Non-Interactive Cryptography in the RAM Model of Computation

Daniel Apon* Xiong Fan* Jonathan Katz* Feng-Hao Liu*
Elaine Shi* Hong-Sheng Zhou†

Abstract

Using recently developed techniques for program obfuscation, we show several constructions of non-interactive cryptosystems in the random-access machine (RAM) model of computation that are asymptotically more efficient than what would be obtained using generic RAM-to-circuit compilation. In particular, let T denote the running time and n the memory size of a RAM program. We show that using differing-inputs obfuscation, functional encryption for arbitrary RAM programs can be achieved with evaluation time $\tilde{O}(T + n)$.

Additionally, we provide a number of RAM-model constructions assuming the stronger notion of virtual black-box (VBB) obfuscation. We view these as initial feasibility results and leave instantiating similar protocols from weaker assumptions for future work. Specifically, using VBB obfuscation we show how to construct RAM-model functional encryption with function privacy, fully homomorphic encryption, and stateful, privacy-preserving verifiable computation in the memory-delegation model.

Keywords. random access machine; program obfuscation; functional encryption; fully homomorphic encryption; verifiable computation

1 Introduction

Most cryptographic feasibility results that apply to arbitrary (polynomial-time computable) functions begin by modeling the function of interest as a polynomial-size *circuit*. In contrast, most real-life computations are expressed in terms of programs working on a *von Neumann architecture*, which resembles the *random-access machine* (RAM) model of computation. Working with a circuit-based model of computation can reduce the efficiency of cryptographic constructions relative to their non-cryptographic counterparts. In particular, the running time T of a RAM program can be sublinear in the length n of its memory array D, whereas the circuit corresponding to this program (assuming the function it computes is non-trivial) must have size at least n. Even when $T = \Omega(n)$, generic RAM-to-circuit compilation [41] results in a circuit of size $O(nT)$, a blowup compared to the original running time T. Moreover, RAM programs may run in a different number of steps on different inputs, while circuits must run in the worst-case time on all inputs.

To overcome the drawbacks of using a circuit-based representation, several recent works have investigated cryptography based on alternative models of computation [31, 6, 7, 27, 15, 9, 36]. In particular, the RAM model has attracted considerable attention, and a flurry of recent works have demonstrated the feasibility of constructing SNARKs [6, 7, 9], verifiable computation (without privacy) [6, 7, 15, 9], attribute-based encryption [26], and secure multi-party computation [31, 36] in the RAM model.

*Dept. of Computer Science, University of Maryland. Email: {dapon, xfan, jkatz, fenghao, elaine}@cs.umd.edu
†Dept. of Computer Science, Virginia Commonwealth University. Email: hszhou@vcu.edu

This line of research has left open the question of how to evaluate RAM programs *non-interactively* over encrypted data; this is explicitly mentioned as an open question by Goldwasser et al. [26]. What makes this problem challenging is the inherent tension arising from (1) the need for data to remain encrypted, and (2) the need for the evaluator to securely "decrypt" intermediate memory addresses during the evaluation. Relying on recently developed tools for program obfuscation [20, 14, 4], we show constructions for several cryptographic tasks related to this goal. We summarize our results, as well as our techniques, below.

1.1 Functional Encryption in the RAM Model

In a functional encryption (FE) scheme [10], roughly speaking, secret keys correspond to functions; a user in possession of a ciphertext $\mathsf{ct} = \mathsf{Enc}(x)$ and a secret key sk_f for a function f can compute $f(x)$ but nothing else about x. In most prior work, the time required to compute $f(x)$ depends on the circuit size of f. Here, we explore constructions whose complexity depends instead on the time to compute f in the RAM model. (We focus on indistinguishability-based security, though techniques from De Caro et al. [19] could be used to achieve simulation-based security.)

We identify a RAM program with a memory array D; the program takes an input x, reads/writes data from/to D, and eventually outputs a result. (The function computed by the program is stored in the initial portion of D.) Relying on differing-inputs obfuscation[1] ($\mathsf{di}\mathcal{O}$) [2], we show a construction of functional encryption for RAM programs in which the evaluation time for a RAM program that runs in time T and uses memory of size n is[2] $\tilde{O}(T+n)$ (omitting $\mathsf{poly}(\lambda)$ terms for simplicity). Our scheme imposes no *a priori* bounds on T or n.

Techniques. We describe the high-level intuition of our construction. Given a RAM program specified by its memory array D, we first use standard oblivious-RAM techniques to compile this into a functionally equivalent program in which memory-access patterns are independent of the input. Both the input x and the CPU states will be encrypted using a fully homomorphic encryption (FHE) scheme. This way, the evaluator can homomorphically evaluate every step of the next-instruction circuit, obtaining new (encrypted) CPU states, (encrypted) memory addresses, and (encrypted) values to write to memory. The missing ingredient is that the evaluator must obtain values of the next memory addresses to read and write *in the clear*. Our idea is to obfuscate (using $\mathsf{di}\mathcal{O}$) homomorphic evaluation of the next-instruction function, followed by decryption of the needed memory addresses. The evaluator can use this obfuscated function to generate the memory addresses, one-by-one, without learning any additional information.

Intuitively, the above suffices as long as the evaluator behaves honestly. To prevent a dishonst evaluator from feeding arbitrary inputs to the obfuscated program and learning additional information, the obfuscated next-instruction circuit at time step τ will check that its inputs represent results from an honest execution of the previous $\tau - 1$ steps. To achieve this we use a succinct proof—constructed using the idea of proof-carrying data (PCD) [9] combined with a Merkle-tree construction to verify correctness of memory accesses—to verify that all previous steps of the evaluation were carried out correctly. Getting this high-level idea to work is non-trivial. One obstacle is circular dependence: the obfuscated circuit must include a PCD verifier to check the legitimacy of the inputs before evaluation; on the other hand, the PCD's statement includes the requirement that the obfuscated circuit itself was evaluated correctly in the previous step. To break this circular

[1] Although Garg et al. [21] show the implausibility of differing-inputs obfuscation in the presence of general auxiliary inputs, there is no evidence that their attack on the differing-inputs property applies to the specific circuits considered in our construction. In any case, we discuss further below how our construction can be adapted to be based on indistinguishability obfuscation.

[2] Throughout the paper, $\tilde{O}(f(\cdot))$ means $O(f(\cdot) \cdot \mathsf{polylog}(f(\cdot)))$.

dependence, we have the obfuscated circuit output a signature for the next-memory addresses it outputs. (For technical reasons, this signature must be deterministic.)

Aside from resolving the circular dependence challenge, other technical challenges also arise in our constructions. For example, we rely on the double-encryption and NIZK trick [20, 39, 37] to ensure some form of non-malleability property in our construction. We further borrow ideas from the randomized Functional Encryption construction by Goyal et al. [33] to handle the randomness of ORAM.

Using indistinguishability obfuscation. We can adapt our construction so that it is based on the (presumably) weaker assumption of indistinguishability obfuscation ($i\mathcal{O}$) [20]. Instead of using PCD proofs and Merkle trees, the obfuscated next-instruction circuit now takes the entire previous evaluation trace as input and checks its correctness. The resulting construction, described in Appendix D, has evaluation time $\tilde{O}((T+n)^2)$. Although this does not offer any asymptotic improvement over what can be obtained by converting the RAM to a circuit, it still has the advantage of having *input-specific* running time as defined by Goldwasser et al. for the case of programs specified as Turing machines [26, 27]. (This leaks information, but may be acceptable in some scenarios.) In comparison with the results of Goldwasser et al. [26], our $i\mathcal{O}$-based construction does not rely on non-falsifiable assumptions. We also remark that working in the RAM model is inherently more challenging precisely because of the random memory accesses. It was previously unknown how to construct a functional encryption scheme in the RAM model, even based on non-falsifiable assumptions or in idealized models (without converting the RAM to a circuit).

1.2 Additional Results Using Virtual Black-Box Obfuscation

We view the preceding results as the main contributions of this paper. For completeness, however, we also show how to realize additional tasks in the RAM model based on the stronger notion of virtual black-box (VBB) obfuscation. Although VBB obfuscation is known not to exist for general functions, recent results have demonstrated the feasibility of constructing general-purpose VBB obfuscators in certain idealized models [14, 4] or in the real-world for restricted functions [3]. VBB obfuscation can also be realized using hardware tokens [14, 16, 28]. Our results relying on VBB obfuscation are subject to the same caveats as in those works, and can be viewed as initial feasibility results secure against "generic" or "algebraic-only" attacks. A natural direction of future research is to explore whether similar results can be achieved under weaker assumptions, akin perhaps to our $i\mathcal{O}$- or $\mathsf{di}\mathcal{O}$-based FE-RAM schemes.

Functional encryption with function privacy. Using VBB obfuscation we can add *function privacy* to our functional encryption scheme while maintaining evaluation time $\tilde{O}(T+n)$.

Fully homomorphic encryption for RAM programs. A fully homomorphic encryption (FHE) scheme provides a public evaluation method that enables anyone holding a ciphertext $\mathsf{ct} = \mathsf{Enc}(x)$ to compute a ciphertext $\mathsf{Enc}(f(x))$ for any desired function f. In existing schemes, the complexity of this evaluation procedure depends on the size of the circuit computing f. Here, we give constructions in which the complexity instead depends on the time to compute f on a RAM. Specifically, if f can be computed by a RAM program in T time steps using memory of size n, then evaluation takes time $\tilde{O}(T+n)$ (again omitting terms that depend on the security parameter). As in the case of our functional encryption schemes, our FHE scheme imposes no *a priori* bounds on T or n.

Similar to the Turing machine FHE of [26], this provides the option of FHE with input-specific run-time, but also has the additional property of random access to working memory, which can provide polynomial speed-ups for specific algorithms.

Stateful verifiable computation of RAM programs, with privacy. In a stateful verifiable-computation (VC) scheme, a client outsources some data D to an untrusted server. The client then makes a series of queries to the server. The queries can include *updates* to the data (i.e., inserting, deleting, or modifying entries in the database); other queries require the server to compute some function over the current contents of the data. The challenge is to guarantee the *integrity* of the computation performed by the server. We require *privacy* of the outsourced data as well. (Prior work on verifiable computation in the RAM model [6, 7, 15, 9] does not achieve privacy.) To make the problem interesting, the client should perform asymptotically less work per function evaluation than the time to compute the function on its own, locally. In prior work, efficiency was measured in terms of the circuit size of the function(s) being computed. Here, we focus on the time complexity of the function(s) in the RAM model.

We show that a stateful VC scheme with nearly optimal online performance: for a function that can be implemented by a RAM program in time T using memory of size n, the server's online cost for computing the function is $\widetilde{O}(T)$, and the client's overhead for verification is linear in the input and output lengths only. (Once again, we omit dependence on the security parameter.)

2 Definitions

We present some definitions specific to our work, and refer the reader to Appendix A for general background and various additional building blocks.

2.1 The RAM Model of Computation

We use the following notation to denote parameters associated with a random access machine (RAM): NextIns is the next instruction circuit; n is the maximum number of memory words consumed by the RAM; ℓ is the bit length of each memory word; and $|\mathsf{cpustate}|$ is the bit length of the $\mathsf{cpustate}$.

A random access machine $\mathsf{RAM} := D$ is defined by initial memory array $D \in \{0,1\}^{n\ell}$. Throughout the paper, we assume that a RAM's initial $\mathsf{cpustate}_0 := 0$. We assume that the RAM's program f is stored in the memory array D, and that the NextIns circuit is independent of the program text size. We use the following notation to denote a RAM's execution.

waddr_t	write address at time t		data_t	memory word to write at time t
raddr_t	read address at time t		$\mathsf{fetched}_t$	memory word fetched at time t
x	input of RAM		$\mathsf{cpustate}_t$	CPU state at time t
y	output of RAM		T	RAM's execution time

Based on this notation, our RAM's execution can be described as below. Without loss of generality, we assume $\mathsf{cpustate}_0 := 0$ throughout the paper.

$\mathsf{fetched}_0 = x$
For $t = 1, 2, \ldots$:
  $(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}\right)$
  $\mathsf{fetched}_t := D[\mathsf{raddr}_t]$
  $D[\mathsf{waddr}_t] := \mathsf{data}_t$

For simplicity, we adopt the convention that in the final step $t = T$ of the execution, the NextIns circuit additionally outputs the final output value y. This assumes that the RAM always

runs for exactly T steps on all inputs x. One can also allow input-specific run-time on a per-application basis, under the condition that this information will leak to an adversary in any of our cryptosystems, by modifying the above in the natural way.

We use the notation $y := \mathsf{RAM}(x)$ to denote that executing RAM on input x yields outcome y. We use $\{(\mathsf{waddr}_t, \mathsf{raddr}_t)\}_{1 \leq t \leq T} := \mathsf{addresses}(\mathsf{RAM}, x)$ to represent the memory address sequence (both read and write addresses) accessed during the execution of RAM.

2.2 Oblivious RAM

An Oblivious RAM (ORAM) is a special random-access machine with a special, deterministic next instruction circuit that takes in a random seed rk:

$$(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}, \mathsf{rk}\right)$$

Informally, an adversary who can observe the memory address sequence emitted during the ORAM's execution cannot gain any additional information about the input to the ORAM. Here we use the notation $y := \mathsf{ORAM}[\mathsf{rk}](x)$ denotes the outcome of the ORAM on input x and random seed rk. We use $\{(\mathsf{waddr}_t, \mathsf{raddr}_t)\}_{1 \leq t \leq T_{\mathsf{ORAM}}} := \mathsf{addresses}(\mathsf{ORAM}, x; \mathsf{rk})$ to represent the memory address sequence (both read and write addresses) accessed during the execution of ORAM on input x and random seed rk.

As mentioned earlier, we assume that the initial CPU state of any RAM or ORAM is $\vec{0}$. Therefore, a RAM can be characterized by its initial memory array D. Any (non-oblivious) $\mathsf{RAM} := D$ can be compiled into an equivalent oblivious RAM, denoted $\mathsf{ORAM} := D||\vec{0}$. In particular, the ORAM's initial memory array is just the RAM's initial memory array padded to the desired number of words, and the desired bit-length for each memory word, which we denote with the short-hand $D||\vec{0}$. We stress that the ORAM has different parameters, including memory word size, number of memory words, and CPU state size than the original RAM. Further, note that the ORAM also has a different next instruction circuit than that of the RAM.

Definition 2.1 (Secure ORAM). *We say* $\mathsf{ORAM} := D||\vec{0}$ *is a secure oblivious RAM for* $\mathsf{RAM} := D$, *if the following properties are satisfied:*

- **Correctness.** *For any input x, for any* rk, *it holds that* $\mathsf{ORAM}[\mathsf{rk}](x) = \mathsf{RAM}(x)$.
- **Address simulatability**[3]**.** *The security of* ORAM *states that there exists a simulator* Sim *such that no PPT adversary can distinguish between a real address sequence (obtained by executing* ORAM *on input x over a randomly-chosen seed* rk*) and a simulated address sequence. We assume that both $\mathcal{A}$ and* Sim *know the parameters of the* RAM *and* ORAM *and their next instruction circuits. Note that the $\mathcal{A}$ does not see the random seed* rk *that used by the* ORAM*'s next instruction circuit.*

Real world:

$(D, x) \leftarrow \mathcal{A}$

Pick random $\mathsf{rk} \in \{0,1\}^\lambda$

$\mathsf{ORAM} := D||\vec{0}$

$b \leftarrow \mathcal{A}\left(\mathsf{addresses}(\mathsf{ORAM}, x; \mathsf{rk})\right)$

Simulated world:

$(D, x) \leftarrow \mathcal{A}$

$b \leftarrow \mathcal{A}(\mathsf{Sim}())$

[3]Our security definition does not stipulate the encryption of memory contents since memory contents will be separately encrypted later in our construction.

- **Overhead.** *The new* ORAM *has number of memory words* $n_{\mathsf{ORAM}} = \tilde{O}(n)$*; word bit-length* $\ell_{\mathsf{ORAM}} = \max(\ell, c \log T)$ *for some appropriate constant* $c > 2$*, and where* T *is the original* RAM*'s maximum run-time; bit-length of CPU state* $|\mathsf{cpustate}_{\mathsf{ORAM}}| = O(|\mathsf{cpustate}| + \lambda)$*; and* $|\textsc{NextIns}_{\mathsf{ORAM}}| := O(|\textsc{NextIns}|) \cdot \mathsf{poly}(\lambda)$*. Additionally, the new* ORAM *runs in* $\tilde{O}(T + n)$ *number of time steps if the original RAM runs in* T *number of time steps.*

Lemma 2.2 (Restatement of [25, 35, 42, 44, 23, 18])**.** *Assuming the existence of* PRF*, then for any* $\mathsf{RAM} := D$*, there exists a secure oblivious RAM* $\mathsf{ORAM} := D||\vec{0}$ *with an appropriate next instruction circuit.*

Remark: Prevent correlated randomness of multiple executions of the same ORAM. In our FE schemes, the same ORAM (defined by its initial memory array) will be used to evaluate multiple ciphertexts. In order for the addresses of these multiple executions of the same ORAM to be uncorrelated, we assume that the ORAM shuffles its entire memory prior to starting executing the logic. More specifically, our ORAM does the following:

1. **ORAM setup.** At the start of the its execution, the ORAM first shuffles the entire memory. In other words, the ORAM shuffles the original data array D and builds an ORAM data structure. This takes $\tilde{O}(n)$ time, and explains the added $\tilde{O}(n)$ run-time on top of T in our FE-RAM.
2. **Execution.** The ORAM then executes the logic of the original RAM, reading and writing data from and to the memory oblivious as needed.

Note also that in our construction, the seed rk is generated pseudorandomly by computing $\mathsf{PRF}(K, \mathsf{ct})$; this will ensure that each ciphertext ct leads to a different seed, thus preventing correlated randomness of multiple executions.

2.3 Functional Encryption

Syntax. A functional encryption scheme for RAM programs (FE-RAM) consists of the following algorithms $\mathsf{FE} = \mathsf{FE}.(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.

- **Setup:** $\mathsf{FE.Setup}(1^\lambda)$ is a PPT algorithm that takes as input a security parameter 1^λ and outputs a pair of master public and secret keys $(\mathsf{mpk}, \mathsf{msk})$.
- **Key Generation:** $\mathsf{FE.KeyGen}(\mathsf{msk}, \mathsf{RAM})$ is a PPT algorithm that takes as input the master secret key msk and a RAM program RAM and outputs a corresponding secret key $\mathsf{sk}_{\mathsf{RAM}}$.
- **Encryption:** $\mathsf{FE.Enc}(\mathsf{mpk}, x)$ is a PPT algorithm that takes as input the master public key mpk and a message x and outputs a ciphertext ct.
- **Decryption:** $\mathsf{FE.Dec}(\mathsf{sk}_{\mathsf{RAM}}, \mathsf{ct})$ is a deterministic algorithm that takes as input the secret key $\mathsf{sk}_{\mathsf{RAM}}$ and a ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{mpk}, x)$ and outputs $\mathsf{RAM}(x)$.

Definition 2.3 (Correctness)**.** *A functional encryption scheme* FE *is correct if for every* $\lambda \in \mathbb{N}$*,* RAM*,* x*,*

$$\Pr\left[\begin{array}{l}(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda); \\ \mathsf{FE.Dec}(\mathsf{FE.KeyGen}(\mathsf{msk}, \mathsf{RAM}), \mathsf{FE.Enc}(\mathsf{mpk}, x)) \neq \mathsf{RAM}(x)\end{array}\right] = \mathsf{negl}(\lambda)$$

where the probability is taken over the coins of $\mathsf{FE.Setup}$*,* $\mathsf{FE.KeyGen}$*, and* $\mathsf{FE.Enc}$*.*

We consider both indistinguishability- and simulation-based notions of security for RAM-model functional encryption, following [10, 38]. See Appendix B for a full treatment. Here, we state the definition under which we prove security directly.

Definition 2.4 (Selective Indistinguishability, Multiple-Key). *Let* FE *be a functional encryption scheme for RAM programs computing a functionality class* $\mathcal{F}$*. For every* PPT *stateful adversary* $\mathcal{A}$*, consider the following experiment.*

$\mathbf{Expt}_{\mathcal{A}}^{\mathsf{selective}}(1^\lambda)$
1: $(x_0, x_1) \leftarrow \mathcal{A}(1^\lambda)$;
2: $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$;
3: $b \leftarrow \{0, 1\}$;
4: $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x_b)$;
5: $b' \leftarrow \mathcal{A}^{\mathsf{FE.KeyGen}(\mathsf{msk}, \cdot)}(\mathsf{mpk}, \mathsf{ct})$;

Define an adversary to be non-trivial *if* $\mathsf{RAM}(x_0) = \mathsf{RAM}(x_1)$*, for every query* RAM *made to the* KeyGen *oracle. We say that an FE-RAM scheme is* selectively IND-secure *if for all* PPT *stateful* non-trivial *adversaries* $\mathcal{A}$*, it holds that* $\Pr[b' = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$ *in the above experiment.*

3 RAM-Model Functional Encryption from Differing-Inputs Obfuscation

3.1 Intuition

As mentioned earlier in the introduction, the main idea is to 1) homomorphically encrypt the input and the CPU state; and 2) create obfuscated next-instruction circuits that will emit next memory addresses to read and write. When a token is being generated for a RAM, the RAM is first converted into an ORAM. The ORAM's memory contents are initially not encrypted (since function privacy is not our goal here); however, contents newly written to memory will be homomorphically encrypted. As a result, data fetched from memory can be a cleartext or ciphertext, depending on whether the memory location has been written during the evaluation.

One key challenge is to secure against a malicious evaluator. For this reason, every obfuscated next instruction circuit must only yield output if the evaluator is behaving correctly. We therefore rely on Proof-Carrying-Data (PCD) [9] to ensure that all previous steps of the encrypted RAM evaluation are done correctly. Unfortunately, if done naively, a *circularity* issue arises:

- We must check a PCD proof inside the obfuscated next-instruction circuit before releasing any output; since otherwise an adversary can supply malicious inputs to the obfuscated next-instruction circuit and learn additional information;
- The PCD proof must verify that the obfuscated next instruction circuit is evaluated correctly.

In order for us to fix the PCD verifier circuit to embed in the obfuscated next-instruction circuit, we must fix the PCD predicate first, which requires fixing the obfuscated next-instruction circuit first.

Ideas. To fix this problem, our idea is to have the obfuscated next instruction circuit additionally output a signature for the $(\mathsf{raddr}_\tau, \mathsf{waddr}_\tau)$ addresses it outputs. Therefore, in the PCD predicate, we simply need to verify that this signature is correct, instead of having the entire obfuscated next-instruction circuit in the PCD predicate.

However, to leverage the properties of $\mathsf{di}\mathcal{O}$ in our proof, this signature output by the $\mathsf{di}\mathcal{O}$ must be deterministic, such that at time step τ of evaluating the ciphertext ct, there is one unique proof for a tuple $(\mathsf{ct}, \tau, \mathsf{raddr}_\tau, \mathsf{waddr}_\tau)$. To construct this signature, we borrow ideas from the NIZK construction of Sahai and Waters [40]. Although this scheme has only non-adaptive soundness, this suffices for our setting because the space of possible (valid) statements has polynomial size.

3.2 Construction

Notational convention. We explain our notational conventions below.

$\boxed{\mathsf{var}}$	double-encrypted ciphertext based on cFHE, under public keys hpk and hpk' respectively
$\overline{\mathsf{var}}$	This variable (in particular $\mathsf{fetched}_t$) is sometimes double-encrypted and sometimes in cleartext. Memory contents fetched during the RAM's evaluation are in cleartext if this is the first time they are accessed; otherwise, they are double-encrypted under hpk and hpk'.
$\{\mathsf{var}\}$ or $\{\overline{\mathsf{var}}\}$ or $\{\boxed{\mathsf{var}}\}$	We use $\{\}$ to denote either that the variable has a Merkle proof or a PCD proof vouching for its correctness.

We introduce the notation $\mathsf{cFHE.Eval}_{hpk,hpk'}$ as a short-hand to express simultaneously evaluating two copies of the FHE ciphertexts, encrypted under hpk and hpk' respectively. Concretely, we use $\boxed{w} = \mathsf{cFHE.Eval}_{hpk,hpk'}(g(\boxed{v}))$ to denote the homomorphic evaluation of function $g(\cdot)$ on double-encrypted ciphertext $\boxed{v} = (c_v, c'_v)$ to obtain a new double-encrypted ciphertext $\boxed{w} = (c_w, c'_w)$, where $c_v = \mathsf{cFHE.Enc}(hpk, v)$ and $c'_v = \mathsf{cFHE.Enc}(hpk', v))$, and $c_w = \mathsf{cFHE.Eval}(hpk, g(\cdot), c_v)$ and $c'_w = \mathsf{cFHE.Eval}(hpk', g(\cdot), c'_v)$.

Detailed construction. We now describe our $\mathsf{di}\mathcal{O}$-based FE-RAM construction. Besides the differing-inputs obfuscation $\mathsf{di}\mathcal{O}$, we use an FHE scheme $\mathsf{cFHE}.(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, and a *simulation sound* NIZK scheme $\mathsf{NIZK}.(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$. Further, we use a Proof-Carrying-Data system [9, 17] and collision-resistant Merkle hash tree construction.

Setup. On input 1^λ, compute $(hpk, hsk) \leftarrow \mathsf{cFHE.Gen}(1^\lambda)$, $(hpk', hsk') \leftarrow \mathsf{cFHE.Gen}(1^\lambda)$, $crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$, and $(\mathsf{rs}, \mathsf{vrs}) \leftarrow \mathsf{PCD}.\mathbb{G}(1^\lambda)$. Set public parameter $\mathsf{mpk} := (hpk, hpk', crs, \mathsf{rs}, \mathsf{vrs})$, and master secret key $\mathsf{msk} := hsk$.

Key Generation. Convert the RAM into an ORAM, and denote $\mathsf{ORAM} := D$. Compute the initial Merkle digest denoted digest_0 of ORAM i.e.,

$$\mathsf{digest}_0 := \mathsf{MerkleDigest}(D)$$

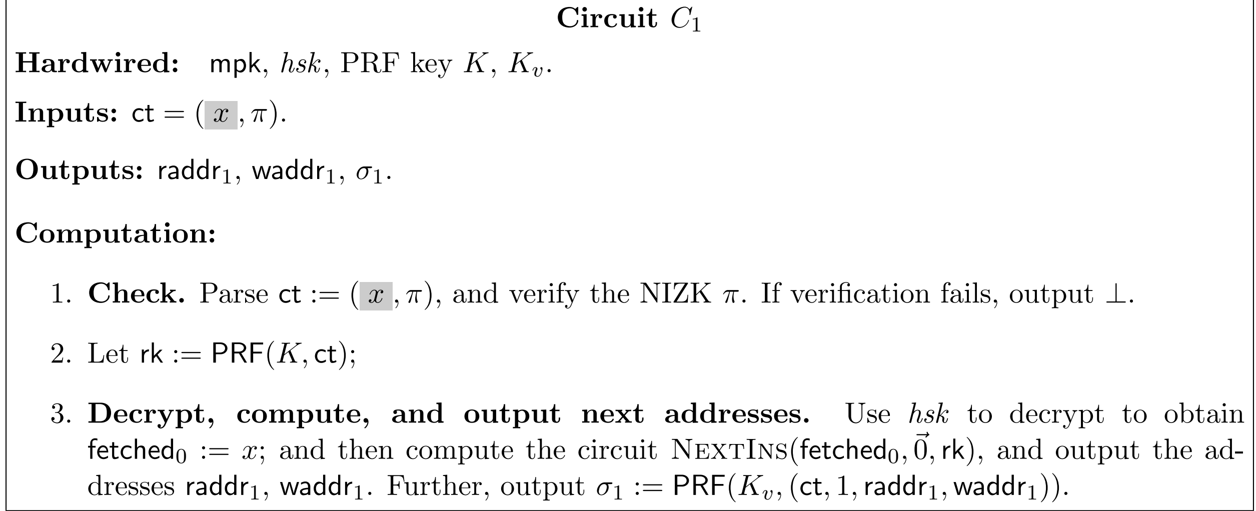
Circuit V

Hardwired: PRF key K_v.

Inputs: ct, τ, raddr_τ, waddr_τ, σ_τ.

Test if $\mathsf{OWF}(\sigma_\tau) = \mathsf{OWF}(\mathsf{PRF}(K_v, (\mathsf{ct}, \tau, \mathsf{raddr}_\tau, \mathsf{waddr}_\tau)))$ where OWF is a one-way function. Output accept if true, reject if false.

Figure 1: Circuit V to be obfuscated using $i\mathcal{O}$. This verifier circuit will be used as part of the predicate of the PCD system.



Circuit C_1

Hardwired: mpk, hsk, PRF key K, K_v.

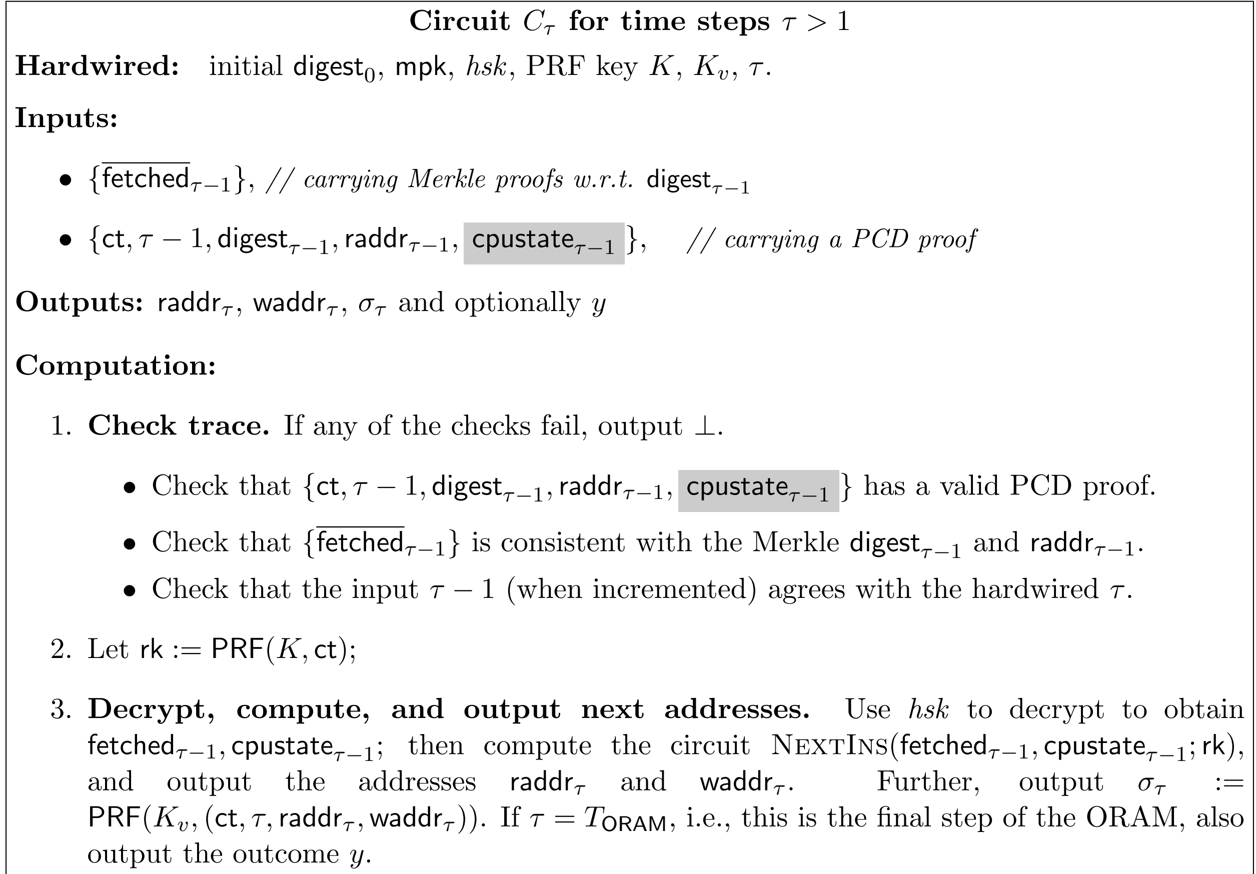
Inputs: $\mathsf{ct} = (\boxed{x}, \pi)$.

Outputs: raddr_1, waddr_1, σ_1.

Computation:

1. **Check.** Parse $\mathsf{ct} := (\boxed{x}, \pi)$, and verify the NIZK π. If verification fails, output $\perp$.
2. Let $\mathsf{rk} := \mathsf{PRF}(K, \mathsf{ct})$;
3. **Decrypt, compute, and output next addresses.** Use hsk to decrypt to obtain $\mathsf{fetched}_0 := x$; and then compute the circuit $\textsc{NextIns}(\mathsf{fetched}_0, \vec{0}, \mathsf{rk})$, and output the addresses raddr_1, waddr_1. Further, output $\sigma_1 := \mathsf{PRF}(K_v, (\mathsf{ct}, 1, \mathsf{raddr}_1, \mathsf{waddr}_1))$.

Figure 2: Circuit C_1 to be obfuscated using $\mathsf{di}\mathcal{O}$.



Circuit C_τ for time steps $\tau > 1$

Hardwired: initial digest_0, mpk, hsk, PRF key K, K_v, τ.

Inputs:

- $\{\overline{\mathsf{fetched}}_{\tau-1}\}$, *// carrying Merkle proofs w.r.t.* $\mathsf{digest}_{\tau-1}$
- $\{\mathsf{ct}, \tau - 1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}\}$, *// carrying a PCD proof*

Outputs: raddr_τ, waddr_τ, σ_τ and optionally y

Computation:

1. **Check trace.** If any of the checks fail, output $\perp$.
   - Check that $\{\mathsf{ct}, \tau - 1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}\}$ has a valid PCD proof.
   - Check that $\{\overline{\mathsf{fetched}}_{\tau-1}\}$ is consistent with the Merkle $\mathsf{digest}_{\tau-1}$ and $\mathsf{raddr}_{\tau-1}$.
   - Check that the input $\tau - 1$ (when incremented) agrees with the hardwired τ.
2. Let $\mathsf{rk} := \mathsf{PRF}(K, \mathsf{ct})$;
3. **Decrypt, compute, and output next addresses.** Use hsk to decrypt to obtain $\mathsf{fetched}_{\tau-1}, \mathsf{cpustate}_{\tau-1}$; then compute the circuit $\textsc{NextIns}(\mathsf{fetched}_{\tau-1}, \mathsf{cpustate}_{\tau-1}; \mathsf{rk})$, and output the addresses raddr_τ and waddr_τ. Further, output $\sigma_\tau := \mathsf{PRF}(K_v, (\mathsf{ct}, \tau, \mathsf{raddr}_\tau, \mathsf{waddr}_\tau))$. If $\tau = T_{\mathsf{ORAM}}$, i.e., this is the final step of the ORAM, also output the outcome y.

Figure 3: Circuit C_τ to be obfuscated using $\mathsf{di}\mathcal{O}$ for $\tau > 1$. Note that the PCD verifier algorithm for each time step may be different, therefore, the circuit for each time step is different.

Sample random PRF key $K \in \{0,1\}^\lambda$ to embed in the obfuscated next instruction circuits. Let $\boxed{K} := \big(\mathsf{cFHE.Enc}(hpk, K), \mathsf{cFHE.Enc}(hpk', K)\big)$.

Sample random PRF key $K_v \in \{0,1\}^\lambda$. Include $i\mathcal{O}(V)$ in the token, where V is defined as in Figure 1.

Generate $T_{\mathsf{ORAM}} = \tilde{O}(T)$ obfuscated circuits: 1) An obfuscated circuit $\mathsf{di}\mathcal{O}(C_1)$for the first step of evaluation, this circuit is of size $\tilde{O}(T)$ as shown in Figure 2; and 2) an obfuscated circuit $\mathsf{di}\mathcal{O}(C_\tau)$ for each time step $\tau > 1$ as described in Figure 3.

The token $\mathsf{sk}_{\mathsf{ORAM}} := (\mathsf{digest}_0, \boxed{K}, i\mathcal{O}(V), \mathsf{di}\mathcal{O}(C_1), \mathsf{di}\mathcal{O}(C_2), \ldots, \mathsf{di}\mathcal{O}(C_{T_{\mathsf{ORAM}}}))$.

Encryption. Upon inputting the public parameter mpk and a message x, pick random $\rho, \rho' \in \{0,1\}^\lambda$, and compute $c = \mathsf{cFHE.Enc}(hpk, x; \rho)$, $c' = \mathsf{cFHE.Enc}(hpk', x; \rho')$, then compute a NIZK (denoted π) for the following statement parameterized by (c, c'):

$$\exists x, \rho, \rho' \text{ s.t. } (c = \mathsf{cFHE.Enc}(hpk, x; \rho)) \wedge \big(c' = \mathsf{cFHE.Enc}(hpk', x; \rho')\big)$$

The ciphertext $\mathsf{ct} := (\boxed{x}, \pi)$, where $\boxed{x} := (c, c')$.

Decryption. Parse the token $\mathsf{sk}_{\mathsf{ORAM}} := (\mathsf{digest}_0, \boxed{K}, i\mathcal{O}(V), \mathsf{di}\mathcal{O}(C_1), \mathsf{di}\mathcal{O}(C_2), \ldots, \mathsf{di}\mathcal{O}(C_{T_{\mathsf{ORAM}}}))$. Initialize $\overline{D} = D$. Compute $\boxed{\mathsf{rk}} := \mathsf{cFHE.Eval}_{hpk,hpk'}\Big(\mathsf{PRF}(\boxed{K}, \mathsf{ct})\Big)$[4].

Next, for $\tau \in [T_{\mathsf{ORAM}}]$,

- **Perform homomorphic evaluation.**
  If $\tau = 1$, use homomorphic evaluation to obtain:

  $$\Big(\boxed{\mathsf{data}_1}, \boxed{\mathsf{cpustate}_1}\Big) := \mathsf{cFHE.Eval}_{hpk,hpk'}(\textsc{NextIns}(\boxed{x}, \vec{0}; \boxed{\mathsf{rk}}))$$

  Else if $\tau > 1$, use the homomorphic evaluation to obtain:

  $$\Big(\boxed{\mathsf{data}_\tau}, \boxed{\mathsf{cpustate}_\tau}\Big) := \mathsf{cFHE.Eval}_{hpk,hpk'}(\textsc{NextIns}(\overline{\mathsf{fetched}}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}; \boxed{\mathsf{rk}}))$$

  Here we *ignore* the encrypted addresses output by the homomorphic evaluations of the $\textsc{NextIns}$.

- **Use $\mathsf{di}\mathcal{O}$ to evaluate next addresses.** If $\tau = 1$, compute

  $$(\mathsf{waddr}_1, \mathsf{raddr}_1, \sigma_1) := \mathsf{di}\mathcal{O}(C_1)(\mathsf{ct})$$

  Else if $\tau > 1$, compute

  $$(\mathsf{waddr}_\tau, \mathsf{raddr}_\tau, \sigma_\tau) := \mathsf{di}\mathcal{O}(C_\tau)(\{\overline{\mathsf{fetched}}_{\tau-1}\}, \{\mathsf{ct}, \tau, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}\})$$

  If this is the final step of evaluation, an output y is additionally output from $\mathsf{di}\mathcal{O}(C_\tau)$.

- **Perform memory read and write.** The evaluator maintains a Merkle-tree authenticated data structure to efficiently compute the Merkle proofs for any memory location, and to efficiently update the Merkle digest upon memory writes. At this moment, the

[4] For technical reasons in the proof, here we assume that PRF is the circuit that first checks whether input $\boxed{K}$ is a punctured key (which can be indicated by adding an indicator bit to the key), and if not, use real PRF evaluation algorithm; if so, use the punctured PRF evaluation algorithm.

evaluator would update its Merkle-tree authenticated data structure to reflect the updated $\boxed{\mathsf{data}_\tau}$, the updated $\boxed{\mathsf{cpustate}_\tau}$, as well as the new time τ. Suppose that the new Merkle digest is digest_τ.

Perform memory read and write:

$$\overline{\mathsf{fetched}}_\tau := \overline{D}[\mathsf{raddr}_\tau], \qquad \overline{D}[\mathsf{waddr}_\tau] := \boxed{\mathsf{data}_\tau}$$

Construct the Merkle proofs for $\overline{\mathsf{fetched}}_\tau$ with respect to digest_τ. Let $\{\overline{\mathsf{fetched}}_\tau\}$, denote the version that has been attached with its Merkle proof w.r.t digest_τ.

- **Compute a PCD proof** for $\{\mathsf{ct}, \tau, \mathsf{digest}_\tau, \mathsf{raddr}_\tau, \boxed{\mathsf{cpustate}_\tau}\}$.

3.2.1 PCD Proof Computation Details

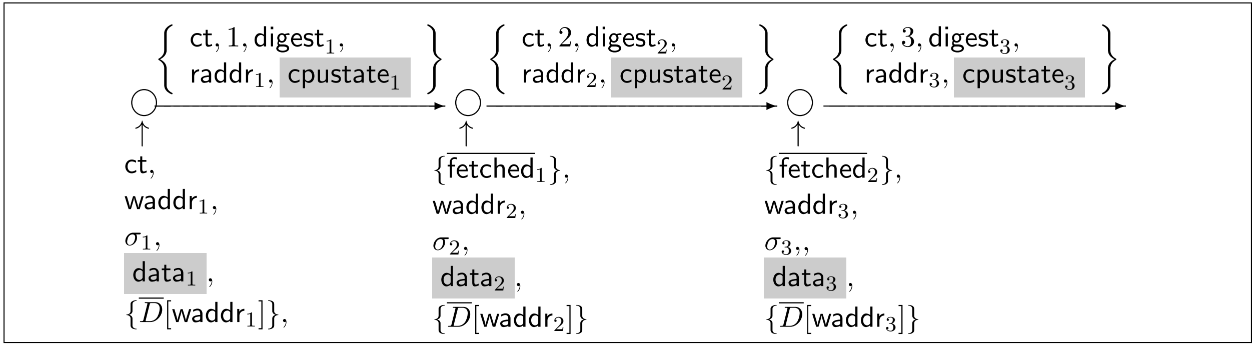


Figure 4: Our Path PCD system.

Effectively we have a Path PCD system as depicted in Figure 4. In every time step of the evaluation, a succinct proof is output vouching for the correctness of the terms $\{\mathsf{ct}, \tau, \mathsf{digest}_\tau, \mathsf{raddr}_\tau, \boxed{\mathsf{cpustate}_\tau}\}$ which will serve as input to the next step.

Statement for $\tau > 1$. At every node $\tau > 1$, the statement being verified is the conjunction of the following:

- The input has a valid PCD proof.
- The output ct agrees with the input ct.
- $i\mathcal{O}(V)(\mathsf{ct}, \tau, \mathsf{raddr}_\tau, \mathsf{waddr}_\tau, \sigma_\tau) = 1$.
- $\{\overline{\mathsf{fetched}}_{\tau-1}\}$ has a Merkle proof consistent with $\mathsf{digest}_{\tau-1}$ and $\mathsf{raddr}_{\tau-1}$.
- Values $\boxed{\mathsf{cpustate}_\tau}$ and $\boxed{\mathsf{data}_\tau}$ are correct homomorphic evaluations using the honest evaluator's algorithm based on $\overline{\mathsf{fetched}}_{\tau-1}$, $\boxed{\mathsf{cpustate}_{\tau-1}}$, and $\boxed{\mathsf{rk}} := \mathsf{cFHE.Eval}_{hpk,hpk'}(\boxed{K}, \mathsf{ct})$.
- The new digest digest_τ is updated correctly from $\mathsf{digest}_{\tau-1}$, based on $\{\overline{D}[\mathsf{waddr}_\tau]\}$, and the new value $\boxed{\mathsf{data}_\tau}$.

Statement for $\tau = 1$. At node $\tau = 1$, the statement verified is the conjunction of the following:

- The output ct agrees with the local input ct.
- $i\mathcal{O}(V)(\mathsf{ct}, 1, \mathsf{raddr}_1, \mathsf{waddr}_1, \sigma_1) = 1$.
- Values $\mathsf{cpustate}_1$ and data_1 are correct homomorphic evaluations using the honest evaluator's algorithm based on ct, $\vec{0}$, and $\mathsf{rk} := \mathsf{cFHE.Eval}_{hpk,hpk'}(K, \mathsf{ct})$.
- The new digest digest_1 is updated correctly from digest_0, based on $\{\overline{D}[\mathsf{waddr}_1]\}$, and the new value data_1.

Theorem 3.1. *Assuming that* $\mathsf{di}\mathcal{O}$ *is a secure differing-inputs obfuscator,* $i\mathcal{O}$ *is a secure indistinguishability obfuscator,* cFHE *is a FHE scheme for circuits with perfect correctness and semantic security,* NIZK *is a simulation sound NIZK scheme, the PCD system is a proof of knowledge, the Merkle tree construction is collision-resistant,* PRF *is a correct and secure puncturable PRF,* ORAM *is secure as in Definition 2.1 and* OWF *is a one-way function, then the above FE-RAM construction is selectively IND-secure as in Definition 2.4.*

The full proofs are deferred to Appendix C.

Lifting security. By a standard argument of complexity leveraging, we can achieve the (full, as opposed to selective) indistinguishability security from the selective security at a cost of stronger complexity assumptions. Then we can achieve a simulation-based security using the trapdoor circuit technique from the work of De Caro et al. [19], who showed how to construct a (selective/full) simulation secure FE from a (selective/full) indistinguishability secure one. Also, the construction supports multiple key queries. Thus, we are able to achieve the following corollary:

Corollary 3.2 (Theorem 3.1 + (complexity leveraging) + [19])**.** *Assume that* $\mathsf{di}\mathcal{O}$ *is a secure differing-inputs obfuscator,* $i\mathcal{O}$ *is a secure indistinguishablility obfuscator,* cFHE *is FHE scheme for all circuits with perfect correctness and semantic security,* NIZK *is a simulation sound NIZK scheme, the PCD system is a sound proof-of-knowledge system, the Merkle tree construction is collision-resistant,* PRF *is a correct and secure puncturable PRF,* ORAM *is secure as in Definition 2.1, and* OWF *is a one-way function. In addition, assume that these primitives remain secure when all polynomial-sized adversaries have sub-exponentially small advantages. Then the above FE-RAM can be made fully SIM-secure as defined in Definition B.1.*

Cost. Our IND-secure construction achieves $\mathsf{poly}(\lambda)$ ciphertext size and $\tilde{O}(n+T)\mathsf{poly}(\lambda)$ evaluation time. For the simulation-secure setting, our cost is preserved (same as the IND-secure setting), for a scheme secure under a single key query. To support q key queries, the ciphertext size blows up by a factor of q due to the use of De Caro et al. 's compiler [19]. Again, recall that it has been shown that in the standard model, it is impossible to achieve fully SIM-secure FE with succinct ciphertexts [19]. In contrast, our IND-secure scheme can support unbounded polynomially many key queries without blowups in ciphertext size.

In the above, we assume that the underlying $\mathsf{di}\mathcal{O}$ scheme has linear blowup. We note that for the weaker primitive of indistinguishable obfuscation, candidate $i\mathcal{O}$ schemes with linear blowup have been proposed: Gordon et al. observe that for the construction of [20], the size of the obfuscated circuit has only linear blowup, i.e. $|i\mathcal{O}(C)| = |C| \cdot \mathsf{poly}(\lambda)$. Therefore, if we conjecture that the $i\mathcal{O}$ scheme is also a secure $\mathsf{di}\mathcal{O}$, then it will also be linear blowup. Please also see Remark A.6 for more explanations.

Acknowledgments

This research was funded by NSF under grant number CNS-1314857, by a Google Faculty Research Award, and by the US Army Research Laboratory and the UK Ministry of Defence under Agreement Number W911NF-06-3-0001. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] Shweta Agrawal, Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption: New perspectives and lower bounds. In *CRYPTO*, 2013. `http://eprint.iacr.org/2012/468`.

[2] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. Cryptology ePrint Archive, Report 2013/689, 2013. `http://eprint.iacr.org/`.

[3] Boaz Barak, Nir Bitansky, Ran Canetti, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Obfuscation for evasive functions. In *TCC*, pages 26–51, 2014.

[4] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. In *EUROCRYPT*, 2014. http://eprint.iacr.org/2013/631.

[5] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[6] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, ITCS '13, pages 401–414, 2013.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, pages 90–108, 2013.

[8] Nir Bitansky and Ran Canetti. On strong simulation and composable point obfuscation. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 520–537. Springer, August 2010.

[9] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. In *STOC*, 2013.

[10] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *TCC 2011: 8th Theory of Cryptography Conference*, volume 6597 of *Lecture Notes in Computer Science*, pages 253–273. Springer, 2011.

[11] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, pages 280–300, 2013.

[12] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *TCC*, pages 52–73, 2014.

[13] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Public Key Cryptography*, 2014.

[14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *TCC*, pages 1–25, 2014.

[15] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 341–357, 2013.

[16] Ran Canetti and Vinod Vaikuntanathan. Obfuscating branching programs using black-box pseudo-free groups. Cryptology ePrint Archive, Report 2013/500, 2013. http://eprint.iacr.org/.

[17] Alessandro Chiesa and Eran Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, pages 310–331, 2010.

[18] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. *CoRR*, abs/1307.3699, 2013.

[19] Angelo De Caro, Vincenzo Iovino, Abhishek Jain, Adam O'Neill, Omer Paneth, and Giuseppe Persiano. On the achievability of simulation-based security for functional encryption. In *CRYPTO*, 2013.

[20] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *FOCS*, 2013. http://eprint.iacr.org/2013/451.

[21] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. *IACR Cryptology ePrint Archive*, 2013:860, 2013.

[22] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, August 2010.

[23] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, pages 1–18, 2013.

[24] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, August 1986.

[25] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[26] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. How to run turing machines on encrypted data. In *CRYPTO*, 2013.

[27] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Succinct functional encryption and applications: Reusable garbled circuits and beyond. In *STOC*, 2013.

[28] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. One-time programs. In *CRYPTO*, pages 39–56, 2008.

[29] Shafi Goldwasser and Guy N. Rothblum. On best-possible obfuscation. In *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2007.

[30] Sergey Gorbunov, Vinod Vaikuntanathan, and Hoeteck Wee. Functional encryption with bounded collusions via multi-party computation. In *CRYPTO*, pages 162–179, 2012.

[31] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 513–524, 2012.

[32] S. Dov Gordon, Jonathan Katz, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. Cryptology ePrint Archive, Report 2013/774, 2013. `http://eprint.iacr.org/`.

[33] Vipul Goyal, Abhishek Jain, Venkata Koppula, and Amit Sahai. Functional encryption for randomized functionalities. Cryptology ePrint Archive, Report 2013/729, 2013. `http://eprint.iacr.org/`.

[34] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *ACM CCS*, 2013.

[35] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[36] Steve Lu and Rafail Ostrovsky. How to garble ram programs. In *EUROCRYPT*, pages 719–734, 2013.

[37] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *In Proc. of the 22nd STOC*, pages 427–437, 1995.

[38] Adam O'Neill. Definitional issues in functional encryption. Cryptology ePrint Archive, Report 2010/556, 2010. `http://eprint.iacr.org/`.

[39] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *40th Annual Symposium on Foundations of Computer Science*, pages 543–553. IEEE Computer Society Press, October 1999.

[40] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: Deniable encryption, and more. Cryptology ePrint Archive, Report 2013/454, 2013. `http://eprint.iacr.org/`.

[41] John E. Savage. Models of computation: Exploring the power of computing. Addison-Wesley Longman Publishing Co., Inc., 1997.

[42] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[43] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *NDSS*, 2012.

[44] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *ACM CCS*, 2013.

A General Background

A.1 Puncturable Pseudorandom Functions

Puncturable family of PRFs are a special case of constrained PRFs [11, 13, 34], where the PRF is defined on all input strings except for a set of size polynomial in the security parameter. Below we recall their definition, as given by [40].

Syntax. A puncturable family of PRFs is defined by a tuple of algorithms $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture})$ and a pair of polynomials $n()$ and $m()$:

- **Key Generation** $\mathsf{Gen}(1^\lambda)$ is a PPT algorithm that takes as input the security parameter λ and outputs a PRF key K.
- **Punctured Key Generation** $\mathsf{Puncture}(K, S)$ is a PPT algorithm that takes as input a PRF key K, a set $S \subset \{0,1\}^{n(\lambda)}$ and outputs a punctured key K_S.
- **Evaluation** $\mathsf{Eval}(K, x)$ is a deterministic algorithm that takes as input a key K (punctured key or PRF key), a string $x \in \{0,1\}^{n(\lambda)}$ and outputs $y \in \{0,1\}^{m(\lambda)}$

Definition A.1. *A family of PRFs* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture})$ *is puncturable if it satisfies the following properties*

- **Functionality preserved under puncturing.** *Let* $K \leftarrow \mathsf{Gen}(1^\lambda)$ *and* $K_S \leftarrow \mathsf{Puncture}(K, S)$. *Then for all* $x \notin S$, $\mathsf{Eval}(K, x) = \mathsf{Eval}(K_S, x)$.
- **Pseudorandom at punctured points.** *For every PPT adversary* $(\mathcal{A}_1, \mathcal{A}_2)$ *such that* $\mathcal{A}_1()$ *outputs a set* $S \subset \{0,1\}^{n(\lambda)}$ *and* $x \in S$, *consider an experiment* $K \leftarrow \mathsf{Gen}(1^\lambda)$ *and* $K_S \leftarrow \mathsf{Puncture}(K, S)$. *Then*

$$\left|\Pr[\mathcal{A}_2(K_S, x, \mathsf{Eval}(K, x)) = 1] - \Pr[\mathcal{A}_2(K_S, x, U_{m(\lambda)}) = 1]\right| \le \mathsf{negl}(\lambda)$$

where $U_{m(\lambda)}$ *denotes the the uniform distribution over* $m(\lambda)$ *bits.*

Theorem A.2 ([24, 11, 13, 34]). *If one-way functions exist, then for all polynomial* $n()$ *and* $m()$, *there exists a puncturable PRF family that maps* $n()$ *bits to* $m()$ *bits.*

A.2 Fully Homomorphic Encryption

A circuit-based fully homomorphic scheme $\mathsf{cFHE}.(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ is a tuple of algorithms described as follows:

$(hpk, hsk) \leftarrow \mathsf{cFHE.Setup}(1^\lambda)$ takes in the security parameter λ, then outputs the public and secret key pair (hpk, hsk).

$c \leftarrow \mathsf{cFHE.Enc}(hpk, x)$ takes in a plaintext message x and public key hpk, then outputs the ciphertext c.

$x \leftarrow \mathsf{cFHE.Dec}(hsk, c)$ takes in a ciphertext c and secret key hsk, then outputs the plaintext x.

$c' \leftarrow \mathsf{cFHE.Eval}(hpk, C, (c_1, \ldots, c_k))$ takes in a circuit description C, input ciphertexts $c_1, \ldots, c_k$ and public key hpk, then outputs the ciphertext c'.

The definition of semantic security for $\mathsf{cFHE}.(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ follows the definition of public-key encryption; we omit a formal statement.

Definition A.3. *A circuit-based homomorphic encryption scheme* $\mathsf{cFHE}.(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ *is* (perfectly) correct *if for all $\lambda \in \mathbb{N}$, for all polynomial-size circuits C, for all honest $(hpk, hsk) \leftarrow \mathsf{Setup}(1^\lambda)$, for all honestly generated $c_1 := \mathsf{Enc}(hpk, x_1), \ldots, c_k := \mathsf{Enc}(hpk, x_k)$, it holds that* $\mathsf{Dec}(hsk, \mathsf{Eval}(hpk, C, c_1, \ldots, c_k)) = C(x_1, \ldots, x_k)$.

Definition A.4. *A circuit-based homomorphic encryption scheme $c\mathcal{FHE}$ is* compact*, if there exists a polynomial* poly *such that for all $\lambda \in \mathbb{N}$, the ciphertexts output by* Eval *have size at most* $\mathsf{poly}(\lambda)$.

A.3 Non-Interactive Proof Systems

A non-interactive proof system consists of three efficient algorithms $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$. The generation algorithm $crs \leftarrow \mathsf{Gen}(1^\lambda)$ produces a common random string crs. The proves algorithm $\pi \leftarrow \mathsf{Prove}(crs, u, w)$ produces a proof π for a statement u using a witness w. The verification algorithm $\mathsf{Verify}(crs, u, \pi)$ decides whether π is a valid proof for the statement u using common reference string crs.

Definition A.5. *We say that* $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Verify})$ *is a non-interactive proof system for an* NP *language L with a corresponding* NP *relation R, if it satisfies the following two properties:*

Completeness : For all $(u, w) \in R$, it holds that:

$$\Pr[\mathsf{Verify}(crs, u, \pi) = 0 | crs \leftarrow \mathsf{Gen}(1^\lambda), \pi \leftarrow \mathsf{Prove}(crs, u, w)] = \mathsf{negl}(\lambda)$$

Soundness : For all efficient Prove', it holds that:

$$\Pr[\mathsf{Verify}(crs, u, \pi) = 1, u \notin L | (crs) \leftarrow \mathsf{Gen}(1^\lambda), \pi \leftarrow \mathsf{Prove}'(crs, u, w)] = \mathsf{negl}(\lambda)$$

Zero-Knowledge. We first define the syntax of the simulator: $(crs, \tau) \leftarrow \mathsf{Sim}_1$ outputs a simulated crs and trapdoor τ, $\pi \leftarrow \mathsf{Sim}_2(crs, u, \tau)$ outputs a simulated proof. The proof system is zero-knowledge if there exist a poly-time simulator $(\mathsf{Sim}_1, \mathsf{Sim}_2)$ such for any adversary $\mathcal{A}$, it holds:

$$\Pr[\mathcal{A}(\pi) = 1 | crs \leftarrow \mathsf{Gen}(1^\lambda), \pi \leftarrow \mathsf{Prove}(crs, u, w)] \approx \Pr[\mathcal{A}(\pi) = 1 | (crs, \tau) \leftarrow \mathsf{Sim}_1, \pi \leftarrow \mathsf{Sim}_2(crs, u, \tau)]$$

where $(u, w) \in R, (u, w) \leftarrow \mathcal{A}(crs)$.

A.4 Statistical Simulation-Sound NIZKs

Let R be a polynomial-time computable binary relation. For $(\mathsf{stmt}, w) \in R$, we call stmt the statement, and w the witness . Let L be the language consisting of all statements in R.

A Non-Interactive Zero-knowledge Proof system (NIZK) is a collection of three algorithms $\mathsf{NIZK} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$:

- $crs \leftarrow \mathsf{Setup}(1^\lambda)$: Takes in the security parameter λ, and generates a common reference string crs.
- $\pi \leftarrow \mathsf{Prove}(crs, \mathsf{stmt}, w)$: Takes in crs, a statement stmt, and a witness w such that $(\mathsf{stmt}, w) \in L$, outputs a proof π.
- $b \leftarrow \mathsf{Verify}(crs, \mathsf{stmt}, \pi)$: Takes in the crs, a statement stmt, and a proof π, and outputs 0 or 1, denoting rejection or acceptance. stmt, and a witness w,

Perfect completeness. A NIZK system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any $(\mathsf{stmt}, w) \in R$, we have

$$\Pr\Big[crs \leftarrow \mathsf{Setup}(1^\lambda),\ \pi \leftarrow \mathsf{Prove}(crs, \mathsf{stmt}, w) : \quad \mathsf{Verify}(crs, \mathsf{stmt}, \pi) = 1\Big] = 1$$

Statistical soundness. A NIZK system is said to be statistically sound, if there does not exist a valid proof for any no false statement. More formally,

$$\Pr\Big[crs \leftarrow \mathsf{Setup}(1^\lambda),\ \exists(\mathsf{stmt}, \pi)\text{:} \quad (\mathsf{stmt} \notin L)\ \wedge\ (\mathsf{Verify}(crs, \mathsf{stmt}, \pi) = 1)\Big] = \mathsf{negl}(\lambda)$$

Computational zero-knowlege. Informally, a NIZK system is computationally zero-knowledge, if the proof does not reveal any information about the witness to any polynomial-time adversary. More formally, a NIZK system is said to computationally zero-knowledge, if there exists a simulator $S = (\mathsf{SimSetup}, \mathsf{SimProve})$, such that for all non-uniform polynomial-time adversary $\mathcal{A}$, for any stmt, w such that $(\mathsf{stmt}, w) \in R$, it holds that

$$\left| \Pr\begin{bmatrix} crs \leftarrow \mathsf{Setup}(1^\lambda), \\ \pi \leftarrow \mathsf{Prove}(crs, \mathsf{stmt}, w) : \\ \mathcal{A}(crs, \mathsf{stmt}, \pi) = 1 \end{bmatrix} - \Pr\begin{bmatrix} (\widetilde{crs}, \mathsf{trap}) \leftarrow \mathsf{SimSetup}(1^\lambda, \mathsf{stmt}), \\ \widetilde{\pi} \leftarrow \mathsf{SimProve}(crs, \mathsf{stmt}, \mathsf{trap}) : \\ \mathcal{A}(\widetilde{crs}, \mathsf{stmt}, \widetilde{\pi}) = 1 \end{bmatrix} \right| = \mathsf{negl}(\lambda)$$

Statistical simulation soundness [20]. Informally, a NIZK system is statistically simulation sound, if under a simulated $\widetilde{crs}$, no proof for a false statement exists, except for the simulated proof for statement fed into the $\mathsf{SimSetup}$ algorithm to generate $\widetilde{crs}$. More formally, a NIZK system is said to be statistically simulation sound, if

$$\Pr\begin{bmatrix} (\widetilde{crs}, \mathsf{trap}) \leftarrow \mathsf{SimSetup}(1^\lambda, \mathsf{stmt}), \pi \leftarrow \mathsf{SimProve}(crs, \mathsf{stmt}, \mathsf{trap}) : \\ \exists(\mathsf{stmt}', \pi') \text{ s.t. } \mathsf{stmt}' \neq \mathsf{stmt})\ \wedge\ (\mathsf{Verify}(\widetilde{crs}, \mathsf{stmt}', \pi') = 1) \end{bmatrix} = \mathsf{negl}(\lambda).$$

A.5 Indistinguishability Obfuscation

A uniform PPT machine $i\mathcal{O}$ is called an indistinguishable obfucastor [5, 29, 20], for a circuit family $\{\mathcal{C}_\lambda\}$, if the following conditions hold:

- **Correctness.** For all $\lambda \in \mathbb{N}$, for all $C \in \mathcal{C}_\lambda$, for all inputs x, we have
$$\Pr\left[C' \leftarrow i\mathcal{O}(\lambda, C) :\ C'(x) = C(x)\right] = 1$$
- For any uniform or non-uniform PPT distinguisher D, for all security parameter $\lambda \in \mathbb{N}$, for all pairs of circuits $C_0, C_1 \in \mathcal{C}_\lambda$ such that $C_0(x) = C_1(x)$ for all inputs x, then
$$|\Pr\left[D(i\mathcal{O}(\lambda, C_0)) = 1\right] - \Pr\left[D(i\mathcal{O}(\lambda, C_1)) = 1\right]| \leq \mathsf{negl}(\lambda)$$

For simplicity, when the security parameter λ is clear, we write $i\mathcal{O}(C)$ in short.

Remark A.6. In a recent work [32], Gordon et al. observe that for the construction of [20], the size of the obfuscated circuit has only linear blowup, i.e. $|i\mathcal{O}(C)| = |C| \cdot \mathsf{poly}(\lambda)$. In the construction, in order to obfuscate any poly-sized C, they consider another NC1 program P that checks the trace of the computation of the FHE evaluation, and then decrypts. Essentially P is an "and" of $|C|$ local verifications (each of which has size $\mathsf{poly}(\lambda)$), and then a decryption, which has size $\mathsf{poly}(\lambda)$. Gordon et al. observe that each local verification can be written as a constant width, $\mathsf{poly}(\lambda)$ length branching program by the Barrington's theorem; also it takes an additive linear blow up to "AND" multiple branching programs. Thus, P can be transformed to a branching program of size $O(|C|) \cdot \mathsf{poly}(\lambda)$. Then the construction of [20] blows up the branching program by $\mathsf{poly}(\lambda)$. Thus, the obfuscated circuit has size $O(|C|) \cdot \mathsf{poly}(\lambda)$.

A.6 Differing-Inputs Obfuscation for Circuits

Barak et al. [5] defined the notion of differing-inputs obfuscation. We present the notion of differing-inputs circuit family as the formulation in the works of Ananth et al. and Boyle et. al [2, 12]

Definition A.7 ([4, 2, 12])**.** *A circuit family $\mathcal{C}$ associated with a sampler* $\mathsf{Sampler}$ *is said to be a differing-inputs circuit family if for every PPT adversary $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\Pr[C_0(x) \neq C_1(x) : (C_0, C_1, \mathsf{aux}) \leftarrow \mathsf{Sampler}(1^\lambda), x \leftarrow \mathcal{A}(1^\lambda, C_0, C_1, \mathsf{aux})] \leq \mathsf{negl}(\lambda).$$

We now define the notion of differing-inputs obfuscation for a differing-inputs circuit family.

Definition A.8 (Differing-Inputs Obfuscators for circuits)**.** *A uniform PPT machine* $\mathsf{di}\mathcal{O}$ *is called a Differing-inputs Obfuscator for a differing-inputs circuit family $\mathcal{C} = \{C_\lambda\}$ if the following conditions are satisfied:*

- *(Correctness): For all security parameter λ, all $C \in \mathcal{C}$, all inputs x, we have*
$$\Pr[C'(x) = C(x) : C' \leftarrow \mathsf{di}\mathcal{O}(\lambda, C)] = 1.$$
- *(Polynomial slowdown): There exists a universal polynomial p such that for any circuit C, we have $|C'| \leq p(|C|)$ for all $C' = \mathsf{di}\mathcal{O}(\lambda, C)$ under all random coins.*
- *(Differing-inputs): For any (not necessarily uniform) PPT distinguisher D, there exists a negligible function* negl *such that the following holds: for all security parameters λ, for $(C_0, C_1, \mathsf{aux}) \leftarrow \mathsf{Sampler}(1^\lambda)$, we have that*
$$|\Pr[D(\mathsf{di}\mathcal{O}(\lambda, C_0, \mathsf{aux})) = 1] - \Pr[D(\mathsf{di}\mathcal{O}(\lambda, C_1, \mathsf{aux})) = 1]| \leq \mathsf{negl}(\lambda).$$

A.7 Virtual Black-Box (VBB) Obfuscation

Definition A.9 ([5]). *Let $\mathcal{C} = \{\mathcal{C}_n\}_{n\in\mathbb{N}}$ be a family of polynomial-size circuits, where $\mathcal{C}_n$ is a set of Boolean circuits operating on inputs of length n. And let $\mathcal{O}$ be a PPTM algorithm, which takes as input an input length $n \in \mathbb{N}$, a circuit $C \in \mathcal{C}_n$, a security parameter $\lambda \in \mathbb{N}$, and outputs a Boolean circuit $\mathcal{O}(C)$ (not necessarily in $\mathcal{C}$).*

$\mathcal{O}$ is a black box obfuscator *for the circuit family $\mathcal{C}$ if it satisfies:*

1. Preserving Functionality: *For every $n \in \mathbb{N}$, and every $C \in \mathcal{C}_n$, and every $\vec{x} \in \{0,1\}^n$, with all but negl(λ) probability over the coins of $\mathcal{O}$:*

$$(\mathcal{O}(C, 1^n, 1^\lambda))(\vec{x}) = C(\vec{x})$$

2. Polynomial Slowdown: *For every $n, \lambda \in \mathbb{N}$ and $C \in \mathcal{C}$, the circuit $\mathcal{O}(C, 1^n, 1^\lambda)$ is of size at most poly($|C|, n, \lambda$).*

3. Virtual Black Box: *For every (non-uniform) polynomial-size adversary $\mathcal{A}$, there exists a (non-uniform) polynomial-size simulator $\mathcal{S}$, such that for every $n \in \mathbb{N}$ and* for every *$C \in \mathcal{C}_n$:*

$$\left| \Pr_{\mathcal{O},\mathcal{A}}[\mathcal{A}(\mathcal{O}(C, 1^n, 1^\lambda)) = 1] - \Pr_{\mathcal{S}}[\mathcal{S}^C(1^{|C|}, 1^n, 1^\lambda) = 1] \right| = negl(\lambda)$$

A.8 Proof-Carrying Data

For completeness, we present the Proof-Carrying-Data (PCD) definition in exactly the same way as Bitansky et al. [9].

We view a distributed computation as a directed acyclic graph $G = (V, E)$ with node labels $\mathsf{linp} : V \to \{0,1\}^*$ and edge labels $\mathsf{data} : E \to \{0,1\}^*$. The node label $\mathsf{linp}(v)$ of a node v represents the *local input* (which may include a local program) used by v in his local computation. (Whenever v is a source or a sink, we require that $\mathsf{linp}(v) = \perp$.) The edge label $\mathsf{data}(u, v)$ of a directed edged (u, v) represents the message sent from node u to node v. Typically, a party at node v uses the local input $\mathsf{linp}(v)$ and input messages $(\mathsf{data}(u_1, v), \ldots, \mathsf{data}(u_c, v))$, where $u_1, \ldots, u_c$ are the parents of v in lexicographic order, to compute an output message $\mathsf{data}(v, w)$ for a child node w; the party also similarly computes a message for every other child node. We can think of the messages on edges going out from sources as the "inputs" to the distributed computation, and the messages on edges going into sinks as the "outputs" of the distributed computation; for convenience we will want to identify a single distinguished output.

Definition A.10. *A (distributed computation) transcript is a triple $\mathsf{T} = (G, \mathsf{linp}, \mathsf{data})$, where $G = (V, E)$ is a directed acyclic graph G, $\mathsf{linp} : V \to \{0,1\}^*$ are node labels, and $\mathsf{data} : E \to \{0,1\}^*$ are edge labels; we require that $\mathsf{linp}(v) = \perp$ whenever v is a source or a sink. The output of T, denoted $\mathsf{out}(\mathsf{T})$, is equal to $\mathsf{data}(\tilde{u}, \tilde{v})$ where $(\tilde{u}, \tilde{v})$ is the lexicographically first edge such that $\tilde{v}$ is a sink.*

A proof-carrying transcript is a transcript where messages are augmented by proof strings, i.e., a function $\mathsf{proof} : E \to \{0,1\}^*$ provides for each edge (u, v) an additional label $\mathsf{proof}(u, v)$, to be interpreted as a proof string for the message $\mathsf{data}(u, v)$.

Definition A.11. *A proof-carrying (distributed computation) transcript PCT is a pair $(\mathsf{T}, \mathsf{proof})$ where T is a transcript and $\mathsf{proof} : E \to \{0,1\}^*$ is an edge label.*

Next, we define what it means for a distributed computation to be *compliant*, which is the notion of "correctness with respect to a given local property". Compliance is captured via an efficiently-computable compliance predicate $\mathbb{C}$, which must be locally satisfied at each vertex; here, "locally" means with respect to a node's local input, incoming data, and outgoing data. For convenience, for any vertex v, we let $\mathsf{children}(v)$ and $\mathsf{parents}(v)$ be the vector of v's children and parents respectively, listed in lexicographic order.

Definition A.12. *Given a polynomial-time predicate $\mathbb{C}$, we say that a distributed computation transcript $\mathsf{T} = (G, \mathsf{linp}, \mathsf{data})$ is $\mathbb{C}$-compliant (denoted by $\mathbb{C}(\mathsf{T}) = 1$) if, for every $v \in V$ and $w \in \mathsf{children}(v)$, it holds that*

$$\mathbb{C}(\mathsf{data}(v, w); \mathsf{linp}(v); \mathsf{inputs}(v)) = 1$$

where $\mathsf{inputs}(v) := (\mathsf{data}(u_1, v), \ldots, \mathsf{data}(u_c, v))$ and $(u_1, \ldots, u_c) := \mathsf{parents}(v)$. Furthermore, we say that a message z is $\mathbb{C}$-compliant if there is T such that $\mathbb{C}(\mathsf{T}) = 1$ and $\mathsf{out}(\mathsf{T}) = 1$.

Proof-Carrying Data Systems. A proof-carrying data (PCD) system for a class of compliance predicates $\mathbf{C}$ is a triple of algorithms $(\mathbb{G}, \mathbb{P}, \mathbb{V})$ that works as follows:

- The (probabilistic) generator $\mathbb{G}$, on input the security parameter λ, outputs a reference string rs and a corresponding verification state vrs.
- For any $\mathbb{C} \in \mathbf{C}$, the (honest) prover $\mathbb{P}_{\mathbb{C}} := \mathbb{P}(\mathbb{C}, \cdots)$ is given a reference string rs inputs $\vec{\mathsf{z}}_i$ with corresponding proofs $\vec{\pi}_i$, a local input linp, and an output z_o, and then produces a proof π_o attesting to the fact that z_o is consistent with some $\mathbb{C}$-compliant transcript.
- For any $\mathbb{C} \in \mathbf{C}$, the verifier $\mathbb{V}_{\mathbb{C}} := \mathbb{V}(\mathbb{C}, \cdots)$ is given the verification state vrs, an output z_o, and a proof string π_o, and accept if it is convinced that z_o is consistent with some $\mathbb{C}$-compliant transcript.

After the generator $\mathbb{G}$ has been run to obtain rs and vrs, the prover $\mathbb{P}_{\mathbb{C}}$ is used (along with rs) at each node of a distributed computation transcript to dynamically compile it into a proof-carrying transcript by generating and adding a proof to each edge. Each of these proofs can be checked using the verifier $\mathbb{V}_{\mathbb{C}}$ (along with vrs).

The formal definition. We now formally define the notion of PCD systems. We begin by introducing the dynamic proof-generation process, which we call $\mathsf{ProofGen}$. We define $\mathsf{ProofGen}$ as an interactive protocol between a (not necessarily efficient) distributed-computation generator S and the PCD prover $\mathbb{P}$, in which both are given a compliance predicate $\mathbb{C} \in \mathbf{C}$ and a reference string rs. Essentially, at every time step, S chooses to do one of the following actions: add a new unlabeled vertex to the computation transcript so far (this corresponds to adding a new computing node to the computation), label an unlabeled vertex (this corresponds to a choice of local input by a computing node), or add a new labeled edge (this corresponds to a new message from one node to another). In case S chooses the third action, the PCD prover $\mathbb{P}_{\mathbb{C}}$ produces a proof for the $\mathbb{C}$-compliance of the new message, and adds this new proof as an additional label to the new edge. When S halts, the interactive protocol outputs the distributed computation transcript T, as well as T's output and corresponding proof. Intuitively, the completeness property requires that if T is compliant with $\mathbb{C}$, then the proof attached to the output (which is the result of dynamically invoking $\mathbb{P}_{\mathbb{C}}$ for each message in T, as T was being constructed by S) is accepted by the verifier. Formally the interactive protocol $\mathsf{ProofGen}(\mathbb{C}, \mathsf{rs}, S, \mathbb{P})$ is defined as follows:

$\mathsf{ProofGen}(\mathbb{C}, \mathsf{rs}, S, \mathbb{P})$

1. Set T and PCT to be "empty transcripts".
2. Until S halts and outputs a message-proof pair (z_o, π_o), do the following:
   (a) Give $(\mathbb{C}, \mathsf{rs}, \mathsf{PCT})$ as input to S and obtain as output (b, x, y).
   (b) If $b = \texttt{add unlabeled vertex}$ and $x \notin V$, then set $V := V \cup \{x\}$ and $\mathsf{linp}(x) := \bot$.
   (c) If $b = \texttt{label vertex}$, $x \in V$, x is nor a source or sink, and $\mathsf{linp}(x) = \bot$, then $\mathsf{linp}(x) := y$.
   (d) If $b = \texttt{add labeled edge}$ and $x \notin E$:
      i. Parse x as (v, w) with $v, w \in V$
      ii. Set $E := E \cup \{(u, v)\}$
      iii. Set $\mathsf{data}(v, w) := y$.
      iv. If v is a source, set $\pi := \bot$
      v. If v is not a source, set $\pi := \mathbb{P}_{\mathbb{C}}(\mathsf{rs}, \mathsf{data}(v, w), \mathsf{linp}(v), \mathsf{inputs}(v), \mathsf{inproofs}(v))$, where $\mathsf{inputs}(v) := \mathsf{data}(u_1, v), \ldots, \mathsf{data}(u_c, v)$, $\mathsf{inproofs}(v) := \mathsf{proof}(u_1, v), \ldots, \mathsf{proof}(u_c, v)$, and $(u_1, \ldots, u_c) := \mathsf{parents}(v)$.
      vi. Set $\mathsf{proof}(v, w) := \pi$

Definition A.13. *A proof-carrying data system for a class of compliance predicates* $\mathbf{C}$ *is a triple of algorithms* $(\mathbb{G}, \mathbb{P}, \mathbb{V})$*, where* $\mathbb{G}$ *is probabilistic and* $\mathbb{V}$ *is deterministic, such that:*

1. *Completeness:*

   For every compliance predicate $\mathbb{C} \in \mathbf{C}$ *and (possibly unbounded) distributed computation generator* S*,*

$$\Pr\left[\begin{array}{l} \mathsf{T} \textit{ is } B \textit{ bounded} \wedge \mathbb{C}(\mathsf{T}) = 1 \wedge \mathbb{V}_{\mathbb{C}}(\mathsf{vrs}, \mathsf{z}_o, \pi_o) \neq 1 \\ \quad : (\mathsf{rs}, \mathsf{vrs}) \leftarrow \mathbb{G}(B), (\mathsf{z}_o, \pi_o, \mathsf{T}) \leftarrow \mathsf{ProofGen}(\mathbb{C}, \mathsf{rs}, S, \mathbb{P}) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

2. *Proof of Knowledge:*

   For every polynomial-size prover $\mathbb{P}^*$ *there exists a polynomial-size extractor* $\mathbb{E}_{\mathbb{P}^*}$ *such that for every compliance predicate* $\mathbb{C} \in \mathbf{C}$*, every large enough security parameter* $\lambda \in \mathbb{N}$*, every auxiliary input* $z \in \{0,1\}^{\mathsf{poly}(\lambda)}$*, and every time bound* $B \in \mathbb{N}$*,*

$$\Pr\left[\begin{array}{l} \mathbb{V}_{\mathbb{C}}(\mathsf{rs}, \mathsf{z}, \pi) = 1 \wedge (\mathsf{out}(\mathsf{T}) \neq \mathsf{z} \vee \mathbb{C}(\mathsf{T}) \neq 1) \\ \quad : (\mathsf{rs}, \mathsf{vrs}) \leftarrow \mathbb{G}(B), (\mathsf{z}_o, \pi_o) \leftarrow \mathbb{P}^*(\mathsf{rs}, z), \mathsf{T} \leftarrow \mathbb{E}_{\mathbb{P}^*}(\mathsf{rs}, z) \end{array}\right] \leq \mathsf{negl}(\lambda)$$

3. *Efficiency:*

   There exists a universal polynomial p *such that, for every compliance predicate* $\mathbb{C} \in \mathbf{C}$*, every large enough security parameter* $\lambda \in \mathbb{N}$*, every time bound* $B \in \mathbb{N}$*, and every* B*-bounded distributed computation transcript* T*,*

   - *the generator* $\mathbb{G}(1^\lambda, B)$ *runs in time* $\begin{cases} p(\lambda + B) & \textit{for a fully-succinct PCD} \\ p(\lambda + \log B) & \textit{for a preprocessing PCD} \end{cases}$
   - *the prover* $\mathbb{P}_{\mathbb{C}}(\mathsf{rs}, \mathsf{data}(v, w), \mathsf{linp}(v), \mathsf{inputs}(v), \vec{\pi}_i)$ *runs in time*

$$\begin{cases} p(\lambda + |\mathbb{C}| + t_{\mathsf{T},\mathbb{C}}(v, w) + \log B) & \textit{for a fully-succinct PCD} \\ p(\lambda + |\mathbb{C}| + B) & \textit{for a preprocessing PCD} \end{cases}$$

   where $t_{\mathsf{T},\mathbb{C}}(v, w)$ *denotes the time to evaluate* $\mathbb{C}(\mathsf{data}(v, w), \mathsf{linp}(v), \mathsf{inputs}(v))$ *at an edge* (v, w)*;*

- *the verifier* $\mathbb{V}_{\mathbb{C}}(\mathsf{vrs}, \mathsf{z}, \pi)$ *runs in time* $p(\lambda + |\mathbb{C}| + |\mathsf{z}| + \log B)$
- *an honestly generated proof has size* $p(\lambda + \log B)$*.*

We shall also consider a restricted notion of PCD system: a **path PCD** *system is a PCD system where completeness is guaranteed to hold only for distributed computations transcripts* T *whose graph is a line.*

B RAM-Model Functional Encryption Security Definitions

B.1 Simulation-Based Security

Simulation-based security for functional encryption has been defined in several prior works [10, 38, 30, 1]. Here we follow the definition in [38, 30], and define the simulation based security for functional encryption in the RAM model. Note that the Setup and the pre-challenge KeyGen are carried out honestly (not by the simulator). As demonstrated in [30], single message security defined below implies multiple-message security where the adversary is allowed to provide many messages in the challenging phase.

Definition B.1 (Full Simulation Security, Non-Adaptive/Adaptive, Multiple-Key, Single-Message)**.** *Let* FE *be a functional encryption scheme for RAM programs computing a functionality class* $\mathcal{F}$*. For every* PPT *stateful adversary* $\mathcal{A}$ *and* PPT *stateful simulator* $\mathcal{S}$*, consider the following two experiments.*

$\textsc{Real}_{\mathcal{A}}(1^\lambda)$	$\textsc{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)$
1: $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$;	1: $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$;
2: $x \leftarrow \mathcal{A}^{\mathsf{FE.KeyGen}(\mathsf{msk},\cdot)}(\mathsf{mpk})$;	2: $x \leftarrow \mathcal{A}^{\mathsf{FE.KeyGen}(\mathsf{msk},\cdot)}(\mathsf{mpk})$; Let $\mathsf{RAM}_1, ..., \mathsf{RAM}_q$ be $\mathcal{A}$'s oracle queries, $\mathsf{sk}_{\mathsf{RAM}_i}$ be the oracle reply to RAM_i, and $\mathsf{view} := \{y_i = \mathsf{RAM}_i(x), \mathsf{RAM}_i, \mathsf{sk}_{\mathsf{RAM}_i}\}_{i\in[q]}$.
3: $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x)$;	3: $\mathsf{ct} \leftarrow \mathcal{S}(\mathsf{mpk}, \mathsf{view}, 1^{|x|})$;
4: $\alpha \leftarrow \mathcal{A}^{\mathcal{O}(\mathsf{msk},\cdot)}(\mathsf{mpk}, \mathsf{ct})$;	4: $\alpha \leftarrow \mathcal{A}^{\mathcal{O}'(\mathsf{msk},\cdot)}(\mathsf{mpk}, \mathsf{ct})$;
5: OUTPUT (α, x);	5: OUTPUT (α, x);

We consider two cases of the above experiments:

1. The non-adaptive case*, where the oracles* $\mathcal{O}(\mathsf{msk}, \cdot)$ *and* $\mathcal{O}'(\mathsf{msk}, \cdot)$ *are both "empty oracles" that return nothing.*

2. The adaptive case*, where*

   - *the oracle* $\mathcal{O}(\mathsf{msk}, \cdot) = \mathsf{FE.KeyGen}(\mathsf{msk}, \cdot)$*, and*
   - *the oracle* $\mathcal{O}'(\mathsf{msk}, \cdot)$ *is the (stateful) second stage of the simulator; namely,* $\mathcal{O}'(\mathsf{msk}, \cdot) = \mathcal{S}^{U(\cdot,x)}(\mathsf{msk}, \cdot)$ *where* $U(\mathsf{RAM}, x) = \mathsf{RAM}(x)$ *for all* RAM*.*

We call a simulator algorithm $\mathcal{S}$ admissible *if, on each of* $\mathcal{A}$*'s queries* RAM *to the second stage of the simulator, the simulator makes just a single query to its oracle* $U(\cdot, x)$ *on* RAM *itself.*

The functional encryption scheme FE *is said to be* fully SIM-secure against non-adaptive (resp. adaptive) adversaries *if there is an admissible* PPT *stateful simulator* $\mathcal{S}$ *such that for every* PPT *stateful adversary* $\mathcal{A}$*, the following two distributions are computationally indistinguishable:*

$$\left\{\textsc{Real}_{\mathcal{A}}(1^\lambda)\right\}_\lambda \overset{c}{\approx} \left\{\textsc{Ideal}_{\mathcal{A},\mathcal{S}}(1^\lambda)\right\}_\lambda$$

We can easily define the *selective-simulation security* by enforcing the adversary to reveal the challenge messages before seeing the master public key mpk. We can also easily define the full simulation security to support multiple messages.

B.2 Indistinguishability-Based Security

Indistinguishability-based security for functional encryption has also been considered [10, 38]. We provide a definition using our notation:

Definition B.2 (Full Indistinguishability, Multiple-Key). *Let* FE *be a functional encryption scheme for RAM programs computing a functionality class* $\mathcal{F}$*. For every* PPT *stateful adversary* $\mathcal{A}$ *, consider the following experiment.*

$\mathbf{Expt}^{\text{full}}_{\mathcal{A}}(1^\lambda)$

1: $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$;
2: $(x_0, x_1) \leftarrow \mathcal{A}^{\mathsf{FE.KeyGen}(\mathsf{msk},\cdot)}(\mathsf{mpk})$;
3: $b \leftarrow \{0,1\}$;
4: $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x_b)$;
5: $b' \leftarrow \mathcal{A}^{\mathsf{FE.KeyGen}(\mathsf{msk},\cdot)}(\mathsf{mpk}, \mathsf{ct})$;

Define an adversary to be non-trivial *if* $\mathsf{RAM}(x_0) = \mathsf{RAM}(x_1)$*, for every query* RAM *made to the* KeyGen *oracle. We say that a FE-RAM scheme is* fully IND-secure *if for all* PPT *stateful* non-trivial *adversaries* $\mathcal{A}$*, it holds that* $\Pr[b' = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$ *in the above experiment.*

We also consider selective security for indistinguishability. In the selective security game, the adversary must reveal the challenge messages (x_0, x_1) before seeing the master public key mpk. To be complete, we restate the details of Definition 2.4 here:

Definition B.3 (Selective Indistinguishability, Multiple-Key). *Let* FE *be a functional encryption scheme for RAM programs computing a functionality class* $\mathcal{F}$*. For every* PPT *stateful adversary* $\mathcal{A}$*, consider the following experiment.*

$\mathbf{Expt}^{\text{selective}}_{\mathcal{A}}(1^\lambda)$

1: $(x_0, x_1) \leftarrow \mathcal{A}(1^\lambda)$;
2: $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{FE.Setup}(1^\lambda)$;
3: $b \leftarrow \{0,1\}$;
4: $\mathsf{ct} \leftarrow \mathsf{FE.Enc}(\mathsf{mpk}, x_b)$;
5: $b' \leftarrow \mathcal{A}^{\mathsf{FE.KeyGen}(\mathsf{msk},\cdot)}(\mathsf{mpk}, \mathsf{ct})$;

Define an adversary to be non-trivial *if* $\mathsf{RAM}(x_0) = \mathsf{RAM}(x_1)$*, for every query* RAM *made to the* KeyGen *oracle. We say that an FE-RAM scheme is* selectively IND-secure *if for all* PPT *stateful* non-trivial *adversaries* $\mathcal{A}$*, it holds that* $\Pr[b' = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda)$ *in the above experiment.*

Remark B.4. In the circuit model, selective-indistinguishability can be transformed into full-indistinguishability by using a standard complexity leveraging argument but losing a factor of $|\mathcal{M}|$ in the security reduction [20, 19] where $|\mathcal{M}|$ is the size of message space. Here we can also apply the same argument in the RAM model to transform selective-indistinguishability as defined in Definition 2.4 into full-indistinguishability in Definition B.2.

Furthermore, we consider a single challenge in the security games, but remark that it can be generalized to multiple-message security by using a standard hybrid argument. Finally, we can lift our indistinguishability security into simulation-based security as defined in Definition B.1 by using similar techniques from the work [19].

C Security Proof for $\mathsf{di}\mathcal{O}$-based FE-RAM Construction

The security of our $\mathsf{di}\mathcal{O}$-based functional encryption scheme for RAMs is proven in two steps. First we prove a technical lemma regarding "address unforgeability" or "trace correctness." Given this, indistinguishability security, i.e. Theorem 3.1, is then shown through a sequence of hybrids.

C.1 Trace Correctness

For simplicity, we consider Hybrid X0 for the time being, which is a real world where the challenger returns an honest encryption of x_0^* upon a challenge ciphertext query. However, the same lemma below can also be proved in a similar manner for other hybrid games described later.

Lemma C.1. *Assume that the PCD is a proof of knowledge system, the one-way function is secure, the Merkle-tree is collision resistant, the* $\mathsf{di}\mathcal{O}$ *is a secure differing-inputs obfuscation, the* $i\mathcal{O}$ *is a secure indistinguishable obfuscation, the PRF is a correct and secure puncturable PRF, and that the* cFHE *encryption scheme is perfectly correct and semantically secure.*

Let ct^* *denote the challenge ciphertext. For any* $\tau \in [T_{\mathsf{ORAM}}]$*, except with negligible probability, no polynomial-time adversary is able to produce a tuple* $\{\mathsf{ct}^*, \tau, \mathsf{digest}_\tau, \mathsf{raddr}_\tau, \boxed{\mathsf{cpustate}_\tau}\}$ *with a valid PCD proof, such that* digest_τ*,* raddr_τ*, and* $\boxed{\mathsf{cpustate}_\tau}$ *do not agree with correct values at time* τ *obtained through an honest evaluation algorithm on the challenge ciphertext* ct^**, and token* sk_{ORAM}*. The adversary is given* mpk*,* sk_{ORAM}*, challenge ciphertext* ct^**, and challenge plaintext* x_0^**; further the adversary has oracle access to the key generation oracle.*

Proof. If a tuple $\{\mathsf{ct}, \tau, \mathsf{digest}_\tau, \mathsf{raddr}_\tau, \boxed{\mathsf{cpustate}_\tau}\}$ has a valid PCD proof, then since PCD is a sound proof of knowledge system, there exists a polynomial-time extractor that can extract an evaluation trace (consistent with the ct, digest_τ, raddr_τ, and $\boxed{\mathsf{cpustate}_\tau}$ values included in the PCD statement),

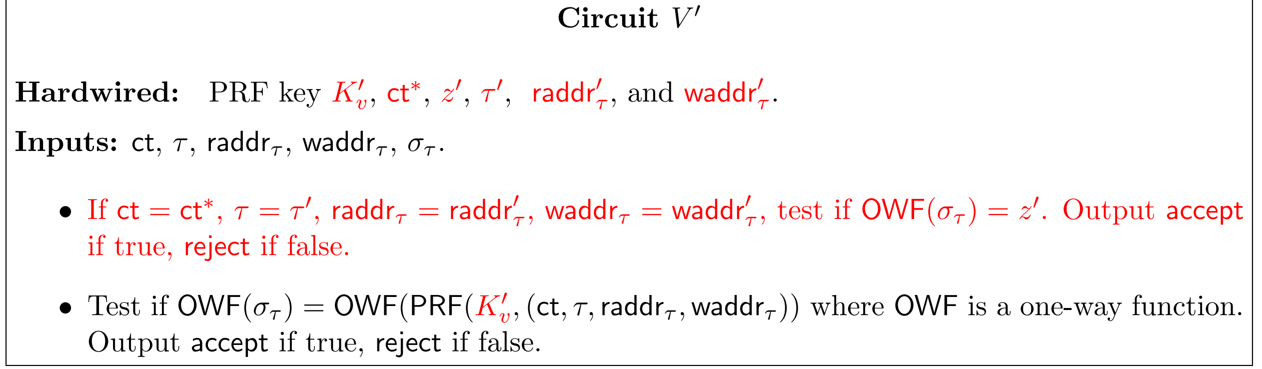
$$\begin{aligned} \mathsf{ct}, \quad \forall t \in [\tau] : &\quad \{\mathsf{waddr}_t, \mathsf{raddr}_t, \boxed{\mathsf{cpustate}_t}, \boxed{\mathsf{data}_t}, \{\overline{D}[\mathsf{waddr}_t]\}, \mathsf{digest}_t\} \\ \forall t \in [\tau - 1] : &\quad \overline{\mathsf{fetched}}_t, \end{aligned}$$

such that

- For all $t \in [\tau]$: all $\boxed{\mathsf{cpustate}_t}$, $\boxed{\mathsf{data}_t}$ values are correct homomorphic evaluations using the honest evaluator's algorithm based on $\overline{\mathsf{fetched}}_{t-1}$, $\boxed{\mathsf{cpustate}_{t-1}}$, and $\boxed{\mathsf{rk}} := \mathsf{cFHE.Eval}_{hpk,hpk'}(\boxed{K}, \mathsf{ct})$.
- For all $t \in [\tau]$: $i\mathcal{O}(V)(\mathsf{ct}, t, \mathsf{raddr}_t, \mathsf{waddr}_t, \sigma_t) = 1$.
- For all $t \in [\tau - 1]$: the $\overline{\mathsf{fetched}}_t$ value has a valid Merkle proof with respect to digest_t.
- For all $t \in [\tau]$: digest_t are updated from digest_{t-1} correctly according to the purported waddr_t, $\boxed{\mathsf{data}_t}$, and $\{\overline{D}[\mathsf{waddr}_t]\}$ values.

To complete proof of the lemma, it suffices to show that the extracted $\mathsf{raddr}_t, \mathsf{waddr}_t$ values must agree with that of an honest evaluation except with negligible probability. If we can show this, then by the collision resistance of the Merkle tree, every memory read must reflect the last write, or the correct initial value in D if this is the first time a memory cell is fetched from. Therefore, we can conclude that the extracted trace agrees with the honest evaluation trace.

We now show that no polynomial-time time adversary can forge an incorrect (i.e., does not agree with honest evaluation) tuple $(\mathsf{ct}^*, \tau, \mathsf{raddr}'_\tau, \mathsf{waddr}'_\tau, \sigma'_\tau)$ that passes $i\mathcal{O}(V)$'s check except with negligible probability. We show this through an inductive argument, starting from $t = 1$ as the base case, and then in an inductive step, showing it for $t \leq \tau$ assuming it holds for $t \leq \tau - 1$. $\square$



Circuit V'

Hardwired: PRF key K_v', ct^*, z', τ', raddr_τ', and waddr_τ'.

Inputs: ct, τ, raddr_τ, waddr_τ, σ_τ.

- If $\mathsf{ct} = \mathsf{ct}^*$, $\tau = \tau'$, $\mathsf{raddr}_\tau = \mathsf{raddr}_\tau'$, $\mathsf{waddr}_\tau = \mathsf{waddr}_\tau'$, test if $\mathsf{OWF}(\sigma_\tau) = z'$. Output accept if true, reject if false.
- Test if $\mathsf{OWF}(\sigma_\tau) = \mathsf{OWF}(\mathsf{PRF}(K_v', (\mathsf{ct}, \tau, \mathsf{raddr}_\tau, \mathsf{waddr}_\tau))$ where OWF is a one-way function. Output accept if true, reject if false.

Figure 5: Circuit V' to be obfuscated using $i\mathcal{O}$.

Definition C.2. *Let $\mathcal{A}$ denote an adversary. If the adversary can output tuple $(\mathsf{ct}^*, \tau, \mathsf{raddr}_\tau', \mathsf{waddr}_\tau', \sigma_\tau')$ such that $i\mathcal{O}(V)((\mathsf{ct}^*, \mathsf{raddr}_\tau', \mathsf{waddr}_\tau'), \sigma_\tau') = 1$; however, tuple $(\mathsf{raddr}_\tau', \mathsf{waddr}_\tau', \sigma_\tau')$ is not the result of honest evaluation for the τ-th time step of evaluating ciphertext ct^*, then we say that the adversary succeeds in an address forgery for (ct^*, τ).*

Claim C.3 (Base case)**.** *Assume that $\mathsf{di}\mathcal{O}$ is a secure differing-inputs obfuscation, $i\mathcal{O}$ is a secure indistinguishability obfuscation, PRF is a secure and correct puncturable PRF, the one-way function is secure, and that cFHE encryption is perfectly correct.*

For any polynomial-time adversary $\mathcal{A}$ the probability that it succeeds in an address forgery for $(\mathsf{ct}^*, \tau = 1)$ is negligible. As before, the adversary $\mathcal{A}$ is given mpk, sk_{ORAM}, ct^*, the challenge plaintext x_0, and has oracle access to the key generation oracle.

Proof. Let denote the $(\mathsf{raddr}_\tau^*, \mathsf{waddr}_\tau^*, \sigma_\tau^*)$ correct addresses and proof at time τ for the challenge ciphertext ct^*.

If the forgery is for $(\mathsf{raddr}_1^*, \mathsf{waddr}_1^*)$, obviously the only way for V to output 1 is to supply the correct σ_1^*.

Suppose the forgery is for some wrong $(\mathsf{raddr}_1, \mathsf{waddr}_1) \neq (\mathsf{raddr}_1^*, \mathsf{waddr}_1^*)$. The simulator guesses the forged $(\mathsf{raddr}_1', \mathsf{waddr}_1')$. There are only polynomially many such pairs, therefore, the simulator guesses correctly with non-negligible probability.

The rest of the proof folllows in a similar manner as the proof for the NIZK construction by Sahai and Waters [40]. However, note that Sahai and Waters prove selective security, where the adversary commits to the statement before the NIZK's setup. In our case, we can prove adaptive security simply because the space of statements is polynomial in size.

Hybrid A: Compute a punctured version $K_v' := \mathsf{Puncture}(K, (\mathsf{ct}^*, 1, \mathsf{raddr}_1', \mathsf{waddr}_1'))$. Use this K_v' in all C_τ for $\tau \in [T_{\mathsf{ORAM}}]$.

Note that $\tau > 1$, none of the C_τ's will need to query the PRF on the punctured point, since C_τ checks that the input $\tau - 1$ (when incremented) agrees with the hardwired $\tau > 1$. Therefore, due to the fact that the PRF preserves functionality under puncturing, using K_v or K_v' result in functionally equivalent circuits. For C_1, if the cFHE scheme is perfectly correct, it is obvious that C_1 will never query the PRF on the punctured point either. By the security of $\mathsf{di}\mathcal{O}$, Hybrid A is computationally indistinguishable from Hybrid X0.

Hybrid B: Change V to V', where $z' = \mathsf{PRF}(K_v, (\mathsf{ct}^*, 1, \mathsf{raddr}_1', \mathsf{waddr}_1'))$. Please refer to Figure 5. .

It is not hard to see that due to the fact that the puncturable PRF preserves functionality under puncturing, circuit V' is equivalent to V. By the security of $i\mathcal{O}$, $i\mathcal{O}(V')$ is computationally indistinguishable from $i\mathcal{O}(V)$.

Hybrid C: Change that z' to a truly random string. If the puncturable PRF is pseudorandom at the punctured point, it is obvious that Hybrid B is computationally indistinguishable from Hybrid C.

Therefore, Hybrids X0, A, B, C are computationally indistinguishable. Now, if the adversary can forge an incorrect address pair that passes verification in Hybrid C, it can break the one-wayness of the OWF with non-negligible probability. ☐

Claim C.4 (Base case)**.** *Lemma C.1 holds for $\tau = 1$.*

Proof. Due to the proof of Lemma C.1 and the above claim. ☐

Claim C.5 (Inductive step)**.** *Assume that $\mathsf{di}\mathcal{O}$ is a secure differing-inputs obfuscation, $i\mathcal{O}$ is a secure indistinguishability obfuscation, PRF is a secure and correct puncturable PRF, the PCD is a proof of knowledge system, the one-way function is secure, and that cFHE encryption is perfectly correct.*

If for all $t < \tau$, no polynomial-time adversary can succeed in an address forgery for (ct^*, t) with non-negligible probability, and Lemma C.1 holds for all $t < \tau$, then no polynomial-time adversary is able to succeed in an address forgery for (ct^*, τ) with more than negligible probability; and further, Lemma C.1 holds for all $t \in [\tau]$.

Proof. (sketch.) Let denote the $(\mathsf{raddr}^*_\tau, \mathsf{waddr}^*_\tau, \sigma^*)$ correct addresses and proof at time τ for the challenge ciphertext ct^*, obtained from an honest evaluation.

If the forgery is for $(\mathsf{raddr}^*_\tau, \mathsf{waddr}^*_\tau)$, obviously the only way for V to output 1 is to supply the correct σ^*_τ.

Suppose the forgery is for some wrong $(\mathsf{raddr}_\tau, \mathsf{waddr}_\tau) \neq (\mathsf{raddr}^*_1, \mathsf{waddr}^*_1)$. The simulator guesses the forged $(\mathsf{raddr}'_\tau, \mathsf{waddr}'_\tau)$. There are only polynomially many such pairs, therefore, the simulator guesses correctly with non-negligible probability.

Hybrid A. Compute a punctured version $K'_v := \mathsf{Puncture}(K, (\mathsf{ct}^*, \tau, \mathsf{raddr}'_\tau, \mathsf{waddr}'_\tau))$. Use this K'_v in all C_τ for $\tau \in [T_{\mathsf{ORAM}}]$.

To show that Hybrid A is indistinguishable from the real world, we need to show that for any polynomial time adversary who is given mpk and msk, except with negligible probability, it cannot produce output an input in such that C_τ and C'_τ differ in output, where C'_τ denotes the circuit that uses K'_v in place of K_v.

To show this, it suffices to show that if the program reaches the statement where K_v or K'_v is used, then the input to the PRF cannot be the punctured point.

If the program reaches the point where K_v or K'_v is used, then we know that the checks all passed. We can therefore use PCD extractor to extrace entire trace of length $\tau - 1$. By assumption, for $\tau - 1$, the tuple $(\mathsf{ct}^*, \tau-1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \mathsf{cpustate}_{\tau-1})$ must agree with an honest evaluation. By the collision resistance of the Merkle tree, the input value $\overline{\mathsf{fetched}}_{\tau-1}$ must agree with honest evaluation (otherwise a collision can be produced from the honest evaluation trace and the adversary-supplied one). Because of this reason, and further, assuming that the cFHE scheme is perfectly correct, then if the program reaches the point where K_v or K'_v is used, the $(\mathsf{raddr}_\tau, \mathsf{waddr}_\tau)$ computed must be the correct values. Therefore, the punctured point will not be evaluated — otherwise either the security of the PCD or the security of the Merkle tree will be violated.

Hybrid B. Use V' instead of V, where $K'_v := \mathsf{Puncture}(K_v, (\mathsf{ct}^*, \tau, \mathsf{raddr}'_\tau, \mathsf{waddr}'_\tau))$, and $z' = \mathsf{OWF}(\mathsf{PRF}(K, (\mathsf{ct}^*, \tau, (\mathsf{raddr}'_\tau, \mathsf{waddr}'_\tau))))$. If $\mathsf{ct} = \mathsf{ct}^*$, K'_v would use z' in the check.

Hybrid C. Replace z' with a truly random string.

The proofs for the indistinguishability of Hybrid A, B, and C follow in the same manner as the proof of Claim C.3.

In a similar manner as Claim C.3, if an adversary can forge an incorrect address pair that pass the verification in Hybrid C, we can break the one-wayness of the one way function.

Finally, by the proof of Lemma C.1, we obtain that Lemma C.1 holds for $t = \tau$. This completes the proof of the inductive step. $\square$

C.2 Sequence of Hybrids

Hybrid X0. Same as the real world where x_0^* is encrypted during the challenge phase.

Hybrid 0. The experiment modified the way to generate the token as follows. On receiving challenge plaintext x_0^* and x_1^* such that $\mathsf{RAM}(x_0^*) = \mathsf{RAM}(x_1^*)$:

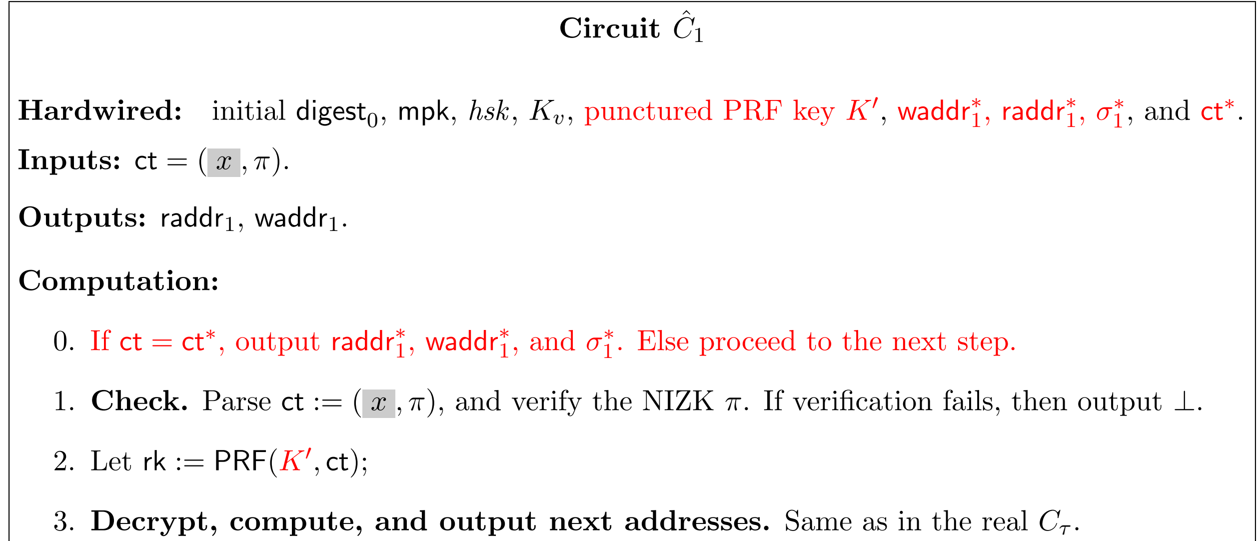
1. Pre-compute the challenge ciphertext (c^*, c'^*) by honestly encrypting x_0^* under hpk and hpk' respectively.

   Run the honest CRS generation algorithm of the NIZK scheme. Let $\mathsf{ct}^* := (c^*, c'^*, \pi)$ where π is an honest NIZK proof for the statement defined by (c^*, c'^*).

2. Pick $\mathsf{rk}^* := \mathsf{PRF}(K, \mathsf{ct}^*)$.

3. Precompute the entire address sequence emitted by the challenge plaintext x_0^* by honestly executing the ORAM on the input x_0^*, using randomness rk^*. This address sequence is referred to as the challenge address sequence, denoted $\mathsf{addresses}^* := (\mathsf{raddr}_\tau^*, \mathsf{waddr}_\tau^*)_{\tau \in [T_{\mathsf{ORAM}}]}$. Also, for each $\tau \in [T_{\mathsf{ORAM}}]$, precompute $\sigma_\tau^* := \mathsf{PRF}(K_v, (\mathsf{ct}^*, \tau, \mathsf{raddr}_\tau^*, \mathsf{waddr}_\tau^*))$.

On a key generation query, compute a punctured PRF key $K' := \mathsf{Puncture}(K, \mathsf{ct}^*)$ such that input ct^* will be punctured. Generate modified $sk_{\mathsf{ORAM}} := \{ K , \mathsf{di}\mathcal{O}(\hat{C}_\tau)\}_{\tau \in [T_{\mathsf{ORAM}}]}$ where the modified circuits $\hat{C}_\tau$'s are as depicted in Figures 6 and 7.



Circuit $\hat{C}_1$

Hardwired: initial digest_0, mpk, hsk, K_v, punctured PRF key K', waddr_1^*, raddr_1^*, σ_1^*, and ct^*.

Inputs: $\mathsf{ct} = (x , \pi)$.

Outputs: raddr_1, waddr_1.

Computation:

0. If $\mathsf{ct} = \mathsf{ct}^*$, output raddr_1^*, waddr_1^*, and σ_1^*. Else proceed to the next step.
1. **Check.** Parse $\mathsf{ct} := (x , \pi)$, and verify the NIZK π. If verification fails, then output $\perp$.
2. Let $\mathsf{rk} := \mathsf{PRF}(K', \mathsf{ct})$;
3. **Decrypt, compute, and output next addresses.** Same as in the real C_τ.

Figure 6: Circuit $\hat{C}_1$ to be obfuscated using $\mathsf{di}\mathcal{O}$.

Hybrid 1. When answering key generation queries, replace K with K' in the sk_{ORAM} given out.

Circuit $\hat{C}_\tau$ for time steps $\tau > 1$

Hardwired: initial digest_0, mpk, hsk, K_v, τ, punctured PRF key K', waddr^*_τ, raddr^*_τ, σ^*_τ, ct^*, and optionally y^* if this is the final step.

Inputs:

- $\{\overline{\mathsf{fetched}}_{\tau-1}\}$, // *carrying Merkle proofs w.r.t.* $\mathsf{digest}_{\tau-1}$
- $\{\mathsf{ct}, \tau - 1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \mathsf{cpustate}_{\tau-1}\}$, // *carrying a PCD proof*

Outputs: raddr_τ, waddr_τ, and optionally y

Computation:

1. **Check trace.** If any of the checks fail, output $\perp$.
   - Check that $\{\mathsf{ct}, \tau - 1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \mathsf{cpustate}_{\tau-1}\}$ has a valid PCD proof.
   - Check that $\{\overline{\mathsf{fetched}}_{\tau-1}\}$ is consistent with the Merkle $\mathsf{digest}_{\tau-1}$ and $\mathsf{raddr}_{\tau-1}$.
   - Check that the input τ agrees with the hardwired expected τ value.
2. If $\mathsf{ct} = \mathsf{ct}^*$, output raddr^*_τ, waddr^*_τ, and σ^*_τ. If this is the final step, also output y^*. Else, let $\mathsf{rk} := \mathsf{PRF}(K', \mathsf{ct})$; and proceed to the next step.
3. **Decrypt, compute, and output next addresses.** Same as in the real $\mathbf{C}_\tau$.

Figure 7: Circuit $\hat{C}_\tau$ to be obfuscated using $\mathsf{di}\mathcal{O}$ for $\tau > 1$. Note that the PCD verifier algorithm for each time step may be different, therefore, the circuit for each time step is different.

Hybrid 2. Replace rk^* with a random string when computing the challenge address sequence.

Hybrid 3. The experiment is the same as Hybrid 2 except, in Step 1, replace the NIZK's real crs with simulated $\widetilde{crs}$, where the statement to be simulated is defined by (c^*, c'^*) — note that this requires selective security. Simulate the NIZK for the challenge ciphertext ct^*.

Hybrid 4. The experiment is the same as Hybrid 3 except, it replaces the second copy of the ciphertext to an encryption of x_1^*. More specifically, in Step 1 above, upon receiving challenge plaintext x_0^* and x_1^*, compute

$$\mathsf{ct}^* := \left(\mathsf{cFHE.Enc}(hpk, x_0^*; \rho), \mathsf{cFHE.Enc}(hpk', x_1^*; \rho'), \widetilde{\pi}\right),$$

where $\widetilde{\pi}$ is a simulated NIZK.

Hybrid 5. When answering key generation queries, create $\mathsf{di}\mathcal{O}_\tau(\hat{C}'_\tau)$ for $\tau \in [T_{\mathsf{ORAM}}]$, where $\hat{C}'_\tau$ is exactly the same as $\hat{C}_\tau$ except that now instead of using hsk to decrypt, $\hat{C}'_\tau$ uses hsk' to decrypt.

Hybrid 6. Replace the first copy of the ciphertext with an encryption of x_1^*. More specifically, in Step 1 above, upon receiving challenge plaintexts x_0^* and x_1^*, compute

$$\mathsf{ct}^* := \left(\mathsf{cFHE.Enc}(hpk, x_1^*; \rho), \mathsf{cFHE.Enc}(hpk', x_1^*; \rho'), \widetilde{\pi}\right),$$

where $\widetilde{\pi}$ is a simulated NIZK.

Hybrid 7. When answering key generation queries, create $\mathsf{di}\mathcal{O}_\tau(\hat{C}_\tau)$ for $\tau \in [T_{\mathsf{ORAM}}]$, i.e., use hsk instead of hsk'.

Hybrid 8. In Step 1 above, run the honest CRS generation algorithm of the NIZK scheme, and compute the NIZK for the challenge ciphertext honestly.

Hybrid 9. In Step 3 above, instead of picking rk^* at random and using x_0^* to compute the challenge address sequence, pick rk^* at random and using x_1^* to compute the challenge address sequence.

Hybrid 10. Instead of picking rk^* at random, pick them based on the true outcome of the PRF at the punctured points, i.e., $\mathsf{rk}^* := \mathsf{PRF}(K, \mathsf{ct}^*)$).

Hybrid 11. Replace occurrences of K' back with K,

Hybrid X1. Switch to using the real-world C_τ in computing the $i\mathcal{O}$. This is the same as the real-world where both encryptions encrypt x_1^*.

Claim C.6. *Assume that the PCD system is a sound proof of knowledge, the one-way function is secure, the Merkle-tree is collision resistant, the* $\mathsf{di}\mathcal{O}$ *is a secure differing-inputs obfuscation, the* $i\mathcal{O}$ *is a secure indistinguishable obfuscation, the PRF is a correct and secure puncturable PRF, and that the* cFHE *encryption scheme is perfectly correct and semantically secure, then Hybrids X0 and 0 are computationally indistinguishable.*

Proof. For C_1 and $\hat{C}_1$, if the puncturable PRF preserves functionality under puncturing, it is obvious that they are functionally equivalent. Therefore, by the security of differing-inputs obfuscation, $\mathsf{di}\mathcal{O}(C_1)$ and $\mathsf{di}\mathcal{O}(\hat{C}_1)$ are computationally indistinguishable.

For $\tau > 1$, by the security of differing-inputs obfuscation, it suffices to show that a polynomial-time adversary cannot find an input in such that $C_\tau(\mathsf{in}) \neq \hat{C}_\tau(\mathsf{in})$ even when the adversary knows all of mpk and msk. If such an adversary exists, the forged input in must contain $\mathsf{ct} = \mathsf{ct}^*$, and all the checks must verify: since if $\mathsf{ct} \neq \mathsf{ct}^*$ by the fact that the puncturable PRF preserves functionality under puncturing, $C_\tau(\mathsf{in})$ and $\hat{C}_\tau(\mathsf{in})$ will yield the same output.

If the checks do not verify, then both $C_\tau(\mathsf{in})$ and $\hat{C}_\tau(\mathsf{in})$ output $\perp$.

If for some input in, its ct satisfies $\mathsf{ct} = \mathsf{ct}^*$, and all the checks verify, According to Lemma C.1 the input tuple $(\mathsf{ct}^*, \tau - 1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \mathsf{cpustate}_{\tau-1})$ must agree with the honest evaluation. We note that we can prove an equivalent of Lemma C.1 for Hybrid 0 too, using the same argument. Further, assuming that the Merkle tree scheme is collision resistant, then the input $\overline{\mathsf{fetched}}_{\tau-1}$ must agree with honest evaluation too. In this case, both $C_\tau(\mathsf{in})$ and $\hat{C}_\tau(\mathsf{in})$ will output the same outcome, raddr^*_τ, waddr^*_τ, and σ^*_τ. Unless the adversary can break Merkle tree or Lemma C.1 it is unable to find an input in such that the two circuits differ. □

Claim C.7. *Assuming that the* cFHE *encryption scheme is semantically secure and perfectly correct, that the puncturable PRF preserves functionality under puncturing, that the NIZK is simulation sound, that the PCD system is a sound proof of knowledge, that the Merkle tree is collision resistant, and that the differing-inputs obfuscator is secure, Hybrid 0 and Hybrid 1 are computationally indistinguishable.*

Proof. This can be proven using a few inner Hybrid games.

- **Hybrid 0a**: When receiving key generation queries, change the second copy of encryption to K'. Hybrid 0a is computationally indistinguishable from Hybrid 0 due to a trivial reduction to the semantic security of the cFHE encryption scheme.

- **Hybrid 0b**: Use key hsk' in the $\mathsf{di}\mathcal{O}$'s instead of hsk.

  We now show that Hybrid 0a is computationally indistinguishable from Hybrid 0b. Due to the security of the $\mathsf{di}\mathcal{O}$, it suffices to show that no adversary knowing mpk and msk can produce an input in such that the two circuits (using hsk and hsk' respectively) differ in output.

  There are two cases:

  – When the input contains ct^*. In this case, neither circuit requires any decryption. They are obviously functionally equivalent.

  – When the input does not contain ct^*. If the input does not survive the checks, then both circuits output $\perp$. If the inputs survive the checks, due to Lemma C.8, using hsk or hsk' in the circuits will give the same outcome except with negligible probability.

- **Hybrid 0c**: Change the first copy of encryption to encrypting K'. This is clearly indistinguishable from Hybrid 0b assuming the semantic security of cFHE encryption.

- **Hybrid 0d**: Use hsk to decrypt inside the $\mathsf{di}\mathcal{O}$'s instead of hsk'.

  We show that Hybrid 0d is computationally indistinguishable from Hybrid 0c. By the security of $\mathsf{di}\mathcal{O}$, it suffices to show that no polynomial-time adversary given msk and mpk can find an input in such that the two circuits differ in output.

If the input $\mathsf{ct} = \mathsf{ct}^*$, no decryption is needed. Otherwise, if input $\mathsf{ct} \neq \mathsf{ct}^*$, by Lemma C.8 (in particular, it is not hard to see that this lemma can be proven for Hybrids 0c and 0d as well), a polynomial-time adversary with knowledge of mpk and msk cannot produce two inputs such that the two circuits differ in output.

Observe also that Hybrid 0d is equivalent to Hybrid 1. □

Lemma C.8. *Assume that the PCD system is a sound proof of knowledge, that the Merkle tree is collision resistant, that the NIZK is simulation sound, and that the* cFHE *scheme is perfectly correct. In Hybrids 0a and 0b, no polynomial time adversary can produce with non-negligible probability a tuple* $\{\mathsf{ct}, \tau-1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}\}$*, and* $\{\overline{\mathsf{fetched}}_{\tau-1}\}$ *such that 1)* $\mathsf{ct} \neq \mathsf{ct}^*$*; 2) the PCD proof for* $\{\mathsf{ct}, \tau-1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}\}$ *verifies; and 3)* $\{\overline{\mathsf{fetched}}_{\tau-1}\}$ *has a Merkle digest consistent with* $\mathsf{digest}_{\tau-1}$*; and 4) the following does NOT hold: the two copies of encryption in* $\boxed{\mathsf{cpustate}_{\tau-1}}$ *decrypt to consistent plaintexts, using hsk and hsk′ respectively, and further,* $\overline{\mathsf{fetched}}_{\tau-1}$ *is either a cleartext value or decrypts to consistent plaintexts using hsk and hsk′ respectively.*

Proof. If a tuple $\{\mathsf{ct}, \tau-1, \mathsf{digest}_{\tau-1}, \mathsf{raddr}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}\}$ has a valid PCD proof, then since PCD is a sound proof of knowledge system, there exists a polynomial-time extractor that can extract an evaluation trace (consistent with the ct, digest_τ, raddr_τ, and $\boxed{\mathsf{cpustate}_\tau}$ values included in the PCD statement),

$$\begin{aligned} \mathsf{ct}, \quad \forall t \in [\tau]: &\quad \{\mathsf{waddr}_t, \mathsf{raddr}_t, \boxed{\mathsf{cpustate}_t}, \boxed{\mathsf{data}_t}, \{\overline{D}[\mathsf{waddr}_t]\}, \mathsf{digest}_t\} \\ \forall t \in [\tau-1]: &\quad \overline{\mathsf{fetched}}_t, \end{aligned}$$

such that

- For all $t \in [\tau]$: all $\boxed{\mathsf{cpustate}_t}, \boxed{\mathsf{data}_t}$ values are correct homomorphic evaluations using the honest evaluator's algorithm based on $\overline{\mathsf{fetched}}_{t-1}$, $\boxed{\mathsf{cpustate}_{t-1}}$, and $\boxed{\mathsf{rk}} := \mathsf{cFHE.Eval}_{hpk,hpk'}(\boxed{(K, K')}, \mathsf{ct})$, where $\boxed{(K, K')}$ denotes that the two copies of encryption encrypt K and K' respectively.
- For all $t \in [\tau]$: $i\mathcal{O}(V)(\mathsf{ct}, t, \mathsf{raddr}_t, \mathsf{waddr}_t, \sigma_t) = 1$.
- For all $t \in [\tau-1]$: the $\overline{\mathsf{fetched}}_t$ value has a valid Merkle proof with respect to digest_t.
- For all $t \in [\tau]$: digest_t are updated from digest_{t-1} correctly according to the purported waddr_t, $\boxed{\mathsf{data}_t}$, and $\{\overline{D}[\mathsf{waddr}_t]\}$ values.

Note that we cannot argue that this evaluation trace must agree with that of an honest evaluation (since we are in the case where $\mathsf{ct} \neq \mathsf{ct}^*$). However, the PCD proof does guarantee that this trace is almost correct, and the only thing the adversary can lie about are the memory addresses. However, every memory fetch result is either the original value of the memory cell in D, or is the previous value (double-encrypted ciphertext) written to a memory cell.

Assume that the cFHE scheme is perfectly correct. Since the puncturable PRF preserves functionality under puncturing, using K' or K for $\mathsf{ct} \neq \mathsf{ct}^*$ result in the same outcome. Now examine the homomorphic evaluation trace. Then if a derived pair of ciphertexts decrypt to different values using hsk and hsk', the initial ct must encrypt two different plaintexts as well. However, the intial ct has a NIZK proof that the plaintexts are consistent. This means that the adversary can forge a NIZK proof.

□

Claim C.9. *Assume that the puncturable PRF is pseudorandom at the punctured point, then Hybrid 1 and Hybrid 2 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim C.10. *Assuming that the NIZK is computationally zero-knowledge, then Hybrid 2 is computationally indistinguishable from Hybrid 3.*

Proof. By a straightforward reduction. □

Claim C.11. *Assuming the encryption scheme is semantically secure, then Hybrids 3 and 4 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim C.12. *Assuming the NIZK is simulation sound, and that the differing-inputs obfuscator is secure, the PCD is a sound proof of knowledge, the Merkle tree is collision resistant, and that the* cFHE *scheme is perfectly correct, then Hybrids 4 and 5 are computationally indistinguishable.*

Proof. By the security of $\mathsf{di}\mathcal{O}$, it suffices to show that no polynomial-time adversary given msk and mpk can find an input in such that the two circuits differ in output.

If the input $\mathsf{ct} = \mathsf{ct}^*$, no decryption is needed. Otherwise, if input $\mathsf{ct} \neq \mathsf{ct}^*$, by Lemma C.8 (in particular, it is not hard to see that this lemma can be proven for Hybrids 4 and 5 as well), a polynomial-time adversary with knowledge of mpk and msk cannot produce two inputs such that the two circuits differ in output. □

Indistinguishability of Hybrids 5 through 8 can be proven in a similar manner as above.

Claim C.13. *Assume that* ORAM *is oblivious RAM as defined in Definition 2.1, then Hybrids 8 and 9 are computationally indistinguishable.*

Proof. In a similar manner as the Proof of Claim D.9 in the $i\mathcal{O}$-based construction. □

Claim C.14. *Assuming that the puncturable PRF satisfies pseudorandomness at punctured points, then Hybrid 9 and Hybrid 10 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim C.15. *Assuming that the* cFHE *encryption scheme is semantically secure, that the puncturable PRF preserves functionality under puncturing, that the NIZK is simulation sound, that the PCD system is a sound proof of knowledge, that the Merkle tree is collision resistant, that the* cFHE *scheme is perfectly correct, and that the differing inputs obfuscator is secure, then Hybrid 10 and Hybrid 11 are computationally indistinguishable.*

Proof. The proof can be accomplished in a similar manner as that of Claim C.7, with the help of a sequence of inner Hybrid games. □

Claim C.16. *Assuming that the puncturable PRF preserves functionality under puncturing, the differing-inputs obfuscator is secure, the Merkle tree is collision resistent, and that the PCD system is a secure proof of knowledge, , then Hybrids 11 and X1 are computationally indistinguishable.*

Proof. Similar to the proof of Claim C.6. □

D RAM-Model Functional Encryption from Indistiguishability Obfuscation

In this Section, we describe an $i\mathcal{O}$-based construction for RAM-model functional encryption, achieving $\tilde{O}((T+n)^2)$ evaluation time. While in the worst-case scenario, this may not be better than converting the RAM to a circuit, our construction still has the advantage of having input-specific running time as Goldwasser et al. defined it for Turing Machines [26, 27]. Note also that achieving input-specific running time for RAM is much more challenging that for Turing Machines [26, 27]. Therefore, our $i\mathcal{O}$-based construction can be considered as a more powerful generalization of the results by Goldwasser et al. [26, 27].

D.1 Construction

Notational convention. We explain our notational conventions below.

$\boxed{\mathsf{var}}$	double-encrypted ciphertext based on cFHE, under public keys hpk and hpk' respectively
$\overline{\mathsf{var}}$	This variable is sometimes double-encrypted and sometimes in cleartext. In particular, initially the memory array D is all in cleartext; however values written to memory will be double-encrypted under hpk and hpk'.

We recall the notation $\mathsf{cFHE.Eval}_{hpk,hpk'}$ as a short-hand to express simultaneously evaluating two copies of the FHE ciphertexts, encrypted under hpk and hpk' respectively. Concretely, we use $\boxed{w} = \mathsf{cFHE.Eval}_{hpk,hpk'}(g(\boxed{v}))$ to denote the homomorphic evaluation of function $g(\cdot)$ on double-encrypted ciphertext $\boxed{v} = (c_v, c'_v)$ to obtain a new double-encrypted ciphertext $\boxed{w} = (c_w, c'_w)$, where $c_v = \mathsf{cFHE.Enc}(hpk, v)$ and $c'_v = \mathsf{cFHE.Enc}(hpk', v))$, and $c_w = \mathsf{cFHE.Eval}(hpk, g(\cdot), c_v)$ and $c'_w = \mathsf{cFHE.Eval}(hpk', g(\cdot), c'_v)$.

Detailed construction. We now describe our $i\mathcal{O}$-based FE-RAM construction. Besides the indistinguishability obfuscation $i\mathcal{O}$ and secure oblivious RAM ORAM, we use an FHE scheme $\mathsf{cFHE}.(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$, a NIZK scheme $\mathsf{NIZK}.(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$, and a puncturable PRF scheme.

Setup. On input 1^λ, compute $(hpk, hsk) \leftarrow \mathsf{cFHE.Gen}(1^\lambda)$, $(hpk', hsk') \leftarrow \mathsf{cFHE.Gen}(1^\lambda)$, and $crs \leftarrow \mathsf{NIZK.Setup}(1^\lambda)$. Set public parameter $\mathsf{mpk} := (hpk, hpk', crs)$, and master secret key $\mathsf{msk} := hsk$.

Encryption. Upon inputting the public parameter mpk and a message x, pick random $\rho, \rho' \in \{0,1\}^\lambda$, and compute $c = \mathsf{cFHE.Enc}(hpk, x; \rho)$, $c' = \mathsf{cFHE.Enc}(hpk', x; \rho')$, then compute a NIZK (denoted π) for the following statement parameterized by (c, c'):

$$\exists x, \rho, \rho' \text{ s.t. } (c = \mathsf{cFHE.Enc}(hpk, x; \rho)) \wedge \big(c' = \mathsf{cFHE.Enc}(hpk', x; \rho')\big)$$

The ciphertext $\mathsf{ct} := (\boxed{x}, \pi)$, where $\boxed{x} := (c, c')$.

Key Generation. Convert the queried $\mathsf{RAM} := D'$ into an ORAM whose initial memory array is $D := D'||\vec{0}$. Let T_{ORAM} denote the maximum run-time of the ORAM.

Sample a random PRF key $K \in \{0,1\}^\lambda$ to embed in the obfuscated next instruction circuits.

Produce T_{ORAM} number of indistinguishable obfuscations, $i\mathcal{O}_1, i\mathcal{O}_2, \ldots, i\mathcal{O}_{T_{\mathsf{ORAM}}}$, for circuits C_1, C_2, ..., $C_{T_{\mathsf{ORAM}}}$ respectively. Please refer to Figure 8 for the description of each C_τ, where $\tau \in [T_{\mathsf{ORAM}}]$.

Define the evaluation trace Tr_τ up to time τ where $1 \le \tau \le T_{\mathsf{ORAM}}$ as:

$$\mathsf{Tr}_\tau := \Big(\mathsf{ct} := (\boxed{x}, \pi), \quad \forall 1 \le t \le \tau : \mathsf{raddr}_t, \mathsf{waddr}_t, \overline{\mathsf{fetched}}_t, \boxed{\mathsf{data}_t}, \boxed{\mathsf{cpustate}_t}\Big)$$

where $\overline{\mathsf{fetched}}_t$ is

- either a cleartext word if the fetched memory location has not been written before; or
- a pair of FHE ciphertexts, i.e., $\overline{\mathsf{fetched}}_t = \boxed{\mathsf{fetched}_t} := (c, c')$ if the fetched memory location has been written before.

Compute $\boxed{K} := \big(\mathsf{cFHE.Enc}(hpk, K), \mathsf{cFHE.Enc}(hpk', K)\big)$.

The token $\mathsf{sk}_{\mathsf{ORAM}}$ for ORAM is defined as $\mathsf{sk}_{\mathsf{ORAM}} := \Big(\boxed{K}, i\mathcal{O}_1, \ldots, i\mathcal{O}_{T_{\mathsf{ORAM}}}\Big)$

Decryption. Initialize $\overline{D} = D$. Compute $\boxed{\mathsf{rk}} := \mathsf{cFHE.Eval}_{hpk,hpk'}(\mathsf{PRF}(\boxed{K}, \mathsf{ct}))$[5].

For $\tau \in [T_{\mathsf{ORAM}}]$,

- **Perform homomorphic evaluation.**
  If $\tau = 1$, use the homomorphic evaluation to obtain:
  $$\Big(\boxed{\mathsf{data}_1}, \boxed{\mathsf{cpustate}_1}\Big) := \mathsf{cFHE.Eval}_{hpk,hpk'}(\textsc{NextIns}(\boxed{x}, \vec{0}, \boxed{\mathsf{rk}}))$$
  Else if $\tau > 1$, use the homomorphic evaluation to obtain:[6]
  $$\Big(\boxed{\mathsf{data}_\tau}, \boxed{\mathsf{cpustate}_\tau}\Big) := \mathsf{cFHE.Eval}_{hpk,hpk'}(\textsc{NextIns}(\overline{\mathsf{fetched}}_{\tau-1}, \boxed{\mathsf{cpustate}_{\tau-1}}, \boxed{\mathsf{rk}}))$$
  Here we ignore the encrypted addresses output by the homomorphic evaluations of the NextIns.
- **Use $i\mathcal{O}$ to evaluate next addresses.** Collect the trace so far $\mathsf{Tr}_{\tau-1}$ and compute
  $$(\mathsf{waddr}_\tau, \mathsf{raddr}_\tau) := i\mathcal{O}_\tau(\mathsf{Tr}_{\tau-1})$$
  If this is the final step of the evaluation, an output y is also output by the obfuscated next instruction circuit $i\mathcal{O}_\tau$.
- **Perform memory read and write:**
  $$\overline{\mathsf{fetched}}_\tau := \overline{D}[\mathsf{raddr}_\tau], \qquad \overline{D}[\mathsf{waddr}_\tau] := \boxed{\mathsf{data}_\tau}$$

[5]For technical reasons in the proof, here we assume that PRF is the circuit that first checks whether input $\boxed{K}$ is a punctured key (which can be indicated by adding an indicator bit to the key), and if not, use real PRF evaluation algorithm; if so, use the punctured PRF evaluation algorithm.

[6]We use the notation $\mathsf{cFHE.Eval}_{hpk,hpk'}(C(\boxed{\mathsf{encrypted_var}}, \mathsf{cleartext_var}))$ as a short-hand to express simutaneously performing homomorphic evaluation of circuit C on two copies of FHE ciphertexts, encrypted under hpk and hpk' respectively. The resulting ciphertexts are double-encrypted under hpk and hpk'. Also, note that sometimes, one input to the NextIns function is a constant, and the other is a double-encrypted ciphertext. Other times, both are double-encrypted ciphertexts. In either case, the homomorphic evaluation happens for both copies encrypted under hpk and hpk' respectively.

Circuit C_τ

Hardwired: initial memory D, mpk, hsk, PRF key K, K.

Inputs:

$$\mathsf{Tr}_{\tau-1} := \Big(\mathsf{ct} := (x, \pi),\quad \forall 1 \le t \le \tau - 1 : \mathsf{raddr}_t, \mathsf{waddr}_t, \overline{\mathsf{fetched}}_t, \mathsf{data}_t, \mathsf{cpustate}_t\Big)$$

Outputs: raddr_τ, waddr_τ, (and optionally y)

Computation:

0. Compute $\mathsf{rk} := \mathsf{PRF}(K, \mathsf{ct})$.

1. **Check trace: computation.** For each $t < \tau$:
   - Check that $\mathsf{cpustate}_t$ and data_t are obtained from correctly performing homomorphic evaluation of NextIns over $\mathsf{cpustate}_{t-1}$ (or cleartext $\mathsf{cpustate}_0 := \vec{0}$ if $t = 1$), $\overline{\mathsf{fetched}}_{t-1}$ (or ct if $t = 1$), and $\mathsf{rk} := \mathsf{cFHE.Eval}_{hpk,hpk'}(\mathsf{PRF}(K, \mathsf{ct}))$. If the check fails, output $\perp$.
   - Use hsk to decrypt and obtain $\mathsf{cpustate}_{t-1}$ and $\mathsf{fetched}_{t-1}$, and check that waddr_t and raddr_t in the trace $\mathsf{Tr}_{\tau-1}$ are the correct outcome of evaluating NextIns on $\mathsf{cpustate}_{t-1}$, $\mathsf{fetched}_{t-1}$, and rk, where rk is computed at the beginning of C_τ.

2. **Check trace: memory consistency**[a]**.**

   Parse $\mathsf{ct} := (x, \pi)$ where ct is part of the trace $\mathsf{Tr}_{\tau-1}$. Verify that the NIZK π is valid.

   For each $t < \tau$, perform memory consistency check:
   - If this is the first time the memory location is read in $\mathsf{Tr}_{\tau-1}$, verify that $\overline{\mathsf{fetched}}_t = D[\mathsf{raddr}_t]$ where D is hard-coded in this circuit. If any check fails, output $\perp$.
   - Else, if memory location raddr_t has been written previously in $\mathsf{Tr}_{\tau-1}$, check that $\overline{\mathsf{fetched}}_t$ is equal to the last written encrypted value in $\mathsf{Tr}_{\tau-1}$.

3. **Decrypt, compute, and output next addresses.** If the above checks all verify, use hsk to decrypt to obtain $\mathsf{fetched}_{\tau-1}, \mathsf{cpustate}_{\tau-1}$, then compute the circuit NextIns($\mathsf{fetched}_{\tau-1}, \mathsf{cpustate}_{\tau-1}, \mathsf{rk}$), and output the addresses raddr_τ and waddr_τ. If $\tau = T_{\mathsf{ORAM}}$, i.e., this is the final step of the ORAM, also output the outcome y.

[a] Ben-Sasson *et al.* [6], Bitansky *et al.* [9], and Ben-Sasson *et al.* [7] show that this step can be done with an $O(t \log t)$-sized circuit.

Figure 8: Circuit C_τ to be obfuscated using $i\mathcal{O}$.

D.2 Cost

Our IND-secure construction achieves $\mathsf{poly}(\lambda)$ ciphertext size and $\tilde{O}((n+T)^2)\mathsf{poly}(\lambda)$ decryption time, since there are $T_{\mathsf{ORAM}} = \tilde{O}(n+T)$ obfuscated circuits to run, where each circuit takes at most $O(n + T_{\mathsf{ORAM}} \log T_{\mathsf{ORAM}}) \cdot \mathsf{poly}(\lambda)$. To achieve this, we require an $i\mathcal{O}$ scheme with linear blowup. Gordon et al. [32] point out that the $i\mathcal{O}$ construction by Garg et al. [20] satisfies this property. Assume $T = \Omega(n)$, this means we need $\tilde{O}(T^2)\mathsf{poly}(\lambda)$ time — roughly equivalent in cost to known results for circuits, as the known conversion from RAM to circuit when $T = \Omega(n)$ incurs $O(nT)$ overhead [41].

For the simulation-secure setting, our cost is preserved (same as the IND-secure setting), for a scheme secure under a single key query. To support q key queries, the ciphertext size blows up by a factor of q due to the use of De Caro et al. 's compiler [19]. Note also that it has been shown that in the standard model, it is impossible to achieve fully SIM-secure FE with succinct ciphertexts [19]. In contrast, our IND-secure scheme can support unbounded polynomially many key queries without blowups in ciphertext size.

D.3 Security Proof

We can show the above construction is a secure functional encryption in the RAM model.

Theorem D.1. *Assuming that $i\mathcal{O}$ is a secure indistinguishability obfuscator,* ORAM *is a secure oblivious RAM scheme as defined in Definition 2.1,* cFHE *is a FHE scheme for circuits with perfect correctness and semantic security,* NIZK *is a statistically simulation sound NIZK scheme, and* PRF *is a correct and secure puncturable PRF, then the FE-RAM construction in Appendix D is selectively IND-secure as defined in Definition B.2.*

By a standard argument of complexity leveraging, we can achieve the (full, as opposed to selective) indistinguishability security from the selective security at a cost of stronger complexity assumptions. Then we can achieve a simulation-based security using the trapdoor circuit technique from the work of De Caro et al. [19], who showed how to construct a (selective/full) simulation secure FE from a (selective/full) indistinguishability secure one. Also, the construction supports multiple key queries. Thus, we are able to achieve the following corollary:

Corollary D.2 (Theorem D.1 + (complexity leveraging) + [19])**.** *Assume that $i\mathcal{O}$ is a secure indistinguishability obfuscator,* ORAM *is a secure oblivious RAM scheme as defined in Definition 2.1,* cFHE *is FHE scheme for circuits with perfect correctness and semantic security,* NIZK *is a statistically simulation sound NIZK scheme, and* PRF *is a correct and secure puncturable PRF. In addition, assume that these primitives remain secure when all polynomial-sized adversaries have sub-exponentially small advantages. Then the above FE-RAM can be made fully SIM-secure as defined in Definition B.1.*

We begin the proof of Theorem D.1 with a sequence of hybrids:

Hybrid X0. This is exactly the real world where both encryptions encrypt x_0^*.

Hybrid 0. The experiment modified the way to generate the token as follows. On receiving challenge plaintext x_0^* and x_1^* such that $\mathsf{RAM}(x_0^*) = \mathsf{RAM}(x_1^*)$:

1. Pre-compute the challenge ciphertext (c^*, c'^*) by honestly encrypting x_0^* under hpk and hpk' respectively.

Circuit $\hat{C}_\tau$

Hardwired: initial memory D, K, mpk, hsk, punctured PRF key K', ct^*, rk^*, $\{\mathsf{raddr}^*_t, \mathsf{waddr}^*_t\}_t$, and optionally the challenge outcome y^* if $\tau = T_{\mathsf{ORAM}}$.

Inputs: $\mathsf{Tr}_{\tau-1}$

Outputs: raddr_τ, waddr_τ, (and optionally y)

Computation:

0. Compute $\mathsf{rk} := \mathsf{PRF}(K', \mathsf{ct})$ if $\mathsf{ct} \neq \mathsf{ct}^*$; else $\mathsf{rk} := \mathsf{rk}^*$.
1. **Check trace: computation.** For each $t < \tau$:
   - Check that $\mathsf{cpustate}_t$ and data_t are obtained from correctly performing homomorphic evaluation of NextIns over $\mathsf{cpustate}_{t-1}$ (or cleartext $\mathsf{cpustate}_0 = \vec{0}$ if $t = 1$), $\mathsf{fetched}_{t-1}$ (or ct if $t = 1$), and $\mathsf{rk} := \mathsf{cFHE.Eval}_{hpk,hpk'}(\mathsf{PRF}(K, \mathsf{ct}))$. If the check fails, output $\perp$.
   - If $\mathsf{ct} \neq \mathsf{ct}^*$: use hsk to decrypt and obtain $\mathsf{cpustate}_{t-1}$ and $\mathsf{fetched}_{t-1}$ and check that waddr_t and raddr_t in the trace $\mathsf{Tr}_{\tau-1}$ are the correct outcome of evaluating NextIns on $\mathsf{cpustate}_{t-1}$, $\mathsf{fetched}_{t-1}$, and rk.
     Else if $\mathsf{ct} = \mathsf{ct}^*$, check if the sequence of $\{\mathsf{raddr}_t, \mathsf{waddr}_t\}_t$ in the trace $\mathsf{Tr}_{\tau-1}$ agree with hardwired challenge address sequence $\{\mathsf{raddr}^*_t, \mathsf{waddr}^*_t\}_t$.
2. **Check trace: memory consistency.** Same as in the real C_τ.
3. If the ciphertext ct in Tr_τ is equal to ct^*, simply output $(\mathsf{raddr}^*_\tau, \mathsf{waddr}^*_\tau)$. If $\tau = T_{\mathsf{ORAM}}$, also output y^*.

   If the ciphertext ct in Tr_τ is not equal to ct^*, proceed to the next step.
4. **Decrypt, compute, and output next addresses.** Same as in the real C_τ.

Figure 9: Circuit $\hat{C}_\tau$ to be obfuscated using $i\mathcal{O}$.

Run the honest CRS generation algorithm of the NIZK scheme. Let $\mathsf{ct}^* := (c^*, c'^*, \pi)$ where π is an honest NIZK proof for the statement defined by (c^*, c'^*).

2. Pick $\mathsf{rk}^* := \mathsf{PRF}(K, \mathsf{ct}^*)$.
3. Precompute the entire address sequence emitted by the challenge plaintext x_0^* by honestly executing the ORAM on the input x_0^*, using randomness rk^*. This address sequence is referred to as the challenge address sequence, denoted $\mathsf{addresses}^* := (\mathsf{raddr}^*_\tau, \mathsf{waddr}^*_\tau)_{\tau \in [T_{\mathsf{ORAM}}]}$.
4. Now, assuming that the addresses output from the $i\mathcal{O}$'s in each step of the evaluation is exactly the above challenge addresses, follow the honest evaluator's algorithm to precompute the correct trace Tr^* for ct^*.

On a key generation query, compute a punctured PRF key $K' := \mathsf{Puncture}(K, \mathsf{ct}^*)$ such that input ct^* will be punctured. Generate modified $\mathsf{sk}_{\mathsf{ORAM}} := \left(K, \{i\mathcal{O}(\hat{C}_\tau)\}_{\tau \in [T_{\mathsf{ORAM}}]} \right)$ where the modified circuits $\hat{C}_\tau$'s are as depicted in Figure 9.

Hybrid 1. Replace any occurrence of K with K', specifically, in the $\mathsf{sk}_{\mathsf{ORAM}}$ for any key generation query, and in the hardwired values in all the $i\mathcal{O}$'s.

Hybrid 2. The experiment is the same as Hybrid 1 except in Step 2 above, pick truly random rk^* for all $\tau \in [T_{\mathsf{ORAM}}]$.

Hybrid 3. The experiment is the same as Hybrid 2 except, in Step 1, replace the NIZK's real crs with simulated $\widetilde{crs}$, where the statement to be simulated is defined by (c^*, c'^*) — note that this requires selective security. Simulate the NIZK for the challenge ciphertext ct^*.

Hybrid 4. The experiment is the same as Hybrid 3 except, it replaces the second copy of the ciphertext to an encryption of x_1^*. More specifically, in Step 1 above, upon receiving challenge plaintext x_0^* and x_1^*, compute

$$\mathsf{ct}^* := \left(\mathsf{cFHE.Enc}(hpk, x_0^*; \rho), \mathsf{cFHE.Enc}(hpk', x_1^*; \rho'), \widetilde{\pi} \right),$$

where $\widetilde{\pi}$ is a simulated NIZK.

Hybrid 5. When answering key generation queries, create $i\mathcal{O}_\tau(\hat{C}'_\tau)$ for $\tau \in [T_{\mathsf{ORAM}}]$, where $\hat{C}'_\tau$ is exactly the same as $\hat{C}_\tau$ except that now instead of using hsk to decrypt, $\hat{C}'_\tau$ uses hsk' to decrypt.

Hybrid 6. Replace the first copy of the ciphertext with an encryption of x_1^*. More specifically, in Step 1 above, upon receiving challenge plaintexts x_0^* and x_1^*, compute

$$\mathsf{ct}^* := \left(\mathsf{cFHE.Enc}(hpk, x_1^*; \rho), \mathsf{cFHE.Enc}(hpk', x_1^*; \rho'), \widetilde{\pi} \right),$$

where $\widetilde{\pi}$ is a simulated NIZK.

Hybrid 7. When answering key generation queries, create $i\mathcal{O}_\tau(\hat{C}_\tau)$ for $\tau \in [T_{\mathsf{ORAM}}]$, i.e., use hsk instead of hsk'.

Hybrid 8. In Step 1 above, run the honest CRS generation algorithm of the NIZK scheme, and compute the NIZK for the challenge ciphertext honestly.

Hybrid 9. In Step 3 above, instead of picking rk^* at random and using x_0^* to compute the challenge address sequence, pick rk^* at random and using x_1^* to compute the challenge address sequence.

Hybrid 10. Instead of picking rk^* at random, pick them based on the true outcome of the PRF at the punctured points, i.e., $\mathsf{rk}^* := \mathsf{PRF}(K, \mathsf{ct}^*)$).

Hybrid 11. Replace occurrences of K' back with K,

Hybrid X1. Switch to using the real-world C_τ in computing the $i\mathcal{O}$. This is the same as the real-world where both encryptions encrypt x_1^*.

We are going to show all the adjacent hybrids are indistinguishable by the following claims:

Claim D.3. *Assuming that the puncturable PRF preserves functionality under puncturing, and that the indistinguishable obfuscator is secure, then Hybrids X0 and 0 are computationally indistinguishable.*

Proof. By the security of the indistinguishability obfuscator, it suffices to show that the circuits C_τ and $\hat{C}_\tau$ are functionally equivalent. There are three cases:

- When a trace $\mathsf{Tr}_{\tau-1}$ does not survive the computation and memory consistency checks. In this case, both C_τ and $\hat{C}_\tau$ output $\perp$.
- For a trace $\mathsf{Tr}_{\tau-1}$ (surviving the checks) that contains ct^*, line 3 will be invoked in $\hat{C}_\tau$. But since the trace survives the checks, it must be a correct trace derived by honest evaluation. Therefore, the output raddr_τ^* and waddr_τ^* values (and optionally y^*) are exactly the same as what C_τ would output.
- For a trace $\mathsf{Tr}_{\tau-1}$ (surviving the checks) that does not contain ct^*, the only difference between $\hat{C}_\tau$ and C_τ is whether K or K' is used. Since the puncturable PRF preserves functionality under puncturing, $\hat{C}_\tau$ and C_τ give exactly the same output.

$\square$

Claim D.4. *Assuming that the* cFHE *encryption scheme is semantically secure, that the puncturable PRF preserves functionality under puncturing, that the NIZK is statistically simulation sound, that the* cFHE *scheme is perfectly correct, and that the indistinguishable obfuscator is secure, then Hybrid 0 and Hybrid 1 are computationally indistinguishable.*

Proof. This can be proven using a few inner Hybrid games.

- **Hybrid 0a**: When receiving key generation queries, change the second copy of encryption to K'. Hybrid 0a is computationally indistinguishable from Hybrid 0 due to a trivial reduction to the semantic security of the cFHE encryption scheme.

- **Hybrid 0b**: Use key hsk' in the $i\mathcal{O}$'s instead of hsk.

  We now show that Hybrid 0a is computationally indistinguishable from Hybrid 0b. Due to the security of the $i\mathcal{O}$, it suffices to show that whether using hsk or hsk' to decrypt inside the $i\mathcal{O}$ result in functionally equivalent circuits. There are two cases:

  – When the input trace $\mathsf{Tr}_{\tau-1}$ contains ct^*. In this case, neither circuit requires any decryption. They are obviously functionally equivalent.

  – When the input trace $\mathsf{Tr}_{\tau-1}$ does not contain ct^*. Due to the statistically simulation soundness of the NIZK, and the fact that the puncturable PRF preserves functionality under puncturing, and that the cFHE scheme is perfectly correct, it is not hard to see that any trace that will survive the checks must decrypt to consistent plaintexts no matter whether hsk or hsk' is used in decryption. Therefore, the two circuits are functionally equivalent.

- **Hybrid 0c**: Change the first copy of encryption to encrypting K'. This is clearly indistinguishable from Hybrid 0b assuming the semantic security of cFHE encryption.

- **Hybrid 0d**: Use hsk to decrypt inside the $i\mathcal{O}$'s instead of hsk'. It is not hard to see that due to the statistical simulation soundness of NIZK, and the perfect correctness of cFHE encryption scheme, whether or not we use hsk or hsk' to decrypt are functionally equivalent. Therefore, by the security of $i\mathcal{O}$, Hybrid 0c and Hybrid 0d are indistinguishable.

Observe also that Hybrid 0d is equivalent to Hybrid 1. □

Claim D.5. *Assuming that the puncturable PRF satisfies pseudo-randomness at punctured points, then Hybrids 1 and 2 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim D.6. *Assuming the NIZK is computational zero-knowledge, then Hybrids 2 and 3 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim D.7. *Assuming the encryption scheme is semantically secure, then Hybrids 3 and 4 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim D.8. *Assuming the NIZK is statistically simulation sound, and the security of the indistinguishable obfuscator, then Hybrids 4 and 5 are computationally indistinguishable.*

Proof. By the security of the indistinguishability obfuscator, it suffices to show that the two circuits $\hat{C}_\tau$ and $\hat{C'}_\tau$ are functionally equivalent on all inputs. There are three cases:

- When the input trace $\mathsf{Tr}_{\tau-1}$ fail the computation or memory consistency checks. In this case, both circuits output $\perp$.

- When the input trace $\mathsf{Tr}_{\tau-1}$ pass all the computation and memory consistency checks, and contain the challenge ct^*. In this case both circuits output the same since in both circuits line 3 is triggered.

- When the input trace $\mathsf{Tr}_{\tau-1}$ pass all the computation and memory consistency checks, and does not contain the challenge ct^*. In this case, due to the statistical simulation soundness of the NIZK, both copies of encryption in the challenge ciphertext must encrypt the same plaintext (otherwise we can easily construct an adversary to break the statistical simulation soundness of NIZK). Therefore, using either decryption key hsk or hsk' to decrypt result in the same output.

□

Using similar arguments as above, we can show that Hybrids 5-8 are indistinguishable.

Claim D.9. *Assuming that the ORAM is secure by Definition 2.1, then Hybrids 8 and 9 are computationally indistinguishable.*

Proof. The only difference from Hybrid 8 and 9 comes from the computation of the challenge address sequence which is computed at the beginning of the game (Step 3).

Define the following distributions where randomness comes from choice of rk^*: $\mathsf{addresses}(\mathsf{ORAM}, x; \mathsf{rk}^*)$ and $\mathsf{addresses}(\mathsf{ORAM}, y; \mathsf{rk}^*)$.

By the security of ORAM,

$$\mathsf{addresses}(\mathsf{ORAM}, x_0; \mathsf{rk}^*) \equiv \mathsf{Sim}() \equiv \mathsf{addresses}(\mathsf{ORAM}, x_1; \mathsf{rk}^*)$$

Therefore the $\hat{C}_\tau$ to be obfuscated in Hybrid 8 and the $\hat{C}_\tau$ in Hybrid 9 are indistinguishable even in the clear. Thus, their obfuscated versions are indistinguishable, and so are the hybrids. □

Claim D.10. *Assuming that the puncturable PRF satisfies pseudorandomness at punctured points, then Hybrid 9 and Hybrid 10 are computationally indistinguishable.*

Proof. By a straightforward reduction. □

Claim D.11. *Assuming that the* cFHE *encryption scheme is semantically secure, that the puncturable PRF preserves functionality under puncturing, that the NIZK is statistically simulation sound, that the* cFHE *scheme is perfectly correct, and that the indistinguishable obfuscator is secure, then Hybrid 10 and Hybrid 11 are computationally indistinguishable.*

Proof. The proof can be accomplished in a similar manner as that of Claim D.4, with the help of a sequence of inner Hybrid games. □

Claim D.12. *Assume that the puncturable PRF preserves functionality under puncturing (perfectly), and that the indistinguishable obfuscator is secure. Then Hybrid 11 and Hybrid X1 are computationally indistinguishable.*

Proof. Similar to the proof of Claim D.3. □

E RAM-Model Functional Encryption from VBB Obfuscation

E.1 Predictive-Memory ORAM

In our VBB-based construction, we not only need ORAM, but a special type of ORAM called predictive-memory ORAM (or predictive ORAM for short)[7], first defined by Lu and Ostrovsky [36].

[7] Instead of requiring the predictive-memory property, we can alternatively use a Merkle hash tree to ensure freshness of memory reads. However, using a predictive ORAM makes our presentation simpler.

Roughly speaking a predictive ORAM is an ORAM for which the CPU can decide when each read location was last accessed (with a small $\mathsf{cpustate}$). Below we formally define predictive ORAM. While Lu and Ostrovsky show that the original ORAM construction by Goldreich and Ostrovsky is a predictive ORAM, we show something a bit stronger, that any ORAM scheme can be transformed into a predictive ORAM incurring only logarithmic overhead (see Appendix G.5).

In predictive ORAM, the NEXTINS circuit additionally outputs the last write time $\mathsf{lastwritten}_t$ for the next raddr emitted as defined below:

$$(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t, \mathsf{lastwritten}_t) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}, \mathsf{rk}\right)$$

E.2 Function Privacy for Functional Encryption Schemes

We formally define function privacy below.

Definition E.1 (Function Privacy). *We say an FE scheme* FE *for RAM programs computing a functionality class* $\mathcal{F}$ *is* function private *if, for all* $\mathsf{RAM}_1, \ldots, \mathsf{RAM}_q$ *for some polynomial* q*, for all* PPT *adversaries* $\mathcal{A}$*, there exists a simulator* $\mathcal{S}$ *such that*

$$\left|\Pr[\mathcal{A}(\mathsf{mpk}, \mathsf{sk}_{\mathsf{RAM}_1}, \ldots, \mathsf{sk}_{\mathsf{RAM}_q}) = 1] - \Pr[\mathcal{S}^{\mathsf{RAM}_1, \ldots, \mathsf{RAM}_q}(1^\lambda) = 1]\right| < \mathsf{negl}(\lambda),$$

for some negligible function $\mathsf{negl}(\cdot)$.

Remark E.2. *Our definition of function privacy is related to, but distinct from, the notion of composable obfuscation of [8], which requires that any attacker who has access to many obfuscated programs can be simulated (even without knowledge of the functions computed by each program). The key difference is the existence of the public parameters* mpk *in our definition, which can be shared among all the* $\mathsf{sk}_{\mathsf{RAM}_i}$*'s. Thus, the function privacy is our setting is weaker than achieving composable obfuscation, and can be achieved by the standard obfuscation.*

E.3 Intuition

A straightforward but insecure idea is to create a virtual blackbox obfuscation $\mathcal{O}$(NEXTINS) of the ORAM's next instruction circuit denoted NEXTINS (omitting the subscript ORAM for simplicity) that would decrypt the current encrypted CPU state, encrypted variable fetched from memory, and compute an encryption of the next CPU state, the value to write to memory, and cleartext memory read and write addresses. However, the issue is that a malicious evaluator may feed this obfuscated circuit $\mathcal{O}$(NEXTINS) with arbitrary inputs to try to obtain additional information.

To enforce honest evaluation, our idea is to sign all legitimate inputs, such that our obfuscated NEXTINS circuit will only output meaningful values if these signatures all verify, i.e., the inputs are legitimate. Otherwise, the obfuscated NEXTINS circuit will simply output $\perp$. More concretely, in addition to encrypting the initial compiled $\mathsf{ORAM} := D$, the key generator also signs every encrypted memory location $D[i]$. Further, before the obfuscator $\mathcal{O}$ emits any value, including raddr_t, waddr_t, $\mathsf{cpustate}_t$, and data_t, all of these values will be signed inside $\mathcal{O}$(NEXTINS) too.

We still need to prevent mix-and-match in two ways:

- To prevent the adversary from mixing and matching values from different time steps, all the signatures will be *tagged with the current time step.*
- To prevent the adversary from mixing and matching values from different tokens, all the signatures will be *tagged with a* nonce *that is unique for each token.*

In this way, we essentially enforce that given a token and a ciphertext, there is only a unique evaluation trace that is legitimate, and will be accepted by the obfuscated NEXTINS circuit, $\mathcal{O}$(NEXTINS).

E.4 Construction

Notational conventions. In the following scheme description, we will use the following notational conventions.

$\boxed{\mathsf{var}}$	an encrypted variable
$\{\mathsf{var}\}$, $\{\boxed{\mathsf{var}}\}$	a variable or encrypted variable, attached with a signature

Construction. Let $\mathcal{O}$ be a virtual black-box circuit obfuscator, $\mathcal{E}$ be a CCA-secure public-key encryption scheme, Σ be a *deterministic* signature scheme existentially unforgeable under adaptive chosen-message attack, and PRF be a pseudorandom function.

We construct the FE-RAM scheme $\mathsf{FE} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ as:

Setup. $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}(1^\lambda)$:

Run $(ek, dk) \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$, and $(sk, vk) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$. Choose PRF keys k and K.

Create a VBB obfuscation $\mathcal{O}(C^{\mathrm{FE}}_{\textsc{NextIns}})$ for the circuit $C^{\mathrm{FE}}_{\textsc{NextIns}}$ as described in Figure 10.

Set $\mathsf{mpk} := (ek, \mathcal{O}(C^{\mathrm{FE}}_{\textsc{NextIns}}))$, and $\mathsf{msk} := (sk, k, K)$.

Key Generation. $\mathsf{sk}_{\mathsf{RAM}} \leftarrow \mathsf{KeyGen}(\mathsf{msk}, \mathsf{RAM})$:

Upon receiving RAM, choose uniform $\mathsf{nonce} \in \{0,1\}^\lambda$, and define $\mathsf{RAM}_{\mathsf{nonce}} = D'$, where

$$\mathsf{RAM}_{\mathsf{nonce}}(x, ((v_1, \mathsf{nonce}_1), \dots, (v_q, \mathsf{nonce}_q)), b) = \begin{cases} \mathsf{RAM}(x) & \text{if } b = 0 \\ v_i & \text{if } b = 1 \wedge \mathsf{nonce} = \mathsf{nonce}_i. \end{cases}$$

Next, let $\mathsf{ORAM} := D$, be the compiled ORAM counterpart for $\mathsf{RAM}_{\mathsf{nonce}}$, where $D = D' || \vec{0}$.

For $i = 1$ to n_{ORAM}, compute $\left\{\boxed{D}[i]\right\} := \left(\boxed{D}[i],\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{D}[i], (0, i, \mathsf{nonce}, \text{“mem”}))\right)$ where $\boxed{D}[i] := D[i] \oplus \mathsf{PRF}_k(0, i, \mathsf{nonce}, \text{“mem”})$.

The token for RAM is then the following tuple:

$$\mathsf{sk}_{\mathsf{RAM}} := \left(\mathsf{nonce}; \forall i \in [n_{\mathsf{ORAM}}], \left\{\boxed{D}[i]\right\}\right)$$

Encryption . (See footnote [8]):

Compute $\boxed{x} \leftarrow \mathsf{Enc}(\mathsf{mpk}, x)$:

Run $\boxed{x} \leftarrow \mathcal{E}.\mathsf{Enc}_{ek}(x, ((0^\lambda, 0^\lambda), \dots, (0^\lambda, 0^\lambda)), 0)$.

Decryption. $y \leftarrow \mathsf{Dec}(\mathsf{sk}_{\mathsf{RAM}}, \boxed{x})$:

Given $\mathsf{sk}_{\mathsf{RAM}}$ and a ciphertext $\boxed{x}$, parse $\mathsf{sk}_{\mathsf{RAM}} := \left(\mathsf{nonce}, \left(\left\{\boxed{D}[i]\right\}\right)_{i \in [n_{\mathsf{ORAM}}]}\right)$. Then, for $t = 1, 2, \dots, T_{\mathsf{ORAM}}$:

[8] We give a multi-key construction directly, where the ciphertext size is linear in q, the number of key queries. De Caro et al. [19] show the impossibililty of succinct simulation secure functional encryption under multiple key queries, we note that this lower bound still holds even assuming virtual black-box obfuscation.

$C^{\mathrm{FE}}_{\textsc{NextIns}}$:

Hardwired state: ek, dk, vk, sk, k, K

Inputs:

$$t, \mathsf{nonce}, \left[\begin{array}{rl} \text{if } t > 1: & \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{ \boxed{\mathsf{fetched}_{t-1}} \right\}, \left\{ \boxed{\mathsf{cpustate}_{t-1}} \right\}, \left\{ \boxed{\mathsf{rk}} \right\} \\ \text{else (if } t = 1): & \boxed{x}, \end{array} \right]$$

Outputs: $\{\mathsf{raddr}_t\}$, $\{\mathsf{lastwritten}_t\}$, waddr_t, $\left\{ \boxed{\mathsf{data}_t} \right\}$, $\left\{ \boxed{\mathsf{cpustate}_t} \right\}$.
For $t = T_{\mathsf{ORAM}}$, the final outcome y is also output.

Circuit description:

1. **Verify.** If any verification fails, output $\perp$. If $t > 1$, verify that[a]
   - $\Sigma.\mathsf{Verify}_{vk}\left(\left\{ \boxed{\mathsf{cpustate}_{t-1}} \right\}, (t-1, \mathsf{nonce}, \text{“cpu”})\right) = 1,$
   - $\Sigma.\mathsf{Verify}_{vk}\left(\left\{ \boxed{\mathsf{rk}} \right\}, (t-1, \mathsf{nonce}, \text{“prf”})\right) = 1,$
   - $\Sigma.\mathsf{Verify}_{vk}\left(\{\mathsf{raddr}_{t-1}\}, (t-1, \mathsf{nonce}, \text{“raddr”})\right) = 1,$
   - $\Sigma.\mathsf{Verify}_{vk}\left(\{\mathsf{lastwritten}_{t-1}\}, (t-1, \mathsf{nonce}, \text{“lastwritten”})\right) = 1,$
   - $\Sigma.\mathsf{Verify}_{vk}\left(\left\{ \boxed{\mathsf{fetched}_{t-1}} \right\}, (\mathsf{lastwritten}_{t-1}, \mathsf{raddr}_{t-1}, \mathsf{nonce}, \text{“mem”})\right) = 1.$

2. **Decrypt.** If $t = 1$, set $\mathsf{cpustate}_0 := \vec{0}$, and decrypt $\mathsf{fetched}_0 := \mathcal{E}.\mathsf{Dec}_{dk}(\boxed{x})$, and compute $\mathsf{rk} := \mathsf{PRF}_K(\boxed{x})$, and $\boxed{\mathsf{rk}} := \mathsf{rk} \oplus \mathsf{PRF}_k(\mathsf{nonce}, \text{“prf seed”})$.

   Else if $t > 1$, compute $\mathsf{cpustate}_{t-1} := \boxed{\mathsf{cpustate}_{t-1}} \oplus \mathsf{PRF}_k(t-1, \mathsf{nonce}, \text{“cpu”})$, decrypt $\mathsf{fetched}_{t-1} := \boxed{\mathsf{fetched}_{t-1}} \oplus \mathsf{PRF}_k(\mathsf{lastwritten}_{t-1}, \mathsf{raddr}_{t-1}, \mathsf{nonce}, \text{“mem”})$, and decrypt $\mathsf{rk} := \boxed{\mathsf{rk}} \oplus \mathsf{PRF}_k(\mathsf{nonce}, \text{“prf seed”})$.

3. **Compute.**

   $(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t, \mathsf{lastwritten}_t, y) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}, \mathsf{rk}\right)$

   where the outcome y is only output if $t = T_{\mathsf{ORAM}}$.

4. **Encrypt.** Compute $\boxed{\mathsf{data}_t} := \mathsf{data}_t \oplus \mathsf{PRF}_k(t, \mathsf{waddr}_t, \mathsf{nonce}, \text{“mem”})$, and $\boxed{\mathsf{cpustate}_t} := \mathsf{cpustate}_t \oplus \mathsf{PRF}_k(t, \mathsf{nonce}, \text{“cpu”})$.

5. **Sign.** $\{\mathsf{raddr}_t\} := (\mathsf{raddr}_t,\ \Sigma.\mathsf{Sign}_{sk}(\mathsf{raddr}_t, (t, \mathsf{nonce}, \text{“raddr”}))),$

   $\{\mathsf{lastwritten}_t\} := (\mathsf{lastwritten}_t,\ \Sigma.\mathsf{Sign}_{sk}(\mathsf{lastwritten}_t, (t, \mathsf{nonce}, \text{“lastwritten”})))$

   $\left\{ \boxed{\mathsf{data}_t} \right\} := \left(\boxed{\mathsf{data}_t},\ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{data}_t}, (t, \mathsf{waddr}_t, \mathsf{nonce}, \text{“mem”})) \right)$

   $\left\{ \boxed{\mathsf{cpustate}_t} \right\} := \left(\boxed{\mathsf{cpustate}_t},\ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{cpustate}_t}, (t, \mathsf{nonce}, \text{“cpu”})) \right)$

   $\left\{ \boxed{\mathsf{rk}} \right\} := \left(\boxed{\mathsf{rk}},\ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{rk}}, (t, \mathsf{nonce}, \text{“prf”})) \right)$

[a] We overload notation $\Sigma.\mathsf{Verify}_{vk}(\{\mathsf{var}\}, \mathsf{tag})$ to mean the following: 1) Parse $\{\mathsf{var}\}$ as $\{\mathsf{var}\} := (\mathsf{var}, \sigma)$; and 2) Compute $\Sigma.\mathsf{Verify}_{vk}(\mathsf{var}||\mathsf{tag}, \sigma)$. In other words, $\Sigma.\mathsf{Verify}_{vk}(\{\mathsf{var}\}, \mathsf{tag})$ verifies that the variable var is properly signed with the specified tag.

Figure 10: $C^{\mathrm{FE}}_{\textsc{NextIns}}$ to be obfuscated using VBB obfuscation.

$$\left(\{\mathsf{raddr}_t\}, \{\mathsf{lastwritten}_t\}, \mathsf{waddr}_t, \left\{\boxed{\mathsf{data}_t}\right\}, \left\{\boxed{\mathsf{cpustate}_t}\right\}\right)$$
$$:= \begin{cases} \mathcal{O}(C^{\text{FE}}_{\text{NextIns}})\,(t=1, \mathsf{nonce}, \boxed{x}\,) \\ \mathcal{O}(C^{\text{FE}}_{\text{NextIns}})\left(t>1, \mathsf{nonce}, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{fetched}_{t-1}}\right\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right) \end{cases}$$
$$\left\{\boxed{\mathsf{fetched}_t}\right\} := \{\,\boxed{D}\,[\mathsf{raddr}_t]\}$$
$$\{\,\boxed{D}\,[\mathsf{waddr}_t]\} := \left\{\boxed{\mathsf{data}_t}\right\}$$

The T_{ORAM}-th run of $\mathcal{O}(C^{\text{FE}}_{\text{NextIns}})$ also yields y. Output y as the decryption of $\boxed{x}$ under $\mathsf{sk}_{\mathsf{RAM}}$.

E.5 Security Proof

We can show the above construction is a secure functional encryption in the RAM model.

Theorem E.3. *Assume* ORAM *is a predictive oblivious RAM secure by Definition 2.1, $\mathcal{O}$ is a VBB obfuscator, Σ is an unforgeable signature scheme,* PRF *is a secure PRF, $\mathcal{E}$ is a CCA-secure public-key encryption scheme, then the above constructed FE-RAM scheme is simulation secure against non-adaptive adversaries, under multiple non-adaptive key queries and a single message.*

Handling adaptive key queries. We can use a similar technique of the trapdoor circuit as the work [19] to handle adaptive key queries. The idea is that the simulator embeds the answers to the adaptive key queries into the tokens. We can use the technique in a straightforward way. For clarity of presentation, we present the non-adaptive version for exposition of our main ideas.

To prove that our scheme achieves the simulation-based security, we need to construct a simulator $\mathcal{S}$ so that for all adversary $\mathcal{A}$, the real experiment is computationally indistinguishable from the ideal experiment. The simulator $\mathcal{S}$ interacts with an adversary $\mathcal{A}$ and operates as follows:

- The simulator $\mathcal{S}$ carries out the setup and key generation honestly for the adversary $\mathcal{A}$.
- When the adversary $\mathcal{A}$ makes a ciphertext query on input x, the simulator $\mathcal{S}$ now is provided with $|x|$ and the values $y_i = \mathsf{RAM}_i(x)$ for all $i \in [q]$. Now the simulator generates a fake ciphertext $\mathsf{ct} \leftarrow \mathcal{E}.\mathsf{Enc}_{ek}(0^\lambda, ((y_1, \mathsf{nonce}_1), \ldots, (y_q, \mathsf{nonce}_q)), 1)$, for the adversary $\mathcal{A}$. Here nonce_i is the nonce used for RAM_i in the key generation.

Based on the above simulator $\mathcal{S}$, we can obtain the ideal experiment $\text{Ideal}_{\mathcal{A},\mathcal{S}}$. In addition, based on the FE construction, we can easily obtain the real experiment $\text{Real}_{\mathcal{A}}$. Given any key query for RAM and challenge plaintext x, the distributions in the both experiments are:

$$\mathsf{Dist}_{\text{Ideal}} = \left(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \tilde{\mathsf{ct}}, \mathcal{O}(C^{\text{FE}}_{\text{NextIns}})\right)$$

$$\mathsf{Dist}_{\text{Real}} = \left(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \mathsf{ct}, \mathcal{O}(C^{\text{FE}}_{\text{NextIns}})\right).$$

Next, we modify the real and ideal experiments respectively and obtain Real' and Ideal'.

Real′ and Ideal′. We define Real' and Ideal' the same as Real and Ideal except the experiments exclude the obfuscated circuits. (We give the distinguisher an oracle O^{FE}_1 instead, providing black box access to the circuit $C^{\text{FE}}_{\text{NextIns}}$.) In particular,

$$\mathsf{Dist}_{\text{Ideal}'} = \left(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \tilde{\mathsf{ct}}\right)$$

$$\mathsf{Dist}_{\text{Real}'} = \left(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \mathsf{ct}\right).$$

Lemma E.4. *Assume $\mathcal{O}$ is a VBB obfuscator. If there exists a* RAM*, a message x, a* PPT *algorithm $\mathcal{D}$ and some non-negligible $\delta()$ such that* $\Pr[\mathcal{D}(\mathsf{Dist}_{\text{REAL}}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\text{IDEAL}}) = 1] > \delta(\lambda)$*, then there exists another* PPT *algorithm $\hat{\mathcal{D}}$ so that* $\Pr[\hat{\mathcal{D}}^{O_1^{\text{FE}}}(\mathsf{Dist}_{\text{REAL}'}) = 1] - \Pr[\hat{\mathcal{D}}^{O_1^{\text{FE}}}(\mathsf{Dist}_{\text{IDEAL}'}) = 1] > \delta(\lambda) - \mathsf{negl}(\lambda)$*.*

Proof. Let $\alpha = \big(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \mathsf{ct}\big)$ and let $\mathcal{D}_\alpha$ denote algorithm $\mathcal{D}$ with α hardwired. Based on the assumption that $\mathcal{O}$ is VBB obfuscator, for algorithm $\mathcal{D}_\alpha$, there exists a VBB simulator Sim so that

$$\Big|\Pr[\mathcal{D}_\alpha(\mathcal{O}(C^{\text{FE}}_{\text{NextIns}})) = 1] - \Pr[\mathsf{Sim}^{O_1^{\text{FE}}}() = 1]\Big| < \mathsf{negl}(\lambda)$$

Now we can define $\hat{\mathcal{D}}$ as follows: upon input α, and oracle access to O_1^{FE}, $\hat{\mathcal{D}}$ internally runs Sim, providing Sim oracle access to O_1^{FE}; $\hat{\mathcal{D}}$ returns the bit that Sim returns. We can obtain

$$\begin{aligned}
&\Pr[\hat{\mathcal{D}}^{O_1^{\text{FE}}}(\mathsf{Dist}_{\text{REAL}'}) = 1] - \Pr[\hat{\mathcal{D}}^{O_1^{\text{FE}}}(\mathsf{Dist}_{\text{IDEAL}'}) = 1]\\
&= \Pr[\mathsf{Sim}^{O_1^{\text{FE}}}() = 1 \mid \text{REAL}'] - \Pr[\mathsf{Sim}^{O_1^{\text{FE}}}() = 1 \mid \text{IDEAL}']\\
&> \Pr[\mathcal{D}_\alpha(\mathcal{O}(C^{\text{FE}}_{\text{NextIns}})) = 1 \mid \text{REAL}] - \Pr[\mathcal{D}_\alpha(\mathcal{O}(C^{\text{FE}}_{\text{NextIns}})) = 1 \mid \text{IDEAL}] - \mathsf{negl}(\lambda)\\
&= \Pr[\mathcal{D}(\mathsf{Dist}_{\text{REAL}}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\text{IDEAL}}) = 1] - \mathsf{negl}(\lambda)\\
&> \delta(\lambda) - \mathsf{negl}(\lambda)
\end{aligned}$$

This completes the proof of the lemma. □

Modified oracle O_2^{fe}. Here we consider a modified oracle O_2^{FE}. Here, whenever there is an oracle call for $t = 1$, decrypts the input, and precomputes all the "expected" inputs for all future time steps and remembers them in a table Γ. Later, if there are queries for all $t > 1$, the circuit looks into the table Γ, to find if it is one of the expected inputs. If so, output the correct outputs for this time step. Else, output $\perp$. The above modified oracle O_2^{FE} is based on honest setup and key generation in the real experiment. Similarly, we can define modified O_2^{FE} based on the setup and key generation in the ideal experiment.

Lemma E.5. *If Σ is unforgeable, then for any* PPT *algorithm $\mathcal{D}$, $\mathcal{D}$ cannot distinguish oracle O_1^{FE} from O_2^{FE} via black box queries, i.e.* $\Pr[\mathcal{D}^{O_1^{\text{FE}}}(1^\lambda) = 1] \approx \Pr[\mathcal{D}^{O_2^{\text{FE}}}(1^\lambda) = 1]$.

Proof. The two oracles are the same except the following bad event occurs: the input vector can be verified but such input vector is not computed by the circuit based on any input at $t = 1$. Note that the computation by the circuit is deterministic and it has been decided by the input at $t = 1$ (and other initial state). For any vector which is not generated by the circuit, if it can be verified, then we immediately obtain a forgery for the underlying digital signature. Under the assumption the signature scheme is unforgeable, the event only occurs with negligible probability. Otherwise, we can use $\mathcal{D}$ to make a forgery to the signature scheme. □

O_2^{FE}:

Initial state: ek, dk, vk, sk, k, K;
Store all information in the key generation stage for each RAM_j, i.e., for all $j \in [q]$, store $\mathsf{RAM}_j.\mathsf{nonce}$, $\mathsf{RAM}_j.\left(D[i]\right)_{i\in[n_{\mathsf{ORAM}}]}$, and $\mathsf{RAM}_j.\left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n_{\mathsf{ORAM}}]}$.

Inputs: same as in $C^{\mathrm{FE}}_{\textsc{NextIns}}$
Outputs: same as in $C^{\mathrm{FE}}_{\textsc{NextIns}}$

Oracle description:

Different from $\Sigma.\mathsf{Verify}_{vk}()$ used in $C^{\mathrm{FE}}_{\textsc{NextIns}}$, here the circuit carries out verification through "book-keeping" mechanism.

If $t = 1$, then the input vector is $(\mathsf{nonce}, \boxed{x})$, carry out the following:

- Compute $x \leftarrow \mathcal{E}.\mathsf{Dec}_{dk}(\boxed{x})$, and set $\mathsf{fetched}_0 := x$; set $\mathsf{cpustate}_0 := \vec{0}$; compute $\mathsf{rk} := \mathsf{PRF}_K(\boxed{x})$, and $\boxed{\mathsf{rk}} := \mathsf{rk} \oplus \mathsf{PRF}_k(\mathsf{nonce}, \text{"prf seed"})$.
- For $t = 1, 2, \ldots, T_{\mathsf{ORAM}}$:
  $(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t, \mathsf{lastwritten}_t, y) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}, \mathsf{rk}\right)$
  $\mathsf{fetched}_t := D[\mathsf{raddr}_t]$
  $D[\mathsf{waddr}_t] := \mathsf{data}_t$
  where the outcome y is only output if $t = T_{\mathsf{ORAM}}$
- For $t = 1, 2, \ldots, T_{\mathsf{ORAM}}$:
  $\boxed{\mathsf{data}_t} := \mathsf{data}_t \oplus \mathsf{PRF}_k(t, \mathsf{waddr}_t, \mathsf{nonce}, \text{"mem"})$.
  $\boxed{\mathsf{cpustate}_t} := \mathsf{cpustate}_t \oplus \mathsf{PRF}_k(t, \mathsf{nonce}, \text{"cpu"})$.
- For $t = 1, 2, \ldots, T_{\mathsf{ORAM}}$:
  $\{\mathsf{raddr}_t\} := (\mathsf{raddr}_t,\ \ \Sigma.\mathsf{Sign}_{sk}(\mathsf{raddr}_t, (t, \mathsf{nonce}, \text{"raddr"}))),$
  $\{\mathsf{lastwritten}_t\} := (\mathsf{lastwritten}_t,\ \ \Sigma.\mathsf{Sign}_{sk}(\mathsf{lastwritten}_t, (t, \mathsf{nonce}, \text{"lastwritten"})))$
  $\left\{\boxed{\mathsf{data}_t}\right\} := \left(\boxed{\mathsf{data}_t},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{data}_t}, (t, \mathsf{waddr}_t, \mathsf{nonce}, \text{"mem"}))\right)$
  $\left\{\boxed{\mathsf{cpustate}_t}\right\} := \left(\boxed{\mathsf{cpustate}_t},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{cpustate}_t}, (t, \mathsf{nonce}, \text{"cpu"}))\right)$
  $\left\{\boxed{\mathsf{rk}}\right\} := \left(\boxed{\mathsf{rk}},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{rk}}, (t, \mathsf{nonce}, \text{"prf"}))\right)$
- Create a table Γ_x as follows.
  For $t = 1, 2, \ldots, T_{\mathsf{ORAM}}$,
  in $\Gamma_x[t]$, input vector Input_t and output vector Output_t are stored, where $\mathsf{Input}_t = \left(\mathsf{nonce}, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{fetched}_{t-1}}\right\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$,
  and $\mathsf{Output}_t = \left(\{\mathsf{raddr}_t\}, \{\mathsf{lastwritten}_t\}, \mathsf{waddr}_t, \left\{\boxed{\mathsf{data}_t}\right\}, \left\{\boxed{\mathsf{cpustate}_t}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$. Note that for $t = T_{\mathsf{ORAM}}$, the final outcome y is also included in Output_T.
- Finally return Output_1 as the output.

If $t > 1$, if the input vector has not been stored in any table, then return $\perp$. Otherwise, if the input vector has been stored in a table, then return Output_t as the output.

Real$''$ and Ideal$''$. We define REAL'' and IDEAL'' the same as REAL' and IDEAL' except that in the key generation, for each RAM_j, instead of using PRF to generate the one time pad, use random strings to encrypt $\mathsf{cpustate}_0$ and $(D[i])_{i\in[n_{\mathsf{ORAM}}]}$. The key generation information will be stored as initial state in oracle O_3^{FE}. Concretely, generate $\mathsf{sk}_{\mathsf{RAM}_j} := \mathsf{RAM}_j.\left(\mathsf{nonce}, \left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n_{\mathsf{ORAM}}]}\right)$, where and $\boxed{D}[i] := D[i] \oplus R_i$, for randomly chosen R_i's. The distributions of the two experiments are

$$\mathsf{Dist}_{\text{IDEAL}''} = \left(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \tilde{\mathsf{ct}}\right)$$
$$\mathsf{Dist}_{\text{REAL}''} = \left(ek, \{\mathsf{sk}_{\mathsf{RAM}_j}\}_{j\in[q]}, x, \mathsf{ct}\right).$$

Lemma E.6. *If* PRF *is secure, for any* PPT $\mathcal{D}$, $|\Pr[\mathcal{D}(\mathsf{Dist}_{\text{REAL}'}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\text{REAL}''}) = 1]| < \mathsf{negl}(\lambda)$ *and* $|\Pr[\mathcal{D}(\mathsf{Dist}_{\text{IDEAL}'}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\text{IDEAL}''}) = 1]| < \mathsf{negl}(\lambda)$.

Proof. This follows directly from the security of the PRF. □

Modified oracle $O_3^{\mathbf{fe}}$. O_3^{FE} is the same as O_2^{FE} except that in Encrypt step, a random string is used as one time pad.

O_3^{FE}:

Initial state: secret keys dk, sk, and a random string as one time pad.
Store all information in the key generation stage for each RAM_j, i.e., for all $j \in [q]$, store $\mathsf{RAM}_j.\mathsf{nonce}$, $\mathsf{RAM}_j.(D[i])_{i\in[n_{\mathsf{ORAM}}]}$, and $\mathsf{RAM}_j.\left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n_{\mathsf{ORAM}}]}$. Here random strings are used as one time pad for encrypting.

Inputs: same as in $C^{\text{FE}}_{\text{NEXTINS}}$
Outputs: same as in $C^{\text{FE}}_{\text{NEXTINS}}$

Oracle description:

Same as that in oracle O_2^{FE} except that in Encrypt steps, instead of using PRF to generate a pseudorandom string as the one-time pad, here a random string is used as the one time pad.

Similarly, we can define modified O_3^{FE} based on the setup and key generation in the ideal experiment.

Lemma E.7. *If* PRF *is secure, for any* PPT *algorithm* $\mathcal{D}$, $\mathcal{D}$ *cannot distinguish oracle* O_2^{FE} *from* O_3^{FE} *via black box queries, i.e.* $\Pr[\mathcal{D}^{O_2^{\text{FE}}}(1^\lambda) = 1] \approx \Pr[\mathcal{D}^{O_3^{\text{FE}}}(1^\lambda) = 1]$.

Proof. This follows directly from the security of the PRF. □

Modified oracle $O_4^{\mathbf{fe}}$. O_4^{FE} is the same as O_3^{FE} except on the query corresponding to $t = 1$, instead of using real memory addresses, the new circuit will use simulated memory addresses generated by predictive ORAM simulator Sim'.

O_4^{FE}:

Intial state: same as that in O_3^{FE}

Inputs: same as in $C^{\mathrm{FE}}_{\textsc{NextIns}}$
Outputs: same as in $C^{\mathrm{FE}}_{\textsc{NextIns}}$

Oracle description:

Same as that in oracle O_3^{FE} except that for each input value x, the corresponding table Γ_x is created in a different way. Instead of generating addresses honestly, now the addresses are simulated by running the predictive ORAM simulator $\mathsf{Sim}'()$, i.e., computing $\left(\{\mathsf{waddr}_t, \mathsf{raddr}_t\}_{t\in[T_{\mathsf{ORAM}}]}\right) \leftarrow \mathsf{Sim}'()$. The circuit then computes $\mathsf{lastwritten}_t$ for each $t \in [T_{\mathsf{ORAM}}]$.

Similarly, we can define modified O_4^{FE} based on the setup and key generation in the ideal experiment.

Lemma E.8. *Assume* ORAM *is secure as defined in Defintion 2.1. Then, for any* PPT *algorithm $\mathcal{D}$, $\mathcal{D}$ cannot distinguish oracle O_3^{FE} from O_4^{FE} via black box queries, i.e.* $\Pr[\mathcal{D}^{O_3^{\mathrm{FE}}}(1^\lambda) = 1] \approx \Pr[\mathcal{D}^{O_4^{\mathrm{FE}}}(1^\lambda) = 1]$.

Proof. This directly follows by the transformation introduced in Defintion 2.1, Assume there is an algorithm $\mathcal{D}$ who can distinguish O_3^{FE} from O_4^{FE} via black box access, then we can construct another algorithm $\mathcal{D}'$ who can break the computational security of the corresponding ORAM. $\mathcal{D}'$ internally simulates $\mathcal{D}$ and the oracle except that the addresses are generated by an external algorithm either by $\mathsf{addresses}(\mathsf{ORAM}, \cdot, \mathsf{rk})$ or by $\mathsf{Sim}'()$. If the external algorithm is $\mathsf{addresses}()$, then $\mathcal{D}$'s view is the same as that with O_3^{FE}; and if the external algorithm is Sim', then $\mathcal{D}$'s view is the same as that with O_4^{FE}. Since $\mathcal{D}$ can distinguish the two experiments with non-negligible probability, that means $\mathcal{D}'$ can distinguish the real addressees generated by $\mathsf{addresses}()$ from the simulated addresses produced by $\mathsf{Sim}'()$ with non-negligible probability. That means $\mathcal{D}'$ can break the computational security of ORAM, which contracts to the assumption. □

Lemma E.9. *Assume $\mathcal{E}$ is CCA-secure PKE, then no* PPT *distinguisher with oracle access to O_4^{FE} can distinguish* $\textsc{Real}''$ *from* $\textsc{Ideal}''$*.*

Proof. Assume there is an algorithm $\hat{\mathcal{D}}$ who can distinguish $\textsc{Real}''$ from $\textsc{Ideal}''$ with oracle access to O_4^{FE}. We now construct another algorithm $\mathcal{D}'$ who can break the security of the underlying CCA encryption scheme $\mathcal{E}$. The construction of $\mathcal{D}'$ is as follows. $\mathcal{D}'$ internally simulates $\hat{\mathcal{D}}$ and O_4^{FE} with the following:

- Now $\mathcal{D}'$ is provided the encryption key ek and decryption oracle, but not the decryption key dk. In oracle O_4^{FE}, the decryption key dk is not included. For any decryption queries of a CCA ciphertext, $\mathcal{D}'$ will send it to his own decryption oracle to get the corresponding plaintext.

- Whenever $\hat{\mathcal{D}}$ provides x in the challenging stage, $\mathcal{D}'$ sends x' to his own challenger to get the ciphertext ct, where $x' = (x, ((0^\lambda, 0^\lambda), \ldots, (0^\lambda, 0^\lambda)), 0)$ or $x' = (0^\lambda, ((y_1, \mathsf{nonce}_1), \ldots, (y_q, \mathsf{nonce}_q)), 1)$, $y_j = \mathsf{RAM}_j(x)$ for $j \in [q]$.

We note that, when $\mathcal{D}'$ uses $x' = (x, ((0^\lambda, 0^\lambda), \ldots, (0^\lambda, 0^\lambda)), 0)$, the view of the internally simulated $\hat{\mathcal{D}}$ is the same as that of $\textsc{Real}''$; and when $\mathcal{D}'$ uses $x' = (0^\lambda, ((y_1, \mathsf{nonce}_1), \ldots, (y_q, \mathsf{nonce}_q)), 1)$, the view of the internally simulated $\hat{\mathcal{D}}$ is the same as that of $\textsc{Ideal}''$. Since $\mathcal{D}^{O_4^{\text{FE}}}$ can distinguish $\textsc{Real}''$ from $\textsc{Ideal}''$ with non-negligible probability, and $\mathcal{D}'$ faithfully simulates O_4^{FE}, the algorithm $\mathcal{D}'$ is a successful attacker on the CCA encryption scheme $\mathcal{E}$. □

We can now complete the proof of security.

Proof of Theorem E.3. From Lemma E.4, assume that there exists a PPT algorithm $\mathcal{D}$ which can distinguish $\textsc{Real}_{\mathcal{A}}$ from $\textsc{Ideal}_{\mathcal{A}}$ with non-negligible probability, then we can construct a PPT algorithm $\hat{\mathcal{D}}$ to distinguish $\textsc{Real}''$ from $\textsc{Ideal}''$ with non-negligible probability. Then from Lemmas E.5, E.6, E.7, and E.8, we know $\hat{\mathcal{D}}^{O_4^{\text{FE}}}$ distinguishes $\textsc{Real}''$ from $\textsc{Ideal}''$. This is a contradiction to Lemma E.9. That means, the real experiment and the ideal experiment are not distinguishable, which completes the proof. □

A Remark on Function Privacy. Our construction achieves the definition of function privacy. Intuitively this follows from the definition of VBB obfuscation, using a similar argument to Lemma E.4. The reason is that the public parameters pp contain a single obfuscated universal circuit, while the functions computed by RAM_i are stored as encrypted code on the database, which can only be accessed using the hardwired decryption key of the obfuscated universal circuit. The only remaining function-dependent information in the view of an attacker is the input-specific running time. We can hide this as well by padding the running time of all computation to a common upper bound. Thus, an attacker on function privacy can be used to construct an attacker on the virtual black-box property of the obfuscated universal circuit.

F RAM-Model Fully Homomorphic Encryption from VBB Obfuscation

F.1 Model

Let $\mathsf{RAM} := D$ denote a random access machine. The program text f can be regarded as part of memory array D. A fully homomorphic encryption for RAM programs (FHE-RAM) consists of a suite of algorithms $\mathsf{FHE} = \mathsf{FHE}.(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ defined as follows.

- **Setup:** $\mathsf{FHE.Setup}(1^\lambda)$ is a PPT algorithm that takes as input a security parameter 1^λ and outputs a pair of master public and secret keys $(\mathsf{hpk}, \mathsf{hsk})$.
- **Encryption:** $\mathsf{FHE.Enc}(\mathsf{hpk}, x)$ is a PPT algorithm that takes as input the master public key hpk and a message x and outputs a ciphertext ct.
- **Decryption:** $\mathsf{FHE.Dec}(\mathsf{hsk}, \mathsf{ct})$ is a deterministic algorithm that takes as input the secret key hsk and ciphertext ct and outputs a message x.
- **Evaluation:** $\mathsf{FHE.Eval}(\mathsf{hpk}, \mathsf{RAM}, \mathsf{ct}_1, \mathsf{ct}_2)$ is a deterministic algorithm that takes as input the evaluation key hpk, a RAM program RAM, two ciphertexts $\mathsf{ct}_1, \mathsf{ct}_2$, and outputs a ciphertext ct.

Correctness The scheme described above is correct if and only if for arbitrary RAM, for every sufficiently large security parameter λ, for every input x_1, x_2, we have:

$$\Pr[\mathsf{FHE.Dec}(\mathsf{hsk}, c) \neq \mathsf{RAM}(x_1, x_2)] = \mathsf{negl}(\lambda)$$

where $c \leftarrow \mathsf{FHE.Eval}(\mathsf{hpk}, \mathsf{RAM}, c_1, c_2)$, $c_1 \leftarrow \mathsf{FHE.Enc}(\mathsf{hpk}, x_1)$, $c_2 \leftarrow \mathsf{FHE.Enc}(\mathsf{hpk}, x_2)$ and $(\mathsf{hpk}, \mathsf{hsk}) \leftarrow \mathsf{FHE.Setup}(1^\lambda)$.

Definition F.1. *We say a homomorphic encryption scheme is semantically secure if for* PPT *distinguisher $\mathcal{D}$ and any two messages x_0, x_1, we have:*

$$|\Pr[\mathcal{D}(\mathsf{hpk}, \mathsf{FHE.Enc}(\mathsf{hpk}, x_0)) = 1] - \Pr[\mathcal{D}(\mathsf{hpk}, \mathsf{FHE.Enc}(\mathsf{hpk}, x_1)) = 1]| < \mathsf{negl}(\lambda)$$

for some negligible function $\mathsf{negl}(\cdot)$.

F.2 Intuition

Our idea is to encrypt a master key K under a circuit FHE scheme in the public parameters, and have the evaluator homomorphically generate a (homomorphically encrypted) PRF key specific to this RAM as well as the inputed ciphertexts during the evaluation. To do this, the evaluator computes a Merkle digest of the initial memory contents, and homomorphically generates a PRF key denoted $\boxed{\mathsf{rk}}$ based on this digest as well as the inputed ciphertexts – this operation will also be checked later inside the obfuscation[9].

To prevent mix-and-match, the obfuscator will sign all intermediate outcomes (encrypted or non-encrypted) along with the Merkle digest of the RAM. While memory writes during the RAM's execution will be encrypted and signed, the initial memory contains non-encrypted data. As a result, every time a memory is read for the first time (before it is ever written), the obfuscation checks that the read is consistent with the declared Merkle digest.

F.3 Construction

Notational convention. We define the following notational conventions for clarity.

$\boxed{\mathsf{var}}$	encryption of the variable var under key *chpk*.
$\overline{\mathsf{var}}$	This variable is sometimes encrypted and sometimes in cleartext. In particular, initially the memory array D is all in cleartext; however values written to memory will be encrypted (and signed).
$\{\mathsf{var}\}$ or $\{\boxed{\mathsf{var}}\}$	We use $\{\}$ to denote either that the variable has a Merkle proof or signature.

Let $\mathsf{cFHE}.(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ be a fully homomorphic encryption scheme for circuits, $\mathsf{Merkle}.(\mathsf{Hash}, \mathsf{Verify})$ be a collision-resistant Merkle hash tree scheme, $\mathcal{O}$ be a virtual black-box circuit obfuscator, $\mathcal{E}.(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a CPA-secure public-key encryption scheme, $\Sigma.(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ be a *deterministic* signature scheme existentially unforgeable under adaptive chosen-message attack, and PRF be a pseudorandom function. We construct the RAM-model fully homomorphic encryption scheme $\mathsf{FHE} = \mathsf{FHE}.(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval})$ as:

[9] Alternatively, one could also just perform this operation inside the obfuscation.

$C^{\text{FHE}}_{\textsc{NextIns}}$:

Initial hardcoded state: $chpk, chsk, ek, dk, sk, vk, k, \boxed{K}$

Inputs:

$$t,\ \mathsf{digest},\ \left[\begin{array}{rl} \text{if } t>1: & \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \{\widetilde{\mathsf{fetched}}_{t-1}\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\} \\ \text{else (if } t=1): & \{\boxed{x_1}\}, \{\boxed{x_2}\}, \end{array}\right]$$

Outputs: $\{\mathsf{raddr}_t\}$, $\{\mathsf{lastwritten}_t\}$, waddr_t, $\left\{\boxed{\mathsf{data}_t}\right\}$, $\left\{\boxed{\mathsf{cpustate}_t}\right\}$, $\left\{\boxed{\mathsf{rk}}\right\}$. For $t = T_{\mathsf{ORAM}}$, the final encrypted outcome $\boxed{y}$ is also output.

Circuit description:

1. **Verify.** If any of the following verification fails, immediately output $\perp$.
   If $t = 1$, verify that $\{\boxed{x_1}\}$ and $\{\boxed{x_2}\}$ carry valid Merkle proofs w.r.t digest. If so, compute $\boxed{\mathsf{rk}} = \mathsf{cFHE.Eval}_{chpk}(G, \boxed{K})$, where $G(\cdot) \triangleq \mathsf{PRF}_{(\cdot)}(\mathsf{digest})$.
   Else if $t > 1$, verify that
   - $\Sigma.\mathsf{Verify}_{vk}\left(\left\{\boxed{\mathsf{rk}}\right\}, (t-1, \mathsf{digest}, \text{"prf"})\right) = 1$,
   - $\Sigma.\mathsf{Verify}_{vk}\left(\left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, (t-1, \mathsf{digest}, \text{"cpu"})\right) = 1$,
   - $\Sigma.\mathsf{Verify}_{vk}\left(\{\mathsf{raddr}_{t-1}\}, (t-1, \mathsf{digest}, \text{"raddr"})\right) = 1$,
   - $\Sigma.\mathsf{Verify}_{vk}\left(\{\mathsf{lastwritten}_{t-1}\}, (t-1, \mathsf{digest}, \text{"lastwritten"})\right) = 1$,
   - If $\mathsf{lastwritten}_{t-1} > 0$: parse $\{\widetilde{\mathsf{fetched}}_{t-1}\} = \{\boxed{\mathsf{fetched}_{t-1}}\}$, i.e., the memory variable is encrypted and signed; verify that $\Sigma.\mathsf{Verify}_{vk}\left(\left\{\boxed{\mathsf{fetched}_{t-1}}\right\}, (\mathsf{lastwritten}_{t-1}, \mathsf{raddr}_{t-1}, \mathsf{digest}, \text{"mem"})\right) = 1$ Otherwise, if $\mathsf{lastwritten}_{t-1} = 0$: parse $\{\widetilde{\mathsf{fetched}}_{t-1}\} = (\mathsf{fetched}_{t-1}, \sigma)$ and verify that $\mathsf{Merkle.Verify}(\sigma, \mathsf{fetched}_{t-1}, \mathsf{raddr}_{t-1}, \mathsf{digest}) = 1$
2. **Decrypt.** Compute $\mathsf{rk} := \mathsf{cFHE.Dec}_{chsk}(\boxed{\mathsf{rk}})$.
   If $t > 1$: decrypt $\mathsf{cpustate}_{t-1} := \boxed{\mathsf{cpustate}_{t-1}} \oplus \mathsf{PRF}_k(t-1, \mathsf{digest}, \text{"cpu"})$. Further, if $\mathsf{lastwritten}_{t-1} > 0$, decrypt $\mathsf{fetched}_{t-1} := \boxed{\mathsf{fetched}_{t-1}} \oplus \mathsf{PRF}_k(\mathsf{lastwritten}_{t-1}, \mathsf{raddr}_{t-1}, \mathsf{digest}, \text{"mem"})$.
   Else if $t = 1$, decrypt $\mathsf{fetched}_0 := \left(\mathcal{E}.\mathsf{Dec}_{dk}(\boxed{x_1}), \mathcal{E}.\mathsf{Dec}_{dk}(\boxed{x_2})\right)$.
3. **Compute.** $(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t, \mathsf{lastwritten}_t, y) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}, \mathsf{rk}\right)$ where the outcome y is only output if $t = T_{\mathsf{ORAM}}$.
4. **Encrypt.** Compute $\boxed{\mathsf{data}_t} := \mathsf{data}_t \oplus \mathsf{PRF}_k(t, \mathsf{waddr}_t, \mathsf{digest}, \text{"mem"})$, and $\boxed{\mathsf{cpustate}_t} := \mathsf{cpustate}_t \oplus \mathsf{PRF}_k(t, \mathsf{digest}, \text{"cpu"})$. If $t = T_{\mathsf{ORAM}}$, use public-key encryption $\mathcal{E}$ to encrypt y, using randomness seeded by rk.
5. **Sign.** $\{\mathsf{raddr}_t\} := (\mathsf{raddr}_t,\ \ \Sigma.\mathsf{Sign}_{sk}(\mathsf{raddr}_t, (t, \mathsf{digest}, \text{"raddr"})))$,
   $\{\mathsf{lastwritten}_t\} := (\mathsf{lastwritten}_t,\ \ \Sigma.\mathsf{Sign}_{sk}(\mathsf{lastwritten}_t, (t, \mathsf{digest}, \text{"lastwritten"})))$
   $\left\{\boxed{\mathsf{data}_t}\right\} := \left(\boxed{\mathsf{data}_t},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{data}_t}, (t, \mathsf{waddr}_t, \mathsf{digest}, \text{"mem"}))\right)$
   $\left\{\boxed{\mathsf{cpustate}_t}\right\} := \left(\boxed{\mathsf{cpustate}_t},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{cpustate}_t}, (t, \mathsf{digest}, \text{"cpu"}))\right)$
   $\left\{\boxed{\mathsf{rk}}\right\} := \left(\boxed{\mathsf{rk}},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{rk}}, (t, \mathsf{digest}, \text{"prf"}))\right)$

Figure 11: Circuit $C^{\text{FHE}}_{\textsc{NextIns}}$ to be obfuscated using VBB obfuscation.

Setup. $(\mathsf{hpk}, \mathsf{hsk}) \leftarrow \mathsf{Setup}(1^\lambda)$:

Run $(chpk, chsk) \leftarrow \mathsf{cFHE.Setup}(1^\lambda)$, $(ek, dk) \leftarrow \mathcal{E}.\mathsf{Gen}(1^\lambda)$ and $(sk, vk) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$.

Choose PRF keys $K \in \{0,1\}^\lambda, k \in \{0,1\}^\lambda$.

Compute $\boxed{K} := \mathsf{cFHE.Enc}_{chpk}(K)$.

Compute a VBB obfuscation $\mathcal{O}(C^{\text{FHE}}_{\text{NextIns}})$ of the circuit $C^{\text{FHE}}_{\text{NextIns}}$ as described in Figure 11, where NextIns is the next-instruction circuit of an ORAM.

The public and secret keys are

$$\mathsf{hpk} := \left(\mathcal{O}(C^{\text{FHE}}_{\text{NextIns}}),\ chpk,\ ek,\ \boxed{K}\right) \qquad \mathsf{hsk} := dk$$

Encryption. $\mathsf{Enc}(\mathsf{hpk}, x)$:

Parse $\mathsf{hpk} := (\mathcal{O}, ek, chpk, \boxed{K})$. Compute $\boxed{x} := \mathcal{E}.\mathsf{Enc}_{ek}(x)$

Decryption. $\mathsf{Dec}(\mathsf{hsk}, \boxed{x})$:

Parse $\mathsf{hsk} := dk$. Compute $x := \mathcal{E}.\mathsf{Dec}_{dk}(\boxed{x})$.

Evaluation. $\mathsf{Eval}(\mathsf{hpk}, \mathsf{RAM}, \boxed{x_1}, \boxed{x_2})$:

ORAM Setup:

Let $\mathsf{ORAM} := D$ be the complied ORAM counterpart for $\mathsf{RAM} = D'$, where $D = D' || \vec{0}$.

Compute $\mathsf{digest} = \mathsf{Merkle.Hash}(D, \boxed{x_1}, \boxed{x_2})$. Let $\{\boxed{x_1}\}$ and $\{\boxed{x_2}\}$ denote the input ciphertext with their Merkle proofs w.r.t digest.

Make the initial memory contents attached with their respective Merkle proofs: For $\mathsf{addr} \in \{1, \ldots, |D|\}$, let σ_{addr} be the Merkle hash proof that $D[\mathsf{addr}]$ is consistent with digest, tag each memory word with its $\mathsf{Merkle\text{-}Hash}$ proof, i.e., $D[\mathsf{addr}] := (D[\mathsf{addr}], \sigma_{\mathsf{addr}})$.

Execution: Initialize $\overline{D} = D$. For $t = 1$ to T_{ORAM}, compute:

$$\left(\{\mathsf{raddr}_t\}, \{\mathsf{lastwritten}_t\}, \mathsf{waddr}_t, \left\{\boxed{\mathsf{data}_t}\right\}, \left\{\boxed{\mathsf{cpustate}_t}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$$
$$:= \begin{cases} \mathcal{O}(C^{\text{FHE}}_{\text{NextIns}})\left(\{\boxed{x_1}\}, \{\boxed{x_2}\}\right) & \text{if } t = 1 \\ \mathcal{O}(C^{\text{FHE}}_{\text{NextIns}})\left(\{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \{\overline{\mathsf{fetched}}_{t-1}\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\}.\right) & \text{o.w.} \end{cases}$$

Note that if $t = T_{\mathsf{ORAM}}$, i.e. this is the final step of a query, the outcome $\left\{\boxed{y}\right\}$ is additionally output by $\mathcal{O}(C^{\text{FHE}}_{\text{NextIns}})$ above.

If $t = T_{\mathsf{ORAM}}$, output $\boxed{y}$ as the homomorphic evaluation of RAM on $\boxed{x_1}$ and $\boxed{x_2}$.

If $t < T_{\mathsf{ORAM}}$, read and write to memory.

$$\{\overline{\mathsf{fetched}}_t\} := \overline{D}[\mathsf{raddr}_t]$$

Note that the resulting $\{\overline{\mathsf{fetched}}_t\}$ is either attached with a Merkle proof if the memory location has not been written before, or it is attached with a time-dependent signature. Further, perform memory write:

$$\overline{D}[\mathsf{waddr}_t] := \{\boxed{\mathsf{data}_t}\}$$

F.4 Security Proof

Theorem F.2. *Assume* ORAM *is oblivious RAM as defined in Definition 2.1,* $\mathcal{O}$ *is a VBB obfuscator,* Σ *is an unforgeable signature scheme,* Merkle-Hash *is collision resilient,* PRF *is a secure PRF,* $\mathcal{E}$ *is a semantically secure encryption scheme, and* cFHE *is a semantically secure fully homomorphic encryption scheme for circuits, then the construction* FHE-RAM *is a semantically secure scheme in the RAM model.*

For any two messages m_0 and m_1, we define the two distributions $\mathsf{Dist}_b := (\mathsf{hpk}, \mathsf{Enc}(\mathsf{hpk}, m_b))$ for $b \in \{0, 1\}$, where hpk contains the obfuscated program $\mathcal{O}(C^{\text{FHE}}_{\text{NextIns}})$. To show the theorem, we argue that for any distinguisher $\mathcal{D}$ of the semantic security game of the FHE-RAM, and any two messages m_0, m_1, we have

$$|\Pr[\mathcal{D}(\mathsf{Dist}_0) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_1) = 1]| < \mathsf{negl}(\lambda)$$

for some negligible function $\mathsf{negl}(\cdot)$.

Lemma F.3. *Suppose that the obfuscator* $\mathcal{O}$ *is virtual-blackbox secure. If there exists a polynomial-time distinguisher* $\mathcal{D}$ *such that* $|\Pr[\mathcal{D}(\mathsf{Dist}_0) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_1) = 1]| > \delta$ *for some non-negligible* δ*, then there exists a distinguisher* $\widehat{\mathcal{D}}$ *such that*

$$|\Pr[\widehat{\mathcal{D}}^{O_1^{\text{FHE}}}(\widehat{\mathsf{Dist}_0}) = 1] - \Pr[\widehat{\mathcal{D}}^{O_1^{\text{FHE}}}(\widehat{\mathsf{Dist}_1}) = 1]| > \delta - \mathsf{negl}(\lambda)$$

where $\widehat{\mathsf{Dist}_0}$ *and* $\widehat{\mathsf{Dist}_1}$ *are the same as* Dist_0 *and* Dist_1*, except for removing the obfuscated program* $\mathcal{O}(C^{\text{FHE}}_{\text{NextIns}})$ *from the adversary's view. Furthermore, the distinguisher* $\widehat{\mathcal{D}}$ *is given oracle access to the circuit* $C^{\text{FHE}}_{\text{NextIns}}$ *(also denoted as* O_1^{FHE}*) .*

Proof. The proof of this lemma follows in a similar manner as Lemma E.4 □

Modified Oracle O_2^{fhe}. We now modify the oracle O_1^{FHE} into O_2^{FHE} in the following manner. At a high level, O_2^{FHE} changes the signature verification to a table look-up process.

More concretely, O_2^{FHE} keeps track of one "thread" for every digest queried. Basically for every digest, the oracle stores the next expected input

$$\mathsf{input} := \Big(t, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\} \Big\{ \boxed{\mathsf{cpustate}_{t-1}} \Big\}, \Big\{ \boxed{\mathsf{rk}} \Big\}\Big)$$

and the current status of the memory array D (containing only information about locations that have been written).

Upon any input corresponding to a digest: If $t = 1$, O_2^{FHE} checks that $\boxed{x_1}$ and $\boxed{x_2}$ have valid Merkle proofs with respect to digest. If $t > 1$, O_2^{FHE} will check the inputs as follows: it first checks that the raddr_{t-1}, $\mathsf{lastwritten}_{t-1}$, $\boxed{\mathsf{cpustate}_{t-1}}$, $\boxed{\mathsf{rk}}$ are the previously stored value using a one-step look-ahead book-keeping mechanism (as mentioned later). Then, if $\mathsf{lastwritten}_{t-1} = 0$, it checks that the claimed $\overline{\mathsf{fetched}}_{t-1}$ is consistent with digest. Otherwise, $\mathsf{lastwritten}_{t-1} > 0$, it checks that the claimed $\overline{\mathsf{fetched}}_{t-1}$ matches with the previously stored value using the one-step look-ahead book-keeping mechanism (as mentioned later). If any check fails, the input is rejected.

If not, go to the look-ahead book-keeping mechanism: The oracle now follows the honest algorithm of the remainder of $C^{\text{FHE}}_{\text{NextIns}}$ to compute the outputs of this step. It then uses the honest evaluation to compute the correct inputs of the next step and stores them.

O_2^{FHE}:

Initial state: $chpk, chsk, ek, dk, sk, k,$ $\boxed{K}$
Additionally, this oracle maintains a table where $\Gamma[\mathsf{digest}]$ is either $\perp$ or is a pair (input, D). Initially, $\Gamma[\mathsf{digest}]$ is $\perp$ for every digest.
The oracle also stores a table of all previously made queries and answers.

Inputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.
Outputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.

Oracle description:

1. If the queried input has been seen before, just give exactly the same answer as before. Otherwise, continue to the following steps.
2. If $t = 1$, the input is $(t, \mathsf{digest}, \{\boxed{x_1}\}, \{\boxed{x_2}\}, \mathsf{cpustate}_0)$. Verify that $\{\boxed{x_1}\}, \{\boxed{x_2}\}$ have valid Merkle proofs w.r.t digest. If verification fails, output $\perp$. Otherwise, compute $\boxed{\mathsf{rk}} = \mathsf{cFHE.Eval}_{chpk}(G, \boxed{K})$, where $G(\cdot) = \mathsf{PRF}_{(\cdot)}(\mathsf{digest})$. Let $D := \vec{0}$, i.e., no location in memory has been written yet.
3. Else, if $t > 1$, the input is $\left(t, \mathsf{digest}, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \{\overline{\mathsf{fetched}}_{t-1}\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$. Let
$$\mathsf{input} := \left(t, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\} \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$$
Check that input is consistent with the expected input saved in $\Gamma[\mathsf{digest}]$. If the check fails, output $\perp$.
If $\mathsf{lastwritten}_{t-1} = 0$, check that $\{\overline{\mathsf{fetched}}_{t-1}\}$ are consistent with Merkle digest. Else, let $(\ldots, D) := \Gamma[\mathsf{digest}]$, check that $\{\overline{\mathsf{fetched}}_{t-1}\} = D[\mathsf{raddr}_{t-1}]$.
4. Carry out the steps "Decrypt", "Compute", "Encrypt" and "Sign" as in $C^{\text{FHE}}_{\text{NextIns}}$. Perform the corresponding memory write to D. Suppose that this gives us new D', and that the correct input for the next time step is input':
$$\mathsf{input}' := \left(t+1, \{\mathsf{raddr}_t\}, \{\mathsf{lastwritten}_t\} \left\{\boxed{\mathsf{cpustate}_t}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$$
5. Set $\Gamma[\mathsf{digest}] := (\mathsf{input}', D')$.

Lemma F.4. *Assuming that the signature scheme is unforgeable, then no distinguisher can distinguish whether it is interacting with O_1^{FHE} or O_2^{FHE}.*

Proof. There are two cases we need to consider: 1) If the adversary can find a different $\mathsf{input} := \left(t, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\} \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left\{\boxed{\mathsf{rk}}\right\}\right)$ than what is stored in the one-step look-ahead book-keeping mechanism, 2) If $\mathsf{lastwritten}_{t-1} > 0$, and the adversary finds a different $\{\overline{\mathsf{fetched}}_t\}$ than what is stored in the one-step look-ahead book-keeping mechanism. Either will constitute a signature forgery. $\square$

Modified Oracle O_3^{fhe}. We now modify the oracle O_2^{FHE} into O_3^{FHE} as in the following figure.

O_3^{FHE}:

Initial state: $chpk, chsk, ek, dk, sk, \boxed{K}$ and a truly random function.

Inputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.
Outputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.

Oracle description:

Same as that in oracle O_2^{FHE} except the following:

- upon receiving the input $\big(t = 1, \mathsf{digest}, \{\boxed{x_1}\}, \{\boxed{x_2}\}, \mathsf{cpustate}_0\big)$, if $\big(t = 1, \mathsf{digest}, C_1, C_2, \cdot\big)$ has been recorded in $\Gamma[\mathsf{digest}]$, and $(\boxed{x_1}, \boxed{x_2}) \neq (C_1, C_2)$, then abort.
- upon receiving the input

$$\big(t, \mathsf{digest}, \{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \{\overline{\mathsf{fetched}}_{t-1}\}, \Big\{\boxed{\mathsf{cpustate}_{t-1}}\Big\}, \Big\{\boxed{\mathsf{rk}}\Big\}\big)$$

where $t > 1$ and $\mathsf{lastwritten}_{t-1} = 0$, if in table $\Gamma[\mathsf{digest}]$, the following tuple has been recorded: $\big(t', \mathsf{digest}, \{\mathsf{raddr}_{t'-1}\}, \{\mathsf{lastwritten}_{t'-1}\}, \{\overline{\mathsf{fetched}}_{t'-1}\}, \Big\{\boxed{\mathsf{cpustate}_{t'-1}}\Big\}, \Big\{\boxed{\mathsf{rk}}\Big\}\big)$, where $t' > 1$ and $\mathsf{lastwritten}_{t'-1} = 0$, and in addition $\mathsf{raddr}_{t-1} = \mathsf{raddr}_{t'-1}$, but $\mathsf{fetched}_{t-1} \neq \mathsf{fetched}_{t'-1}$, then abort.

Lemma F.5. *Assume that the hash function used to compute the Merkle digest is collision resistant, then no PPT distinguisher can distinguish whether it is interacting with O_2^{FHE} or O_3^{FHE}.*

Proof. Since the oracles differ only if the adversary can find a consistent input (other than the stored one), which results in a collision of the Merkle digest. Thus, by the security of the collision resistance of the Merkle digest, no PPT adversary can distinguish the two oracles. □

Modified Oracle O_4^{fhe}. We now modify the oracle O_3^{FHE} into O_4^{FHE} in the following manner: O_4^{FHE} replaces the PRF with a truly random function F. Everything else is the same as O_3^{FHE}.

O_4^{FHE}:

Initial state: $chpk, chsk, ek, dk, sk, \boxed{K}$ and a truly random function.

Inputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.
Outputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.

Oracle description:

Same as that in oracle O_3^{FHE} except that in Encrypt steps, instead of using PRF to generate a pseudorandom string as the one-time pad, here a truly random function is used as the one time pad. The use of PRF in computing or verifying the random seed rk is left unchanged.

Lemma F.6. *Assume that the PRF is secure, then no PPT distinguisher can distinguish whether it is interacting with O_3^{FHE} or O_4^{FHE}.*

Proof. This follows directly from the security of the PRF. □

Modified Oracle O_5^{fhe}. We now modify the oracle O_4^{FHE} into O_5^{FHE} in the following manner: We store both K and its encryption $\boxed{K}$ in the initial hardwired state, thus rk can be computed directly from K instead of decryption of the homomorphic encryption.

O_5^{FHE}:

Initial state: $chpk, ek, dk, sk, K,$ K and a truly random function

Inputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.
Outputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.

Oracle description:

Same as that in oracle O_4^{FHE} except that now rk is computed as follows: $\mathsf{rk} = \mathsf{PRF}_{(K)}(\mathsf{digest})$.

Lemma F.7. *Assume the correctness of* cFHE*, then no PPT distinguisher can distinguish whether it is interacting with* O_4^{FHE} *or* O_5^{FHE}*.*

Proof. This follows directly from the correctness of cFHE. □

Modified Experiment. We now modify the experiment such that the simulator will encode 0 in the FHE ciphertext instead of K. Note that K is still hardwired in the oracle. We denote for $b \in \{0, 1\}$: $\widehat{\mathsf{Dist}}'_b$ to be the same as $\widehat{\mathsf{Dist}}_b$ except that the FHE ciphertext of K is now replaced with a ciphertext of 0.

Lemma F.8. *Assume that* cFHE *is semantically secure, no PPT distinguisher with oracle access to* O_5^{FHE} *can distinguish* $\widehat{\mathsf{Dist}}_b$ *from* $\widehat{\mathsf{Dist}}'_b$ *for* $b \in \{0, 1\}$*.*

Proof. We show this by contradiction. Suppose there exists a PPT distinguisher $\mathcal{D}$ such that $\mathcal{D}$ distinguishes $\widehat{\mathsf{Dist}}_b$ from $\widehat{\mathsf{Dist}}'_b$ with oracle access to O_5^{FHE}, then we can show a reduction that breaks the semantic security of FHE. The reduction simply simulates the distinguisher $\mathcal{D}$ and the oracle O_5^{FHE}, and embed the challenge ciphertext $\mathsf{cFHE.Enc}(m_b)$ into $\widehat{\mathsf{Dist}}_b$. Since the oracle O_5^{FHE} is independent of the message m_b, the advantage of the reduction follows directly from the advantage of $\mathcal{D}^{O_5^{\text{FHE}}}$. This is a contradiction. □

Modified Oracle $O_6^{\mathbf{fhe}}$. We now modify the oracle O_5^{FHE} into O_6^{FHE} in the following manner: Instead of computing the random seed rk from K, we use a randomly chosen rk.

O_6^{FHE}:

Initial state: $chpk, ek, dk, sk$

Inputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.
Outputs: Same as in $C^{\text{FHE}}_{\text{NextIns}}$.

Oracle description:

Same as that in oracle O_5^{FHE} except that for the computation of random seed rk, we use a randomly chosen rk.

Lemma F.9. *Assume the security of PRF, no PPT distinguisher can distinguish whether it is interacting with* O_5^{FHE} *or* O_6^{FHE}*.*

Proof. This follows directly from the security of the PRF. □

Modified Oracle $O_7^{\mathbf{fhe}}$. We now modify the oracle O_6^{FHE} into O_7^{FHE} in the following manner: O_7^{FHE} replaces emitted memory addresses with simulated ORAM addresses. The simulated ORAM addresses can be generated and saved the first time a digest is queried, such that if the same query is made twice, the same answer will be given.

O_7^{FHE}:

Initial state: $chpk, ek, dk, sk$ and a truly random function.

Inputs: Same as in $C_{\text{NextIns}}^{\text{FHE}}$.
Outputs: Same as in $C_{\text{NextIns}}^{\text{FHE}}$.

Oracle description:

Same as that in oracle O_6^{FHE} except that each table $\Gamma[\mathsf{digest}]$ is created in a different way. Instead of generating addresses by running NextIns(), now the addresses are computed by running the predictive ORAM simulator $\mathsf{Sim}'()$, i.e., compute $\left(\{\mathsf{waddr}_t, \mathsf{raddr}_t\}_{t\in[T_{\mathsf{ORAM}}]}\right) \leftarrow \mathsf{Sim}'()$. The circuit then computes $\mathsf{lastwritten}_t$ for each $t \in [T_{\mathsf{ORAM}}]$.

Lemma F.10. *Assume that the ORAM is secure, then no PPT distinguisher can distinguish whether it is interacting with O_6^{FHE} or O_7^{FHE}.*

Proof. Assume there is an algorithm $\mathcal{D}$ who can distinguish O_6^{FHE} from O_7^{FHE} via black box access, then we can construct another algorithm $\mathcal{D}'$ who can break the computational security of the corresponding ORAM. $\mathcal{D}'$ internally simulates $\mathcal{D}$ and the oracle except that the addresses are generated by an external algorithm either by $\mathsf{addresses}(\mathsf{ORAM}, \cdot, \mathsf{rk})$ or by $\mathsf{Sim}'()$. If the external algorithm is $\mathsf{addresses}()$, then $\mathcal{D}$'s view is the same as that in O_6^{FHE}; and if the external algorithm is Sim', then $\mathcal{D}$'s view is the same as that in O_6^{FHE}. Since $\mathcal{D}$ can distinguish the two experiments with non-negligible probability, that means $\mathcal{D}'$ can distinguish the real addressees generated by $\mathsf{addresses}()$ from the simulated addresses produced by $\mathsf{Sim}'()$ with non-negligible probability. That means $\mathcal{D}'$ can break the computational security of ORAM, as defined in Definition 2.1, which reaches a contradiction. □

Modified Oracle $O_8^{\mathbf{fhe}}$. We now modify the oracle O_7^{FHE} into O_8^{FHE} as in the following figure.

O_8^{FHE}:

Initial state: $chpk, ek, sk$ and a truly random function.

Inputs: Same as in $C_{\text{NextIns}}^{\text{FHE}}$.
Outputs: Same as in $C_{\text{NextIns}}^{\text{FHE}}$.

Oracle description:

Same as that in oracle O_7^{FHE} except in the decryption step, the oracle does not decrypt the input ciphertexts ($\boxed{x_1}$, $\boxed{x_2}$) for $t = 1$. Instead, it sets $\mathsf{fetched}_0 := (0, 0)$, and then continue as O_7^{FHE}. Thus, the oracle does not need dk.

We have the following lemma immediately.

Lemma F.11. *There is no PPT distinguisher can distinguish whether it is interacting with O_7^{FHE} or O_8^{FHE}.*

Proof. We observe that the distributions of the outputs of the two oracle are identical. This follows from the fact that all the data (i.e. data) and cpu states (i.e. $\mathsf{cpustate}$), except the addresses,

output by the oracle are encrypted under the truly random function as one-time pads. Therefore, along the computations, of (x_1, x_2) and $(0, 0)$, the distributions of the data and the cpu states are identical. Thus, no adversary can distinguish the oracles from interactions with them. $\square$

Lemma F.12. *Assume that the public-key encryption scheme is semantically secure, then no PPT distinguisher with* O_8^{FHE} *can distinguish* $\widehat{\mathsf{Dist}'}_0$ *from* $\widehat{\mathsf{Dist}'}_1$*.*

Proof. We show this by contradiction. Assume there exists a PPT distinguisher $\mathcal{D}$ such that $\mathcal{D}$ distinguishes $\widehat{\mathsf{Dist}'}_0$ from $\widehat{\mathsf{Dist}'}_1$ with oracle access to O_8^{FHE}, then we can show a reduction that breaks the semantic security of the public key encryption scheme. The reduction simply simulates the distinguisher $\mathcal{D}$ and the oracle O_8^{FHE}, and embeds the challenge ciphertext $\mathcal{E}.\mathsf{Enc}_{ek}(m_b)$ into the $\widehat{\mathsf{Dist}'}_b$. This is because O_8^{FHE} is independent of the message m_b. Recall that it uses a truly random function to encrypt and the ORAM simulator to generate addresses. Both are independent of the message. Thus, the advantage of the reduction follows directly from the advantage of $\mathcal{D}^{O_8^{\mathrm{FHE}}}$. This is a contradiction. $\square$

Recall that we have assumed that there exists a distinguisher $\mathcal{D}$, and messages m_0, m_1, and some non-negligible ϵ such that $\Pr[\mathcal{D}(\mathsf{Dist}_0) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_1) = 1] > \epsilon$. Then by the above lemmas, there exists a distinguisher $\mathcal{D}'$ that with oracle O_8^{FHE} can distinguish $\widehat{\mathsf{Dist}'}_0$ from $\widehat{\mathsf{Dist}'}_1$. This contradicts the above lemmas, which completes the proof of the theorem that the FHE-RAM is semantically secure.

G Non-Interactive Verifiable Computation from VBB Obfuscation

Cloud computing allows users and organizations to outsource both their *data* and *computation* to cloud servers. Imagine that a client C outsources a database DB (e.g., a SQL database, a key-value store, a graph, etc.) to an untrusted cloud server S. The client will then make a series of queries to the server. For example, the client can ask the server to compute a function over the outsourced DB, and return the answer; the client can also make update queries such as inserting, deleting, or modifying entries in the database. Since the cloud server is outside the client's control, a big challenge is how to guarantee the *privacy* of outsourced data, and the *integrity* of computation performed by the server.

Using the circuit model of verifiable computation would require building a circuit that embeds the entire database. Therefore, the server would have to run in at least linear time, even when the query is sublinear-time in nature (e.g., binary search, range queries). The only known RAM-model solution that ensures the privacy of the database is the Garbled RAM scheme by Lu and Ostrovsky [36] – however, it requires the client to also compute in time linear in T for each query.

Stateful RAM for repeated queries. In this section, we consider an interesting new model of *stateful*, RAM-model VC suitable for outsourcing a large database and performing repeated queries over the database afterwards. Let $\mathsf{RAM} = (D, \mathsf{cpustate}_0)$ denote a RAM's initial configuration, then we write

$$y_m \leftarrow \mathsf{RAM}(x_1, x_2, \ldots, x_m)$$

to denote that the result of answering the m-th query of the sequence $x_1, \ldots, x_m$ is y_m.

Remark. We note that our VC-RAM notion is closely related to that of symetric-key, function private FE-RAM. In particulary, a symetric-key, function private FE-RAM would give rise to a stateless VC-RAM scheme. Based on our VC-RAM techniques, it is also straightforward to construct a symetric-key, function private FE-RAM scheme.

G.1 Stateful VC-RAM Model

Definition G.1 (Verifiable RAM Computation). *A Verifiable RAM Computation (VC-RAM) scheme consists of the following algorithms:*

$(Z, z) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{RAM})$: The Setup algorithm is a one-time setup algorithm run by the client. Setup takes in the security parameter 1^λ, initial RAM configuration $\mathsf{RAM} = (D, \mathsf{cpustate}_0)$, and outputs server initial state Z, and client state z. The client hands Z to the server, and retains state z locally.

$(\overline{x}, z) \leftarrow \mathsf{ProbGen}(x, z)$: Given input x, prepare the input and obtain the encoding $\overline{x}$. The client state z is updated[10].

$(\overline{y}, Z) \leftarrow \mathsf{Compute}(\overline{x}, Z)$: Given current server state Z and encoded input $\overline{x}$, the server computes an encoded answer $\overline{y}$, which typically embeds the output as well as a proof of correct computation. The server also updates its state Z as a result of $\mathsf{Compute}$.

$(y, b, z) \leftarrow \mathsf{Verify}(\overline{y}, z)$: Outputs the decoded answer y, a bit $b \in \{0, 1\}$ indicating whether the answer is accepted, and updates the client local state z.

Correctness is defined in the obvious way. We require that for any initial RAM configuration $\mathsf{RAM} := (D, \mathsf{cpustate}_0)$, for any query sequence $x_1, x_2, \ldots, x_m$ where $m = \mathsf{poly}(\lambda)$,

$$\Pr\left[\exists i : \begin{array}{l} (y_i \neq \mathsf{RAM}(x_1, x_2, \ldots, x_i)) \\ \vee (b_i = 0) \end{array} \;\middle|\; \begin{array}{l} (Z_0, z) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{RAM}) \\ \forall i \in \{1, 2, \ldots, m\} : \\ \quad (\overline{x}_i, z) \leftarrow \mathsf{ProbGen}(x_i, z) \\ \quad (\overline{y}_i, Z_i) \leftarrow \mathsf{Compute}(\overline{x}_i, Z_{i-1}) \\ \quad (y_i, b_i, z) \leftarrow \mathsf{Verify}(\overline{y}_i, z) \end{array}\right] = \mathsf{negl}(\lambda)$$

We give a simulation-based definition that incorporates both verifiability and privacy. Our definition handles the selective abort issue which was not addressed by the GGP construction [22]. Particularly, a malicious server is allowed to choose potentially malformed answers and ask the client to decode, and learn the corresponding outcome.

Definition G.2 (Simulation-based security: verifiability + privacy). *We say that a VC-RAM scheme is simulation secure, if there exists a polynomial-time algorithm* $\mathcal{S} = (\mathcal{S}.\mathsf{Setup}, \mathcal{S}.\mathsf{ProbGen}, \mathcal{S}.\mathsf{Verify})$ *such that for any params, no polynomial-time (stateful) adversary* $\mathcal{A}$ *can distinguish the following real- and ideal-world games except with negligible probability. Particularly, in line [**] below, if* $b_i = 1$, $y_i := \mathsf{RAM}(x_1, \ldots, x_i)$ *represents the correct answer to the i-th query; otherwise* $y_i := \perp$.

[10] We use the notation $(\mathit{output}, \mathit{state}) \leftarrow \mathsf{Alg}(\mathit{input}, \mathit{state})$ to denote that a party runs algorithm Alg, which results in updating its local state.

Real world:	**Ideal world:**
$\mathsf{RAM} \leftarrow \mathcal{A}(1^\lambda, params)$	$\mathsf{RAM} \leftarrow \mathcal{A}(1^\lambda, params)$
$(Z_0, z) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{RAM})$	$(Z_0, z) \leftarrow \mathcal{S}.\mathsf{Setup}(1^\lambda, params)$
$x_1 \leftarrow \mathcal{A}(Z_0)$	$x_1 \leftarrow \mathcal{A}(Z_0)$
$\forall i \in \{1, 2, \ldots, \text{poly}(\lambda)\}$:	$\forall i \in \{1, 2, \ldots, \text{poly}(\lambda)\}$:
$(\overline{x}_i, z) \leftarrow \mathsf{ProbGen}(x_i, z)$	$(\overline{x}_i, z) \leftarrow \mathcal{S}.\mathsf{ProbGen}(z)$
$\overline{y}_i \leftarrow \mathcal{A}(\overline{x}_i)$	$\overline{y}_i \leftarrow \mathcal{A}(\overline{x}_i)$
$(y_i, b_i, z) \leftarrow \mathsf{Verify}(\overline{y}_i, z)$	$(b_i, z) \leftarrow \mathcal{S}.\mathsf{Verify}(\overline{y}_i, z)$
$x_{i+1} \leftarrow \mathcal{A}(y_i, b_i)$	$x_{i+1} \leftarrow \mathcal{A}(y_i, b_i)$ [**] *//see definition of y_i above*
$\{0, 1\} \leftarrow \mathcal{A}$	$\{0, 1\} \leftarrow \mathcal{A}$

G.2 Preliminary: Predictive-Memory Stateful ORAM for Multiple Inputs

Remark: No rewinding and one-time ORAM setup. Earlier in our FE-RAM construction, we use an ORAM abstraction where the ORAM will start by shuffling memory prior to evaluating the RAM program, thus incurring $\tilde{O}(n)$ cost.

In our VC setting, our RAM is stateful. Even though the client submits multiple queries, the stateful ORAM cannot be rewinded to the beginning state. Therefore, the ORAM setup only needs to be performed once, when the client outsources the database to the server. Afterwards, during the online query phase, the memory need not be reshuffled again. This is also important for amortizing the setup time over multiple queries and achieving sublinear (in n) amortized cost for database queries that take sublinear time (e.g., binary search and range queries).

In comparison, in our FE-RAM scheme, the shuffling must be done at evaluation time, because the same ORAM can be "rewinded" to its initial state, and applied to multiple inputs. Unless we re-perform the shuffling for each input using fresh (pseudo)-randomness, the address sequence emitted can be correlated for multiple inputs.

We use $\mathsf{OCompile}$ to denote an ORAM compiler that takes a initial RAM configuraion $\mathsf{RAM} := (D', \mathsf{cpustate}'_0)$, and performs ORAM setup. The resulting memory configuration is denoted $\mathsf{ORAM} := (D, \mathsf{cpustate}_0)$. In particular $\mathsf{OCompile}$ stores secret state (including a random seed) inside the ORAM's initial CPU state $\mathsf{cpustate}_0$[11].

Lemma G.3 (Stateful, predictive-memory ORAM compiler for multiple queries)**.** *There exists a predictive oblivious compiler denoted* $\mathsf{OCompile}$*, such that on given a* $\mathsf{RAM} := (D', \mathsf{cpustate}'_0)$*, it outputs a predictive-memory, stateful* $\mathsf{ORAM} := (D, \mathsf{cpustate}_0)$ *(where the* ORAM*'s* $\mathsf{cpustate}_0$ *contains a random pseudorandom key, and D is a shuffled version of the old D') such that the following conditions hold:*

- **Correctness.** For any RAM, let $\mathsf{ORAM} := \mathsf{OCompile}(\mathsf{RAM})$; then for any input $x_1, x_2, \ldots, x_m$, it holds that $\Pr[\mathsf{ORAM}(x_1, \ldots, x_i) = \mathsf{RAM}(x_1, \ldots, x_i)] = 1$ for any $i \in [m]$, where probability is taken over the randomness used by $\mathsf{OCompile}$. Further, with probability 1, the last written address $\mathsf{lastwritten}_t$ emitted by the predictive ORAM's NextIns function accurately represents the last time raddr_t is written. If address raddr_t has never been written, our convention is to set $\mathsf{lastwritten}_t := 0$ in this case.
- **Computational security.** An ORAM compiler (for a stateful RAM) is secure, if there exists a simulator Sim, such that no PPT adversary $\mathcal{A}$ can distinguish the following real and

[11] For simplicity, we assume that in this section, the ORAM's next instruction circuit need not take a random string, since the random seed needed is embedded in the ORAM's initial CPU state.

simulated worlds: We assume that both $\mathcal{A}$ and Sim knows the parameters of the RAM and ORAM, and their next instruction circuits.

Real world:

- $\mathsf{RAM} \leftarrow \mathcal{A}$
- $\mathsf{ORAM} \leftarrow \mathsf{OCompile}(\mathsf{RAM})$
- $b \leftarrow \mathcal{A}^{\mathsf{addresses}(\mathsf{ORAM},\cdot)}$

  where $\mathsf{addresses}(\mathsf{ORAM},\cdot)$ is a stateful oracle, where on a series of inputs $x_1, x_2, \ldots, x_m$, the oracle outputs the memory address sequence accessed by the stateful ORAM during the execution of each query $x_1, x_2, \ldots, x_m$.

Simulated world:

- $\mathsf{RAM} \leftarrow \mathcal{A}$
- $\mathsf{ORAM} \leftarrow \mathsf{OCompile}(\mathsf{RAM})$
- $b \leftarrow \mathcal{A}^{\mathsf{Sim}}$ where Sim is a stateful simulator that outputs a sequence of simulated addresses upon each invocation (with no input).

- **Overhead.** The compile ORAM has the following overhead: $n_{\mathsf{ORAM}} = \tilde{O}(n)$, $\ell_{\mathsf{ORAM}} = \max(\ell, c \log T)$ for some appropriate constant $c > 2$, and where T is the original RAM's maximum run-time, $|\mathsf{cpustate}_{\mathsf{ORAM}}| = O(|\mathsf{cpustate}| + \lambda)$, and $|\textsc{NextIns}_{\mathsf{ORAM}}| := O(|\textsc{NextIns}|)\mathsf{poly}(\lambda)$. Additionally, the transformed ORAM runs in $\tilde{O}(T)$ number of time steps if the original RAM runs in T number of time steps.

Lu and Ostrovsky [36] are the first ones to define predictive ORAM, and show that a predictive ORAM exists. In Section G.5, we give a more generic approach to convert any ORAM into a predictive ORAM. In particular, if we use an ORAM scheme with polylogarithmic worst-case (as opposed to amortized) cost, the resulting predictive ORAM will also have polylogarithmic cost.

G.3 Construction

Notational conventions. In the following scheme description, we will use the following notational conventions.

$\boxed{\mathsf{var}}$	an encrypted variable
$\{\mathsf{var}\}$, $\{\boxed{\mathsf{var}}\}$	a variable or encrypted variable, attached with a signature

Detailed scheme. We describe the detailed scheme below.

Setup. $(Z, z) \leftarrow \mathsf{Setup}(1^\lambda, \mathsf{RAM})$.

Let $\mathsf{ORAM} := (D, \mathsf{cpustate}_0) := \mathsf{OCompile}(\mathsf{RAM})$.

Generate secret PRF key k, and run the setup algorithm of a deterministic signature scheme $(sk, vk) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$.

Compute $\left\{\boxed{\mathsf{cpustate}_0}\right\} := \left(\boxed{\mathsf{cpustate}_0}, \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{cpustate}_0}, (0, \text{“cpu”}))\right)$ where $\boxed{\mathsf{cpustate}_0} := \mathsf{cpustate}_0 \oplus \mathsf{PRF}_k(0, \text{“cpu”})$.

For $i = 1$ to n_{ORAM}, compute $\left\{\boxed{D}[i]\right\} := \left(\boxed{D}[i], \ \Sigma.\mathsf{Sign}_{sk}(\boxed{D}[i], (0, i, \text{“mem”}))\right)$ where $\boxed{D}[i] := D[i] \oplus \mathsf{PRF}_k(0, i, \text{“mem”})$.

Construct a VBB obfuscation $\mathcal{O}(C^{\mathrm{VC}}_{\textsc{NextIns}})$ of the circuit $C^{\mathrm{VC}}_{\textsc{NextIns}}$ as described in Figure 12

Give the server the initial server state Z, consisting of the following:

$$Z := \left(\mathcal{O}(C^{\text{VC}}_{\textsc{NextIns}}), \left\{\boxed{\mathsf{cpustate}_0}\right\}, \left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n]}\right)$$

The client retains the following state $z := (k, sk, vk, q = 1)$.

Problem Generation. $(\overline{x}, z) \leftarrow \mathsf{ProbGen}(x, z)$.

Parse $z := (k, vk, q)$. Let $t := (q-1)T_{\mathsf{ORAM}}+1$. The client encrypts $\boxed{x} := x \oplus \mathsf{PRF}_k(t, \text{“input”})$, and signs it $\{\boxed{x}\} := \Sigma.\mathsf{Sign}_{sk}(\boxed{x}, (t, \text{“input”}))$. The client sends the resulting $\overline{x} := \{\boxed{x}\}$ to the server. The new client state is $z := (k, sk, vk, q)$.

Computation. $(\overline{y}, Z) \leftarrow \mathsf{Compute}(\overline{x}, Z)$.

Parse $\overline{x} := \{\boxed{x}\}$. Parse Z as $Z := \left(\mathcal{O}(C^{\text{VC}}_{\textsc{NextIns}}), \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n]}\right)$.

For $t = (q-1)T_{\mathsf{ORAM}} + 1$ to qT_{ORAM}, compute:

$$\begin{aligned}&\left(\{\mathsf{raddr}_t\}, \{\mathsf{lastwritten}_t\}, \mathsf{waddr}_t, \left\{\boxed{\mathsf{data}_t}\right\}, \left\{\boxed{\mathsf{cpustate}_t}\right\}\right)\\ := &\begin{cases}\mathcal{O}(C^{\text{VC}}_{\textsc{NextIns}})\left(\{\boxed{x}\}, \{\boxed{\mathsf{cpustate}_0}\}\right) & \text{if } t = 1\\ \mathcal{O}(C^{\text{VC}}_{\textsc{NextIns}})\left(\{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \{\boxed{x}\},\right) & \text{else if } t = qT_{\mathsf{ORAM}} + 1\\ \mathcal{O}(C^{\text{VC}}_{\textsc{NextIns}})\left(\{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{fetched}_{t-1}}\right\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}\right) & \text{o.w.}\end{cases}\end{aligned}$$

Note that if $t = qT_{\mathsf{ORAM}}$, i.e., this is the final step of a query, the outcome $\left\{\boxed{y}\right\}$ is additionally output by the above. The server now reads and writes to memory:

$$\left\{\boxed{\mathsf{fetched}_t}\right\} := \left\{\boxed{D}[\mathsf{raddr}_t]\right\}$$
$$\left\{\boxed{D}[\mathsf{waddr}_t]\right\} := \left\{\boxed{\mathsf{data}_t}\right\}$$

Finally, at the end of all T_{ORAM} time steps for this query, the server sends $\left\{\boxed{y}\right\}$ back to the client, and updates its state as

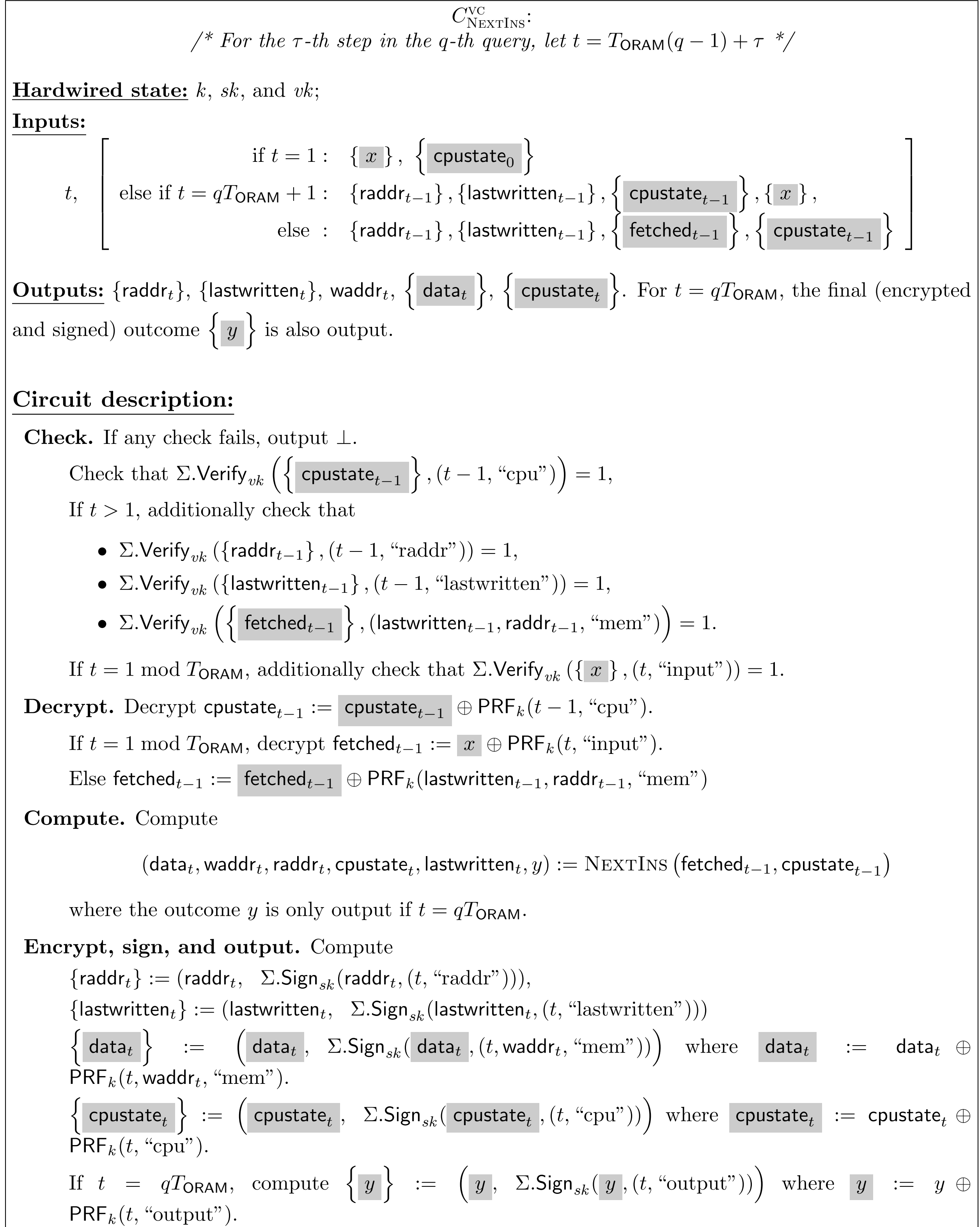
$$Z := \left(\mathcal{O}(C^{\text{VC}}_{\textsc{NextIns}}), \left\{\boxed{\mathsf{cpustate}_t}\right\}, \left\{\boxed{\mathsf{fetched}_t}\right\}, \left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n]}\right)$$

where $t = qT_{\mathsf{ORAM}}$ at the end of the q-th query.

Verification. $(y, b, z) \leftarrow \mathsf{Verify}(\overline{y}, z)$: Parse $z := (k, sk, vk, q)$. Parse $\overline{y} := \left\{\boxed{y}\right\}$. The client verifies the signature

$$b := \Sigma.\mathsf{Verify}_{vk}\left(\left\{\boxed{y}\right\}, (qT_{\mathsf{ORAM}}, \text{“output”})\right)$$

The client decrypts $y := \boxed{y} \oplus \mathsf{PRF}_k(qT_{\mathsf{ORAM}}, \text{“output”})$. The client outputs (y, b), and updates its state to be $z := (k, sk, vk, q + 1)$.



$C^{\text{VC}}_{\textsc{NextIns}}$:
/* *For the τ-th step in the q-th query, let $t = T_{\mathsf{ORAM}}(q-1)+\tau$* */

Hardwired state: k, sk, and vk;

Inputs:

$$t,\ \left[\begin{array}{rl} \text{if } t=1: & \{\boxed{x}\},\ \left\{\boxed{\mathsf{cpustate}_0}\right\} \\ \text{else if } t = qT_{\mathsf{ORAM}}+1: & \{\mathsf{raddr}_{t-1}\},\{\mathsf{lastwritten}_{t-1}\},\left\{\boxed{\mathsf{cpustate}_{t-1}}\right\},\{\boxed{x}\}, \\ \text{else}: & \{\mathsf{raddr}_{t-1}\},\{\mathsf{lastwritten}_{t-1}\},\left\{\boxed{\mathsf{fetched}_{t-1}}\right\},\left\{\boxed{\mathsf{cpustate}_{t-1}}\right\} \end{array}\right]$$

Outputs: $\{\mathsf{raddr}_t\}$, $\{\mathsf{lastwritten}_t\}$, waddr_t, $\left\{\boxed{\mathsf{data}_t}\right\}$, $\left\{\boxed{\mathsf{cpustate}_t}\right\}$. For $t = qT_{\mathsf{ORAM}}$, the final (encrypted and signed) outcome $\left\{\boxed{y}\right\}$ is also output.

Circuit description:

Check. If any check fails, output $\perp$.

Check that $\Sigma.\mathsf{Verify}_{vk}\left(\left\{\boxed{\mathsf{cpustate}_{t-1}}\right\},(t-1,\text{“cpu”})\right)=1$,

If $t>1$, additionally check that

- $\Sigma.\mathsf{Verify}_{vk}(\{\mathsf{raddr}_{t-1}\},(t-1,\text{“raddr”}))=1$,
- $\Sigma.\mathsf{Verify}_{vk}(\{\mathsf{lastwritten}_{t-1}\},(t-1,\text{“lastwritten”}))=1$,
- $\Sigma.\mathsf{Verify}_{vk}\left(\left\{\boxed{\mathsf{fetched}_{t-1}}\right\},(\mathsf{lastwritten}_{t-1},\mathsf{raddr}_{t-1},\text{“mem”})\right)=1$.

If $t = 1 \bmod T_{\mathsf{ORAM}}$, additionally check that $\Sigma.\mathsf{Verify}_{vk}(\{\boxed{x}\},(t,\text{“input”}))=1$.

Decrypt. Decrypt $\mathsf{cpustate}_{t-1} := \boxed{\mathsf{cpustate}_{t-1}} \oplus \mathsf{PRF}_k(t-1,\text{“cpu”})$.

If $t = 1 \bmod T_{\mathsf{ORAM}}$, decrypt $\mathsf{fetched}_{t-1} := \boxed{x} \oplus \mathsf{PRF}_k(t,\text{“input”})$.

Else $\mathsf{fetched}_{t-1} := \boxed{\mathsf{fetched}_{t-1}} \oplus \mathsf{PRF}_k(\mathsf{lastwritten}_{t-1},\mathsf{raddr}_{t-1},\text{“mem”})$

Compute. Compute

$$(\mathsf{data}_t,\mathsf{waddr}_t,\mathsf{raddr}_t,\mathsf{cpustate}_t,\mathsf{lastwritten}_t,y) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1},\mathsf{cpustate}_{t-1}\right)$$

where the outcome y is only output if $t = qT_{\mathsf{ORAM}}$.

Encrypt, sign, and output. Compute

$\{\mathsf{raddr}_t\} := (\mathsf{raddr}_t,\ \Sigma.\mathsf{Sign}_{sk}(\mathsf{raddr}_t,(t,\text{“raddr”})))$,

$\{\mathsf{lastwritten}_t\} := (\mathsf{lastwritten}_t,\ \Sigma.\mathsf{Sign}_{sk}(\mathsf{lastwritten}_t,(t,\text{“lastwritten”})))$

$\left\{\boxed{\mathsf{data}_t}\right\} := \left(\boxed{\mathsf{data}_t},\ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{data}_t},(t,\mathsf{waddr}_t,\text{“mem”}))\right)$ where $\boxed{\mathsf{data}_t} := \mathsf{data}_t \oplus \mathsf{PRF}_k(t,\mathsf{waddr}_t,\text{“mem”})$.

$\left\{\boxed{\mathsf{cpustate}_t}\right\} := \left(\boxed{\mathsf{cpustate}_t},\ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{cpustate}_t},(t,\text{“cpu”}))\right)$ where $\boxed{\mathsf{cpustate}_t} := \mathsf{cpustate}_t \oplus \mathsf{PRF}_k(t,\text{“cpu”})$.

If $t = qT_{\mathsf{ORAM}}$, compute $\left\{\boxed{y}\right\} := \left(\boxed{y},\ \Sigma.\mathsf{Sign}_{sk}(\boxed{y},(t,\text{“output”}))\right)$ where $\boxed{y} := y \oplus \mathsf{PRF}_k(t,\text{“output”})$.

Figure 12: Circuit $C^{\text{VC}}_{\textsc{NextIns}}$ to be obfuscated using VBB-secure obfuscation.

G.4 Security Proof

Theorem G.4. *Assume* OCompile *is secure stateful predictive ORAM compiler as defined Section G.2,* $\mathcal{O}$ *is a VBB obfuscator,* Σ *is an unforgeable signature scheme,* PRF *is a secure PRF, then the above construction is a simulation-secure VC-RAM as defined in Defintion G.2.*

The main proof idea is similar to that for simulation-based functional encryption in the RAM model. To prove that our scheme achieves the simulation-based security, we need to construct a simulator $\mathcal{S}$ so that for all adversary $\mathcal{A}$, the real experiment is computationally indistinguishable from the ideal experiment. The simulator $\mathcal{S}$ interacts with an adversary $\mathcal{A}$ and operates as follows:

- $\mathcal{S}$**.Setup.** The simulator $\mathcal{S}$ does not know the original D'. So it uses a fake $\widetilde{D}' := \vec{0}$ and follows the rest of the honest Setup algorithm honestly. That is, let $\mathsf{RAM} := (\vec{0}, \vec{0})$. Let $\mathsf{ORAM} := (\widetilde{D}, \widetilde{\mathsf{cpustate}}_0) := \mathsf{OCompile}(\mathsf{RAM})$.

  Generate secret PRF key k, and run the setup algorithm of a deterministic signature scheme $(sk, vk) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$.

  Compute $\left\{ \boxed{\widetilde{\mathsf{cpustate}}_0} \right\} := \left(\boxed{\widetilde{\mathsf{cpustate}}_0}, \;\; \Sigma.\mathsf{Sign}_{sk}(\boxed{\widetilde{\mathsf{cpustate}}_0}, (0, \text{“cpu”})) \right)$ where $\boxed{\widetilde{\mathsf{cpustate}}_0} := \widetilde{\mathsf{cpustate}}_0 \oplus \mathsf{PRF}_k(0, \text{“cpu”})$.

  For $i = 1$ to n_{ORAM}, compute $\left\{ \boxed{\widetilde{D}}\,[i] \right\} := \left(\boxed{\widetilde{D}}\,[i], \;\; \Sigma.\mathsf{Sign}_{sk}(\boxed{\widetilde{D}}\,[i], (0, i, \text{“mem”})) \right)$ where $\boxed{\widetilde{D}}\,[i] := \widetilde{D}[i] \oplus \mathsf{PRF}_k(0, i, \text{“mem”})$.

  Construct a VBB obfuscation $\mathcal{O}(O^{\mathrm{VC}}_{\textsc{NextIns}})$ of the following oracle $O^{\mathrm{VC}}_{\textsc{NextIns}}$ which hardwires the keys k, sk, and vk.

- $\mathcal{S}$.ProbGen. For each input, the simulator does not know the original input x_j, where $j \in [q]$. Therefore, it uses a fake input $\tilde{x}_j := \vec{0}$ and follows the rest of the ProbGen algorithm honestly.

- $\mathcal{S}$.Verify. Simulator $\mathcal{S}$ verifies the signature on the adversary supplied $\{ \boxed{y} \}$ under the tag $(qT_{\mathsf{ORAM}}, \text{“output”})$, where q denotes the number of the current query. If the signature verifies, the simulator outputs $\tilde{b} = 1$, otherwise, it outputs $\tilde{b} = 0$ to the adversary.

Based on the above simulator $\mathcal{S}$, we can obtain the ideal experiment $\textsc{Ideal}_{\mathcal{A},\mathcal{S}}$. In addition, based on the VC construction, we can easily obtain the real experiment $\textsc{Real}_{\mathcal{A}}$. Given any D' and inputs $(x_1, \ldots, x_m)$, the distributions in the both experiments are:

$$\mathsf{Dist}_{\textsc{Real}} = \left(\left(D', \mathcal{O}(C^{\mathrm{VC}}_{\textsc{NextIns}}), \left\{ \boxed{\mathsf{cpustate}_0} \right\}, \left(\left\{ \boxed{D}\,[i] \right\} \right)_{i \in [n_{\mathsf{ORAM}}]} \right), \left(x_i, \{ \boxed{x_i} \}, y_i, b_i \right)_{i \in [m]} \right)$$

$$\mathsf{Dist}_{\textsc{Ideal}} = \left(\left(D', \mathcal{O}(C^{\mathrm{VC}}_{\textsc{NextIns}}), \left\{ \boxed{\widetilde{\mathsf{cpustate}}_0} \right\}, \left(\left\{ \boxed{\widetilde{D}}\,[i] \right\} \right)_{i \in [n_{\mathsf{ORAM}}]} \right), \left(x_i, \{ \boxed{0^\lambda} \}, \tilde{y}_i, \tilde{b}_i \right)_{i \in [m]} \right).$$

Here $\tilde{b}_i$ is generated by $\mathcal{S}$.Verify. When $\tilde{b}_i = 0$, $\tilde{y}_i := \perp$. When $\tilde{b}_i = 1$, $\tilde{y}_i := f(D', x_1, \ldots, x_i)$.

Next, we modify the real and ideal experiments respectively and obtain $\textsc{Real}'$ and $\textsc{Ideal}'$.

Real′ and Ideal′. We define $\textsc{Real}'$ and $\textsc{Ideal}'$ the same as $\textsc{Real}$ and $\textsc{Ideal}$ except the experiments exclude the obfuscated circuits. (We give the distinguisher an oracle O^{VC}_1 instead, providing black box access to the circuit $C^{\mathrm{VC}}_{\textsc{NextIns}}$.) In particular,

$$\mathsf{Dist}_{\textsc{Real}'} = \left(\left(D', \left\{ \boxed{\mathsf{cpustate}_0} \right\}, \left(\left\{ \boxed{D}\,[i] \right\} \right)_{i \in [n_{\mathsf{ORAM}}]} \right), \left(x_i, \{ \boxed{x_i} \}, y_i, b_i \right)_{i \in [m]} \right)$$

$$\mathsf{Dist}_{\text{IDEAL}'} = \left(\left(D', \left\{ \boxed{\widetilde{\mathsf{cpustate}_0}} \right\}, \left(\left\{ \boxed{\widetilde{D}}\, [i] \right\} \right)_{i \in [n_{\mathsf{ORAM}}]} \right), \left(x_i, \{ \boxed{0^\lambda} \}, \tilde{y}_i, \tilde{b}_i \right)_{i \in [m]} \right).$$

Lemma G.5. *Assume $\mathcal{O}$ is VBB obfuscator. If there exist a* RAM*, a message x, a* PPT *algorithm $\mathcal{D}$ and some non-negligible $\delta()$ such that* $\Pr[\mathcal{D}(\mathsf{Dist}_{\text{REAL}}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\text{IDEAL}}) = 1] > \delta(\lambda)$*, then there exists another* PPT *algorithm $\hat{\mathcal{D}}$ so that* $\Pr[\hat{\mathcal{D}}^{O_1^{\text{VC}}}(\mathsf{Dist}_{\text{REAL}'}) = 1] - \Pr[\hat{\mathcal{D}}^{O_1^{\text{VC}}}(\mathsf{Dist}_{\text{IDEAL}'}) = 1] > \delta(\lambda) - \mathsf{negl}(\lambda)$*.*

Proof. Let $\alpha = \left(\left(D', \left\{ \boxed{\mathsf{cpustate}_0} \right\}, \left(\left\{ \boxed{D}\, [i] \right\} \right)_{i \in [n_{\mathsf{ORAM}}]} \right), \left(x_i, \{ \boxed{x_i} \}, y_i, b_i \right)_{i \in [m]} \right)$ and let $\mathcal{D}_\alpha$ denote algorithm $\mathcal{D}$ with α hardwired. Based on the assumption that $\mathcal{O}$ is VBB obfuscator, for algorithm $\mathcal{D}_\alpha$, there exists a VBB simulator Sim so that

$$\left| \Pr[\mathcal{D}_\alpha(\mathcal{O}(C^{\text{VC}}_{\text{NEXTINS}})) = 1] - \Pr[\mathsf{Sim}^{O_1^{\text{VC}}}() = 1] \right| < \mathsf{negl}(\lambda)$$

Now we can define $\hat{\mathcal{D}}$ as follows: upon input α, and oracle access to O_1^{VC}, $\hat{\mathcal{D}}$ internally runs Sim, providing Sim oracle access to O_1^{VC}; $\hat{\mathcal{D}}$ returns the bit that Sim returns. We can obtain

$$\begin{aligned}
&\Pr[\hat{\mathcal{D}}^{O_1^{\text{VC}}}(\mathsf{Dist}_{\text{REAL}'}) = 1] - \Pr[\hat{\mathcal{D}}^{O_1^{\text{VC}}}(\mathsf{Dist}_{\text{IDEAL}'}) = 1] \\
&= \Pr[\mathsf{Sim}^{O_1^{\text{VC}}}() = 1 \mid \text{REAL}'] - \Pr[\mathsf{Sim}^{O_1^{\text{VC}}}() = 1 \mid \text{IDEAL}'] \\
&> \Pr[\mathcal{D}_\alpha(\mathcal{O}(C^{\text{VC}}_{\text{NEXTINS}})) = 1 \mid \text{REAL}] - \Pr[\mathcal{D}_\alpha(\mathcal{O}(C^{\text{VC}}_{\text{NEXTINS}})) = 1 \mid \text{IDEAL}] - \mathsf{negl}(\lambda) \\
&= \Pr[\mathcal{D}(\mathsf{Dist}_{\text{REAL}}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\text{IDEAL}}) = 1] - \mathsf{negl}(\lambda) \\
&> \delta(\lambda) - \mathsf{negl}(\lambda)
\end{aligned}$$

This completes the proof of the lemma. □

Modified oracle O_2^{vc}. Here we consider a modified oracle O_2^{VC}. Here, whenever there is an oracle call for $t = qT_{\mathsf{ORAM}} + 1$, the circuit verifies consistency, decrypts the input, and precomputes all the "expected" inputs for all future time steps $qT_{\mathsf{ORAM}} + 2, qT_{\mathsf{ORAM}} + 3, \ldots, (q+1)T_{\mathsf{ORAM}}$ and remembers them in a table Γ. Later, if there are queries for all $t \in [qT_{\mathsf{ORAM}} + 2, \ldots, (q+1)T_{\mathsf{ORAM}}]$, the circuit looks into the table Γ, to find if it is one of the expected inputs. If so, output the correct outputs for this time step. Else, output $\perp$. The above modified oracle O_2^{VC} is based on honest setup and problem generation in the real experiment. Similarly, we can define modified O_2^{VC} based on the setup and problem generation in the ideal experiment.

Lemma G.6. *If Σ is unforgeable, then for any* PPT *algorithm $\mathcal{D}$, $\mathcal{D}$ cannot distinguish oracle O_1^{VC} from O_2^{VC} via black box queries, i.e.* $\Pr[\mathcal{D}^{O_1^{\text{VC}}}(1^\lambda) = 1] \approx \Pr[\mathcal{D}^{O_2^{\text{VC}}}(1^\lambda) = 1]$.

Proof. The two oracles are the same except the following bad event occurs: the input vector can be verified but such input vector is not computed by the circuit based on any input at $t = 1$. Note that the computation by the circuit is deterministic and it has been decided by the input at $t = 1$ (and other initial state). For any vector which is not generated by the circuit, if it can be verified, then we immediately obtain a forgery for the underlying digital signature. Under the assumption the signature scheme is unforgeable, the event only occurs with negligible probability. Otherwise, we can use $\mathcal{D}$ to make a forgery to the signature scheme. □

O_2^{VC}:

Initial state: k, and sk, vk;
In addition, information in setup and problem generation is stored in an adaptive fashion. That is, $\mathsf{cpustate}_0$ and $(D[i])_{i\in[n_{\mathsf{ORAM}}]}$, as well as $\left\{\boxed{\mathsf{cpustate}_0}\right\}$, $\left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n_{\mathsf{ORAM}}]}$ are stored. In addition, x_j and $\{\boxed{x_j}\}$ for $j \in [m]$ are stored whenever they are used in problem generation.

Inputs and **Outputs:** same as in $C^{\text{VC}}_{\textsc{NextIns}}$

Oracle description: Different from $\Sigma.\mathsf{Verify}_{vk}()$ used in $C^{\text{VC}}_{\textsc{NextIns}}$, here the circuit carries out verification through "book-keeping" mechanism.

- If $t = 1$, the input vector is $\left(\{\boxed{x}\}, \left\{\boxed{\mathsf{cpustate}_0}\right\}\right)$. If $\{\boxed{x}\}$ is different from the hardwired $\left\{\boxed{x_1}\right\}$, or $\left\{\boxed{\mathsf{cpustate}_0}\right\}$ is different from the hardwired one, then return $\perp$. Otherwise return Output_1 stored in table Γ as output. Table Γ will be defined below.
- If $t = qT_{\mathsf{ORAM}} + 1$, then the input vector is $\left(\{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \{\boxed{x}\}\right)$. If $\left(\{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}, \{\boxed{x}\}\right)$ is different from Input_t in table Γ, or $\{\boxed{x}\}$ is different from the hardwired $\left\{\boxed{x_q}\right\}$, then return $\perp$. Otherwise return Output_t.
- If $t = qT_{\mathsf{ORAM}} + \tau$, where $\tau \neq 1 \bmod T_{\mathsf{ORAM}}$, if the input vector has not been stored in table Γ, then return $\perp$. Otherwise, return Output_t as the output.

The table Γ is generated as follows. If $t = qT_{\mathsf{ORAM}} + 1$, the input vector has been verified, then carry out the following:

- Compute $\mathsf{fetched}_{t-1} := \boxed{x} \oplus \mathsf{PRF}_k(t, \text{"input"})$.
- For $t = qT_{\mathsf{ORAM}} + 1, qT_{\mathsf{ORAM}} + 2, \ldots, (q+1)T_{\mathsf{ORAM}}$:
  $(\mathsf{data}_t, \mathsf{waddr}_t, \mathsf{raddr}_t, \mathsf{cpustate}_t, \mathsf{lastwritten}_t, y) := \textsc{NextIns}\left(\mathsf{fetched}_{t-1}, \mathsf{cpustate}_{t-1}\right)$
  $\mathsf{fetched}_t := D[\mathsf{raddr}_t]$
  $D[\mathsf{waddr}_t] := \mathsf{data}_t$
  where the outcome y is only output if $t = (q+1)T_{\mathsf{ORAM}}$
- For $t = qT_{\mathsf{ORAM}} + 1, qT_{\mathsf{ORAM}} + 2, \ldots, (q+1)T_{\mathsf{ORAM}}$:
  $\boxed{\mathsf{data}_t} := \mathsf{data}_t \oplus \mathsf{PRF}_k(t, \mathsf{waddr}_t, \text{"mem"})$.
  $\boxed{\mathsf{cpustate}_t} := \mathsf{cpustate}_t \oplus \mathsf{PRF}_k(t, \text{"cpu"})$.
  $\{\mathsf{raddr}_t\} := (\mathsf{raddr}_t,\ \ \Sigma.\mathsf{Sign}_{sk}(\mathsf{raddr}_t, (t, \text{"raddr"}))),$
  $\{\mathsf{lastwritten}_t\} := (\mathsf{lastwritten}_t,\ \ \Sigma.\mathsf{Sign}_{sk}(\mathsf{lastwritten}_t, (t, \text{"lastwritten"})))$
  $\left\{\boxed{\mathsf{data}_t}\right\} := \left(\boxed{\mathsf{data}_t},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{data}_t}, (t, \mathsf{waddr}_t, \text{"mem"}))\right)$
  $\left\{\boxed{\mathsf{cpustate}_t}\right\} := \left(\boxed{\mathsf{cpustate}_t},\ \ \Sigma.\mathsf{Sign}_{sk}(\boxed{\mathsf{cpustate}_t}, (t, \text{"cpu"}))\right)$
- Define $\mathsf{Input}_t = \left(\{\mathsf{raddr}_{t-1}\}, \{\mathsf{lastwritten}_{t-1}\}, \left\{\boxed{\mathsf{fetched}_{t-1}}\right\}, \left\{\boxed{\mathsf{cpustate}_{t-1}}\right\}\right)$,
  and $\mathsf{Output}_t = \left(\{\mathsf{raddr}_t\}, \{\mathsf{lastwritten}_t\}, \mathsf{waddr}_t, \left\{\boxed{\mathsf{data}_t}\right\}, \left\{\boxed{\mathsf{cpustate}_t}\right\}\right)$. Note that for $t = qT_{\mathsf{ORAM}}$, the final outcome y is also included in $\mathsf{Output}_{qT_{\mathsf{ORAM}}}$. Store input vector Input_t and output vector Output_t in $\Gamma[t]$.

Real″ and Ideal″. We define $\textsc{Real}''$ and $\textsc{Ideal}''$ the same as $\textsc{Real}'$ and $\textsc{Ideal}'$ except that in

setup stage, instead of generating $\boxed{\mathsf{cpustate}_0} := \mathsf{cpustate}_0 \oplus \mathsf{PRF}_k(0, \text{“cpu”})$, and $\boxed{D}[i] := D[i] \oplus \mathsf{PRF}_k(0, i, \text{“mem”})$ for all $i \in [n_{\mathsf{ORAM}}]$, here compute . $\boxed{\mathsf{cpustate}_0} := \mathsf{cpustate}_0 \oplus R_0$, and $\boxed{D}[i] := D[i] \oplus R_i$ where R_0 and R_i are randomly chosen. Similarly, for each input x_j, instead of generating $\boxed{x_j} := x_j \oplus \mathsf{PRF}_k(t, \text{“input”})$, compute $\boxed{x_j} := x_j \oplus R'_j$ where R'_j is randomly chosen. The setup information and the problem generation information will be hardwired in oracle O_3^{vc} below. The distributions of two experiments are

$$\mathsf{Dist}_{\mathrm{REAL}''} = \left(\left(D', \left\{\boxed{\mathsf{cpustate}_0}\right\}, \left(\left\{\boxed{D}[i]\right\}\right)_{i\in[n_{\mathsf{ORAM}}]}\right), \left(x_i, \{\boxed{x_i}\}, y_i, b_i\right)_{i\in[m]}\right)$$

$$\mathsf{Dist}_{\mathrm{IDEAL}''} = \left(\left(D', \left\{\boxed{\widetilde{\mathsf{cpustate}_0}}\right\}, \left(\left\{\boxed{\widetilde{D}}[i]\right\}\right)_{i\in[n_{\mathsf{ORAM}}]}\right), \left(x_i, \{\boxed{0^\lambda}\}, \tilde{y}_i, \tilde{b}_i\right)_{i\in[m]}\right).$$

Lemma G.7. *If* PRF *is secure, for any* PPT $\mathcal{D}$, $|\Pr[\mathcal{D}(\mathsf{Dist}_{\mathrm{REAL}'}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\mathrm{REAL}''}) = 1]| < \mathsf{negl}(\lambda)$ *and* $|\Pr[\mathcal{D}(\mathsf{Dist}_{\mathrm{IDEAL}'}) = 1] - \Pr[\mathcal{D}(\mathsf{Dist}_{\mathrm{IDEAL}''}) = 1]| < \mathsf{negl}(\lambda)$.

Proof. This follows directly from the security of the PRF. □

Modified oracle $O_3^{\mathbf{vc}}$. O_3^{vc} is the same as O_2^{vc} except that in Encrypt steps, a random string is used as one time pad.

O_3^{vc}:

Initial state: secret keys vk, sk, and a random string as one time pad.
As in O_2^{vc}, information in setup and problem generation is stored. But now in setup and problem generation, instead of using PRF to generate one time pad, random keys are used.

Inputs: same as in $C^{\mathrm{vc}}_{\mathrm{NextIns}}$
Outputs: same as in $C^{\mathrm{vc}}_{\mathrm{NextIns}}$

Oracle description:

Same as that in oracle O_2^{vc} except that in Encrypt steps, instead of using PRF to generate a pseudorandom string as the one-time pad, here a random string is used as the one time pad.

Similarly, we can define oracle O_3^{vc} based on setup and problem generation in the ideal experiment.

Lemma G.8. *If* PRF *is secure, for any* PPT *algorithm* $\mathcal{D}$, $\mathcal{D}$ *cannot distinguish oracle* O_2^{vc} *from* O_3^{vc} *via black box queries, i.e.* $\Pr[\mathcal{D}^{O_2^{\mathrm{vc}}}(1^\lambda) = 1] \approx \Pr[\mathcal{D}^{O_3^{\mathrm{vc}}}(1^\lambda) = 1]$.

Proof. This follows directly from the security of the PRF. □

Modified oracle $O_4^{\mathbf{vc}}$. O_4^{vc} is the same as O_3^{vc} except on the query corresponding to $t = 1$, instead of using real memory addresses, the new circuit will use simulated memory addresses generated by predictive ORAM simulator Sim'.

O_4^{VC}:

Initial state: same as that in O_3^{VC}

Inputs: same as in $C^{\text{VC}}_{\text{NextIns}}$
Outputs: same as in $C^{\text{VC}}_{\text{NextIns}}$

Oracle description:

Same as that in oracle O_3^{VC} except that for each input value x, the corresponding table Γ_x is created in a different way. Instead of generating addresses honestly, now the addresses are computed by running the predictive ORAM simulator, i.e., compute $(\{\mathsf{waddr}_t, \mathsf{raddr}_t\}_{t\in[T_{\mathsf{ORAM}}]}) \leftarrow \mathsf{Sim}'()$. The circuit then computes $\mathsf{lastwritten}_t$ for each $t \in [T_{\mathsf{ORAM}}]$.

Similarly, we can define oracle O_4^{VC} based on setup and problem generation in the ideal experiment.

Lemma G.9. *If* OCompile *is secure predictive ORAM compiler as defined in Section G.2, for any* PPT *algorithm $\mathcal{D}$, $\mathcal{D}$ cannot distinguish oracle O_3^{VC} from O_4^{VC} via black box queries, i.e.* $\Pr[\mathcal{D}^{O_3^{\text{VC}}}(1^\lambda) = 1] \approx \Pr[\mathcal{D}^{O_4^{\text{VC}}}(1^\lambda) = 1]$.

Proof. This directly follows by the security of stateful predictive ORAM defined in Section G.2. Assume there is an algorithm $\mathcal{D}$ who can distinguish O_3^{VC} from O_4^{VC} via black box access, then we can construct another algorithm $\mathcal{D}'$ who can break the security of the predictive ORAM complier. $\mathcal{D}'$ internally simulates $\mathcal{D}$ and the oracle except that the addresses are generated by an external algorithm either by $\mathsf{addresses}(\mathsf{ORAM}, \cdot)$, or by $\mathsf{Sim}'()$. If the external algorithm is by $\mathsf{addresses}()$, then $\mathcal{D}$'s view is the same as that in O_3^{VC}; and if the external algorithm is Sim', then $\mathcal{D}$'s view is the same as that in O_4^{VC}. Since $\mathcal{D}$ can distinguish the two experiments with non-negligible probability, that means $\mathcal{D}'$ can distinguish the real addressees generated by $\mathsf{addresses}()$ from the simulated addresses produced by $\mathsf{Sim}'()$ with non-negligible probability. That means $\mathcal{D}'$ can break the security of stateful predictive ORAM complier OCompile, which contracts the assumption. □

Now we can easily see that Real'' and Ideal'' are identically distributed.

Lemma G.10. *For any algorithm $\mathcal{D}$, $\mathcal{D}$ cannot distinguish oracle O_3^{VC} from O_4^{VC} via black box queries, i.e.* $\Pr[\mathcal{D}^{O_4^{\text{VC}}}(\mathsf{Dist}_{\text{Real}''}) = 1] = \Pr[\mathcal{D}^{O_4^{\text{VC}}}(\mathsf{Dist}_{\text{Ideal}''}) = 1]$.

We can now complete the proof of security.

Proof of Theorem G.4. From Lemma G.5, assume that there exists a PPT algorithm $\mathcal{D}$ which can distinguish $\text{Real}_{\mathcal{A}}$ from $\text{Ideal}_{\mathcal{A}}$ with non-negligible probability, then we can construct a PPT algorithm $\hat{\mathcal{D}}$ to distinguish Real'' from Ideal'' with non-negligible probability. Then from Lemmas G.6, G.7, G.8, and G.9, we know $\hat{\mathcal{D}}^{O_4^{\text{VC}}}$ distinguishes Real'' from Ideal''. This is a contradiction to Lemma G.10. That means, the real experiment and the ideal experiment are not distinguishable, which completes the proof. □

G.5 Predictive ORAM Construction

Lu and Ostrovsky [36] are the first to define predictive memory ORAM (or predictive ORAM for short), and show that their ORAM construction is predictive memory.

We now show how to convert any RAM into a predictive RAM, where T is the total RAM execution time. Our constructions also works for stateful RAMs.

We now give a constructive proof. First, given a RAM, we first convert it into an equivalent ORAM using existing ORAM techniques.

Now, we show how to transform an ORAM to be a predictive ORAM. The idea is to store an additional table Γ_1 in memory, where entry i of the table $\Gamma_1[i] := \mathsf{lastwritten}_t(i)$, i.e., stores the last time the i-th memory word is written to at the current time t. When the RAM needs to access the i-th memory word, it first accesses this table to find out $\mathsf{lastwritten}_t(i)$. It seems that this is circular logic: how do you know when the memory location $\Gamma_1[i]$ is last accessed? Fortunately, observe that since $\ell > c \log T$, the table Γ_1 requires $n_\emptyset / c$ words to store, a constant factor smaller than the original number of words $n_\emptyset$. We can now divide entries of Γ_1 into words of size ℓ, and recursively use another table Γ_2 to store when each word of table Γ_1 was last accessed. After $k = O(\log_c n)$ rounds of recursion, we end up with a table Γ_k whose size is $O(1)$ (in terms of number of memory words) – and the RAM can simply read the entire table Γ_k to access it. The RAM also keeps as part its CPU state when this $O(1)$-sized table Γ_k is last accessed.

Whenever a memory address needs to be over-written, we recursively update the corresponding entries in Γ_1, Γ_2, …, Γ_k with the new time of write. We also update the CPU states to remember the new time of write for Γ_k.

This introduces $\log_c(n)$ memory access overhead for reading each memory word, i.e., reading a single memory word in the original RAM now requires reading $\log_c(n)$ memory words in the transformed, predictive memory RAM; and similarly for writing. In other words, if the original RAM executes in T time, the transformed RAM executes in time $T \log_c(n)$.

The additional memory addresses accessed during table lookup sequence depend only on the physical memory address accessed during the original RAM's execution. Clearly, if the original RAM is oblivious, so is the transformed one.

It is not hard to see that transformed ORAM is oblivious, since we first applied the ORAM compiler before we performed this predictive last-write transformation.

We note that a similar recursion trick has been used in some ORAM constructions [43, 42] to recursively reduce the trusted metadata.