-bit output (and discards all other bits).

Non-generically, our construction of transcript-simulatable obfuscation generalizes almost immediately to RAM machines with bit-by-bit output, achieving a runtime of $(T + k) \cdot \mathsf{poly}(\lambda)$ in the single-hop setting, and $(T + k) \cdot N^{\epsilon} \cdot \mathsf{poly}(\lambda)$ in the multi-hop setting. Moving to address-simulatable obfuscation, we must ensure that each output ciphertext uses pseudo-independent randomness. This is only a matter of providing an additional input to the PRF that generates encryption randomness. In particular, we will append i to the PRF's input when encrypting the i^{th} output bit.

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.

[Agr18] Shweta Agrawal. New methods for indistinguishability obfuscation: Bootstrapping and instantiation. *IACR Cryptology ePrint Archive*, 2018:633, 2018.

[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.

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC 2016-A, Proceedings, Part II*, pages 175–204, 2016.

[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+16] 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, 2016.

[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+13] 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, 2013.

[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.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *FOCS 2014, Proceedings*, pages 404–413, 2014.

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

[GK05] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *FOCS 2005, Proceedings*, pages 553–562, 2005.

[GKP+13a] 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, 2013.

[GKP+13b] 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, 2013.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled RAM from one-way functions. In *STOC 2015, Proceedings*, pages 449–458, 2015.

[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.

[GSW13] 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. Springer, 2013.

[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.

A Oblivious RAM for Initially-Empty Databases

In this section, we define the notion of an ORAM for initially-empty databases which, intuitively, is an ORAM scheme for databases that start out empty, and has the feature that obliviousness holds even when the adversary sees the access pattern to physical memory during setup. Formally,

Definition A.1 (ORAM for initially-empty databases)**.** We say that a pair $(\mathsf{Setup}, \mathsf{Access})$ is an ORAM scheme for initially-empty databases if it satisfies the following:

- $\mathsf{Setup}\left(1^\lambda, N\right)$ is a protocol between the client C and server S. C takes as input a security parameter λ, and a bound $N \leq 2^\lambda$ on the database size, and S has no input. The output of the protocol is a client state ck. The execution also updates the server state, which at the end of the execution is denoted by st.
- Access has the same syntax as in Definition 2.4, except that the output value val' might be $\perp$ also in `read` operations, in case addr is an address that has not been previously written to.

- **Correctness:** $\mathsf{Setup}, \mathsf{Access}$ satisfy the correctness property of Definition 2.4.[15]
- **Security.** Every PPT adversary $\mathcal{A}$ wins the following security game with a challenger $\mathcal{C}$ with probability at most $1/2 + \mathsf{negl}(\lambda)$: the game is identical to the security game of Definition 2.4, except that in Step 1 the adversary sends to $\mathcal{C}$ a size bound $N \in \mathbb{N}$ instead of a database DB; and in Step 2 $\mathcal{C}$ runs $\mathsf{Setup}\left(1^\lambda, N\right)$ instead of $\mathsf{Setup}\left(1^\lambda, \mathsf{DB}\right)$.

Next, we construct an ORAM scheme for initially-empty databases by a simple adaptation of the Hierarchical ORAM scheme of [Ost90, GO96]. The hierarchical structure in our construction is initially empty, and levels are added as needed, up to an a-priori determined bound specified during setup. We also support reads for blocks that have not been previously accessed. Specifically, we prove the following:

Claim A.2 (ORAM for initially-empty databases)**.** *Assume OWFs exist. Then there exists an ORAM scheme for initially-empty databases in which* $\mathsf{Access}, \mathsf{Setup}$ *have* $\mathsf{poly}(\lambda)$ *complexity, where* λ *is the security parameter, and the client (server) state has size* $\mathsf{poly}(\lambda)$ ($\mathsf{poly}(\lambda, N)$*, where* N *is the database size).*

Construction 8 (ORAM for initially-empty databases)**.** The scheme uses the following building blocks:

- A PRF F.
- An IND-CPA secure symmetric encryption scheme $(\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt})$.

The scheme consists of the following procedures.

$\mathsf{Setup}(1^\lambda, N)$**:** Recall that λ denotes a security parameter, and N denotes a bound on the supported database size. Proceed as follows.

- Initializing client state and counter: generate a random encryption key by computing $\mathsf{sk} \leftarrow \mathsf{KeyGen}\left(1^\lambda\right)$, pick a random PRF key $K \leftarrow \{0,1\}^\lambda$, and initialize a counter count for the number of operations performed, to 0.
- Initializing the first level: run the $\mathsf{InitLevel}$ procedure of Figure 13 for $i = 0$ to obtain level 0 L_0.
- Output: return the client key $\mathsf{ck} = (\mathsf{sk}, K)$, and the server state $\mathsf{st} = (\mathsf{L}_0, \mathsf{count}, N)$.

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 $\mathsf{ck} = (\mathsf{sk}, K)$, and the server with state $\mathsf{st} = \left(\{\mathsf{L}_i\}_{i \in [\ell]}, \mathsf{count}, N\right)$ for some $\ell \leq \log N$, operates as follows:

1. Resizing the ORAM: Retrieve count and N, and increment $\mathsf{count} := \mathsf{count} + 1$ on the server; if $\mathsf{count} \leq N$ and $\mathsf{count} = 2^\ell$, then run the $\mathsf{InitLevel}$ procedure of Figure 13 on $i = \ell + 1$, which updates the server state to include L_i.[16]
2. **If** $\mathsf{op} = \texttt{read}$ run the Access procedure from [GO96] with $\mathsf{op} = \texttt{read}$ with the following changes:
   (a) Computing bucket index: for each level $0 \leq i \leq \ell$, let $t = \mathsf{count}$ div 2^i. Compute the PRF key for level i in epoch t as $K_i = F(K, (i, t))$. Compute $l_i = F(K_i, \mathsf{addr})$ as the bucket index for level i.

[15] We consider the "correct" value returned for addresses that have not been previously written to be $\perp$.

[16] To achieve a bound on the *worst case* complexity of the scheme, we "spread out" the operations needed to generate level i across accesses number 2^{i-1} to 2^i.

(b) Handling uninitialized blocks: if block addr is not found, set the return value to be "empty", i.e., $\mathsf{val} = \perp$. write the block $(\mathsf{addr}, \perp, \mathsf{valid} = \mathsf{false})$ to the top-level of the ORAM.

(c) Output: return val to the client.

3. **If** $\mathtt{op} = \mathtt{write}$ run the Access procedure from [GO96] with $\mathtt{op} = \mathtt{write}$ where the mapping between addresses and level-i buckets is computed as in Step 2a above.[17]

We prove the following claims regarding Construction 8:

Claim A.3 (ORAM for initially-empty databases security)**.** *Assuming the security of all building blocks, Construction 8 is a secure ORAM scheme for initially-empty databases.*

Claim A.4 (ORAM for initially-empty databases complexity)**.** *When instantiated with a bound N on the database size, and security parameter λ, Construction 8 satisfies the following:*

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

Claims imply Theorem. We now use the claims to prove Claim A.2.

Proof of Claim A.2. We prove that Construction 8 has the required properties. The existence of OWFs implies the existence of a secure ORAM for initially-empty databases by Claim A.3, and thus the security of Construction 8 follows. The stated complexity follows directly from Claim A.4. □

Proofs of Claims. We now prove Claims A.3 and A.4.

Proof Sketch of Claim A.3. Perfect correctness follows from the correctness of the [GO96] scheme, together with the way uninitialized blocks are handled, and because if a bucket overflows then the level contents are stored "in the clear".

We now sketch the security proof, which mirrors that of the hierarchal scheme of [GO96] that it is based on. We note that Construction 8 differs from the hierarchal scheme in two ways: (1) additional levels are added to the ORAM as it grows; and (2) we support accesses for addresses that do not have existing blocks within the ORAM.

We first address obliviousness of growing the ORAM on Access executions. InitLevel is called each time the count of the number of Access operations reaches the size of the current largest level, and is initialized with "empty" blocks. Therefore, the ORAM growth depends only on the number of accesses performed so far, and is independent of the operation type (`read` or `write`) in each access. Thus, the adversary's view in the security game when $b = 0, 1$ remain indistinguishable even when we grow the ORAM.

We next address the obliviousness of blocks when they are accessed for the first time. `write` operations proceed identically to the scheme of [GO96]. However, `read` operations diverge from the hierarchal scheme: while the block is searched for at each level in the assigned bucket, no block associated with the address will be found, which is the reason we create a new "empty" block with that address. This prevents a repeated access to the assigned path on subsequent accesses to that address, and the rest of the analysis is as in [GO96].

We condition our proof on the event that no bucket overflows – this happens with overwhelming probability as shown by Lemma 3.11.

□

[17] To obtain perfect correctness, if a bucket overflows during reshuffle then the contents of the level are stored "in the clear" (i.e., the block encryptions are stored in an array).

Proof of Claim A.4. We first analyze the complexity of Setup. During a Setup execution, the client performs $\mathsf{poly}(\lambda)$ and $O(\log N)$ operations to initialize the secret keys sk, K, and the counter count (respectively). The client performs, for a bucket size of B, $O(\mathsf{B})$ operations to initialize the first level of the ORAM. We set bucket size to $\mathsf{B} = O(\lambda)$, resulting in a total number of operations of $\mathsf{poly}(\lambda, \log N) = \mathsf{poly}(\lambda)$ for Setup.

The complexity of Access is that of [GO96], where the j'th access for $2^{i-1} \leq j < 2^i$ also performs its "share" of initializing level-i. The cost of initializing level-i is $\mathsf{B} \cdot 2^i \cdot \mathsf{poly}(\lambda) = 2^i \cdot \mathsf{poly}(\lambda)$. This is spread-out over 2^{i-1} operations, resulting in an additional $\mathsf{poly}(\lambda)$ number of operations per access. The overall execution of Access is therefor $\mathsf{poly}(\lambda)$. Client state is simply that of [GO96], consisting of encryption and PRF keys (namely, $\mathsf{poly}(\lambda)$). The server state again is simply that of [GO96] (of size $\mathsf{poly}(\lambda, N)$) with an additional counter of size $\log N$, so overall the server state has size $\mathsf{poly}(\lambda, N)$. □

We can derandomize Construction 8 using a similar transformation to that of Construction 9, except that the initial hash h_0 is generated in the first access using the access itself, and the bound on the database size. Using similar argument to the proofs of Claims B.1 and B.3, we can prove the following.

Claim A.5 (ORAM for initially empty databases with a deterministic client)**.** *Let* ORAM *be a randomized ORAM for initially-empty databases, and let* $\mathsf{D-ORAM}$ *be the deterministic ORAM for initially-empty databases obtained by applying Construction 9 to* ORAM*. Then assuming the security of all building blocks,* $\mathsf{D-ORAM}$ *is a secure ORAM for initially-empty databases.*

B Rewindable ORAM with a Deterministic Client

In this section we describe how to transform an ASR or ISR-ORAM scheme with a PPT client, into one with a deterministic client. This will be needed when rewindable ORAM is used to construct RAM-FHE.

Intuitively, the idea is to generate the client randomness using a PRF applied to the entire execution history. However, since the client cannot store the entire history, we instead use a short hash of the history, which we incorporate into the client state.

Construction 9 (Rewindable ORAM with deterministic client)**.** The scheme uses the following building blocks:

- A PRF F.
- A family $\mathcal{H}$ of collision-resistant hash functions.
- A ISR/ASR-ORAM scheme $(\mathsf{ORAM.Setup}, \mathsf{ORAM.Access})$.

The scheme consists of the following procedures.

$\mathsf{Setup}(1^\lambda, \mathsf{DB})$**:** Recall that λ denotes a security parameter, and $\mathsf{DB} \in \{0,1\}^N$. Proceed as follows.

- Initializing the ORAM: run $\mathsf{ORAM.Setup}\left(1^\lambda, \mathsf{DB}\right)$ to obtain client and server states ck, st, respectively.
- Initializing the keys: pick a description of a hash function $H \leftarrow \mathcal{H}$ and generate a random PRF key $K \in_R \{0,1\}^\lambda$.
- Initializing the hash: Compute a hash $h_0 = H(\mathsf{DB})$ of the initial database.
- Output: the server state is st, and the client state is $\mathsf{ck}' = (\mathsf{ck}, H, K, h_0)$.

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 $\mathsf{ck} = (\mathsf{ck}, H, K, h)$, and the server with state st, operate as follows:

- Updating the hash: C generates an updated hash $h' = H(h, \mathsf{op}, \mathsf{addr}, \mathsf{val})$ by hashing his input together with current hash value.
- Generating client randomness: C sets $r = F(K, h')$.
- Emulating ORAM access: C and S emulate the Access protocol of the underlying ORAM, where C emulates the client with input $(\mathsf{op}, \mathsf{addr}, \mathsf{val})$, state ck, and randomness r, and S emulates the server with state st.

Claim B.1 (Deterministic-Client ISR-ORAM Complexity)**.** *Under the assumptions of Theorem 3.5, there exists an ISR-ORAM scheme with a deterministic client where for a database of size N and a security parameter λ, the complexity of* $\mathsf{ORAM.Access}$ *is* $\mathsf{poly}(\lambda)$*, the client state has size* $\mathsf{poly}(\lambda)$*, and the server state has size* $\mathsf{poly}(\lambda, N)$*.*

Proof. We show Construction 9 has the required properties, when instantiated with the ISR-ORAM scheme of Theorem 3.5. The client state in Construction 9 is larger by a $\mathsf{poly}(\lambda)$ additive factor compared to the client state in the scheme of Theorem 9 (since it needs to store the description of the hash function, the PRF key, and the hash). The server state is identical to the underlying ISR-ORAM scheme. As for the complexity of Access, that is only $\mathsf{poly}(\lambda, \log N) = \mathsf{poly}(\lambda)$ larger than that of the underling ISR-ORAM Access algorithm, since the client needs to compute a hash and a PRF image. □

The proof of the following claim follows identically to the proof of Claim B.1, by instantiating Construction 9 with the ASR-ORAM scheme of Theorem 3.8.

Claim B.2 (Deterministic-Client ASR-ORAM Complexity)**.** *Under the assumptions of Theorem 3.8, there exists an ASR-ORAM scheme with a deterministic client where for a database of size N and a security parameter λ:*

- *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* $N^{1+\epsilon} \cdot \mathsf{poly}(\lambda)$*.*

Claim B.3 (Deterministic-Client ISR/ASR-ORAM Security)**.** *Let* ORAM *be a randomized ISR-(ASR-)ORAM, and let* $\mathsf{D-ORAM}$ *be the deterministic ISR- (ASR-)ORAM obtained by applying Construction 9 to* ORAM*. Then assuming the security of all building blocks,* $\mathsf{D-ORAM}$ *is a secure ISR- (ASR-)ORAM.*

Proof. For correctness, by the pseudorandomness of F an honest execution of $\mathsf{D-ORAM}$ (i.e., with no rewinds) is computationally close to an execution in which F is replaced with a random function. Conditioned on the event that there are no H-collisions (which by the collision-resistance of $\mathcal{H}$ happens with overwhelming probability), the execution with a random function is identical to an execution of ORAM with fresh randomness (since in an honest execution history never repeats), for which correctness holds by the correctness of ORAM.

We now argue security, via a sequence of hybrids. We condition all hybrids on the event that throughout the execution there are no collisions of H. (We note that in the rewindable security game, the history might be repeated due to rewinds, but we do not consider these as collisions of H.) This holds with overwhelming probability due to the collision-resistance of H, and is therefore without loss of generality.

$\mathcal{H}_0^b$: Hybrid $\mathcal{H}_0^b$ is the view $(\mathsf{st}_0, (\mathcal{ACC}_i)_b)$ of the adversary $\mathcal{A}$ in the rewindable ORAM security game of Definition 3.2 when the challenger chooses bit b.

$\mathcal{H}_1^b$: In $\mathcal{H}_1^b$, we replace the PRF used to generate the randomness for each $\mathsf{ORAM.Access}$ with a truly random function applied to the entire (unhashed) history. *Hybrids $\mathcal{H}_0^b$ and $\mathcal{H}_1^b$ are computationally indistinguishable by the pseudorandomness of F. This holds because we have conditioned on the event that no H-collisions occur, so throughout the execution, the random function is called multiple times on the same input if and only if F is called multiple times on the same input.*

To conclude the proof, we claim that $\mathcal{H}_1^0$ and $\mathcal{H}_1^1$ are computationally indistinguishable by the security of the underlying ISR/ASR-ORAM scheme. Indeed, the only difference between $\mathcal{H}_1^b$ and the adversary's view in the corresponding execution in ORAM is that in ORAM multiple executions with the same history are executed with fresh randomness, whereas in $\mathcal{H}_1^b$ these executions use the same client randomness. However, we can generate $\mathcal{H}_1^b$ from the adversary's view in the rewindable security game of ORAM as follows: the adversary $\mathcal{A}'$ against $\mathsf{D-ORAM}$ emulates the challenger for the adversary $\mathcal{A}$ against ORAM, where in every iteration of Step 3 of the rewindable security game (Definition 3.2), $\mathcal{A}'$ forwards the query to its own challenger, unless the query is consistent with the execution history generated in a previous iteration, in which case $\mathcal{A}'$ simply provides the access pattern during that execution (which $\mathcal{A}'$ had obtained from its own challenger in a previous iteration) as the challenger's answer to $\mathcal{A}$. □

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 $\mathsf{addr}'_{\mathsf{DB}}, \mathsf{addr}'_{\mathsf{stape}}, \left(\mathsf{addr}'_w, \mathsf{val}'_w\right)$ 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 37). 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 $\sigma' = \mathsf{Tag}\left(K_{\mathsf{MAC}}, \left(b_{\mathsf{fin}}, \mathsf{st}'\right)\right)$, and set $\mathsf{out} = \left(b_{\mathsf{fin}}, \mathsf{st}', \sigma'\right)$.
5. **Output:** return out.

Figure 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 $z = (x, b_{\mathsf{fin}}, \mathsf{st}', \sigma', \mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}, \mathcal{P}_{\mathsf{DB}}, \mathcal{P}_{\mathsf{stape}}, \mathcal{P}_w)$.

Figure 6: Description of the wrapper RAM machine

The RAM machine $M_{\mathsf{sk},K}$

Hard-wired values: an encryption key sk, and a PRF key K.
Internal variables: an encryption c_{st} of an internal state st of M (as outputted by the transition circuit), and a counter count initialized to 0.
RAM access to: a database D (containing an initial database, concatenated with a scratch tape).
Input: input $x \in \{0,1\}^n$ for M, and some additional input $\mathcal{I}$.
Operation: repeat:

1. **Decrypt internal state and values read from database and scratch tape:** Decrypt $\mathsf{st} = \mathsf{Decrypt}(\mathsf{sk}, c_{\mathsf{st}})$.
   - If $\mathsf{count} = 0$ then set $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ to be empty. (This is the first operation of $M_{\mathsf{sk},K}$, no values were previously read from the database and scratch tape.)
   - Otherwise, decrypt $\mathsf{val}_{\mathsf{DB}} = \mathsf{Decrypt}(\mathsf{sk}, c_{\mathsf{DB}})$ and $\mathsf{val}_{\mathsf{stape}} = \mathsf{Decrypt}(\mathsf{sk}, c_{\mathsf{stape}})$, where $c_{\mathsf{DB}}, c_{\mathsf{stape}}$ are the values that were read in Step 3 below.
2. **Emulate the next transition step:**
   (a) Execute the next command of M with internal state st, and using $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ as the values read from the database and the scratch tapes, respectively. (M's internal state st indicates which command should be executed.)
   (b) If M terminated in the current step with output y, then output y.
   (c) Otherwise, the execution step results in an updated internal state st', and 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.
3. **Read from the database and scratch tape:** read addresses $\mathsf{addr}_{\mathsf{DB}}$ and $\mathsf{addr}_{\mathsf{stape}}$ from the database and scratch tape, respectively, and let $c_{\mathsf{DB}}, c_{\mathsf{stape}}$ denote the read values.
4. **Generate encryption randomness:** let $v = (x, \mathsf{st}, \mathsf{count}, \mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}})$. Set $r = F(K, (v, 0))$ and $r_w = F(K, (v, 1))$.
5. **Write to the scratch tape:** encrypt $c_w = \mathsf{Encrypt}(\mathsf{sk}, \mathsf{val}_w; r_w)$, and write c_w to address addr_w of the database.
6. **Update internal state:** encrypt $c'_{\mathsf{st}} = \mathsf{Encrypt}(\mathsf{sk}, \mathsf{st}'; r)$ and set $c_{\mathsf{st}} := c'_{\mathsf{st}}$.
7. $\mathsf{count} := \mathsf{count} + 1$.

Figure 7: RAM machine used by the address-simulatable RAM-VBB obfuscator of Construction 4

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 9 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 10.
3. **Output:** set $r = F(K, (M, T, x))$, encrypt $c = \mathsf{PKE.Encrypt}(\mathsf{pk}, y; r)$ and output c.

Figure 8: RAM machine used in Construction 5

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

Hard-wired values, internal variables, M: as in Figure 8.

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.

Figure 9: 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 8.
Memory accesses $\mathsf{addr}_{\mathsf{DB}}$, $\mathsf{addr}_{\mathsf{stape}}$, $(\mathsf{addr}_w, \mathsf{val}_w)$: as in Figure 9.

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 $\mathsf{val}_{\mathsf{stape}} = \mathsf{val}_{\mathsf{out}}$.

Figure 10: Emulating a database or scratch tape access in M

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

Hard-wired value: a public key pk for a PKE scheme, and a PRF key K.
Internal variables: $\mathsf{ck}, y, \mathsf{fin}, \mathsf{count}$: as in $M_{\mathcal{U}}$ (Figure 8), and additionally a client state $\mathsf{ck}_{\mathsf{stape}}$ in a scratch-tape ORAM (initialized to be empty).
Inputs: M, T, x: as in $M_{\mathcal{U}}$ (Figure 8), as well as an auxiliary input $z \in \{0,1\}^*$.
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.) Let $r_{\mathsf{oram}} = F(K, (M, T, x, z, 0))$ and $r_{\mathsf{Encrypt}} = F(K, (M, T, x, z, 1))$.
2. **Initialize scratch-tape ORAM:** generate an ORAM scheme for initially-empty databases for an empty scratch tape by emulating the Setup protocol between the client and server, by running the ORAM client with randomness r_{oram}, writing to stape the values the client writes to the physical memory on the server. Let $\mathsf{ck}_{\mathsf{stape}}$ denote the client state at the end of the run, and stape denote the server state.
3. **Execute M for T steps:** for $i = 1, \ldots, T$, do:
   - **Emulate a transition step:** execute the procedure from Figure 9 (with $\mathsf{val}_{\mathsf{DB}}, \mathsf{val}_{\mathsf{stape}}$ as the values read from the database and the scratch tape, respectively), except for the following changes:
     - In Step 1, set $\mathsf{addr}_{\mathsf{DB}} = \mathsf{addr}_{\mathsf{stape}} = \mathsf{addr}_{\mathsf{DB},w} = \mathsf{addr}_w = 0$ and $\mathsf{val}_{\mathsf{DB},w} = 0$.
     - In Step 2c, do not change the values of $\mathsf{addr}_{\mathsf{stape}}, \mathsf{addr}_w$. Also, the execution step also outputs a write instruction $(\mathsf{addr}_{\mathsf{DB},w}, \mathsf{val}_{\mathsf{DB},w})$ to the database.
   - **Access DB and scratch tape:** if $\mathsf{count} \le T$ then for $j = 1, \ldots, 4$: execute the procedure from Figure 12.
4. **Output:** encrypt $c = \mathsf{PKE.Encrypt}(\mathsf{pk}, y; r_{\mathsf{Encrypt}})$ and output c.

Figure 11: RAM machine used in Construction 7

Emulating a database or scratch tape access in $M_{\mathcal{U}}^{\mathsf{mh}}$

Hard-wired value, internal variables, j: as in Figure 11.
Memory accesses $\mathsf{addr}_{\mathsf{DB}}$, $\mathsf{addr}_{\mathsf{stape}}$, $(\mathsf{addr}_w, \mathsf{val}_w)$: as in Figure 9, as well as a write instruction $(\mathsf{addr}_{\mathsf{DB},w}, \mathsf{val}_{\mathsf{DB},w})$ to the database.

1. **Determine ORAM client input and server state:** if $j = 1$ (i.e., `read` from database) set $v = (\mathsf{addr}_{\mathsf{DB}}, \perp)$, $\mathsf{tape} = \widetilde{\mathsf{DB}}$, and $\mathsf{ck}' = \mathsf{ck}$. If $j = 2$ (i.e., `read` from scratch tape) set $v = (\mathsf{addr}_{\mathsf{stape}}, \perp)$, $\mathsf{tape} = \mathsf{stape}$ and $\mathsf{ck}' = \mathsf{ck}_{\mathsf{stape}}$. If $j = 3$ (i.e., `write` to database), set $v = (\mathsf{addr}_{\mathsf{DB},w}, \mathsf{val}_{\mathsf{DB},w})$, $\mathsf{tape} = \widetilde{\mathsf{DB}}$ and $\mathsf{ck}' = \mathsf{ck}$. If $j = 4$ (i.e., `write` to scratch tape), set $v = (\mathsf{addr}_w, \mathsf{val}_w)$, $\mathsf{tape} = \mathsf{stape}$ and $\mathsf{ck}' = \mathsf{ck}_{\mathsf{stape}}$.
2. In the following, if $j = 1$ or $j = 4$ then use the ASR-ORAM client, otherwise use the ORAM client.
3. **Initiate ORAM access:** run the 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 $M_{\mathcal{U}}^{\mathsf{mh}}$'s internal state).
4. **Emulate ORAM access:** until the client halts, do:
   - Read the value val written in block q of tape, and perform update on tape.
   - Run the 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 client halts; in either case, this also results in an updated client state).
5. **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 $\mathsf{val}_{\mathsf{stape}} = \mathsf{val}_{\mathsf{out}}$.

Figure 12: Emulating a database or scratch tape access in $M_{\mathcal{U}}^{\mathsf{mh}}$

The InitLevel procedure

Constants: the encryption key sk, and bucket size B.
Input: the index i of the level to initialized.
Operation:

- For every $0 \leq j < 2^i$:
  - Generate a bucket B_j of B "empty" blocks (see remark on physical memory block contents in Section 2.3 for a discussion of empty blocks), and encrypt B by computing $\widetilde{B_j} \leftarrow \mathsf{Encrypt}(\mathsf{sk}, B_j)$.
  - Upload $\widetilde{B_j}$ to the server as bucket j in level i.
- Let $\mathsf{L}_i = \left(\widetilde{B_j}\right)_j$ denote the contents of level i.

Figure 13: The InitLevel procedure used in Construction 8