Cryptography for Parallel RAM from Indistinguishability Obfuscation[*]

Yu-Chi Chen[†] Sherman S. M. Chow[‡] Kai-Min Chung[§] Russell W. F. Lai[¶]
Wei-Kai Lin[‖] Hong-Sheng Zhou[**]

January 1, 2016

Abstract

Since many cryptographic schemes are about performing computation on data, it is important to consider a computation model which captures the prominent features of modern system architecture. Parallel random access machine (PRAM) is such an abstraction which not only models multiprocessor platforms, but also new frameworks supporting massive parallel computation such as MapReduce.

In this work, we explore the feasibility of designing cryptographic solutions for the PRAM model of computation to achieve security while leveraging the power of parallelism and random data access. We demonstrate asymptotically optimal solutions for a wide-range of cryptographic tasks based on indistinguishability obfuscation. In particular, we construct the first publicly verifiable delegation scheme with privacy in the persistent database setting, which allows a client to privately delegate both computation and data to a server with optimal efficiency. Specifically, the server can perform PRAM computation on private data with parallel efficiency preserved (up to poly-logarithmic overhead). Our results also cover succinct randomized encoding, searchable encryption, functional encryption, secure multiparty computation, and indistinguishability obfuscation for PRAM.

We obtain our results in a modular way through a notion of computational-trace indistinguishability obfuscation (CiO), which may be of independent interests.

[*]This is the full version of the extended abstract to appear at ACM Innovations in Theoretical Computer Science (ITCS) 2016. Previous version of this paper is known as "Computation-Trace Indistinguishability Obfuscation and its Applications".

[†]Academia Sinica, Taiwan (wycchen@iis.sinica.edu.tw). This work is partially supported by Academia Sinica Postdoctoral Fellowship.

[‡]The Chinese University of Hong Kong, Hong Kong (sherman@ie.cuhk.edu.hk). This work is supported in part by the Early Career Award and the grants from the Research Grants Council, Hong Kong (CUHK 439713 & 14201914). Part of this work was done while the author was visiting Academia Sinica.

[§]Academia Sinica, Taiwan (kmchung@iis.sinica.edu.tw). This work is partially supported by Ministry of Science and Technology, Taiwan, under Grant no. MOST 103-2221-E-001-022-MY3. This work was done in part while the author was visiting the Simons Institute for the Theory of Computing, supported by the Simons Foundation and by the DIMACS/Simons Collaboration in Cryptography through NSF grant #CNS-1523467; and while the author was visiting The Chinese University of Hong Kong.

[¶]The Chinese University of Hong Kong, Hong Kong (wflai@ie.cuhk.edu.hk).

[‖]Academia Sinica, Taiwan (wklin@iis.sinica.edu.tw).

[**]Virginia Commonwealth University, VA, USA (hszhou@vcu.edu).

Contents

1 Introduction

1.1 The PRAM Model

The parallel random-access machine (PRAM) is an abstract computation or programming model of a canonical structured parallel machine. It consists of a polynomial number of synchronous processors. Each of them is similar to an individual (non-parallel) RAM with its central processing unit (CPU) performing computation locally. In addition to the local memory, CPUs in PRAM have random access of a common array of memory which is potentially unbounded. Parallel and distributed computing community suggested many algorithms which are parallelizable in the PRAM model, resulting in an exponential gap between solving the same problem in the RAM and PRAM models. Examples include parallel sorting or searching in a database, which have linear size input but run in polylogarithmic time.

Being an abstract model, PRAM not only models multiprocessor platforms, but also new frameworks in the big-data era such as MapReduce, GraphLab, Spark, etc. Running time is a critical factor, especially when data is being generated in every second worldwide which are too big to be processed by traditional information processing technique or by a single commodity computer. For individuals, or even enterprises without in-house resource/expertise, there is an emerging demand for *delegation* of both data and computation to a third-party server, often called “the cloud”, a distributed computing platform with a large amount of CPUs to perform computations in *parallel*. We found PRAM a clean theoretical model to work with for these scenarios.

PRAM with Persistent Database With the high volume of data to process and the potentially high volume of output data, it is natural to perform multiple computations over the *“big data”* that persists in the cloud storage. Such functionality is supported by introducing the notion of *persistent database* on top of the PRAM model. A motivating example is a special kind of delegation, known as searchable symmetric encryption (SSE), which features parallel search and update algorithms.

1.2 Crypto for PRAM

Many cryptographic schemes are about performing computation on data. Traditionally, cryptographers worked on the circuit model of computations; for example, the celebrated result of Yao’s garbled circuit for two-party computation [Yao86]. Many cryptographic notions can be benefited by parallelism and persistent database.

Secure Multiparty Computation (SMC) Secure multiparty computation (SMC) generalizes two-party computation. Consider using SMC on electronic health record (EHR) for collaborative research, EHR often involves patients’ medical and genetic information which are often expensive to collect and should be kept confidential as mandated by law. Such kind of large-scale SMC [BCP15] further motivates the benefits of PRAM.

Although (highly optimized) circuit-based SMC protocols and RAM-based solutions of SMC exist, they have inherent drawbacks. Circuit-based solutions are not feasible for big data since circuit representations are huge and the (worst case) runtime can be dependent on the input length. Consequently, it cannot represent sublinear time algorithm. Existing RAM-based solutions cannot exploit parallelism even when the program is parallelizable (which is often the case for processing big data). On the other hand, PRAM is an expressive model to capture the requirements in this case.

Secure and Efficient Delegation Security concern manifests in various forms when we consider outsourcing. For a concrete discussion, we consider the *delegation* problem, faced by an enterprise which is outsourcing a newly-developed big data analytic algorithm for uncovering market trends from the customer preferences collected. Data owners demand confidentiality. Secrecy of the algorithm is also desired, or competitors may gather the same kind of business intelligence (from their own data). The output of data analytics is also sensitive, both its confidentiality and authenticity (i.e., the correctness of the algorithm invocation) are crucial for the success of any corresponding strategic plan. It is risky to place all these strong trust in different dimensions on the cloud. Cloud client should safeguard the outsourcing process by resorting to cryptography.

The next concern is about efficiency. The client would want both storage and computation required for secure delegation to be significantly less than the actual data and computation. On the other hand, the server, who is actually storing the data and performing the computation, would like to operate on the (private) data as a PRAM program, and not to perform too much work when compared to computation on the plaintext data. There exists verifiable delegation with privacy, but the solution is based on the circuit model, and is far from suitable for outsourcing big data. Recent work provides heuristic solution for RAM delegation with persistent database [GHRW14], but the solution is only a heuristic one based on a stronger variant of differing-inputs obfuscation (diO), which is subject to implausibility result.

More formally, we consider secure delegation of PRAM program with persistent database, as follows. A (very large) database x is firstly delegated from the client to the server. The client can make arbitrary number of PRAM program queries to the server, who performs the computation to update the database, and returns the answer with a proof. Ideally, we want the efficiency to match the "unsecured" solution. Namely, delegating database takes $O(|x|)$ cost, and for each PRAM program query, the client's runtime depends only on the description size of the PRAM program, the server's parallel runtime is linear in the parallel runtime of the program, and the client's verification time is independent of the program complexity and the database size.

We pose ourselves this question: Can we outsource both data and computation, leveraging parallelism and random data access, i.e., in the PRAM model?

Functional Encryption (FE) Another primitive in cloud cryptography which attracts much attention recently is functional encryption (FE), a generalized notion of attribute-based encryption (ABE) originally proposed for enforcing cryptographic access control. FE enables a user with the function key for $f(\cdot)$ to learn $f(x)$ given an encryption of x. Consider x to be the encrypted cloud storage and each user can only access part of the shared storage space (for obvious security reason). FE for PRAM means that the function key can be associated to a PRAM program taking the large x as an input. The access control policy can be very general. We can even embed some sort of parallel logic into it for operating on the relevant parts of the cloud storage in parallel.

We remark that a very recent result achieved FE for Turing machines with unbounded input [AS16].

1.3 Our Goal

To summarize, current study of cryptography does not work in a model which fully leverages the important features of modern architecture to handle the computation problem nowadays; namely, massive parallel computation on big data. In this work, we address the following basic question:

> *"How to do Cryptography in the PRAM model — How to design cryptographic solutions that achieve security and simultaneously leverage the power of parallelism and random data access?"*

Our work provides general feasibility and asymptotically optimal results for various important cryptographic primitives based on indistinguishability obfuscation for circuits (i$\mathcal{O}$).

1.4 Summary of Our Results

We develop techniques to obtain (asymptotically) optimal constructions for several cryptographic primitives (i.e., multiparty computation, delegation, and functional encryption) in the PRAM model. We do so in modular steps, and our results are presented below. Please also refer to Table 1 for the efficiency of our schemes.

Computation-Trace Indistinguishability Obfuscation First, we define a new primitive named computation-trace indistinguishability obfuscation (Ci$\mathcal{O}$), which obfuscates a computation instance instead of a program. A computation instance Π is defined by a program P and an input x. Evaluation of Π produces a *computation trace*; namely, all CPU states, memory content, and memory access instructions throughout the computation. A Ci$\mathcal{O}$ obfuscator takes in a computation instance Π as an input, and outputs $\tilde{\Pi}$ as an obfuscated computation instance that can be evaluated to correctly output $P(x)$. We only require a very weak indistinguishability-based security for $\mathsf{Ci}\mathcal{O}$, where two obfuscations $\mathsf{Ci}\mathcal{O}(\Pi)$ and $\mathsf{Ci}\mathcal{O}(\Pi')$ are required to be indistinguishable only when the evaluation of Π and Π' produce identical computation trace (which implies their inputs are the same). While the security is weak, we demand stringent efficiency that the obfuscator's runtime depends only on the instance description size, but not the evaluation runtime.

We construct $\mathsf{Ci}\mathcal{O}$-RAM based on $\mathsf{i}\mathcal{O}$ for circuits and one-way functions, by adopting techniques developed in a very recent result due to Koppula, Lewko, and Waters [KLW15] (hereinafter referred as KLW). We then (non-trivially, to be elaborated in the next section) extend it into $\mathsf{Ci}\mathcal{O}$-PRAM. The main challenge is to avoid linear overhead on the number of CPUs in both *parallel runtime* and *obfuscation size* — note that such overhead would obliterate the gain of parallelism for a PRAM computation. To summarize, we have:

Theorem 1.1 (Informal)**.** *Assuming the existences of indistinguishability obfuscation* ($\mathsf{i}\mathcal{O}$) *and one-way functions* (OWF)*, there exists (fully succinct) computation-trace indistinguishability obfuscation for PRAM computation.*

While the notion of $\mathsf{Ci}\mathcal{O}$ is weak, we immediately obtain optimal publicly-verifiable delegation of PRAM computation. In particular, the program encoding has size independent of the output length.

Corollary 1.2 (Informal)**.** *Assuming the existences of* $\mathsf{i}\mathcal{O}$ *and* OWF*, there exists a two-message publicly-verifiable delegation scheme for PRAM computation, where the delegator's runtime depends only on the program description and input size, and the server's complexity matches the PRAM complexity up to polynomial factor of program description size.*

Fully Succinct Randomized Encoding More importantly, we show how to use our (fully succinct) $\mathsf{Ci}\mathcal{O}$-PRAM to construct the *first fully succinct* randomized encoding ($\mathcal{RE}$) for PRAM computation. The notion of *randomized encoding*, proposed by Ishai and Kushilevitz [IK00], allows a "complex" function f on an input x to be represented by a "simpler to compute" randomized encoding $\hat{f}(x;r)$ whose output distribution encodes $f(x)$, such that the encoding reveals nothing else regarding f and x, and one can decode by extracting $f(x)$ from $\hat{f}(x;r)$. The original measure of simplicity [IK00] considers the circuit depth (i.e., parallel runtime) of the encoding. Very recently, Bitansky, Garg, Lin, Pass, and Telang [BGL+15] focus on encoding time. Bitansky et al. consider f as represented by a RAM program P, and construct *(space-dependent) succinct randomized encodings* where the encoding time is independent of the time complexity of $P(x)$ (as a RAM program evaluation), but depends on the space complexity of $P(x)$.[1]

We extend the $\mathcal{RE}$ notion further to the PRAM model. More precisely, given a PRAM computation instance Π defined by a PRAM program P and an input x, an $\mathcal{RE}$-PRAM generates a randomized encoding $\tilde{\Pi} = \mathcal{RE}.\mathsf{Encode}(\Pi)$ that can be decoded/evaluated to obtain $P(x)$, but reveals nothing else regarding both P and x (except the size/time/space bound of $P(x)$). *Full succinctness* means the encoder's runtime (and thus the encoding size) depends on the description size of P, the input length of x, and the output length of $P(x)$, but is essentially independent of both time and space complexities of $P(x)$. To the best of our knowledge, there was *no known fully succinct construction* of $\mathcal{RE}$, even in the RAM model, before our result.

Theorem 1.3 (Informal)**.** *Assuming the existence of* $\mathsf{i}\mathcal{O}$ *and* OWF*, there exists* fully succinct *randomized encoding for PRAM, where the encoding time depends only on the program description and input/output size, and the server's complexity matches the PRAM complexity of the computation up to polynomial factor of program description size.*

Remark 1.4. *In the $\mathcal{RE}$ construction, the output is not private. Actually, when the output is private, we can provide constructions with slightly better efficiency where the encoding time depends only on program description and input size, but* independent of the output size. *See Table 1.*

[1]Canetti et al. [CHJV15] achieved a similar result in the context of garbling.

Scheme	Encoding Time for Each Program P_i	Encoding Time for Database	Decoding Time	Decoding Space
PRAM unsecured delegation	$\|P_i\|$	$\|D\|$	$T_i \cdot \|P_i\|$	$\tilde{O}(m)+S_i$
Ci$\mathcal{O}$-PRAM / Delegation without privacy	$\tilde{O}(\mathsf{poly}(\|P_i\|)+\ell_i^{\text{out}})$	$\tilde{O}(\|D\|)$	$\tilde{O}(T_i \cdot \mathsf{poly}(\|P_i\|))$	$\tilde{O}(m+S_i)$
$\mathcal{RE}$-PRAM / Delegation with program and input privacy	$\tilde{O}(\mathsf{poly}(\|P_i\|)+\ell_i^{\text{out}})$	$\tilde{O}(\|D\|)$	$\tilde{O}(T_i \cdot \mathsf{poly}(\|P_i\|))$	$\tilde{O}(m+S_i)$
Delegation with full privacy	$\tilde{O}(\mathsf{poly}(\|P_i\|))$	$\tilde{O}(\|D\|)$	$\tilde{O}(T_i \cdot \mathsf{poly}(\|P_i\|))$	$\tilde{O}(m+S_i)$

Table 1: Summarizing efficiency of our schemes in PRAM with persistent database, where the computation consists of L sessions among m parallel CPUs and a shared database D: In each session $i \in [L]$, program F_i is executed with input-size ℓ_i^{in}, output-size ℓ_i^{out}, using time T_i and space S_i. We use $\tilde{O}$ to ignore logarithmic factors. We remark that the efficiency for schemes in the single-session setting can easily be derived from the table by dropping the subscript i, and by replacing $|D|$ (the encoding time for database) with ℓ^{in} (the encoding time for input). The efficiency for schemes in the RAM setting can also be derived by setting $m = 1$.

Building Cryptography for PRAM By plugging our $\mathcal{RE}$-PRAM into various transformations in the literature [GHRW14, BGL$^+$15, CHJV15], we obtain the first constructions of a wide range of cryptographic primitives for PRAM (with the corresponding full succinctness), including non-interactive zero-knowledge, functional encryption, garbling, secure multi-party computation, and indistinguishability obfuscation for PRAM, and we have the following two corollaries.

Corollary 1.5 (Informal)**.** *Assuming the existences of* i$\mathcal{O}$ *and* OWF*, there exist (fully) succinct non-interactive zero-knowledge, functional encryptions with succinct (PRAM) function keys, succinct reusable garbling, and secure multi-party computation for PRAM with optimal communication.*

Notably, while Ci$\mathcal{O}$ is syntactically weaker than i$\mathcal{O}$, sub-exponential Ci$\mathcal{O}$-PRAM still implies i$\mathcal{O}$ for PRAM with sub-exponential security by complexity leveraging (e.g., [BGL$^+$15, CHJV15]).

Corollary 1.6 (Informal)**.** *Sub-exponentially secure* Ci$\mathcal{O}$*-PRAM implies sub-exponentially secure* i$\mathcal{O}$ *for PRAM.*

Optimal Delegation with Persistent Database Finally, we generalize to the persistent database setting where a computation consists of a database and multiple programs. The generalization is straightforward, and leads to optimal delegation with persistent database.

Theorem 1.7 (Informal)**.** *Assuming the existence of* i$\mathcal{O}$ *and* OWF*, there exists* fully succinct *delegation schemes for PRAM with persistent database, where the encoding time depends on the database size and the size of each program description, and the server's complexity matches the PRAM complexity of the computation up to polynomial factor of program description size.*

We remark that this immediately gives us the feasibility of optimal symmetric searchable encryption without leakage.

1.5 Related Works

Independent and Concurrent Work Canetti and Holmgren [CH15] also proposed a fully succinct garbling scheme for RAM programs, based on the same assumption of the existences of i$\mathcal{O}$ and OWF. However, our motivation is different. Specifically, we aim for developing cryptographic solutions for PRAM model of computation to capture the power of both parallelism and random data access. Achieving full succinctness in the PRAM model is a major technical novelty of our result.

On the technical level, we note that both our construction and theirs can be viewed as a natural generalization and modularization of the construction of KLW for succinct encoding for Turing machines. Both works first construct a succinct encoding that satisfies a weak indistinguishability-based security (in our case, the notion of Ci$\mathcal{O}$). With this encoding, both rely on encryption and oblivious RAM (ORAM) to hide the memory content and access pattern of the RAM computation respectively. At the core of both security proofs are approaches to "puncture" ORAM execution to switch the access pattern step by step. From here, Canetti and Holmgren [CH15] additionally introduce a novel dual-encryption mechanism with a security property of tree-based ORAM constructions, which makes their security analysis more modular, at the cost of slightly increasing the security loss in the hybrid. Their techniques can be generalized to provide a more modular proof of $\mathcal{RE}$-PRAM from Ci$\mathcal{O}$-PRAM.[2]

Other Related Works and Open Problems As mentioned, Boyle et al. [BCP16, BCP15] constructed the first oblivious PRAM (OPRAM) compiler and applied it to obtain secure MPC for PRAM. Recently, Chen, Lin, and Tessaro [CLT16] showed a more efficient OPRAM compiler, as well as a generic transformation taking any generic ORAM compiler to an OPRAM compiler. However, the compilers of [CLT16] only apply to PRAM programs with a fixed number of CPUs. Boyle et al. [BCP16] also constructed the first (non-succinct) garbling PRAM schemes based on identity-based encryptions. This is subsequently improved by a very recent work of Lu and Ostrovsky [LO15], who proposed a black-box construction based on the minimal assumption of OWF.

In succinct i$\mathcal{O}$-based setting with persistent database, our construction achieves only selective security, where the database and programs are chosen by the adversary in advance. Two independent subsequent works [ACC$^+$15, CCHR15] achieved stronger adaptive security, where the adversary can make adaptive queries based on previous database and program encodings. Both works rely on additional assumptions than i$\mathcal{O}$ and OWF.

Finally, while we demonstrated general feasibility results for several cryptographic primitives for PRAM, our constructions are based on very strong and less well-understood assumptions of indistinguishability obfuscations. A natural and important research direction is to understand the landscape of cryptography for PRAM without assuming i$\mathcal{O}$. For example, can we construct attribute-based encryptions and functional encryptions for PRAM based on the learning with errors (LWE) assumption?

1.6 Paper Outline

Sections 2 gives a very high level overview of the paper. Experienced readers can jump directly to Section 3 for a more detailed overview. For a more compact presentation, we move the preliminaries to Appendix A.

The formal description of our results starts from Section 4, where we define the new notion of Computation-Trace Indistinguishability Obfuscation (Ci$\mathcal{O}$). The constructions of Ci$\mathcal{O}$ in the RAM and PRAM models are described in Section 5 and 6 respectively. Next, in Sections 7 and 8, we extend Ci$\mathcal{O}$ to randomized encoding ($\mathcal{RE}$) in the RAM and PRAM model respectively. Various extensions of $\mathcal{RE}$ for different delegation scenarios can be found in Section 9. Finally, all security proofs are consolidated in Appendix B.

2 Constructions Overview

Our starting point is the succinct primitives (message-hiding encodings and machine-hiding encodings) for Turing machines constructed by Koppula, Lewko, and Waters (KLW) [KLW15]. Our constructions are natural generalizations of their constructions to handle PRAM with persistent database, where the major challenge is to develop new techniques to handle parallel processors and random access pattern. On the conceptual level, our constructions are modular and simple. Therefore, in this section, we first focus on illustrating our constructions. We will include a brief description of the application of the techniques by KLW in our context, and discuss our new techniques in the next section. We start by describing the way we view (parallel) RAM model.

2.1 The (Parallel) RAM Model

In the RAM model, computation is done by the CPU with random access to the memory in time steps (CPU cycles). At each time step t, the CPU (represented as a *next-step circuit*) receives the read memory content,

[2]We could include the modular proof; yet, we think it would be better to keep the two works separate for the readers.

performs one step of computation to update its CPU state, and outputs a memory access (read or write) instruction for time step $t+1$. The computation terminates when the CPU reaches a special halting state. A RAM computation instance Π is defined by a CPU next-step program P, and an input x stored in the memory as initial memory content (and a default initial CPU state). At the end of the computation, the output y is stored in the CPU state (with the special halting symbol).

The PRAM model is similar to the RAM model, except that there are multiple CPUs executing in parallel with random access to a shared memory (and reaching the halting state at the same time). The CPUs share the same CPU program P, but have distinct CPU id's. In this overview, we assume that there is no conflict writes throughout the computation. We note that our construction can handle the general CRCW (concurrent read concurrent write) model.

2.2 High Level Ideas

Let us motivate our work through the context of delegation, where a client delegates computation of a PRAM instance $\Pi = (P, x)$ to a server. Without security consideration, the client can simply send Π in the clear to the server, who can evaluate the PRAM program and return $y = P(x)$. Our goal is to achieve *publicly verifiable* delegation with *privacy* and (asymptotically) the same client and server efficiencies. Specifically, the server learns nothing except for the output y, whose correctness can be verified publicly.[3]

At a high level, we let the client send an obfuscated program $\tilde{P}$ and encoded input $\tilde{x}$ to the server with the aim that the obfuscation and encoding hide P and x, yet allowing the server to perform PRAM evaluation on $\tilde{P}(\tilde{x})$. (Obfuscation preserves input/output behavior and thus allows PRAM evaluation). To protect the privacy of P, we must restrict $\tilde{P}$ to evaluate *only* on input x, since $P(x')$ may leak additional information about P beyond $y = P(x)$. We need some *authentication mechanism* to *authenticate* the whole evaluation of P on x but nothing else. Moreover, the evaluation of P on x produces a long computation trace in addition to y. We need some *hiding mechanism* to hide the evaluation process. We discuss these two major ingredients in turn. First, we show the design of an authentication mechanism which gives $\mathsf{Ci}\mathcal{O}$ for RAM and PRAM from $\mathsf{i}\mathcal{O}$ for circuits. Next, we show the design of a hiding mechanism which gives $\mathcal{RE}$ for RAM and PRAM from $\mathsf{Ci}\mathcal{O}$ in the respective models.

2.3 $\mathsf{Ci}\mathcal{O}$ Construction

Our construction of $\mathsf{Ci}\mathcal{O}$ for (parallel) RAM computation is based on $\mathsf{i}\mathcal{O}$ for circuits and the novel $\mathsf{i}\mathcal{O}$-friendly authentication techniques developed originally to build $\mathsf{i}\mathcal{O}$ for Turing machines (TM) [KLW15]. Let Π be a computation instance for (parallel) RAM computation defined by (P, x), where P is represented as a next-step circuit for the CPU program and x is the input.[4] The goal is to allow $\tilde{P}$ to evaluate on x but nothing else. At a high level, our $\mathsf{Ci}\mathcal{O}$ construction outputs $\mathsf{i}\mathcal{O}$ of a compiled version of P and a compiled input. We proceed to discuss the intuition of our construction.

Recall that if two computation instances Π and Π', defined by (P, x) and (P', x') respectively, have identical computation trace (which implies $x = x'$), the security of $\mathsf{Ci}\mathcal{O}$ requires that their $\mathsf{Ci}\mathcal{O}$-ed computation instances should be computationally indistinguishable, i.e., $\mathsf{Ci}\mathcal{O}(\Pi) \approx \mathsf{Ci}\mathcal{O}(\Pi')$. The two programs P and P' may behave differently on other inputs $x'' \neq x$. So, to ensure $\mathsf{Ci}\mathcal{O}(\Pi) \approx \mathsf{Ci}\mathcal{O}(\Pi')$, we must restrict the obfuscation to only evaluate the program on the specific input x, but not other inputs. A natural approach is to use *authentication*. Note that computation involves updating CPU states and memory, where the latter may be large in size. We can authenticate CPU states by signatures and memory by signing on a Merkle-tree-like data structure.

More precisely, consider a compiled program P_{auth}, which at each time step expects a signed CPU state from the previous time step, and signs the output state for the next time step. To authenticate the memory, a Merkle tree root is stored in the CPU state, and each memory read/write is authenticated via *authentication path* (i.e., the path from the root to the memory location with siblings in the Merkle tree). In other words, P_{auth} expects from the evaluator signed CPU states and the authentication path for the read memory content before

[3]We consider other settings as well, but focus on this particular setting here.

[4]For uniform programs, the circuit size is polylogarithmic.

evaluating P (otherwise, P_{auth} outputs $\texttt{Reject}$). In this way, the input x can simply be authenticated by signing the initial CPU state with Merkle tree root of x stored. Intuitively (i.e., assuming security of all used primitives "works"), the evaluator can evaluate P_{auth} only on x.

Finally, we consider a $\mathsf{Ci}\mathcal{O}$ obfuscator which outputs $\tilde{\Pi}_{\mathsf{auth}}$ defined by $(\mathsf{i}\mathcal{O}(P_{\mathsf{auth}}), x_{\mathsf{auth}})$, where x_{auth} is an authenticated input. The hope is that the authentication mechanism, together with $\mathsf{i}\mathcal{O}$ security, can ensure that an adversary receiving $\tilde{\Pi}_{\mathsf{auth}}$ can only generate the honest computation trace of Π, and further imply $\mathsf{Ci}\mathcal{O}$ security. Authentication alone implies publicly verifiable delegation without privacy, by using a special signing key to sign the output y and publishing the corresponding verification key for public verification.

We further discuss how this can be done in finer details. We first focus on the simpler case of RAM computation, which can be viewed as an abstraction of the existing techniques for TM. To handle the full-fledged PRAM computation, we introduce several techniques to tackle the challenges in the parallel setting. In particular, we avoid the dependency on the number of CPUs to achieve full succinctness.

2.3.1 $\mathsf{Ci}\mathcal{O}$-RAM Construction

The following is essentially the intuition behind the construction of message-hiding encoding (MHE) [KLW15]. Our $\mathsf{Ci}\mathcal{O}$-RAM construction is heavily inspired by their work, and can be viewed as an abstraction of what is achieved by their techniques. As a result, $\mathsf{Ci}\mathcal{O}$ is closely related to MHE, and the former can be used to construct the latter readily.

For convenience, we assume that P only writes to the memory cell it reads in the previous time step.[5] We also assume that the CPU state stores the time step t.

As mentioned, to authenticate the whole memory, we build a Merkle tree with each memory cell being a leaf node, and store the tree root as the digest in the CPU state. Our $\mathsf{Ci}\mathcal{O}$ construction outputs $\tilde{\Pi}$ defined by $(\mathsf{i}\mathcal{O}(P_{\mathsf{auth}}), x_{\mathsf{auth}})$, where the compiled program P_{auth} expects as input a signed CPU state, the read memory cell ℓ, and its authentication path. If the authentication path and the signature pass verification, P_{auth} outputs a signed next CPU state and memory access instruction. Additionally, if the memory access is a write to the cell ℓ, it also updates the digest stored in the CPU state using the authentication path. The authenticated input x_{auth} consists of the initial memory content x, and a signed initial CPU state that contains the Merkle tree digest of x.

Let Π and Π' be two computation instances defined by (P, x) and (P', x) respectively with identical computation trace. To prove security, we consider a sequence of hybrids starting from $\tilde{\Pi}$ that switches the program from P to P' time step by time step. However, to switch based on $\mathsf{i}\mathcal{O}$ security, the programs in the hybrids need to be functionally equivalent, while P to P' only behave identically during honest execution. Note that normal signatures and Merkle tree cannot guarantee functional equivalence as forgeries exist information-theoretically. Instead, we rely on the powerful $\mathsf{i}\mathcal{O}$-friendly authentication primitives, which are *splittable signatures* and *accumulators* [KLW15], respectively. At a very high level, they allow us to switch to a hybrid program that, at a particular time step t (i.e., input with time step t stored in the CPU state), only accepts the honest input but rejects all other inputs (and outputs $\texttt{Reject}$), which enables us to switch from P to P' at time step t using $\mathsf{i}\mathcal{O}$ security. Additionally, a primitive called *iterators* is introduced [KLW15] to facilitate the argument about the above hybrids. Details can be found in [KLW15] and Section 3.1.1.

2.3.2 $\mathsf{Ci}\mathcal{O}$-PRAM Construction

We then extend the above approach to the PRAM model with some care of the efficiency issues in the parallel setting.

Recall that a PRAM computation instance Π is also specified by a next-step circuit P for the CPU program and an input x stored in the memory. However, instead of a single CPU, there are m CPUs, specified by the same program P but with different CPU id's, performing the computation in parallel with shared random access to the memory. We assume that there are no conflict writes throughout the computation, all CPUs have synchronized read/write memory access, and all terminate at the same time step.[6]

[5] This convention can be imposed without loss of generality.

[6] We note that the latter two conventions can be imposed with $O(\log m)$ blow up in the parallel run-time.

A Naïve Attempt We can view the m copies of next-step circuit as a single giant next-step circuit P^m that accesses m memory locations at each CPU time step. We can then compile P^m to P^m_{auth} and output $\tilde{\Pi}$ defined by $(\mathsf{i}\mathcal{O}(P^m_{\mathsf{auth}}), x_{\mathsf{auth}})$ in a similar way as before. This approach indeed works in terms of security and correctness. However, as P^m has description size $\Omega(m)$ (since it operates on $\Omega(m)$-size input), the obfuscated computation will have description size $\mathsf{poly}(m)$, which incurs $\mathsf{poly}(m)$ overhead in the evaluation time.[7]

Avoiding the Dependency on the Number of CPUs We thus observe that, to preserve the parallel run-time, we can only $\mathsf{i}\mathcal{O}$ a single (compiled) CPU program P_{auth}, and run m copies of $\mathsf{i}\mathcal{O}(P_{\mathsf{auth}})$ in the evaluation of obfuscated instance with different CPU id's (as in evaluating the original Π).

We use accumulator to authenticate the (shared) memory. That is a Merkle-tree-like structure with the tree root as the (shared) accumulator value w stored in the CPU state, which needs to be updated when the memory content is changed. Specifically, consider a time step where m CPUs perform parallel write to distinct memory cells. The CPUs need to update the shared accumulator value w to reflect the m writes in some way. We cannot let a single CPU perform the update in one time step, because it involves processing $\Omega(m)$-size data, which makes the size of the next-step circuit dependent on $\Omega(m)$ again. Also, we cannot afford to update sequentially (i.e., each CPU takes turns to update the digest), since this blows up the parallel run-time of the evaluation algorithm by m and obliterates the gains of parallelism. So, we design an $O(\mathsf{poly}\log m)$-round distributed algorithm to update the digest as follows.

First, we allow the instances of the (compiled) CPU program P_{auth} to communicate with each other. Namely, each CPU can send a message to other CPUs at each time step. Such CPU-to-CPU communications can be emulated readily by storing the messages in the memory for the evaluator of the obfuscation. Recall that P_{auth} needs to authenticate the computation, so the program needs to authenticate the communication as well. Fortunately, this can be done using splittable signatures in a natural way. We can now formulate the problem of updating the accumulator value as a distributed computing problem as follows:

There are m CPUs, each holding an accumulator value w, memory cell index ℓ_i, write value val_i, and an authentication path ap_i for ℓ_i (received from the evaluation algorithm) as its inputs. Their common goal is to compute the updated accumulator value w' with respect to the write instructions $\{(\ell_i, val_i)\}_{i\in[m]}$. Our task is to design a distributed algorithm for this problem with oblivious communication pattern[8], in $\mathsf{poly}\log(m)$ rounds, and with per-CPU space complexity $\mathsf{poly}\log(m)$. If this is achieved, the blow-up in both the parallel run-time and obfuscation size can be reduced from $\Omega(m)$ to $\mathsf{poly}\log(m)$.

We construct such *oblivious update* protocol with desired complexity based on two oblivious protocols by Boyle et al. [BCP16]: an aggregation protocol that allows m CPUs to aggregate information they need in parallel, and a multi-casting protocol that allows m CPUs to receive messages from other CPUs in parallel. Both protocols have run-time poly-logarithmic in m. Roughly, our oblivious update protocol updates the Merkle tree layer-by-layer from leaves to root. For each layer, the CPUs engage in the oblivious aggregation protocol to aggregate information for updating their local branches of the tree. They then distribute their results using the oblivious multi-casting protocol.

Back to our $\mathsf{Ci}\mathcal{O}$ construction, we output $\tilde{\Pi}$ defined by $(\mathsf{i}\mathcal{O}(P_{\mathsf{auth}}), x_{\mathsf{auth}})$, where P_{auth} is a compiled CPU program that can communicate with other CPUs and authenticate the communication by splittable signatures. The evaluation of $\tilde{\Pi}$ runs m copies of $\mathsf{i}\mathcal{O}(P_{\mathsf{auth}})$ and emulates the communication by routing the messages. After each memory-write time step, P_{auth} maintains the accumulator value by invoking the oblivious update protocol. Finally, for the authenticated input x_{auth}, it consists of the initial memory content x, and the accumulator value of x stored in signed CPU states as before. However, it cannot contain initial CPU states with m different signatures, as otherwise the obfuscation has size dependent on m. This can be solved by a simple trick: we let x_{auth} only consists of a single signed "seed" CPU state $\mathsf{st}_{\mathsf{seed}}$, and when P_{auth} takes $\mathsf{st}_{\mathsf{seed}}$ and a CPU id i as input, P_{auth} outputs a signed initial state of CPU i.

[7]While P^m has low depth (independent of m), its obfuscated version may not, as the security of $\mathsf{i}\mathcal{O}$ needs to hide the circuit depth. Thus, the parallel run-time is not preserved.

[8]We mention that the oblivious communication pattern property may not be essential, but a useful feature to make the construction simple, since the CPUs do not need to decide who to send/receive messages.

We stress that there is a more subtle and challenging issue arises when we try to generalize the security proof for TM/RAM model to handle PRAM while preserving parallel run-time. In short, a naïve generalization of the security proof would require hardwiring the m CPU sates at some time steps, which results in $\Omega(m)$ amount of hardwired information, and causes unacceptable $\mathsf{poly}(m)$ overhead in size of the obfuscated program. We address this by developing a "*branch-and-combine*" technique to emulate PRAM computation, which enables us to reduce the amount of hardwired information in the hybrids to $O(\log m)$, in Section 3.1.

2.4 $\mathcal{RE}$ Construction

The next goal is to hide information through the evaluation process of $\tilde{P}$ on $\tilde{x}$. A natural approach is to use encryption schemes to hide the CPU states and the memory content. Namely, $\tilde{P}$ always outputs encrypted CPU states and memory, and on (authenticated) input of ciphtertexts, performs decryption before the actual computation. Note, however, that the memory access pattern cannot be encrypted (otherwise the server cannot evaluate), which may also leak information. A natural approach is to use oblivious (parallel) RAM (OPRAM/ORAM) to hide the access pattern. Namely, we use OPRAM compiler to compile the program (and add an "encryption layer") before obfuscating it. Again, intuitively (i.e., assuming all primitives "works"), the server cannot learn information from the evaluation process.

We note that the construction of machine-hiding encoding for Turing machine [KLW15] uses public-key encryption to hide the memory content of TM evaluation, and hides the TM access pattern by oblivious Turing machine compiler [PF79], which is *deterministic*. In our case, hiding random memory access pattern for (parallel) RAM (which is necessary to capture sublinear time computation) requires new techniques, since OPRAM/ORAM compilers are *probabilistic*, and we cannot use OPRAM/ORAM security in a black-box way. We deal with this issue by developing "puncturing" technique for specific OPRAM construction.

At a high level, constructing fully succinct randomized encoding ($\mathcal{RE}$) from $\mathsf{Ci}\mathcal{O}$-RAM requires hiding both the memory content and the access pattern of the computation. We first consider the simpler case where the computation has oblivious access pattern (so we only need to hide the memory content), which we can rely on techniques of using public-key encryption [KLW15]. In fact, if the access pattern is not required to be hidden, the construction of machine-hiding encoding for TM [KLW15] can be modified in a straightforward way to yield $\mathcal{RE}$-RAM. Our construction can be viewed as a modularization and simplification of their construction through our $\mathsf{Ci}\mathcal{O}$ notion in a black-box way (which in a sense captures security achieved by authentication).

We next discuss how to hide access pattern, which is the major challenge for the construction of fully succinct $\mathcal{RE}$-RAM. We follow a natural approach to use ORAM compiler to hide it. However, the main difficulty is that ORAM only hides the access pattern when the CPU state and the memory contents are hidden from the adversary, which is hard to argue for unless the obfuscation is virtual-black-box (VBB) secure [BGI$^+$12], while $\mathsf{Ci}\mathcal{O}$ (just like $\mathsf{i}\mathcal{O}$) does not hide anything explicitly. We develop a *puncturable* ORAM technique to tackle this issue. We rely on a simple ORAM construction [CP13] (referred to as CP-ORAM) and show that it can be "punctured" at time step i so that the access pattern at the i-th time step of $P(x)$ can be simulated even given the punctured program. Armed with this technique, we can simulate the access pattern at time step i by puncturing the ORAM compiled program at step i (through hybrids), replacing the access pattern at this step, and then unpuncturing the program. Yet, the computation traces of a punctured program can differ from the original ones in many steps. Therefore, arguing the indistinguishability of a hybrid using a punctured program is non-trivial. We do so by defining a sequence of "partially punctured" hybrids that gradually modifies the program step by step.

Finally, we extend the above construction to handle PRAM computation, where we simply replace the ORAM compiler by the oblivious PRAM compiler [BCP16]. The security proof also generalizes in a natural way, except that we need to take care of some issues aroused in the parallel setting. The main issue is to generalize the puncturing argument to puncture OPRAM in a way that avoids dependency on the number of CPU m to maintain full succinctness. This can be done by puncturing the OPRAM CPU by CPU.

2.4.1 $\mathcal{RE}$-ORAM Construction

We first focus on the simpler case of $\mathcal{RE}$ for *oblivious* RAM computation where the given RAM computation instance $\Pi = (P, x)$ has oblivious access pattern. Namely, we assume that there is a public function $\mathsf{ap}(t)$ that predicts the memory access at each time step t, to be given to the simulator.

For this simpler case, we do not need to use ORAM to hide the access pattern. We can modify existing machine-hiding encoding for TM [KLW15] which hides the CPU state and the memory content using $\mathcal{PKE}$ in a straightforward way to yield $\mathcal{RE}$ for oblivious RAM computation. Our construction presented below can be viewed as a modularization and simplification of their construction through our $\mathsf{Ci}\mathcal{O}$ notion.

Consider an encoding algorithm $\mathcal{RE}.\mathsf{Encode}(\Pi)$ which outputs $\mathsf{Ci}\mathcal{O}(P_{\mathcal{PKE}}, x_{\mathcal{PKE}})$, where $P_{\mathcal{PKE}}$ is a compiled version of P, and $x_{\mathcal{PKE}}$ is an encrypted version of x. At a high level, $P_{\mathcal{PKE}}$ emulates P step by step. Instead of outputting the CPU state and the memory content in the clear, $P_{\mathcal{PKE}}$ outputs encrypted versions of them. $P_{\mathcal{PKE}}$ also expects encrypted CPU states and memory contents as input, and emulates P by first decrypting them. A key idea here (following [KLW15]) is to encrypt each message (a CPU state or a memory cell) using a different key, and generate these keys (as well as encryption randomness) using puncturable PRF (PPRF), which allows us to use a standard puncturing argument in the security proof (extended to work with $\mathsf{Ci}\mathcal{O}$ instead of $\mathsf{i}\mathcal{O}$) to move to a hybrid where semantic security holds for a specific message so that we can "erase" it.

To make sure that each key is used to encrypt a single message, at time step t, $P_{\mathcal{PKE}}$ encrypts the output state and memory content using the "t-th" keys, which are generated by PPRF with input t (and some additional information to distinguish between state and memory). Likewise $x_{\mathcal{PKE}}$ contains the encryption of the initial memory x with different keys for each memory cell. To decrypt the input memory, $P_{\mathcal{PKE}}$ needs to know which secret key to use. This can be addressed by simply storing the time tag t with the encrypted memory (as a single memory cell). Namely, each memory cell for $P_{\mathcal{PKE}}$ contains a ciphertext ct_{mem} and a time tag t. As authentication is taken care of by $\mathsf{Ci}\mathcal{O}$ as a black box, no additional authentication mechanisms are needed.

At a high level, we prove the security of the above construction by defining a sequence of hybrids that "erase" the computation *backward in time*, which leads to a simulated encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$ where all ciphertexts generated by P_{Sim}, as well as those in x_{Sim}, are replaced by some encrypted special $\mathtt{dummy}$ symbols. More precisely, P_{Sim} simulates the access pattern using the public access function ap. For each time step $t < t^*$, P_{Sim} simply ignores the input and outputs encrypted $\mathtt{dummy}$ symbols (for both CPU state and memory content), and outputs y at time step $t = t^*$.[9]

By erasing the computation backward in time, we consider the intermediate hybrids $\mathbf{Hyb}_i$ where the computations of the first i time steps are real, and those of the remaining time steps are simulated. Namely, $\mathbf{Hyb}_i$ is a hybrid encoding $\mathsf{Ci}\mathcal{O}(P_{\mathbf{Hyb}_i}, x_{\mathcal{PKE}})$, where $P_{\mathbf{Hyb}_i}$ acts as $P_{\mathcal{PKE}}$ in the first i time steps, and acts as P_{Sim} in the remaining time steps. To argue for the indistinguishability between $\mathbf{Hyb}_i$ and $\mathbf{Hyb}_{i-1}$, which corresponds to erasing the computation at the i-th time step, the key observation is that the i-th decryption key is *not* used in the honest evaluation, which allows us to replace the output of the i-th time step by an encryption of $\mathtt{dummy}$ through a puncturing argument. We can then further remove the computation at the i-th time step readily by $\mathsf{Ci}\mathcal{O}$ security.

In more details, to move from $\mathbf{Hyb}_i$ to $\mathbf{Hyb}_{i-1}$, we further consider an intermediate hybrid $\mathbf{Hyb}'_i$ where the output of the i-th time step is replaced by an encryption of $\mathtt{dummy}$, but the real computation is still performed. Namely, at the i-th time step, $P_{\mathbf{Hyb}'_i}$ in $\mathbf{Hyb}'_i$ still decrypts the input and emulates P, but replaces the output by an encryption of $\mathtt{dummy}$. Note that indistinguishability between $\mathbf{Hyb}'_i$ and $\mathbf{Hyb}_{i-1}$ follows immediately from $\mathsf{Ci}\mathcal{O}$ security by observing that $(P_{\mathbf{Hyb}'_i}, x_{\mathcal{PKE}})$ and $(P_{\mathbf{Hyb}_{i-1}}, x_{\mathcal{PKE}})$ have identical computation trace. To argue for the indistinguishability between $\mathbf{Hyb}_i$ and $\mathbf{Hyb}'_i$, we note that the i-th decryption key is *not* used in the honest evaluation, since the computation after time step i is erased. Thus, we can puncture the randomness and erase the decryption key from the program (which uses $\mathsf{Ci}\mathcal{O}$ security as well), and reach a hybrid (from $\mathbf{Hyb}_i$) where semantic security holds for the output ciphertext at time step i. We can then replace the ciphertext by an encryption of $\mathtt{dummy}$ and undo the puncturing to reach $\mathbf{Hyb}'_i$.

[9] Here we only consider honest evaluation of P_{Sim} on x_{Sim}. Any "dishonest" evaluation is disallowed by $\mathsf{Ci}\mathcal{O}$ security.

There remain some details to complete the proof. First, the real encoding $\mathsf{Ci}\mathcal{O}(P_{\mathcal{PKE}}, x_{\mathcal{PKE}})$ and $\mathbf{Hyb}_{t^*-1}$ have identical computation trace, so indistinguishability follows by $\mathsf{Ci}\mathcal{O}$ security. Second, moving from $\mathbf{Hyb}_0$ to the simulated encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$ requires replacing $x_{\mathcal{PKE}}$ by x_{Sim}, which can be done by a similar puncturing argument.

2.4.2 $\mathcal{RE}$-RAM Construction

We now deal with the main challenge of hiding access pattern by using ORAM. Recall that an ORAM compiler compiles a RAM program by replacing each memory access by a *probabilistic* procedure OACCESS that implements memory access in a way that hides the access pattern.[10]

Given a computation instance $\Pi = (P, x)$, we first compile P using an ORAM compiler, with randomness supplied by puncturable PRF for succinctness, and initiate the ORAM memory by inserting the input x. Let $(P_{\mathsf{ORAM}}, x_{\mathsf{ORAM}})$ be the compiled program and the resulting memory. It is then compiled in the same way as in Section 2.4.1 with $\mathcal{PKE}$. Namely, we use PPRF to generate multiple keys, and use each key to encrypt a single message, including the initial memory x_{ORAM}. Let the resulting instance be $(P_{\mathsf{hide}}, x_{\mathsf{hide}})$. Our randomized encoding of Π is $\mathsf{Ci}\mathcal{O}(P_{\mathsf{hide}}, x_{\mathsf{hide}})$. However, as we discussed earlier, it is unlikely that we can use the security of ORAM in a black-box way, since ORAM security only holds when the adversary does not learn any content of the computation. Indeed, recall in the previous section, we can only use puncturing argument to argue that semantic security holds *locally* for some encryption at a time.

We remark that the work of Canetti et al. [CHJV15] encountered a similar technical problem in their construction of one-time RAM garbling scheme. Their construction has similar high level structure as ours, but based on a quite different machinery called asymmetrically constrained encapsulation (ACE) they built from $\mathsf{i}\mathcal{O}$ for circuits. Canetti et al. provided a novel solution to this problem, but their garbling incurs dependency on the space complexity of the RAM program, and thus is not fully succinct. In more details, their security proof established the indistinguishability of hybrids *forwards in time*: At a certain hybrid $\mathbf{Hyb}_i$, they information-theoretically erase computation before time step i, simulate the memory access pattern, and hardwire the configuration of the $(i+1)$-th step into the program, so as to faithfully perform the correct computation in the later steps. Moving to the $(i+1)$-th hybrid relies on their *new* strong ORAM simulatability (which is satisfied by a specific ORAM construction [CP13] they use), which enables them to replace the actual memory access at time step $(i+1)$ by a simulated one. However, the ORAM security relies on the fact that the first i steps of computation are information-theoretically hidden, and thus the hybrids need to hardwire in an intermediate configuration of size proportion to the space complexity of the program. Such memory content hardwiring forces their garbling scheme to be padded to a size depending on the space complexity of the program, making the scheme non-succinct in space.

A natural approach to avoid such dependency in our construction is to establish indistinguishability of hybrids *backwards in time*, as in the previous section. Namely, we consider intermediate hybrids $\mathbf{Hyb}_i$ where the computations of the first i time steps are real, and those of the remaining time steps are simulated (appropriately). Yet, since the computation trace of the first $(i-1)$ time steps is real, it contains enough information to carry out the rest of the (deterministic) computation. In particular, the access pattern at time step i is determined by the first $(i-1)$ time steps, that means we cannot replace it by a simulated access pattern.

To solve this problem, we develop a puncturing ORAM technique to reason about the simulation specifically for CP-ORAM [CP13][11]. At a very high level, to move from $\mathbf{Hyb}_i$ to $\mathbf{Hyb}_{i-1}$ (i.e., erase the computation at the i-th time step), we "puncture" ORAM at time step i (i.e., the i-th memory access), which enables us to replace the access pattern by a simulated one at this time step. We can then move (from $\mathbf{Hyb}_i$) to $\mathbf{Hyb}_{i-1}$ by erasing the memory content and computation, and undoing the "puncturing."

Roughly speaking, "puncturing" CP-ORAM at i-th time step can be viewed as injecting a piece of "*puncturing*" code in OACCESS to erase the information rand_i about access pattern at time step i information-

[10]In contrast, for Turing machines (TM), one can make the access pattern oblivious by a *deterministic* oblivious TM compiler. This is why [KLW15] does not need to address the issue of hiding access pattern for TM computation.

[11]We believe that our puncturing technique works for any tree-based ORAM constructions, but we work with CP-ORAM for concreteness.

theoretically: rand_i is generated at the latest time step t' that accesses the same memory location as time step i. The puncturing code simply removes the generation of rand_i at time step t'.

However, the last access time t' can be much smaller than i, so the puncturing may cause global changes in the computation. Thus, moving to the punctured mode requires defining a sequence of hybrids that modifies the computation step by step. We do so by further introducing an auxiliary "*partially puncturing*" code that punctures rand_i from certain threshold time step $j \geq t'$. The sequence of hybrids to move to the punctured code corresponds to moving the threshold $j \leq i$ backwards from i to t'.

2.4.3 $\mathcal{RE}$-PRAM Construction

Our construction of $\mathcal{RE}$-PRAM replaces the CP-ORAM compiler of our construction of $\mathcal{RE}$-RAM by the OPRAM compiler of Boyle et al. [BCP16] (referred to as BCP OPRAM hereafter), a generalization of tree-based ORAM to the parallel setting. Namely, given a PRAM computation instance Π defined by (P, x), we first compile P into P_{OPRAM} using the BCP OPRAM compiler with randomness supplied by puncturable PRF. We also initiate the OPRAM memory by inserting the input x. Let x_{OPRAM} be the resulting memory. We then compile $(P_{\mathsf{OPRAM}}, x_{\mathsf{OPRAM}})$ using $\mathcal{PKE}$ in the same way as in Section 2.4.1. A small difference here is that we also need to include CPU id as PPRF input to ensure single usage of each key. Denote the resulting instance by $(P_{\mathsf{hide}}, x_{\mathsf{hide}})$. Our randomized encoding of Π is $\mathsf{Ci}\mathcal{O}(P_{\mathsf{hide}}, x_{\mathsf{hide}})$.

The security proof also follows identical steps, where we prove the security by a sequence of hybrids that erases the computation backward in time, and simulate access patterns by generalizing the puncturing ORAM argument to puncturing BCP OPRAM. At a high level, the arguments generalize naturally with the following two differences: First, as the OPAccess algorithm of BCP OPRAM is more complicated, we need to be slightly more careful in defining the simulated encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$. Second, to avoid dependency on the number m of CPUs, we need to handle a single CPU at a time in the hybrids to puncture OPRAM.

2.5 Extension for Persistent Database

Finally, we note that our construction can be generalized readily to handle delegation with persistent database. In this setting, the client additionally delegates his or her database to the server at beginning, and then delegates multiple computation to evaluate and *update* the database in a verifiable and private way. Recall that we authenticate every step of computation by signatures. We can "connect" two programs by letting P_i to sign its halting state using a special "termination" signing key, and letting the next program P_{i+1}, upon receiving this signed state, initiate itself by signing its initial state, and inheriting the Merkle tree root of the database from P_i (stored in the halting CPU state).

3 Technical Highlights

Here we highlight the technical difficulties on handling multiple parallel CPUs and random memory access in PRAM computations, and our new techniques to resolve them.

3.1 Handling Parallel Processors

Algorithmic Issue To preserve the gain of parallelism, we need to run m (obfuscated) CPU programs in parallel, each of which has small $\mathsf{poly}\log(n)$-sized state. When there is a parallel write to the memory, these m CPUs need to update the digest of the large memory in parallel efficiently. Note that no CPU can have global update information (since each only has $\mathsf{poly}\log(n)$-sized state), so they need to do it in a distributed fashion without incurring $\Omega(m)$ efficiency overhead. We handle this issue based on the techniques in the OPRAM construction of Boyle, Chung, and Pass [BCP16]. At a high level, we allow the CPUs to communicate with each other, and design an $O(\mathsf{poly}\log m)$-round distributed algorithm for updating the digest with *oblivious* communication pattern[12].

Security Proof Issue This is a more subtle and challenging issue arises when we try to generalize the security proof for TM/RAM model to handle PRAM. At a very high level, the security proof consists of a (long) sequence of hybrids in time steps, where in the intermediate hybrids, we need to hardwire the CPU state at some time steps to the obfuscated program. Generalizing the idea to PRAM in a naïve way would require us to hardwire the m CPU sates at some time steps, which results in $\Omega(m)$ amount of hardwired information. This in turn requires us to pad the program to size $\Omega(m)$ and causes $\mathsf{poly}(m)$ overhead in size of the obfuscated program. To see why, and to pave the way for discussing our idea for resolving it, we must take a closer look at the technique of KLW.

3.1.1 Proof Techniques of KLW

We now provide a very high level overview of the security proof of our $\mathsf{Ci}\mathcal{O}$-RAM based on the machinery of KLW. The techniques serve as a basis for the discussion of our construction of $\mathsf{Ci}\mathcal{O}$-PRAM. Recall that our construction can be viewed as $\mathsf{Ci}\mathcal{O}(\Pi) = (\mathsf{i}\mathcal{O}(P_{\mathsf{auth}}), x_{\mathsf{auth}})$, where $(P_{\mathsf{auth}}, x_{\mathsf{auth}})$ is just (P, x) augmented with $\mathsf{i}\mathcal{O}$-friendly authentication mechanism of KLW. Let $\Pi = (P, x)$ and $\Pi' = (P', x)$ be two computation instances with identical computation trace.[13] Our goal is to show $\tilde{\Pi} \leftarrow \mathsf{Ci}\mathcal{O}(\Pi)$ and $\tilde{\Pi}' \leftarrow \mathsf{Ci}\mathcal{O}(\Pi')$ are computationally indistinguishable. To prove security, we consider a sequence of hybrids starting from $\tilde{\Pi}$ that switches the program from P to P' time step by time step. Very roughly, those $\mathsf{i}\mathcal{O}$-friendly authentication mechanisms allow us to switch from P to P' at time step t using $\mathsf{i}\mathcal{O}$ security. It can be viewed as introducing "*check-points*" to hybrid programs as follows (which is implicit in [KLW15])[14]:

- We can place a check-point at the initial step of computation, and move it from a time step t to time step $(t+1)$ through hybrids.
- Check-point is a piece of code that at a time step t, checks if the input (or output) is the same as that in the honest computation, and forces the program to output `Reject` if it is different. This is an information-theoretic guarantee which enables us to switch the program at time step t based on $\mathsf{i}\mathcal{O}$ security.

We can then move the check-point from the beginning to the end of the computation, and along the way switch the program from P to P'. This check-point technique is essential to illustrate our issue on PRAM model.

3.1.2 Proof Issue Illustrated as a "Pebble Game"

The Pebble Game for RAM We now discuss the issue of hardwiring $\Omega(m)$ amount of information in intermediate hybrids when we generalize the KLW techniques to handle PRAM with m CPUs. To illustrate, we cast the security proof as a "*pebble game*" over a graph defined by the computation, and the required amount of hardwire information in the hybrids can be captured by the "*pebble complexity*" of the game.

We first illustrate this pebble game abstraction for the case of RAM. Recall that the security proof relies on a check-point technique that allows us to place a check-point on the initial time step, and move it from a time step t to its next time step $(t+1)$. Placing a check-point at a time step requires hardwiring information proportional to the input (or output) size of the CPU program. The goal is to travel all time steps (to switch the programs on all time steps). In this example, the RAM computation can be viewed as a line graph with each time step being a node. A check-point is a pebble that can be placed on the first node, and can be moved from node t to node $(t+1)$. The winning condition of the pebble game is to "cover" the graph, namely, to ever place

[12] The CPU to CPU communication could be done through memory access (e.g., CPU i writes to a specific memory address and CPU j reads it). However, we cannot do so in our context of OPRAM since communication through memory requires updating the digest, which leads to a circularity.

[13] Recall that it means both the CPU states and memory content are identical throughout the computation.

[14] The description here over-simplifies many details.

a pebble on each node. The pebble complexity is the maximum number of pebbles needed to cover the graph, which is 2 for the case of RAM (since technically we need to place a pebble at $(t+1)$ before removing the pebble at t).

The Pebble Game for PRAM Next, we attempt to generalize the pebble game to the PRAM setting so as to capture the direct generalization of the previous security argument (including generalization of both accumulators and iterators).

The graph is a layered (directed acyclic) graph with each layer corresponds to m CPUs' at a certain time step. Namely, each node is indexed by (t, i) where t is the time step and i is the CPU id. It also consists of a node 0 corresponding to the seed state. Node 0 has an outgoing edge to node $(t = 1, i)$ for every $i \in [m]$. Each node (t, i) has an outgoing edge to $(t+1, i)$ indicating the (trivial) dependency of i-th CPU between time step t and $t + 1$. Recall that the CPUs have communication (to jointly update the digest of the memory). If CPU i sends a message to CPU j at time step t, we also put an outgoing edge from (t, i) to $(t + 1, j)$ to indicate the dependency.[15]

The pebbling rule is defined as follows: First, we can place a pebble on node 0. To place a pebble on a node v, all nodes of v's incoming edges need to have a pebble on it. To remove a pebble on a node v, we need to "cover" all v's outgoing nodes, i.e., ever place a pebble on each outgoing node. These capture the conditions for placing and removing a check-point to a computation step respectively, for our generalization of the i$\mathcal{O}$-friendly authentication techniques of KLW to the parallel setting.

As before, the goal is to cover the whole graph (i.e., ever places a pebble on every node) using a minimal number of pebbles. The pebble complexity of the game is the maximum number of pebbles we need to simultaneously use to cover the graph. Covering the graph corresponds to switching the programs for every computation step, and the pebble complexity captures the amount of hardwire information required in the intermediate hybrids.

Recall that our P_{auth} invokes a distributed protocol to update the digest of the memory for every (synchronized) memory-writes. To play the pebble game induced by multiple invocations of this distributed protocol, a trivial solution with $2m$ pebbles is to place pebbles on two neighboring layers. Unfortunately, it is unclear how to play this pebble game with $o(m)$ pebble complexity, and it seems likely that the pebble complexity is indeed $\Omega(m)$. Therefore, it may seem that hardwiring $\Omega(m)$ amount of information in intermediate hybrids is required. As a result, P_{auth} needs to be padded to size $\Omega(m)$, and thus $|\tilde{\Pi}|$ has $\mathsf{poly}(m)$ overhead.

3.1.3 Reducing Information Hardwiring

To reduce information to be hardwired, we introduce a "*branch-and-combine*" approach to emulate a PRAM computation, which transforms the computation graph to one with $\mathsf{poly}\log(m)$ pebble complexity, and preserves the parallel run-time and obfuscation size with a $\mathsf{poly}\log(m)$ overhead.

At a high level, after one parallel computation step, we *combine* m CPU states into one "digest" state, then *branch* out from the digest state for another parallel computation step, which results in m CPU states to be *combined* again. The PRAM computation is emulated by alternating the branch and combine steps. The combine step involves $\log m$ rounds where we combine two states into one in parallel (which forms a complete binary tree). The branch step is done in one shot which branches out m CPUs in one step in parallel. Thus, the branch-and-combine emulation only incurs $O(\log m)$ overhead in parallel run-time. Note that this transforms the computation graph into a sequence of complete binary trees where each time step of the original PRAM computation corresponds to a tree, and the root of a time step connects to all leaf nodes of the next time step.

Now, we observe that only $O(\log m)$ pebbles are used to traverse the computation graph of the branch-and-combine PRAM emulation. This is because whenever we put two pebbles at a pair of sibling nodes in the complete binary tree of the combine step, we can merge them into a single one at their parent node. This means that only one pebble is necessary for each height level of the tree. More precisely, we can move pebbles from one root to the next one by simply putting pebbles at its branched out nodes one by one in order, and merging

[15]In general, the memory accesses may create dependency. We ignore it here as the pebble complexity is already high.

the pebbles greedily whenever it is possible. Very roughly, the combine step corresponds to constructing an accumulator tree for CPU states, and the branch step verifies an input CPU state using the accumulator and performs one computation step.

3.2 Handling Memory Access

We next discuss the difficulties in hiding memory access pattern. Recall that our construction to achieve privacy is very natural: We hide CPU states and memory content using public-key encryption ($\mathcal{PKE}$) and hide access pattern using oblivious (parallel) RAM. Note that KLW already showed how to use $\mathcal{PKE}$ to hide the memory content of Turing machine evaluation. Roughly speaking, the security proof is done by a sequence of hybrids that "erases" the computation step by step *backward* in time. However, hiding access pattern for Turing machine is simple since oblivious Turing machine compiler [PF79] is *deterministic*.

In contrast, ORAM and OPRAM compilers are probabilistic, and they only hide the access pattern statistically when the adversary learns only the access pattern, but not the CPU states and the memory content. However, since the obfuscated program contains the hardwired secret key of $\mathcal{PKE}$, we can only argue that the memory content is hidden by puncturing argument at the cost of hardwiring information. In other words, we can only afford to argue hiding holds "locally" but not "globally". This is why the proof of KLW erases the computation step by step. More importantly, this prevents using ORAM/OPRAM security in a black-box way.

We remark that the seminal work of Canetti et al. [CHJV15] encountered this technical problem in the context of one-time RAM garbling scheme, where they resolve the issue by identifying stronger security of a specific ORAM construction [CP13]. However, this approach requires "erasing" the computation *forward* in time in the security hybrids, which in turn require hardwiring information proportional to the space complexity of the RAM computation. We instead resolve this issue by a puncturing ORAM/OPRAM technique that also relies on specific ORAM/OPRAM constructions [CP13, BCP16], to be elaborated below for the case of RAM.

3.2.1 ORAM **Puncturing Technique for Simulation**

We develop an ORAM *puncturing* technique to reason about the simulation for a specific ORAM construction [CP13] (referred to as CP-ORAM hereafter).[16] The high level strategy is to switch the ORAM access pattern from a real one to a simulated one step by step (backward in time). To enable switching at time step i (i.e., for the i-th memory access), we "puncture" the real execution in a way that ensures that the i-th memory access is information theoretically hidden even given the memory content of the first $(i-1)$ steps execution. Since we do hybrids backward in time, the computation after i-th step is already erased, and so once the ORAM is "punctured", we can replace the real i-th step access pattern by a simulated one (both of which are random given full information of the punctured real ORAM execution). To further explain how this is done, we first review the CP-ORAM construction.

3.2.2 Review of the CP-ORAM Construction

In a tree-based ORAM (CP-ORAM), the memory is stored in a complete binary tree (as known as ORAM tree), where each node is associated with a bucket. A bucket is a vector of K elements, where each element is either a memory block, or a unique symbol `dummy` stands for an empty slot. Note that a memory block is the smallest unit of operation in CP-ORAM, which consists of a fixed small number of memory cells. In particular, a position map *pos* (stored in CPU state) records where each memory block is stored in the tree, i.e., a node somewhere along a path from the root to the leaf indexed by $pos[\ell]$. Each memory block ℓ in the ORAM tree also stores its index ℓ and position map value $pos[\ell]$ as meta data. Each memory access to block ℓ is performed by OACCESS, which (i) reads the position map value $p = pos[\ell]$ and refreshes $pos[\ell]$ to a random value, (ii) fetches and removes the block ℓ (i.e., replaces it by `dummy`) from the path, (iii) updates the block content and puts it back to the root (i.e., replace a `dummy` block by the updated memory block), and (iv) performs a flush operation along another random path p' to move the blocks down along p' (subject to the condition that each

[16] We believe that our puncturing technique works for any tree-based ORAM, but we work with CP-ORAM for concreteness.

block is stored in the path specified by their position map value). At a high level, the security follows by the fact that the position map values are uniformly random and hidden from the adversary, and thus the access pattern of each OACCESS is simply two uniformly random paths, which is trivial to simulate.

Although the position map as described above is large, it can be recursively outsourced to lower level ORAM structures to reduce its size. For simpler illustration, we consider here a non-recursive version of the CP-ORAM, where the large position map is stored in the CPU state. We note that our construction can handle full-fledged recursion.

3.2.3 Key Observation for "Puncturing" CP-ORAM

Consider the execution of an CP-ORAM[17] compiled program which accesses memory block ℓ in the i-th time step (corresponds to the i-th OACCESS call), the access pattern at this time step is determined by the position map value $p = pos[\ell]$ at this time.[18] So, as long as this value p is information-theoretically hidden from the adversary, we can simulate the access pattern by a random path even if everything else is leaked to the adversary. On the other hand, the value is generated at the last access time t' of this block, which can be much smaller than i, stored in both the position map and the block ℓ (as part of the meta data), and can be touched multiple times from time step t' to i. Thus, the value may appear many times in the computation trace of the evaluation. Nevertheless, by a sequence of carefully defined hybrids (which we refer to as partially punctured ORAM hybrids), we can erase the information of p step by step with constant-size data hardwired, which allows us to carry through the puncturing ORAM argument (see our later section for detailed hybrids).

3.2.4 Details for Puncturing CP-ORAM

Below is a more detailed sketch of the security proof to illustrate the puncturing ORAM technique in depth.

$\mathcal{RE}$ Simulator and Backward-in-time Hybrids Our construction is $\mathcal{RE}.\mathsf{Encode}(\Pi) = \mathsf{Ci}\mathcal{O}(P_{\mathsf{hide}}, x_{\mathsf{hide}})$, where P_{hide} is $\mathcal{PKE}$ and CP-ORAM compiled version of P. We construct simulated encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$ where P_{Sim} simulates P_{hide} for each time step of $P(x)$ (corresponding to each OACCESS call), and it simulates the access pattern by two (pseudo-)random paths supplied by PPRF (with a (different) key used in simulation). For each access, it ignores the input and outputs encryptions of $\mathtt{dummy}$ (as before). Note that in the rest of this section, a time step refers to a time step of $P(x)$, as opposed to a time step of $P_{\mathsf{hide}}(x_{\mathsf{hide}})$.

We prove the security by a sequence of hybrids that erases the computation *backward in time*. Namely, we consider intermediate hybrids $\mathbf{Hyb}_i$, a hybrid encoding $\mathsf{Ci}\mathcal{O}(P_{\mathbf{Hyb}_i}, x_{\mathsf{hide}})$ where $P_{\mathbf{Hyb}_i}$ acts as P_{hide} for the first i time steps, and acts as P_{Sim} in the remaining time steps. A main step in the proof is to show indistinguishability of $\mathbf{Hyb}_i$ and $\mathbf{Hyb}_{i-1}$, which corresponds to erasing the computation at the i-th time step. It is not hard to see that the outputs can be replaced by an encryption of $\mathtt{dummy}$ by a similar puncturing argument as in Section 2.4.1. To replace the access pattern, we define a *punctured* hybrid $\mathbf{Hyb}_i^{\mathrm{punct}}$ that punctures ORAM at time step i.

A Punctured Hybrid $\mathbf{Hyb}_i^{\mathrm{punct}}$ Let ℓ be the memory block accessed at the i-th time step of $P(x)$, $p = pos[\ell]$ be the position map value at the time, and t' be the last access time of block ℓ before time step i. Following the above key observation, our goal is to move to a hybrid where the value p is information-theoretically erased. We do so by injecting a "*puncturing*" code that removes the generation of the value p at time step t'. More precisely, we define a punctured hybrid $\mathbf{Hyb}_i^{\mathrm{punct}}$ with a hybrid encoding $\mathsf{Ci}\mathcal{O}(P_{\mathbf{Hyb}_i^{\mathrm{punct}}}, x_{\mathsf{hide}})$ where $P_{\mathbf{Hyb}_i^{\mathrm{punct}}}$ is $P_{\mathbf{Hyb}_i}$ with the following puncturing code added:

[17] Ideally, it is desirable to formalize a "puncturable" property for ORAM or the CP-ORAM construction, and use the property in the security proof. Unfortunately, we do not know how to formalize such a property without referring to our actual construction, since it is difficult to quantify what an adversary can learn from the $\mathsf{Ci}\mathcal{O}$ obfuscated computation. Yet, in an independent work [CH15] which also constructs succinct randomized encoding for RAM (but not for PRAM), a more modular and cleaner technique is developed to prove security.

[18] We ignore the uniformly random path used in the flush operation here, which is trivial to simulate.

Puncturing Code: At time step $t = t'$, do not generate the value p, and instead of putting back the (encrypted) fetched block ℓ to the root of the ORAM tree, an encryption of a dummy block is put back. Moreover, the position map value $pos[\ell]$ is not updated. Additionally, the value p is hardwired, and is used to emulate the memory access at the i-th time step.

In other words, block ℓ is deleted at time step t' and $pos[\ell]$ remains to store the old value (used to fetch the block at time step t'). So, in $\mathbf{Hyb}_i^{\text{punct}}$, the value p is information-theoretically hidden in the computation trace of the first $(i-1)$ time steps and is only used to determine the access pattern at time step i. We can then use puncturing arguments for PPRF to replace p by one generated by the simulation (as opposed to real) PPRF key.

We also note that since block ℓ is not accessed before time step i, the computation trace of $P_{\mathbf{Hyb}_i^{\text{punct}}}$ before time step i is identical to that of $P_{\mathbf{Hyb}_i}$, except that each time when $P_{\mathbf{Hyb}_i}$ touches the block ℓ (resp., $pos[\ell]$), it is replaced by a `dummy` block (resp., the old value) instead (though this can occur many times).

To complete the argument, we should argue indistinguishability between $\mathbf{Hyb}_i$ and $\mathbf{Hyb}_i^{\text{punct}}$. However, for simplicity of exposition, we consider a simplified goal as follows.

A Simplified Version $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ of Puncturing Hybrid Here we consider a simplified version of $\mathbf{Hyb}_i^{\text{punct}}$, denoted by $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$, where only the block ℓ is removed but the $pos[\ell]$ value is still updated at time step t'. Namely, the puncturing code is replaced by the following simplified version.

Puncturing Code (Simplified): At time step $t = t'$, instead of putting back (an encryption of) the fetched block ℓ to the root of the ORAM tree, (an encryption of) a dummy block is put back.

We focus on indistinguishability between $\mathbf{Hyb}_i$ and $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ (i.e., "deleting" the block ℓ) to simplify the exposition. The $pos[\ell]$ part can be addressed in a similar way to be detailed in the technical section.

Moving from $\mathbf{Hyb}_i$ to $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ As discussed above, the computation traces of $P_{\mathbf{Hyb}_i}$ and $P_{\widehat{\mathbf{Hyb}}_i^{\text{punct}}}$ can differ in many time steps, where each occurrence of the (encrypted) block ℓ is replaced by a dummy (encrypted) block. Thus, we cannot move from $\mathbf{Hyb}_i$ to $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ in one step, but requires a sequence of hybrids that gradually modifies the computation trace of $P_{\mathbf{Hyb}_i}$ to that of $P_{\widehat{\mathbf{Hyb}}_i^{\text{punct}}}$ step by step, using puncturing arguments for PPRF and (local) semantic security of $\mathcal{PKE}$ (as in Section 2.4.1). One natural approach is to keep track of the differing places in the computation trace, and replace the (encrypted) block ℓ by a (encrypted) dummy block one by one. This can indeed be done carefully by a sequence of hybrids backwards in time. However, when we consider parallel RAM computation, the difference of computation traces in the corresponding hybrids is more complicated, and it becomes tedious to keep track of the differences and modify them through hybrids.

We instead introduce an auxiliary "*partially puncturing*" code that punctures the information of p from certain threshold time step $j \geq t'$, and move from $\mathbf{Hyb}_i$ to $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ by a sequence of hybrids that moves the threshold j from $i \geq j$ backwards to t'. Such a *code modification* technique can be generalized to handle corresponding hybrids in the PRAM setting.

Partially Punctured Hybrids $\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}$ We define partially punctured hybrids $\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}$ indexed by a threshold time step j, where the underlying $P_{\widehat{\mathbf{Hyb}}_i^{\text{punct}}}$ is $P_{\mathbf{Hyb}_i}$ with the partially puncturing code below added:

Partially Puncturing Code$[j]$: At any time step $t > j$, if the input CPU state or memory contains the block ℓ, then replace it by a dummy block before performing the computation.

In other words, $P_{\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}}$ punctures the block ℓ after threshold time step j by *deleting the block from its input*. We will see it is important that the code removes block ℓ by deleting it from the input (as opposed to output).

Moving from $\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}$ to $\widehat{\mathbf{Hyb}}_{i,j-1}^{\text{p-punct}}$ We show indistinguishability between $\mathbf{Hyb}_i$ and $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ by moving the threshold j from i to t'. As the main step, we prove indistinguishability between $\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}$ to $\widehat{\mathbf{Hyb}}_{i,j-1}^{\text{p-punct}}$, which corresponds to deleting the block ℓ from the input at time step j. Now there are two cases. If the input at the j-th time step does not contain the block ℓ, then $P_{\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}}$ and $P_{\widehat{\mathbf{Hyb}}_{i,j-1}^{\text{p-punct}}}$ have identical computation trace, and thus indistinguishability follows by $\mathsf{Ci}\mathcal{O}$ security. If the input contains the block ℓ, then the computation traces can be different. We observe that, since the block ℓ is not accessed at time step j, the difference is to correspondingly replace the (encrypted) block ℓ by a dummy block in the output. Thus, to show indistinguishability, we use the puncturing PRF argument and semantic security of $\mathcal{PKE}$ to modify the output.

However, the situation here is more complicated. Previously, we only modify an encryption whose corresponding decryption key is not used in the honest computation, since the computation afterward is erased. Here, the encrypted output block can later be accessed by $P_{\widehat{\mathbf{Hyb}}_{i,j}^{\text{p-punct}}}$ at some time step $j < t < i$, where the computation is not erased and thus the decryption key is still in use. Let ct be the encrypted output block ℓ at the j-th time step, and sk be the corresponding decryption key. In order to replace ct, we proceed in the following steps, which use the property that the partially puncturing code deletes the block ℓ from the input:

- We first move to a hybrid where the block ℓ is hardwired, and the decryption of ct is set to the hardwired value instead of decryption using sk so that the decryption key sk is not used.

- We then replace ct by an encryption of a dummy block using puncturing PRF argument and semantic security of $\mathcal{PKE}$. Note that this creates inconsistency in that the decryption of ct is still set to the hardwired block ℓ, as opposed to the dummy block.

- We now undo the hardwiring and use sk to decrypt ct again to reach $\widehat{\mathbf{Hyb}}_{i,j-1}^{\text{p-punct}}$. This fixes the inconsistency from the previous step. On the other hand, the decryption of ct is changed from the block ℓ to dummy, so when ct is accessed in later time steps (after j), the input block ℓ is replaced by a dummy block. Here is the place we use the property of partially punctured code: after time step j, the input block ℓ is replaced by a dummy block before the computation anyways. Thus, this change does not effect the computation trace, and indistinguishability follows by $\mathsf{Ci}\mathcal{O}$ security.

We can now argue indistinguishability of $\mathbf{Hyb}_i$ to $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$. Indistinguishability of $\mathbf{Hyb}_i$ and $\widehat{\mathbf{Hyb}}_{i,i}^{\text{p-punct}}$ follows from $\mathsf{Ci}\mathcal{O}$ security by observing that they have identical computation trace. The above argument allows us to move from $\widehat{\mathbf{Hyb}}_{i,i}^{\text{p-punct}}$ to $\widehat{\mathbf{Hyb}}_{i,t'}^{\text{p-punct}}$. Now, note that the difference between $\widehat{\mathbf{Hyb}}_{i,t'}^{\text{p-punct}}$ and $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ is that the output block ℓ at time step t' is replaced by a dummy block in the later hybrid. The indistinguishability follows by the same argument using PPRF, $\mathcal{PKE}$, and the property of partially punctured code as above.

Recall that $\widehat{\mathbf{Hyb}}_i^{\text{punct}}$ is a simplified version of punctured hybrid. To move to the actual punctured hybrid $\mathbf{Hyb}_i^{\text{punct}}$, we can apply the same argument to handle the value stored in $\mathit{pos}[\ell]$.

Summarizing the Hybrids We summarize the long (nested) sequences of hybrids we discussed so far. Above we showed how to move from $\mathbf{Hyb}_i$ to $\mathbf{Hyb}_i^{\text{punct}}$, which corresponds to puncturing ORAM at time step i. From $\mathbf{Hyb}_i^{\text{punct}}$, we can replace the output by encryption of dummy and replace the access pattern by a simulated one. We can then reach $\mathbf{Hyb}_{i-1}$ by undo the puncturing.

Finally, we can complete the proof in a similar way as the previous section, where we move from the real encoding to $\mathbf{Hyb}_{t^*-1}$ by $\mathsf{Ci}\mathcal{O}$ security, then move to $\mathbf{Hyb}_0$ by the above puncturing ORAM technique, and then replace x_{hide} by x_{Sim} using puncturable PRF and semantic security of $\mathcal{PKE}$.

3.3 Handling Parallel Memory Access

Lastly, we discuss how the techniques above for ORAM can be extended to the OPRAM setting.

3.3.1 Review of BCP OPRAM Construction

BCP OPRAM [BCP16] is a natural generalization of tree-based ORAM. Consider that each CPU j wants to access memory block ℓ_j in a (parallel) memory access. At a high level, the CPUs first communicate with each other to resolve the conflicts, and recursively invoke OPACCESS to fetch and refresh the position map values. They then fetch the memory blocks from the path, put the blocks back, and flush the tree in parallel. Since the m CPUs want to access m paths p_j of the tree in parallel, they need to communicate with each other to avoid write conflicts. In BCP OPRAM, the CPUs access the tree level by level, and in each level, they aggregate the access instructions, select representative to perform the access, and then distribute the answers via oblivious aggregation and oblivious multi-casting protocols.

3.3.2 Simulated Encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$

As before, P_{Sim} simulates P_{hide} for each (parallel) time step of $P(x)$. At each step P_{Sim} uses the simulated access pattern and erases the computation by ignoring the input and then outputting encryption of `dummy`. Here, we need to simulate the parallel access pattern of OPACCESS, which is more complicated and involves polylogarithmic time steps because all CPUs in OPACCESS interact with each other. In particular, the access pattern of the OPACCESS depends on the paths p_j's each CPU wants to access, where each CPU manipulates its path with its own state and OPACCESS. If we still erase all the CPU states step by step, we would not have enough information to simulate the second half access pattern of OPACCESS once the CPU states in the first half is erased. Nevertheless, the key observation here is that the access pattern is fully determined by the paths p_j's each CPU wants to access, which are *public* information revealed in the execution. So, we can view these p_j's as *public states* of OPACCESS, and do not erase its content in the hybrids. In other words, we generate simulated path p_j for each CPU, and store them as public states to simulate the access pattern of OPACCESS.

3.3.3 Puncturing BCP OPRAM CPU by CPU

As BCP OPRAM is a generalization of tree-based ORAM, it is not hard to see that the puncturing argument generalizes to work for BCP OPRAM as well (while maintaining full succinctness). Namely, it suffices to information-theoretically hide the values of the paths p_j's to simulate the access pattern, and this can be done by injecting a puncturing code. Additionally, we observe that this can be done CPU by CPU. Namely, for each p_j accessed by CPU j, we can (for all CPUs) inject a puncturing code at the corresponding time step t'_j that the value p_j is generated, to remove the generation of p_j. Also, we can move to this punctured hybrid by a sequence of partially punctured hybrids as before, by gradually puncturing the value of p_j backwards in time, per time-step and per CPU. Upon reaching this punctured hybrid, we can switch p_j to a simulated one, undo the puncturing, and move to the next CPU. In this way, we switch the paths p_j's to simulated version one by one, and never need to hardwire information of size depending on m throughout the hybrids, which maintains full succinctness.

4 Computation-Trace Indistinguishability Obfuscation ($\mathsf{Ci}\mathcal{O}$)

We define a new primitive called Computation-trace Indistinguishability Obfuscation ($\mathsf{Ci}\mathcal{O}$), which produces indistinguishably obfuscated computations as long as the input computations give identical computation trace. For this, we need to define a formal notion of computation trace, and before that, a formal notion of (distributed) computation systems.

We define a distributed computation system Π as a tuple consisting of a collection initial states $\{\mathsf{st}_k^0\}$, a shared initial memory mem^0, and a collection of stateful algorithms $\{F_k\}$. The entity (named agent $\mathtt{A}_k$) executes the stateful algorithm F_k by taking as input the state in the previous time step, an access command received from the memory, and some communication messages received from the other agents. It outputs a new state, an access command to be sent to the memory, and some communication messages to be sent to the other agents. The memory which receives access commands from all the agents processes these commands and outputs some new access commands to be returned to all the agents. Having specified these, the computation trace of a distributed computation system is simply defined as the collection of configurations including the states, access commands and communication messages at all time steps.

The philosophy behind such a definition is to decouple the functionality of a program and its computation trace. On one hand, programs with the same functionality can still produce different computation traces. On the other hand, we wish to only focus at one particular instance of a program-input pair (P, x) rather than the entire functionality of P.

4.1 Model of Distributed Computation Systems

Definition 4.1. *We define a* distributed computation system Π *with an evaluation algorithm* **evaluate** *as follows.*

Description of the Computation. *Π consists of m agents $\mathtt{A}_1, \ldots, \mathtt{A}_m$ and a shared memory component $\mathtt{M}$. Each agent $\mathtt{A}_k$ where $k \in [m]$ is associated with:* (i) *a stateful algorithm F_k;* (ii) *a register for storing its local state* st_k*;* (iii) *an incoming communication buffers which allow any other agents $\mathtt{A}_j$ where $j \in [m] \setminus \{k\}$ to send communication messages $c_{k\leftarrow j}$ to agent $\mathtt{A}_k$;* (iv) *an incoming memory access buffer which stores value $a_{k\leftarrow\mathtt{M}}$ that read from the shared memory.*

Memory component $\mathtt{M}$ is a distinguished component in the computation system, associated with: (i) *a memory* mem*;* (ii) *an incoming memory access buffers which allow any agents $\mathtt{A}_j$ where $j \in [m]$ to write the memory with a value $a_{\mathtt{M}\leftarrow j}$.*

For all $k \in [m]$, for all $j \in [m] \setminus \{k\}$, Π is set to be $(\mathsf{st}_k, c_{k\leftarrow j}, a_{k\leftarrow\mathtt{M}}, a_{\mathtt{M}\leftarrow k}, \mathsf{mem})$ with the initialized values $(\mathsf{st}_k^0, c_{k\leftarrow j}^0, a_{k\leftarrow\mathtt{M}}^0, a_{\mathtt{M}\leftarrow k}^0, \mathsf{mem}^0)$ given externally.

We denote the computation system by

$$\Pi = ((\mathsf{mem}^0, \{\mathsf{st}_k^0, \{c_{k\leftarrow j}^0\}_{j\in[m]\setminus\{k\}}, a_{k\leftarrow\mathtt{M}}^0, a_{\mathtt{M}\leftarrow k}^0\}_{k=1}^m), (\{F_k\}_{k=1}^m)).$$

Computation System Evaluation Procedure. *The procedure* **evaluate**() *will evaluate the system Π by rounds. For each round $t > 0$,*

– *Each agent $\mathtt{A}_k$ where $k \in [m]$ operates as follows:*
  - *Reads its incoming communication buffers and memory access buffer, and obtains $\mathbf{c}_{k\leftarrow\cdot}^{t-1} = \{c_{k\leftarrow j}^{t-1}\}_{j\in[m]\setminus\{k\}}$ and $a_{k\leftarrow\mathtt{M}}^{t-1}$ respectively.*
  - *Computes $(\mathsf{st}_k^t, a_{\mathtt{M}\leftarrow k}^t, \mathbf{c}_{\cdot\leftarrow k}^t) \leftarrow F_k(\mathsf{st}_k^{t-1}, a_{k\leftarrow\mathtt{M}}^{t-1}, \mathbf{c}_{k\leftarrow\cdot}^{t-1})$, where $\mathbf{c}_{\cdot\leftarrow k}^t = \{c_{j\leftarrow k}^t\}_{j\in[m]\setminus\{k\}}$.*
    If $\mathsf{st}_k^{t-1} = (\mathtt{halt}, \cdot)$ or $\mathtt{Reject}$ then $a_{\mathtt{M}\leftarrow k}^t := \perp$, $c_{j\leftarrow k}^t := \perp$ for $j \in [m] \setminus \{k\}$, $\mathsf{st}_k^t := \mathsf{st}_k^{t-1}$.
  - *Writes the value $a_{\mathtt{M}\leftarrow k}^t$ to the incoming memory access buffers, and sends the messages $c_{j\leftarrow k}^t$ to the incoming communication buffer of agent $j \in [m] \setminus \{k\}$.*

– *The memory component $\mathtt{M}$ operates as follows:*
  - *Reads its incoming memory access buffers, and obtains $a_{\mathtt{M}\leftarrow 1}^t, \ldots, a_{\mathtt{M}\leftarrow m}^t$.*
  - *Computes $(\mathsf{mem}^t, a_{1\leftarrow\mathtt{M}}^t, \ldots, a_{m\leftarrow\mathtt{M}}^t) \leftarrow \mathsf{access}(\mathsf{mem}^{t-1}, a_{\mathtt{M}\leftarrow 1}^t, \ldots, a_{\mathtt{M}\leftarrow m}^t)$, where* access *performs the memory access command $a_{\mathtt{M}\leftarrow k}^t$ on memory mem^{t-1} and output its corresponding read value $a_{k\leftarrow\mathtt{M}}^t$ and update memory mem^t for each $k \in [m]$.*
  - *Returns value $a_{k\leftarrow\mathtt{M}}^t$ to each agent k's memory access buffer, where $k \in [m]$.*

Terminologies *To facilitate presentation, we introduce the following terminologies.*

- *The* terminating time *t^*, if it exists, is the smallest t such that $\mathsf{st}_k^t = (\mathtt{halt}, \cdot)$ for all $k \in [m]$.*
- *The* computation configuration *at any time $t \geq 0$, defined as*
  $\mathsf{Conf}\langle\Pi, t\rangle = (\{(\mathsf{st}_k^t, a_{k\leftarrow\mathtt{M}}^t, a_{\mathtt{M}\leftarrow k}^t, \mathbf{c}_{k\leftarrow\cdot}^t)\}_{k=1}^m, \mathsf{mem}^t)$*, is the output of the evaluation at time t.*

- *The* computation trace *is defined as* $\mathsf{Trace}\langle\Pi\rangle = \{\mathsf{Conf}\langle\Pi, t\rangle\}_{t\geq 0}$.
- *The* next step *function F is another representation of all F_k such that $F(k, \mathsf{st}, a, \mathbf{c}) = F_k(\mathsf{st}, a, \mathbf{c})$ for all* $\mathsf{st}, a, \mathbf{c}$.

Remark 4.2. *For specific computation systems, we can restrict the initial states, access commands and communication messages to some default values. Such initial values can thus be dropped from the tuple Π, and we use the simplified form $\Pi = \big((\mathsf{mem}^0, \mathsf{st}_1^0, \mathsf{st}_2^0, \ldots, \mathsf{st}_m^0), (F_1, F_2, \ldots, F_m)\big)$.*

Remark 4.3. *For easier presentation, the model defined here is centered around PRAM model. We can easily generalize the definition to support richer syntax for even more fine-grained model of computation.*

4.2 Computation-trace Indistinguishability Obfuscation

In this subsection, we will introduce a new security notion named *Computation-trace Indistinguishability Obfuscation* ($\mathsf{Ci}\mathcal{O}$ for short). Recall that $\mathsf{i}\mathcal{O}$ captures the security intuition that if the *functionalities* of the computations are identical, then the complied / obfuscated versions are indistinguishable. Here, we want to capture an even milder security property that if the *computation traces* are identical, then the obfuscated versions are indistinguishable.

A $\mathsf{Ci}\mathcal{O}$ scheme consists two parts, a probabilistic compilation procedure Obf which transforms a computation system to an "obfuscated" computation system, and a deterministic evaluation algorithm Eval which evaluates the obfuscated system to return the output.

Definition 4.4. *Let $\mathcal{P}$ be a collection of computation systems. A* computation-trace indistinguishability obfuscation scheme w.r.t $\mathcal{P}$*, denoted by $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{Obf}, \mathsf{Eval}\}$, is defined as follows:*

Compilation algorithm $\widetilde{\Pi} := \mathsf{Obf}(1^\lambda, \Pi; \rho)$: $\mathsf{Obf}()$ *is a probabilistic algorithm which takes as input the security parameter λ, the computation system $\Pi \in \mathcal{P}$ and some randomness ρ, and returns a complied / obfuscated system $\widetilde{\Pi}$ as output.*

Evaluation algorithm $\mathsf{conf} := \mathsf{Eval}(\widetilde{\Pi})$**:** $\mathsf{Eval}()$ *is a deterministic algorithm which takes as input the obfuscated system $\widetilde{\Pi}$, and returns a configuration of the original computation system Π as output.*

Correctness *For all $\Pi \in \mathcal{P}$ with termination time t^* and all randomness ρ, let $\widetilde{\Pi} := \mathsf{Obf}(1^\lambda, \Pi; \rho)$. It holds that* $\mathsf{Eval}(\widetilde{\Pi}) = \mathsf{Conf}\langle\Pi, t^*\rangle$.

Security *For any (not necessarily uniform) PPT distinguisher $\mathcal{D}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that, for all security parameters $\lambda \in \mathbb{N}$, $\Pi^0, \Pi^1 \in \mathcal{P}$ where $\mathsf{Trace}\langle\Pi^0\rangle = \mathsf{Trace}\langle\Pi^1\rangle$, it holds that*

$$|\Pr[\mathcal{D}(\mathsf{Obf}(1^\lambda, \Pi^0)) = 1] - \Pr[\mathcal{D}(\mathsf{Obf}(1^\lambda, \Pi^1)) = 1] \leq \mathsf{negl}(\lambda).$$

Efficiency *We require* Obf *runs in time $\tilde{O}(\mathsf{poly}(|\Pi|))$, and efficient* Eval *runs in time $\tilde{O}(t^*)$. That is, a client can efficiently compile Π, and a server carries out evaluation in time comparable to the insecure computation.*

Let t^* be the halting time of the computation system Π. A trivial construction is to perform all computations in the Obf algorithm. However, this trivial case does not work because we require Obf to be efficient and independent to the computation time t^*.

5 Starting Point: Constructing $\mathsf{Ci}\mathcal{O}$ in the RAM Model ($\mathsf{Ci}\mathcal{O}$-RAM)

As introduced in Section 4, positing two computations which give identical computation trace, $\mathsf{Ci}\mathcal{O}$ guarantees the indistinguishability of their obfuscations. Viewing from another perspective, $\mathsf{Ci}\mathcal{O}$ forces the evaluator to evaluate the obfuscated computation as intended, so as to only produce the intended computation trace.

To construct $\mathsf{Ci}\mathcal{O}$, a natural idea is therefore to authenticate the output of a time step and verify the integrity of the input in the next time step. Straightforwardly, the output state (which is small in size) will be signed using a signature scheme. On the other hand, the memory has a much larger size and is controlled by the evaluator outside the obfuscated program. For authenticating the memory, a Merkle-tree-like structure is used to produce a digest which is then stored in the CPU state. Having a similar construction but a more vigorous and ambitious security goal, our $\mathsf{Ci}\mathcal{O}$ can be referred to as an abstraction of KLW.

In this section, we first tackle the simpler task of constructing $\mathsf{Ci}\mathcal{O}$ in the RAM model. For this, we compile the underlying function F into another function $\widehat{F}$ which verifies and authenticates its inputs and outputs respectively. Concretely, upon receiving as input a time t, a CPU state, and a bit read from the memory, function $\widehat{F}$ verifies the signature of the input CPU states and the memory digest, before executing the underlying function F. $\widehat{F}$ then signs the resulting CPU state, updates the Merkle-tree-like structure to obtain an updated memory digest, and eventually outputs the time $(t+1)$, a new CPU state, and an access command. At the compilation stage, we convert the initial state st^0 and the initial memory mem^0 to an authenticated form $(\widetilde{\mathsf{st}}^0, \widetilde{\mathsf{mem}}^0)$, and compute $\mathsf{Ci}\mathcal{O}(\Pi) = ((\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0), \widetilde{F})$ where $\widetilde{F} = \mathsf{i}\mathcal{O}(\widehat{F})$.

5.1 Building Blocks

In our $\mathsf{Ci}\mathcal{O}$ construction in Section 5.2, we will use several building blocks: accumulator, iterator, splittable signature, puncturable PRF, and indistinguishability obfuscation. The formal definitions of these primitives can be found in Appendix A. Below we define the notations and parameters for the building blocks we will use in our $\mathsf{Ci}\mathcal{O}$ construction.

- Accumulator scheme $\mathsf{Acc} = \mathsf{Acc}.\{\mathsf{Setup}, \mathsf{SetupEnforceRead}, \mathsf{SetupEnforceWrite}, \mathsf{PrepRead}, \mathsf{PrepWrite}, \mathsf{VerifyRead}, \mathsf{WriteStore}, \mathsf{Update}\}$ with message space $\{0,1\}^{\ell_{\mathsf{msg}}}$ and accumulated value space $\{0,1\}^{\ell_{\mathsf{Acc}}}$.
- Iterator scheme $\mathsf{Itr} = \mathsf{Itr}.\{\mathsf{Setup}, \mathsf{SetupEnforceIterate}, \mathsf{Iterate}\}$ with message space $\{0,1\}^{\ell_{\mathsf{Acc}}+\ell_{\mathsf{msg}}}$ and iterated value space$\{0,1\}^{\ell_{\mathsf{Itr}}}$.
- Splittable signature scheme $\mathsf{Spl} = \mathsf{Spl}.\{\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Split}, \mathsf{AboSign}\}$, $\mathsf{Spl.Setup}$ uses ℓ_{rnd} bits of randomness, and the message space is $\{0,1\}^{\ell_{\mathsf{Itr}}+\ell_{\mathsf{Acc}}+\ell_{\mathsf{msg}}}$.
- Puncturable PRF scheme $\mathsf{PPRF} = \mathsf{PPRF}.\{\mathsf{Setup}, \mathsf{Puncture}, \mathsf{Eval}\}$ with key space $\mathcal{K}$, punctured key space $\mathcal{K}_{\mathsf{punct}}$, domain $[T]$ where T is a parameter to be defined (which will be the time bound on the terminating time of a RAM program), and range $\{0,1\}^{\ell_{\mathsf{rnd}}}$.
- Indistinguishability obfuscation scheme $\mathsf{i}\mathcal{O}$ for circuits.

5.2 Construction for $\mathsf{Ci}\mathcal{O}$-RAM

We now construct $\mathsf{Ci}\mathcal{O}$ in the RAM model. Informally, a RAM consists of a single CPU, with random access to an external memory, executing a next-step circuit F step by step until reaching the termination state which embeds the computation result $P(x)$ for some program P evaluated on some input x. Hence, a RAM computation Π can be specified by a program P and an initial input x. The evaluator interprets the input x so as to prepare the initial state st^0 and the initial memory mem^0, to which it has random access ability. P is converted to the stateful algorithm F. At each time step, F is executed with the state in the previous time step and a bit read from the memory as input. F outputs a new state for the next step, and a memory access command.

Formally, the class $\mathcal{P}_{\mathrm{RAM}}$ of distributed computation emulating RAM computation is defined as follows:

Definition 5.1 (RAM Computation Class)**.** *We define $\mathcal{P}_{\mathrm{RAM}}$ as a class of distributed computation systems for RAM computation with a single agent* $\mathtt{A}$ *(a.k.a. CPU) and a memory* $\mathtt{M}$ *where*

- *the terminating time t^* is bounded by T;*

– *the memory size* $|\mathsf{mem}|$ *is bounded by* S*;*
– *the state size* $|\mathsf{st}|$ *is bounded by* $\mathsf{poly}\log(T)$*;*
– *the communication buffer sizes* $|a_{\mathtt{A}\leftarrow\mathtt{M}}|$ *and* $|a_{\mathtt{M}\leftarrow\mathtt{A}}|$ *are bounded by* $\mathsf{poly}\log(S)$*;*
– *the initial access commands are empty, i.e.,* $a^0_{\mathtt{A}\leftarrow\mathtt{M}} := \perp$ *and* $a^0_{\mathtt{M}\leftarrow\mathtt{A}} := \perp$.

In this subsection, we describe our scheme $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{Obf}, \mathsf{Eval}\}$ where $\mathsf{Ci}\mathcal{O}.\mathsf{Obf}$ can transform a given computation system $\Pi \in \mathcal{P}_{\mathrm{RAM}}$ into an obfuscated computation system $\widetilde{\Pi}$, where

$$\Pi = ((\mathsf{mem}^0, \mathsf{st}^0), F), \text{ and}$$

$$\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0), \widetilde{F}).$$

Compilation algorithm $\widetilde{\Pi} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \Pi)$ The compilation algorithm $\mathsf{Obf}()$ consists of several steps.
Step 1: Generating parameters. The compilation algorithm computes the following parameters for the obfuscated computation system:

$$K_A \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$$
$$(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0) \leftarrow \mathsf{Acc.Setup}(T)$$
$$(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{Itr.Setup}(T)$$

Step 2: Generating stateful algorithms $\widetilde{F}$*.* Based on the parameters $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$ generated above, as well as program F, we define the program $\widehat{F}$ in Algorithm 1. The compilation procedure then computes an obfuscation of the program $\widehat{F}$. That is, $\widetilde{F} \leftarrow \mathsf{i}\mathcal{O}.\mathsf{Gen}(\widehat{F})$.

Algorithm 1: $\widehat{F}$

```
    Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$, $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
    Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$

 1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output Reject;
 2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$;
 3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$;
 4  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$;
 5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 0$ then output Reject;

 6  Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$ where $a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}})$;
 7  if $\mathsf{st}^{\mathrm{out}} = \mathtt{Reject}$ then output Reject;

 8  $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
 9  if $w^{\mathrm{out}} = \mathtt{Reject}$ then output Reject;
10  Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
11  if $v^{\mathrm{out}} = \mathtt{Reject}$ then output Reject;
12  Compute $r'_A = \mathsf{PRF}(K_A, t)$;
13  Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$;
14  Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
15  Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;

16  Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$, $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Step 3: Generating the initial configuration $(\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0)$*.* Recall that $a^0_{\mathtt{A}\leftarrow\mathtt{M}} = \perp$, $a^0_{\mathtt{M}\leftarrow\mathtt{A}} = \perp$. Based on given $\mathsf{mem}^0, \mathsf{st}^0$, the compilation procedure computes the initial configuration as follows.

– For each $j \in \{1, \dots, |\mathsf{mem}^0|\}$, it computes iteratively:

$$\begin{aligned}\pi_j &\leftarrow \mathsf{Acc.PrepWrite}(\mathsf{pp_{Acc}}, \hat{store}_{j-1}, j)\\ \hat{w}_j &\leftarrow \mathsf{Acc.Update}(\mathsf{pp_{Acc}}, \hat{w}_{j-1}, j, x_j, \pi_j)\\ \hat{store}_j &\leftarrow \mathsf{Acc.WriteStore}(\mathsf{pp_{Acc}}, \hat{store}_{j-1,}, j, \mathsf{mem}^0[j])\end{aligned}$$

Set $w^0 := \hat{w}_{|\mathsf{mem}^0|}$, and $store^0 := \hat{store}_{|\mathsf{mem}^0|}$.

– Compute σ^0 as follows:

$$\begin{aligned} r_A &\leftarrow \mathsf{PRF}(K_A, 0)\\ (\mathrm{sk}^0, \mathrm{vk}^0) &\leftarrow \mathsf{Spl.Setup}(1^\lambda; r_A)\\ \sigma^0 &\leftarrow \mathsf{Spl.Sign}(\mathrm{sk}^0, (0, w^0, v^0))\end{aligned}$$

– Now we can define the initial configuration as

$$\begin{aligned}\widetilde{\mathsf{mem}}^0 &= store^0\\ \widetilde{\mathsf{st}}^0 &= (0, \mathsf{st}^0, v^0, w^0, \sigma^0)\end{aligned}$$

Final step. The compilation procedure returns $\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0), \widetilde{F})$ as output.

Evaluation algorithm $\mathsf{conf} := \mathsf{Eval}(\widetilde{\Pi})$ Upon receiving an obfuscated system $\widetilde{\Pi}$, the evaluation algorithm carries out the following:

Set $\widetilde{a}^0_{\mathtt{A}\leftarrow\mathtt{M}} = \perp$. For $t = 1$ to T, perform following procedures until $\widetilde{F}$ outputs a halting state $\widetilde{\mathsf{st}}^{t^*}$ at that halting time t^*:

– Compute $(\widetilde{\mathsf{st}}^t, \widetilde{a}^t_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow \widetilde{F}(\widetilde{\mathsf{st}}^{t-1}, \widetilde{a}^{t-1}_{\mathtt{A}\leftarrow\mathtt{M}})$;

– Run $(\widetilde{\mathsf{mem}}^t, \widetilde{a}^t_{\mathtt{A}\leftarrow\mathtt{M}}) \leftarrow \widetilde{\mathsf{access}}(\widetilde{\mathsf{mem}}^{t-1}, \widetilde{a}^t_{\mathtt{M}\leftarrow\mathtt{A}})$, where $\widetilde{\mathsf{access}}$ is defined in Algorithm 2.

Parse $\widetilde{\mathsf{mem}}^{t^*} = \mathsf{mem}^{t^*}$ and $\widetilde{\mathsf{st}}^{t^*} = (t^*, \mathsf{st}^{t^*}, v^{t^*}, w^{t^*}, \sigma^{t^*})$. Output $\mathsf{conf} = (\mathsf{mem}^{t^*}, \mathsf{st}^{t^*}, a^{t^*}_{\mathtt{A}\leftarrow\mathtt{M}}, a^{t^*}_{\mathtt{M}\leftarrow\mathtt{A}})$ where $a^{t^*}_{\mathtt{A}\leftarrow\mathtt{M}} = a^{t^*}_{\mathtt{M}\leftarrow\mathtt{A}} = \perp$.

Algorithm 2: $\widetilde{\mathsf{access}}$

```
  Input  : $\widetilde{\mathsf{mem}}^{\mathrm{in}}, \widetilde{a}^{\mathrm{in}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
1 if $\widetilde{a}^{\mathrm{in}}_{\mathtt{M}\leftarrow\mathtt{A}} = \perp$ then output $\widetilde{\mathsf{mem}}^{\mathrm{out}} = \widetilde{\mathsf{mem}}^{\mathrm{in}}$;
2 Compute $\widetilde{\mathsf{mem}}^{\mathrm{out}} \leftarrow \mathsf{WriteStore}(\mathsf{pp_{Acc}}, \widetilde{\mathsf{mem}}^{\mathrm{in}}, (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}))$;
3 Compute $((\mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}), \pi^{\mathrm{out}}) \leftarrow \mathsf{PrepRead}(\mathsf{pp_{Acc}}, \widetilde{\mathsf{mem}}^{\mathrm{out}}, \mathbf{I}^{\mathrm{in}})$;
4 Output $(\widetilde{\mathsf{mem}}^{\mathrm{out}}, \widetilde{a}^{\mathrm{out}}_{\mathtt{A}\leftarrow\mathtt{M}} = ((\mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}), \pi^{\mathrm{out}}))$;
```

Efficiency Let $|F|$ be the description size of program F, n be the description size of initial memory mem^0, computation system Π proceeds with time and space bound T and S. Assuming $\mathsf{i}\mathcal{O}$ is a circuit obfuscator with circuit size $|\mathsf{i}\mathcal{O}(C)| \leq \mathsf{poly}(|C|)$ for given circuit C. Our $\mathsf{Ci}\mathcal{O}$-RAM has following complexity:

– Compilation time is $\tilde{O}(\mathsf{poly}(|F|) + n)$.

– Compilation size is $\tilde{O}(\mathsf{poly}(|F|) + n)$.

– Evaluation time is $\tilde{O}(T \cdot \mathsf{poly}(|F|))$.

– Evaluation space is $\tilde{O}(S)$, where S term is needed by F intrinsically.

Theorem 5.2. *Assume* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscation for circuits scheme,* PPRF *is a secure puncturable PRF scheme,* Itr *is a secure iterator,* Acc *is a secure positional accumulator scheme, and* Spl *is a secure splittable signature scheme; then, our construction of* $\mathsf{Ci}\mathcal{O}$ *is a computation-trace indistinguishability obfuscation scheme with respect to class* $\mathcal{P}_{\mathrm{RAM}}$.

The proof can be found in Appendix B.1.

6 Constructing $\mathsf{Ci}\mathcal{O}$ in the PRAM Model ($\mathsf{Ci}\mathcal{O}$-PRAM)

In this section we construct a computation-trace indistinguishability obfuscation scheme $\mathsf{Ci}\mathcal{O}$ in the PRAM model. Our final scheme $\mathsf{Ci}\mathcal{O}$-PRAM is quite technical. For facilitating presentation, we will illustrate our main ideas gradually via three attempts.

First, as a *naïve attempt*, we directly extend our $\mathsf{Ci}\mathcal{O}$-RAM scheme into that in the PRAM model. However, we face two technical challenges: **(i)** In our $\mathsf{Ci}\mathcal{O}$-RAM construction, memory accumulator has been used. We therefore need a new strategy to efficiently compute the memory accumulator digest in the parallel setting. **(ii)** In the PRAM model, for each CPU step, its output depends on all previous CPU steps (and further depends on their own previous steps). We therefore need to track these dependencies in the security proof. For efficiency, we are unable to hardwire too much information, so the dependencies pose a significant challenge. In Section 6.1, we will formalize the *dependency problem*.

In our *second attempt*, we focus on the dependency problem raised in the attempt above. Instead of directly providing a solution in the standard PRAM model, we here consider a special model, memoryless PRAM model mPRAM, which has no memory component but allows communications between CPUs. To address the dependency issue in constructing $\mathsf{Ci}\mathcal{O}$ in the mPRAM model, we introduce the "branch-and-combine" technique, which allows us to maintain an accumulator that stores m CPU states and messages, without scarifying efficiency too much. In Section 6.5, we will present this technique.

Finally, in our *full-fledged attempt*, we extend the above mPRAM model solution to $\mathsf{Ci}\mathcal{O}$ in the standard PRAM model. In addition to the branch-and-combine technique, we also use parallel accumulator to compute accumulator digest in the standard PRAM model. See Section 6.6 for more details.

Section Outline In the next subsection (Section 6.1), we will describe the naïve attempt based on $\mathsf{Ci}\mathcal{O}$-RAM and briefly discuss the reason why it fails. Specifically, we show that the dependency problem exists in parallel setting. The remainder of this section will be organized as follows. For completeness, we list the building blocks for our constructions in Section 6.2. Among the building blocks, we will introduce in Section 6.3 a new primitive named *Topological Iterators*, which will replace the (ordinary) iterator in our construction. Then, we will present parallel accumulator as the solution to the first challenge **(i)** in Section 6.4. Next, as a warm-up to overcome the second challenge **(ii)**, a construction of $\mathsf{Ci}\mathcal{O}$ for memoryless model mPRAM is shown by using a new branch-and-combine technique in Section 6.5, with its security proof in Appendix B.3. Finally, in Section 6.6, we will show the full-fledged construction of $\mathsf{Ci}\mathcal{O}$ for (standard) PRAM, with its security proof sketched in Appendix B.4.

6.1 Generalizing $\mathsf{Ci}\mathcal{O}$-RAM to $\mathsf{Ci}\mathcal{O}$-PRAM: A "Pebble Game" Illustration

To construct $\mathsf{Ci}\mathcal{O}$-PRAM, a trivial solution is to convert PRAM computation Π to RAM computation Π' and to obfuscate Π' with $\mathsf{Ci}\mathcal{O}$-RAM in a black-box manner. However, this convert-to-RAM solution is not acceptable because it has $\Omega(m)$ multiplicative parallel time overhead, which does not benefit from parallelization. To enjoy the benefits of parallel computation, our goal is to construct an efficient $\mathsf{Ci}\mathcal{O}$-PRAM such that the parallel time overhead is minimized. In general, we require the parallel time overhead to be $O(\log m)$.

An attempt to build efficient $\mathsf{Ci}\mathcal{O}$-PRAM is through the generalization of $\mathsf{Ci}\mathcal{O}$-RAM directly. Let Π be a PRAM computation composed of an m-CPU PRAM program P and an input x. The program P takes CPU id i as input and emulates the i-th CPU. Then we can obfuscate P associated with i and the input x by the same way as in $\mathsf{Ci}\mathcal{O}$-RAM. Finally, the evaluation algorithm runs m copies of the obfuscated program in parallel with different CPU id to emulate the PRAM computation. It preserves the parallel runtime of the evaluation algorithm.

The construction of $\mathsf{Ci}\mathcal{O}$-RAM (presented in Section 5) follows KLW construction, which utilizes splittable signature, iterator, and accumulator. To realize $\mathsf{Ci}\mathcal{O}$-PRAM, we need to address the issue of updating accumulator digest in the parallel setting, i.e., m CPUs can perform parallel writes to different memory cells, and they need to obtain updated accumulator digest in some way. However, we face the following distributed algorithm problem for updating the accumulator digest: There are m CPU agents, where each CPU i holds the same accumulator digest w, memory cell index ℓ_i, write value val_i, and an authentication path π_i for ℓ_i (received from the evaluation algorithm) as its inputs, with the goal of computing the updated accumulator digest w' with respect to write instructions $\{(\ell_i, val_i)\}_{i\in[m]}$. We need a distributed algorithm to solve this problem with oblivious communication pattern, $\mathsf{poly}\log(m)$ rounds, and per-CPU space complexity $\mathsf{poly}\log(m)$, where oblivious communication requires the sender and receiver of any message be public information and independent from the input. With oblivious communication, a message can be signed and verified with a fixed pair of signature keys, which authenticates this message just by signatures to CPU states. This is a non-trivial problem, but not too difficult. Our solution (of using parallel accumulator) is to rely on an oblivious update protocol with desired complexity, which is based on existing oblivious aggregation and oblivious multicasting protocols [BCP16]. For more details, we will introduce the parallel accumulator in Section 6.4.

With the above update protocol, each CPU can concurrently obtain the correct accumulator digest, and then we have a construction that emulates the PRAM execution with $\mathsf{poly}\log(m)$ overhead in both CPU program size and parallel time complexity. Then, it seems that we can directly generalize the proof techniques of $\mathsf{Ci}\mathcal{O}$-RAM to prove the security of $\mathsf{Ci}\mathcal{O}$-PRAM. However, if we were to prove the security of the above construction, in the proof, we will still need to hardwire *all* m CPU states (at some time step t) in some intermediate hybrids. As a result, the obfuscated program must be padded to size $\Omega(m)$, which leads to inefficient constructions because each step takes time $\Omega(m)$ and the total parallel time is multiplied by a factor of $\Omega(m)$. The reason why hardwiring $\Omega(m)$ information in the program is necessary is illustrated in the following "pebble game".

Pebble Game Illustration A pebble game is a type of mathematical game played by moving pebbles on a directed graph. The pebble game that involves placing pebbles on the nodes of a directed acyclic graph $\mathbf{DAG}$ according to certain rules which are given as follows.

- A step of the game is either placing a pebble on an empty node of $\mathbf{DAG}$ or removing a pebble from a previously pebbled node.
- A pebble can be added to node v only if either (1) all its predecessors u, such that $u \to v$, have pebbles, or (2) v is a source node.
- The pebble on a node v can be removed only if all its successors u', such that $v \longrightarrow u'$, had been traversed by some other pebbles.
- The winning condition of the game is to successively put pebbles at each node to traverse the whole graph (in any order) while minimizing the number of pebbles that are ever on the graph simultaneously.

We rely on an (oversimplified) interpretation of the security proof as a pebble game, to show why hardwiring m CPU states is necessary. Firstly, we show that the pebble game illustration is applicable to the KLW proof techniques in proving the security of $\mathsf{Ci}\mathcal{O}$-RAM, where a formal proof has already presented in Appendix B.1. Conceptually, we can interpret each time step t as a node and the connection of two time steps as an edge. The computation will be transformed into a line path, and thus the KLW proof techniques ($\mathbf{Hyb}_{0,2,i}$ and $\mathbf{Hyb}_{0,2',i}$) can be viewed as a way to put and move a *check-point* (pebble) on the nodes. When the pebble is placed on a node t, we obtain an information theoretic guarantee for the correctness of the input/output of the t-th step computation, such that the input/output values from an honest evaluation must pass the verification. This allows us to locally change the program code at time step t, so that the pebble can be moved forward or backward along the computation path. Very interestingly, although the computation path is long, moving a pebble corresponds to hardwiring only $O(1)$ information in the hybrids, independent of the time bound T. In Appendix B.1, the security proof of $\mathsf{Ci}\mathcal{O}$-RAM has shown how to replace program P_0 to P_1 gradually in this way.

However, in the PRAM setting, the computation trace becomes a directed layered graph with width m, where each layer corresponds to a time step and

- each node is indexed by a time t and a CPU id i,
- there is a directed edge (u, v) if node u outputs either a CPU state or a communication message as input to node v.

Note that an edge only exists between the nodes in *neighboring* layers, which means that an edge denotes some dependency between the two nodes in the actual computation. In particular, nodes u and v can be either the same CPU depends on its previous state or different CPUs such that v receives a message from u.[19]

Suppose that we generalize the proof techniques of KLW to this setting in a straightforward manner, which is analogous to putting and moving check-points along the graph. By moving check-points, we introduce the check condition which checks the output m^{out} of a computation step u against the hardwired value m^u. The general pebble game rules bring a constraint that we can only put one pebble on a node v if, for every u such that $(u, v) \in E$, a pebble is put on each node u. This is analogous to that adding another hardwired value m^v must depend on all its input. As such, we can only remove a check-point after we have put check-points on all its outgoing neighbors, which is analogous to the hybrid removing the hardwired signing key and CPU output state m^u. In general, placing (or removing) a pebble on (or from) node u is analogous to hardwiring (or un-hardwiring) the output of computation step u in an intermediate hybrid program. Recall that our goal of the security proof is to traverse all computation steps, and to locally change the program while minimizing the total number of hardwired values for any intermediate hybrid. Accordingly, in the corresponding pebble game abstraction, the goal is to traverse the whole graph while minimizing the total number of pebbles (a.k.a. *pebble complexity*) on the graph.

Using the pebble game abstraction, the graph of $\mathsf{OUpdate}$ is partially depicted in Figure 1. To win this game, the straightforward solution is to put m pebbles on all m CPUs of the first column, and then placing pebbles to the next column while removing from the first greedily whenever possible. We can remove all pebbles on the first column when its next is filled with pebbles, since any column only depends on its previous one. The pebble complexity is at most $2m$. Unfortunately, we have not found a solution yet to winning this pebble game with pebble complexity less than the number of CPUs m. The straightforward solution implies an inefficient $\mathsf{Ci}\mathcal{O}$ scheme with encoding of size overhead $\Omega(m)$. However, we provide a generic transformation that converts any PRAM program with oblivious communication to a special class of PRAM program with $\log m$ pebble complexity (with $\log m$ parallel time overhead in addition). Such transformation allows us to construct efficient $\mathsf{Ci}\mathcal{O}$-PRAM programs.

Branch and Combine Transformation Here we very briefly describe the transformation. It is motivated by the observation that a hardwired value (and thus a pebble) is corresponding to the signature signing and verifying a CPU state or communication, and in a sense we can replace such verification with an additional accumulator structure. Furthermore, computing the digest of the CPU accumulator is much simpler than that of memory accumulator, since each CPU computes directly with its fixed neighbor without additional communication. We divide the transformation into two stages: *branch* and *combine*.

At the beginning, all m input CPU states and messages are stored in a random access buffer buff, and then the evaluator is required to provide each state st_i and message com_i with a proof π_i to each CPU. All CPUs receive as input the same signed accumulator digest w_{buff} of the buff, their corresponding states and messages with proofs. Each CPU then verifies accumulator digest with signature, verifies state and message with accumulator digest and proof, computes its output state and message, and signs the output with the signature scheme. In this *branch* stage, these procedures have one signature as input but m different signatures as output.

Now the evaluator has m new signed states and messages, but the CPU steps at the next branch stage require a new signed accumulator digest w'_{buff}. Our next goal is to design a series of CPU steps to compute such digest and its signature, which should be verifiable at the next stage by the CPUs but unforgeable by the

[19]We assume here we have the protocol to update the memory accumulator digest (Section 6.4) in the parallel setting. Although memory accesses lead to dependencies between two steps, they are verified by memory accumulators, and thus are not hardwired in hybrid programs. Therefore we do not consider memory accesses in this pebble game.

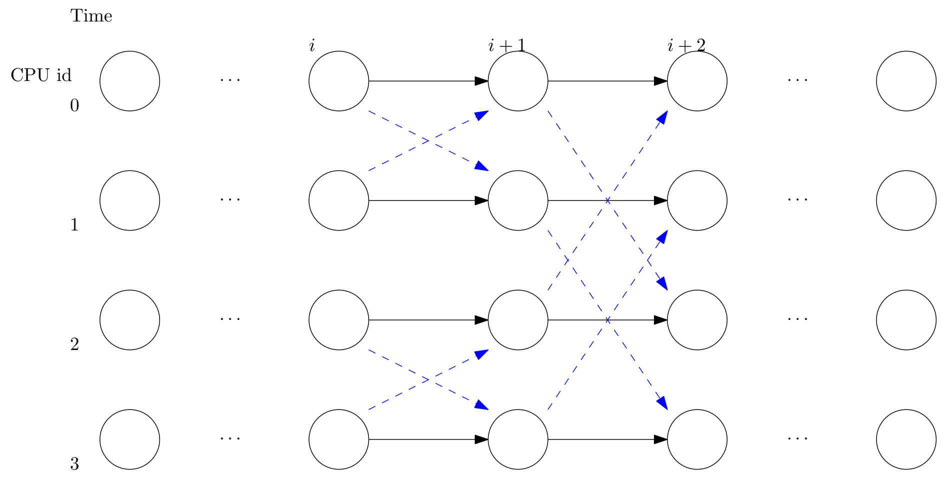

Figure 1: A graph showing partially the computation dependencies of $\mathsf{OUpdate}$ with 4 CPUs, which needs 4+2 pebbles to traverse: Solid black arrows denote CPU state transitions, and dashed blue arrows denote communications between two CPUs.

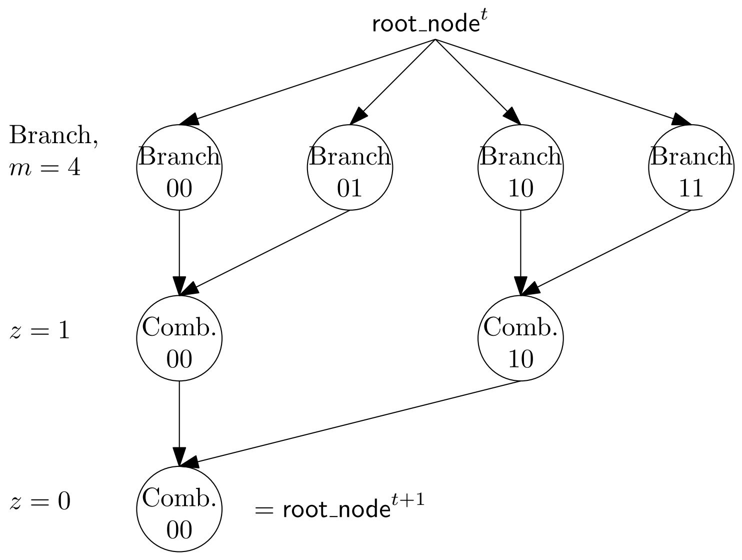

Figure 2: Illustrating an evaluation of “branch and combine” program: each node is a step computation of either F_{branch} or F_{combine} with its CPU id denoted by a binary string, each arrow denotes dispatching an output to the input of another node; dummy computations are omitted, and dispatching is performed in the evaluator.

evaluator. Intuitively, we construct a CPU program following the Merkle-tree-like structure, which takes a pair of signed inputs and outputs one signed output in each step (Figure 2). From a leaf to the root, the program indeed computes the whole tree structure of $buff'$ including the tree root, which is exactly the accumulator digest w'_{buff}, and each step has output signed with a signature. Now that we have this new signed accumulator digest, the evaluator can start the next branch stage iteratively. In this *combine* stage, there are m leaf signatures at the beginning but only one root signature as the final output.

To analyze our *branch and combine* technique, we observe that those states and communications in the original program are transformed into $buff$, and dependencies are simple. In the pebble game abstraction, it is easy to traverse from a root in a combine stage to the next root via post order, where its pebble complexity is $\log m$. Specifically, our strategy is to merge two sibling pebbles into their parent node as soon as possible. There is an invariant that each layer in the binary tree has at most 2 pebbles. Multiplying $\log m$ overhead of branch and combine transformation with $\log m$ pebble complexity, the overall overhead is $O(\log^2 m)$ in parallel time. As a result, it significantly improves the naïve solution. Also note that the evaluation of the branch and combine program is parallel, but the pebble game (and thus the series of hybrids) is sequential because it is not possible to traverse Tm nodes with only $\log m$ pebbles in T moves. We will describe branch-and-combine construction in the simpler mPRAM model in Section 6.5, and then prove its security in Appendix B.3.

6.2 Building Blocks

In our Ci$\mathcal{O}$ construction in Section 6.5, we will use several building blocks: accumulator, topological iterator, splittable signature, puncturable PRF, and indistinguishability obfuscation. Topological iterator is a new building block and we will investigate it in detail in next subsection. The formal definitions for other building blocks can be found in Appendix A. We next define the parameters for the building blocks we will use in our Ci$\mathcal{O}$ construction.

- Accumulator scheme $\mathsf{Acc} = \mathsf{Acc}.\{\mathsf{Setup}, \mathsf{SetupEnforceRead}, \mathsf{SetupEnforceWrite}, \mathsf{PrepRead}, \mathsf{PrepWrite}, \mathsf{VerifyRead}, \mathsf{WriteStore}, \mathsf{Update}\}$ with message space $\{0,1\}^{\ell_{\mathsf{msg}}}$ and accumulated value space $\{0,1\}^{\ell_{\mathsf{Acc}}}$.
- Topological Iterator scheme $\mathsf{TItr} = \mathsf{TItr}.\{\mathsf{Setup}, \mathsf{SetupEnforceIterate}, \mathsf{Iterate}\}$ with message space $\{0,1\}^{\ell_{\mathsf{Acc}}+\ell_{\mathsf{msg}}}$ and iterated value space $\{0,1\}^{\ell_{\mathsf{Itr}}}$ bits. (Section 6.3)
- Splittable signature scheme $\mathsf{Spl} = \mathsf{Spl}.\{\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify}, \mathsf{Split}, \mathsf{AboSign}\}$ with message space $\{0,1\}^{\ell_{\mathsf{Itr}}+\ell_{\mathsf{Acc}}+\ell_{\mathsf{msg}}}$; We will assume $\mathsf{Spl.Setup}$ uses ℓ_{rnd} bits of randomness.
- Puncturable PRF scheme $\mathsf{PPRF} = \mathsf{PPRF}.\{\mathsf{Setup}, \mathsf{Puncture}, \mathsf{Eval}\}$ with key space $\mathcal{K}$, punctured key space $\mathcal{K}_{\mathsf{punct}}$, domain $[T]$, and range $\{0,1\}^{\ell_{\mathsf{rnd}}}$.
- Indistinguishability obfuscation scheme i$\mathcal{O}$ for circuits.
- Parallel accumulator scheme (Section 6.4).

6.3 Topological Iterators

In this section, we will define a primitive named *Topological Iterators* based on the original KLW iterator (Appendix A.3.1). We will demonstrate a construction for this new primitive by using $\mathcal{PKE}$, puncturable PRF and i$\mathcal{O}$, and then prove its security. The main difference between our iterators and KLW iterators is that, we allow iterating two states, instead of one, into a new state.

Syntax Let poly be any polynomial. An iterator TItr with message space $\mathcal{M}_\lambda = \{0,1\}^{\mathsf{poly}(\lambda)}$ and state space $\mathcal{S}_\lambda$ consists of four algorithms - $\mathsf{TItr.Setup}$, $\mathsf{TItr.SetupEnf}$, $\mathsf{TItr.Iterate}$, and $\mathsf{TItr.Iterate2to1}$ defined below.

- $\mathsf{TItr.Setup}(1^\lambda, N)$: The setup algorithm takes as input the security parameter λ (in unary), and an integer bound N (in binary) on the number of iterations. It outputs public parameters $\mathsf{pp}_{\mathsf{Itr}}$ and an initial state $v \in \mathcal{S}_\lambda$.
- $\mathsf{TItr.SetupEnf}(1^\lambda, N, \mathbf{DAG})$: The enforced setup algorithm takes as input the security parameter λ (in unary), an integer bound T (in binary), and a directed acyclic graph $\mathbf{DAG}$ which has following properties:

1. $\mathbf{DAG} = (\mathcal{N}, \mathcal{E}, \mathsf{source}, \mathsf{sink})$ is a directed acyclic graph with $|\mathcal{N}| < N$;
2. Single "source" node and single "sink" node;
3. Each node (except source) has in-degree 1 or 2;
4. Each node n has a *unique* message value $m_n \in \mathcal{M}_\lambda$. This is important for the "enforcing" property.

It outputs public parameters $\mathsf{pp}_{\mathsf{Itr}}$ and an initial state $v \in \mathcal{S}_\lambda$.

– $\mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v, m)$: The iterate algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Itr}}$, a state v, and a message $m \in \mathcal{M}_\lambda$. It outputs a state $v_{\text{out}} \in \mathcal{S}_\lambda$.
– $\mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, v_l, v_r, m)$: The iterate algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Itr}}$, two states (v_l, v_r), and a *unique* message $m \in \mathcal{M}_\lambda$. It outputs a state $v_{\text{out}} \in \mathcal{S}_\lambda$.

Security Let $\mathsf{TItr} = \mathsf{TItr}.\{\mathsf{Setup}, \mathsf{SetupEnforceIterate}, \mathsf{Iterate}, \mathsf{Iterate2to1}\}$ be an iterator with message space $\mathcal{M}_\lambda$ and state space $\mathcal{S}_\lambda$. We require the following notions of security.

Definition 6.1 (Indistinguishability of Setup). *An iterator* TItr *is said to satisfy indistinguishability of Setup phase if any PPT adversary $\mathcal{A}$'s advantage in the security game* ***Exp-Setup-Itr***$(1^\lambda, \mathsf{TItr}, \mathcal{A})$ *is at most negligible in λ, where* ***Exp-Setup-Itr*** *is defined as follows.*

Exp-Setup-Itr$(1^\lambda, \mathsf{TItr}, \mathcal{A})$
– *The adversary $\mathcal{A}$ chooses a bound $N \in \Theta(2^\lambda)$ and sends it to challenger.*
– *$\mathcal{A}$ sends $\mathbf{DAG} = (\mathcal{N}, \mathcal{E}, \mathsf{source}, \mathsf{sink})$ to the challenger.*
– *The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathsf{pp}_{\mathsf{Itr}}, v) \leftarrow \mathsf{TItr.Setup}(1^\lambda, N)$. Else, it outputs $(\mathsf{pp}_{\mathsf{Itr}}, v) \leftarrow \mathsf{TItr.SetupEnf}(1^\lambda, N, \mathbf{DAG})$.*
– *$\mathcal{A}$ sends a bit b'.*
$\mathcal{A}$ wins the security game if $b = b'$.

Definition 6.2 (Enforcing). *Consider any $\lambda \in \mathbb{N}, N \in \Theta(2^\lambda), \mathbf{DAG} = (\mathcal{N}, \mathcal{E}, \mathsf{source}, \mathsf{sink})$. Let $(\mathsf{pp}_{\mathsf{Itr}}, v) \leftarrow \mathsf{TItr.SetupEnf}(1^\lambda, N, \mathbf{DAG})$ and $v_n = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, v_{(l,n)}, v_{(r,n)}, m_n)$ for all $n \in \mathcal{N} \setminus \{\mathsf{source}\}$. $\mathsf{TItr} = (\mathsf{TItr.Setup}, \mathsf{TItr.SetupEnf}, \mathsf{TItr.Iterate2to1})$ is said to be enforcing if*

$$v_{\mathsf{sink}} = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, v_l, v_r, m) \Rightarrow (v_l, v_r, m) = (v_{l,\mathsf{sink}}, v_{r,\mathsf{sink}}, m_{\mathsf{sink}}).$$

Note that this enforcing property is an information-theoretic property.

6.3.1 Construction

– $\mathsf{TItr.Setup}(1^\lambda, T)$: The setup algorithm chooses $(PK, SK) \leftarrow \mathcal{PKE}.\mathsf{Gen}(1^\lambda)$ and puncturable PRF key $K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$. It sets $\mathsf{pp}_{\mathsf{Itr}} \leftarrow i\mathcal{O}(\mathsf{prog}\{K, PK\})$, where prog is defined in Algorithm 3. Let $\mathsf{ct} \leftarrow \mathcal{PKE}.\mathsf{Encrypt}(PK, 0)$. The initial state $v = \mathsf{ct}$. It outputs $(\mathsf{pp}_{\mathsf{Itr}}, v)$.

Algorithm 3: prog

```
  Input : $v_l, v_r, m \in \mathcal{M}_\lambda$
  Data  : Puncturable PRF key $K$, $\mathcal{PKE}$ public key $PK$
1 Compute $r \leftarrow \mathsf{PRF}(K, (v_l, v_r, m))$;
2 Let $\mathsf{ct} \leftarrow \mathcal{PKE}.\mathsf{Encrypt}(PK, 0; r)$;
3 Output $v_{\text{out}} = \mathsf{ct}$;
```

– $\mathsf{TItr.SetupEnf}(1^\lambda, T, \mathbf{DAG})$: The setup algorithm chooses $(PK, SK) \leftarrow \mathcal{PKE}.\mathsf{Gen}(1^\lambda)$ and puncturable PRF key $K \leftarrow \mathsf{PRF.Setup}(1^\lambda)$. Let $\mathbf{DAG} = (\mathcal{N}, \mathcal{E}, \mathsf{source}, \mathsf{sink})$. $\forall n \in \mathcal{N}$, compute v_n as follows.

1. $\forall n \in \mathcal{N}, v_n \leftarrow \perp$.
2. The initial state $v_{\mathsf{source}} = \mathcal{PKE}.\mathsf{Encrypt}(PK, 0)$.

3. $\forall n \in \mathcal{N}$ with definite in-edge, compute v_n. That is,

$$\forall n \in \mathcal{N} \text{ s.t. } (v_n = \bot) \wedge (\exists (l,n),(r,n) \in \mathcal{E}) \wedge (v_l \neq \bot) \wedge (v_r \neq \bot),$$

$r_n \leftarrow \mathsf{PRF}(K,(v_l,v_r,m_n))$, and $v_n \leftarrow \mathcal{PKE}.\mathsf{Encrypt}(PK,0;r_n)$.

4. Repeat previous step until $\forall n \in \mathcal{N}, v_n \neq \bot$.

Let $(l_{\mathsf{sink}},\mathsf{sink}),(r_{\mathsf{sink}},\mathsf{sink}) \in \mathcal{E}$ be the two edges directed to sink. It computes a punctured key $K' \leftarrow \mathsf{PRF.Puncture}(K,(v_{l,\mathsf{sink}},v_{r,\mathsf{sink}},m_{\mathsf{sink}}))$, chooses $r \leftarrow \{0,1\}^r$ and sets $\mathsf{ct}_{\mathsf{sink}} = \mathcal{PKE}.\mathsf{Encrypt}(PK,1;r)$. Finally, for progEnforce which is defined in Algorithm 4, it computes the public parameters $\mathsf{pp}_{\mathsf{Itr}} \leftarrow i\mathcal{O}(\mathsf{progEnforce}\{\mathsf{sink},(v_{l,\mathsf{sink}},v_{r,\mathsf{sink}},m_{\mathsf{sink}}),\mathsf{ct}_{\mathsf{sink}},K',PK\})$. It outputs $(\mathsf{pp}_{\mathsf{Itr}},v_{\mathsf{source}})$.

Algorithm 4: progEnforce

```
Input : $v_l, v_r, m \in \mathcal{M}_\lambda$
Data  : sink, states $(v_{l,\mathsf{sink}}, v_{r,\mathsf{sink}})$, message $m_{\mathsf{sink}}$, $\mathsf{ct}_{\mathsf{sink}}$, puncturable PRF key $K'$, $\mathcal{PKE}$ public key
        $PK$
1 if $(v_l, v_r, m) = (v_{l,\mathsf{sink}}, v_{r,\mathsf{sink}}, m_{\mathsf{sink}})$ then
2     Output $\mathsf{ct}_{\mathsf{sink}}$;
3 Compute $r \leftarrow \mathsf{PRF}(K,(v_l,v_r,m))$;
4 Let ct $\leftarrow \mathcal{PKE}.\mathsf{Encrypt}(PK,0;r)$;
5 Output $v_{\text{out}} = $ ct;
```

– $\mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}},v,m)$: simply outputs $\mathsf{pp}_{\mathsf{Itr}}(v,\bot,m)$.
– $\mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}},v_l,v_r,m)$: simply outputs $\mathsf{pp}_{\mathsf{Itr}}(v_l,v_r,m)$.

6.3.2 Security

We show that the construction described in Section 6.3.1 satisfies indistinguishability of setup (Definition 6.1) and enforcing property (Definition 6.2).

Lemma 6.3 (Indistinguishability of Setup)**.** *Assuming* $i\mathcal{O}$ *is a secure indistinguishable obfuscator,* PRF *is a selectively secure puncturable PRF, and* $\mathcal{PKE}$ *is a semantically-secure public key encryption scheme, any PPT adversary* $\mathcal{A}$ *has only negligible advantage in the* ***Exp-Setup-Itr*** *game.*

The proof can be found in Appendix B.2.

Lemma 6.4 (Enforcing)**.** *Assuming* $\mathcal{PKE}$ *is a perfectly correct public key encryption scheme, then* $\mathsf{TItr} = (\mathsf{TItr.Setup}, \mathsf{TItr.SetupEnf}, \mathsf{TItr.Iterate}, \mathsf{TItr.Iterate2to1})$ *is enforcing.*

Proof. This follows directly from the correctness of $\mathcal{PKE}$ because the sink node has v_{sink} being an encryption of 1 but all other nodes are encryption of 0. □

6.4 Parallel Accumulator

Consider the (standard) PRAM model in which m CPUs have random access to a shared memory at m locations at each time step. An intuitive approach is to extend the construction of $\mathsf{C}i\mathcal{O}$-RAM into the PRAM model, where each bit read by each CPU is verified against the accumulator value (a.k.a. digest) w. The verification is straightforward and can be run in parallel. The problem, however, is that the digest w must be updated correctly after each bit written by each CPU (in order to verify the next bit to be read). Therefore, suppose all m CPUs write in a time step, the new digest w depends on all m newly written bits, which may take roughly m (total) steps to update w. The trivial solution which sequentially updates w with each m bits obviously introduces an unacceptable $\Omega(m)$ multiplicative parallel time overhead. To retain the benefits of using m CPUs, we must design a clever update mechanism with at most $O(\log m)$ overhead.

Recall that by our assumption the m CPUs read and write synchronously and alternatively. Consider a time step when m synchronized writes occur. Our goal here is to let each CPU, with their own bits to be written, exchange information with each others and concurrently compute the same digest w. Ignoring the steps where the bits are actually written to the memory, the procedures for computing the digest alone form a memoryless PRAM computation, which can be obfuscated using techniques in the construction of $\mathsf{Ci}\mathcal{O}$ for memoryless PRAM.

To argue the security of this approach, we observe that the whole accumulator tree structure is accessible to the malicious evaluator/adversary, and thus it is not necessary to hide any intermediates while updating. In fact, the CPUs compute and communicate with their sequences forming a binary tree bottom-up, and leak a partially updated memory to the decoder at each level of the tree. However, any partially updated memory does not give the adversarial decoder additional information because it can always compute these values from the previous memory contents and the newly written bits. For correctness, as long as the CPU states and messages are authenticated by $\mathsf{Ci}\mathcal{O}$, any adversary is not able to forge a malicious message or digest, and hence the CPUs will eventually agree on the same digest w correctly.

The construction of $\mathsf{Ci}\mathcal{O}$ for memoryless PRAM requires the communication pattern between CPUs to be oblivious, which means the receiver of each communication is uniquely determined by the iteration counter t and the sender CPU id. In the following Section 6.4.1, we recall the oblivious aggregation and oblivious multicasting protocols from [BCP16] to build oblivious communication between the CPUs. We then use these protocols to construct the mechanism for computing the digest w in Section 6.4.2.

6.4.1 Oblivious Aggregation and Multicasting

To simplify the "updating" algorithm, we apply two PRAM primitives, namely *oblivious aggregation* and *oblivious multicasting*, which are introduced in [BCP16] and described below for completeness. Both primitive are deterministic PRAM algorithms that run with only CPU states and oblivious communication. By oblivious communication, we mean that the source and destination of all messages are specified by algorithm but not dependent on the input. The functionality and complexity are specified in the following definitions.

Oblivious Aggregation $\mathsf{OblivAgg}$ is a procedure satisfying the following aggregation goal with communication patterns independent of the input, using only $\tilde{O}(\mathsf{poly}\log(m))$ local memory and communication per CPU, in only $\tilde{O}(\mathsf{poly}\log(m))$ sequential time steps.

Input: Each CPU $i \in [m]$ holds $(\mathsf{key}_i, data_i)$. Let $\mathbb{K} = \bigcup\{\mathsf{key}_i\}$ be the set of distinct keys. We assume that any (subset of) data associated with the same key can be aggregated by an aggregation function Agg to a short digest of size at most $\mathsf{poly}(|data_i|, \log m)$.

Goal: Each CPU i outputs out_i such that the following holds.

- for every $\mathsf{key} \in \mathbb{K}$, there exists unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $\mathsf{out}_i = (\mathsf{rep}, \mathsf{key}, \mathsf{agg}_{\mathsf{key}})$, where $\mathsf{agg}_{\mathsf{key}} = \mathsf{Agg}(\{data_j : \mathsf{key}_j = \mathsf{key}\})$.
- for every remaining agent i, $\mathsf{out}_i = (\mathtt{dummy}, \perp, \perp)$.

Oblivious Multicasting $\mathsf{OblivMCast}$ is a procedure satisfying the following multicasting goal with communication patterns independent of the inputs, using only $\tilde{O}(\mathsf{poly}\log(m))$ local memory and communication per CPU, in only $\tilde{O}(\mathsf{poly}\log(m))$ sequential time steps. Namely, a subset of CPUs must deliver information to (unknown) collections of other CPUs who request it. This is abstractly modelled as follows, where key_i denotes which data item is requested by each CPU i.

Input: Each CPU i holds $(\mathsf{key}_i, data_i)$ with the following promise. Let $\mathbb{K} = \bigcup\{\mathsf{key}_i\}$ be the set of distinct keys. For every $\mathsf{key} \in \mathbb{K}$, there exists a unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $data_i \neq \perp$; let $data_{\mathsf{key}}$ be such $data_i$.

Goal: Each agent i outputs $\mathsf{out}_i = (\mathsf{key}_i, data_{\mathsf{key}_i})$.

6.4.2 OUpdate Algorithm

In the following, we construct a distributed Acc.OUpdate algorithm (Algorithm 5) to be run with m CPUs. Each CPU i takes i, loc_i, b_i, π_i as input, communicates with other CPUs obliviously using OblivAgg and OblivMCast defined above, and outputs the same digest w of the whole updated memory, where for CPU i

- loc_i, b_i are the writing location and value, and
- π_i is the authentication path (proof) of the writing location loc_i.

Acc.OUpdate introduces a multiplicative $O(\log S)$ parallel time overhead in the worst case where all CPUs write in parallel m different values at distinct locations of the size-S memory. Acc.OUpdate traverses each authentication path π_i in a bottom-up manner. For each node in π_i, it checks if another CPU possesses a fresh sibling node by the OblivAgg protocol, which exchanges information between all CPUs and yields a pair of fresh nodes (n_l, n_r) that may depend on one or more CPUs, to the representative CPU (Algorithm 6). All CPUs then share the pair of fresh nodes by OblivMCast from those representative CPUs, and each CPU computes the updated parent node of the updated pair directly by the public parameter $\mathsf{pp}_{\mathsf{Acc}}$. With the updated parent node, the procedure is able to continue with the next node in π_i iteratively. The procedure is finished when the root node of π_i is updated.

Some additional notations used in Acc.OUpdate are listed in Table 2.

Notations	
z	The depth of node to be updated in π
loc_x	The x bit prefix of loc
$\mathsf{loc}[x]$	The x-th bit of loc ($\mathsf{loc}[-1] := \epsilon$)
$\mathsf{loc}[x] = 0$ ($\mathsf{loc}[x] = 1$)	Implies that the left (right) node should be updated
$\pi[x]$	The pair of nodes $(n_l, n_r)_x$ at depth x such that $\pi[-1] := root$
$\pi[x][0]$ ($\pi[x][1]$)	The left node n_l (right node n_r) in the pair $(n_l, n_r)_x$ such that $\pi[-1][\epsilon] := root$
flag $=$ `Done`	Implies that both nodes (n_l, n_r) are fresh in aggdata

Table 2: Additional notations for the OUpdate protocol: We omit subscription i and use loc, b and π in OUpdate (Algorithm 5).

Algorithm 5: Acc.OUpdate

```
   Input  : $i, \mathsf{loc}, b, \pi$
   Output : $w$
   Data   : $\mathsf{pp}_{\mathsf{Acc,mem}}$
 1 Parse $\pi = (root, (n_l, n_r)_0, \ldots, (n_l, n_r)_{MemAccDepth-1})$;
 2 Let the leaf node $\pi[MemAccDepth-1][\mathsf{loc}[MemAccDepth-1]] \leftarrow b$;
 3 for $z \leftarrow MemAccDepth$ to 0 do
 4     $\mathsf{key}_i \leftarrow \mathsf{loc}_{z-1}$;
 5     $(\mathsf{rep}, \mathsf{key}_i, \mathsf{aggdata}) \leftarrow$ Run OblivAgg with input $(i, \mathsf{key}_i, (\mathsf{loc}[z], \pi[z][0], \pi[z][1]))$ and aggregation
       function Agg;
 6     $(flag, n_l, n_r) \leftarrow$ Run OblivMCast with input $(\mathsf{key}_i, \mathsf{aggdata})$;
 7     $\pi[z-1][\mathsf{loc}[z-1]] \leftarrow \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,mem}}, n_l, n_r, \mathsf{loc}_{z-1})$;
 8 return $root$;
```

6.4.3 Notation of OUpdate Compiler

In this sub-section, we describe the notations used in describing how to compile a PRAM computation into a *memory checking* PRAM computation with protocol OUpdate described above. Without loss of generality, we assume that there is a step function F_{OUpdate} which runs the protocol OUpdate.

Algorithm 6: Agg

```
Input  : $\mathsf{datatemp}_1, \mathsf{datatemp}_2$
Output : aggdata
1  if $\mathsf{datatemp}_1$ has the form $(\texttt{Done}, n_l, n_r)$ then
2      // If any data is already done, return it.
3      return $\mathsf{datatemp}_1$;
4  if $\mathsf{datatemp}_2$ has the form $(\texttt{Done}, n_l, n_r)$ then
5      return $\mathsf{datatemp}_2$;
6  if $\mathsf{datatemp}_1 = \mathsf{datatemp}_2$ has the form $(\mathsf{loc}[z], n_l, n_r)$ then
7      // If both data are identical, keep any one.
8      return $\mathsf{datatemp}_1$;
9  if $\mathsf{datatemp}_1$ has the form $(\mathsf{loc}[z]_1, n_{l,1}, n_{r,1})$ and $\mathsf{datatemp}_2$ has the form $(\mathsf{loc}[z]_2, n_{l,2}, n_{r,2})$ then
10     // If two data differs, merge by their freshness and mark as done.
11     if $(\mathsf{loc}[z]_1 > \mathsf{loc}[z]_2)$ then
12         Swap $(\mathsf{loc}[z]_1, n_{l,1}, n_{r,1}), (\mathsf{loc}[z]_2, n_{l,2}, n_{r,2})$;
13     return $(\texttt{Done}, n_{l,1}, n_{r,2})$;
```

Given a PRAM computation system Π with synchronous and alternative READ and WRITE,

$$\Pi = \Big((\mathsf{mem}^0, \{\mathsf{st}_k^0, a^0_{k\leftarrow\mathtt{M}}, a^0_{\mathtt{M}\leftarrow k}\}_{k=1}^m), F \Big),$$

and step function F_{OUpdate} described above, we define $\Pi_{\mathsf{check}} = \mathsf{AccCompile}(\Pi, \mathsf{Acc.OUpdate}\{\mathsf{pp}_{\mathsf{Acc,mem}}\})$, which verifies data read from memory with accumulator and use $\mathsf{Acc.OUpdate}$ to compute a new accumulator digest. By the assumption of synchronous and alternative READ and WRITE, we simply assume F always reads at even rounds and writes at odd rounds, and thus Π_{check} has a straightforward construction: repeatedly invokes (i) step function F one reading round, (ii) step function F one writing round, (iii) step function F_{OUpdate} a fixed number D_{Acc} of rounds. Specifically,

$$\Pi_{\mathsf{check}} = \Big((\mathsf{mem}^0, \{\check{\mathsf{st}}_k^0, \check{a}^0_{k\leftarrow\mathtt{M}}, \check{a}^0_{\mathtt{M}\leftarrow k}\}_{k=1}^m), F_{\mathsf{check}} \Big),$$

where F_{check} is defined in Algorithm 7, $\check{\mathsf{st}}$ and $\check{a}$ are defined by augmenting the corresponding st and a from Π. We can use $F_{\mathsf{check}} = \mathsf{AccCompile}(F, \mathsf{Acc.OUpdate}\{\mathsf{pp}_{\mathsf{Acc,mem}}\})$ to specify the compilation.

F_{check} has three major stages, which are READ, WRITE, and OUpdate.

- In a READ state, it has no memory input and just invokes F, which issues a READ command.
- In a WRITE state, the previous state must be READ. Therefore, F_{check} verifies the value read from memory and invokes F, which issues a WRITE command.
- In OUpdate states, F_{check} first verifies the proof π^{in} against old accumulator digest. Then, it initializes F_{OUpdate} with the correct proof π^{in} and runs F_{OUpdate} stepwise to obtain the new accumulator digest.

These stages are controlled by the simple counter d_{Acc} and the fixed running time D_{Acc}, which is defined implicitly by OUpdate.

6.5 Warm-up: Construction for Ci$\mathcal{O}$-mPRAM

As a warm-up, we construct Ci$\mathcal{O}$ in the memoryless PRAM (mPRAM) model. Recall in Appendix A.1.3 that the mPRAM model is similar to the PRAM model except that it has no external memory, but oblivious communication between pairs of CPUs is allowed. Formally, the class of distributed computation for mPRAM, denoted by $\mathcal{P}_{\mathrm{mPRAM}}$, is defined as follows:

Definition 6.5 (mPRAM Computation Class). *$\mathcal{P}_{\mathrm{mPRAM}}$ is a class of distributed computation for mPRAM with m agents without external memory where*

Algorithm 7: F_{check}

```
Input   : $id$, $\check{\mathsf{st}}^{\mathrm{in}}, \check{a}^{\mathrm{in}} = (b^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, \pi^{\mathrm{in}})$
Data    : $D_{\mathsf{Acc}}$
1  Parse $\check{\mathsf{st}}^{\mathrm{in}}$ as $((\mathsf{st}_{\Pi}^{\mathrm{in}}, \mathsf{st}_{\mathsf{Acc}}^{\mathrm{in}}), d_{\mathsf{Acc}}, b, w, \mathsf{loc})$;
2  if $(d_{\mathsf{Acc}} = \mathtt{Read}$ or $d_{\mathsf{Acc}} = \mathtt{Write})$ then
3      $(\mathsf{st}_{\mathsf{Acc}}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow (\bot, \bot)$;
4      if $d_{\mathsf{Acc}} = \mathtt{Read}$ then
5          Compute $(\mathsf{st}_{\Pi}^{\mathrm{out}}, (\mathsf{loc}^{\mathrm{out}}, b^{\mathrm{out}})) \leftarrow F(id, \mathsf{st}_{\Pi}^{\mathrm{in}}, b^{\mathrm{in}})$;
6          if $\mathsf{st}_{\Pi}^{\mathrm{out}} = \mathtt{Reject}$ then output $\mathtt{Reject}$;
7          Set $\check{\mathsf{st}}^{\mathrm{out}} = ((\mathsf{st}_{\Pi}^{\mathrm{out}}, \mathsf{st}_{\mathsf{Acc}}^{\mathrm{out}}), \mathtt{Write}, b^{\mathrm{out}}, w, \mathsf{loc}^{\mathrm{out}})$;
8      else
9          if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,mem}}, w, (\mathsf{loc}, b^{\mathrm{in}}), \pi^{\mathrm{in}}) = 0$ then output $\mathtt{Reject}$;
10         Compute $(\mathsf{st}_{\Pi}^{\mathrm{out}}, (\mathsf{loc}^{\mathrm{out}}, b^{\mathrm{out}})) \leftarrow F(id, \mathsf{st}_{\Pi}^{\mathrm{in}}, b^{\mathrm{in}})$;
11         if $\mathsf{st}_{\Pi}^{\mathrm{out}} = \mathtt{Reject}$ then output $\mathtt{Reject}$;
12         Set $\check{\mathsf{st}}^{\mathrm{out}} \leftarrow ((\mathsf{st}_{\Pi}^{\mathrm{out}}, \mathsf{st}_{\mathsf{Acc}}^{\mathrm{out}}), 0, b^{\mathrm{out}}, w, \mathsf{loc}^{\mathrm{out}})$;
13     Set $\check{a}^{\mathrm{out}} \leftarrow (\mathsf{com}^{\mathrm{out}}, (\mathsf{loc}^{\mathrm{out}}, b^{\mathrm{out}}))$;
14 else
15     if $d_{\mathsf{Acc}} = 0$ then
16         if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,mem}}, w, (\mathsf{loc}, b^{\mathrm{in}}), \pi^{\mathrm{in}}) = 0$ then output $\mathtt{Reject}$;
17         Initialize $\mathsf{st}_{\mathsf{Acc}}^{\mathrm{in}}$ with $(\mathsf{loc}, b, \pi^{\mathrm{in}})$;
18     $(\mathsf{st}_{\mathsf{Acc}}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F_{\mathsf{OUpdate}}(id, \mathsf{st}_{\mathsf{Acc}}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$ ;        // Execute $F_{\mathsf{OUpdate}}$ iteratively
19     $d_{\mathsf{Acc}} \leftarrow d_{\mathsf{Acc}} + 1$;
20     if $d_{\mathsf{Acc}} = D_{\mathsf{Acc}}$ then
21         Parse $\mathsf{st}_{\mathsf{Acc}}^{\mathrm{out}}$ to obtain new accumulator digest $w^{\mathrm{out}}$;
22         $d_{\mathsf{Acc}} \leftarrow \mathtt{Read}$;
23     else
24         Let $w^{\mathrm{out}} = w$;
25     Set $\check{\mathsf{st}}^{\mathrm{out}} \leftarrow ((\mathsf{st}_{\Pi}^{\mathrm{in}}, \mathsf{st}_{\mathsf{Acc}}^{\mathrm{out}}), d_{\mathsf{Acc}}, \bot, w^{\mathrm{out}}, \bot)$;
26     Set $\check{a}^{\mathrm{out}} \leftarrow (\mathsf{com}^{\mathrm{out}}, (\bot, \bot))$;
27 return $(\check{\mathsf{st}}^{\mathrm{out}}, \check{a}^{\mathrm{out}})$;
```

- *the terminating time t^* is bounded by T;*
- *the communication between agents is restricted in the sense that in each round t, each agent k is only allowed to receive a single message c_k^t from agent $\mathsf{src}(t,k)$ where src is a public function, and the agent k is allowed to at most send one message to other agent;*
- *for all $k \in [m]$, the state size $|\mathsf{st}_k|$ is bounded by $\mathsf{poly}\log(T)$;*
- *for all $k \in [m]$, the access commands are restricted to $a^t_{\mathtt{M}\leftarrow k} := \bot$ and $a^t_{k\leftarrow\mathtt{M}} := \bot$;*
- *for all $k \in [m]$, the initial communication messages are restricted to $c_k^0 := \bot$.*

Notations	
m	The total number of CPUs in a PRAM program
node	A node contains $(t, \mathsf{index}, w_{\mathsf{st}}, w_{\mathsf{com}}, v, \sigma)$
$\mathsf{src}(\cdot,\cdot)$	A function decides an oblivious communication (e.g., at time t, $id_{\mathsf{cpu}}(b)$ sends to $id_{\mathsf{cpu}}(a)$, then $\mathsf{src}(t, id_{\mathsf{cpu}}(a)) \to id_{\mathsf{cpu}}(b)$)
$\mathsf{max\text{-}cpu}(\cdot)$	A mapping function, defined on index, outputs a leaf node: If index is a leaf, $\mathsf{max\text{-}cpu}(\mathsf{index}) = \mathsf{index}$. If not, outputs the *maximum* leaf index from index's descendants.
$\mathsf{min\text{-}cpu}(\cdot)$	A mapping function, defined on index, outputs a leaf node: If index is a leaf, $\mathsf{min\text{-}cpu}(\mathsf{index}) = \mathsf{index}$. If not, outputs the *minimum* leaf index from index's descendants.
$\mathbf{C}_{i,j}$	A set of indices of *internal* nodes defined by an index j for $t = i$. (For all $\mathsf{index} \in \mathbf{C}_{i,j}$, $\mathsf{index} < j$ and index's parent $\notin \mathbf{C}_{i,j}$)
$\mathbf{M}_{i,j}$	A set of hardwired output messages corresponding to indices of $\mathbf{C}_{i,j}$

Table 3: Additional notations for $\mathsf{Ci}\mathcal{O}$ in the mPRAM model

We now describe our scheme $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{Obf}, \mathsf{Eval}\}$ in the mPRAM model. For readability, we first introduce additional notations in Table 3. Our construction of $\mathsf{Ci}\mathcal{O}$ for mPRAM follows the structure of the construction of $\mathsf{Ci}\mathcal{O}$-RAM, except for the following differences. First, since there is no memory access in the mPRAM model, no accumulator is needed for storing the memory content. However, we do need two new accumulators to store the states and messages, as we wish to compress m states and messages into their corresponding digests respectively. In order to compute these digests, the next step function $\tilde{F}$ of the obfuscated program is split into the branch stage and the combine stage. The branch stage is essentially the first half of $\tilde{F}$ in the construction of $\mathsf{Ci}\mathcal{O}$-RAM, where the function verifies its inputs and performs the actual computation. The steps for computing the digests are deferred to the combine stage, where the m CPUs collaboratively update the digests for their states and messages.

The compilation procedure $\mathsf{Ci}\mathcal{O}.\mathsf{Obf}$ can transform a given computation system Π in the memoryless PRAM model, i.e., $\Pi \in \mathcal{P}_{\mathrm{mPRAM}}$, into an obfuscated computation system $\widetilde{\Pi}$, where

$$\Pi = (\{\mathsf{st}_k^0\}_{k=1}^m, F), \text{ and}$$

$$\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \{\widetilde{\mathsf{st}}_k^0\}_{k=1}^m), \widetilde{F}).$$

We note that for $\Pi \in \mathcal{P}_{\mathrm{mPRAM}}$, the variables mem, $a_{k\leftarrow\mathtt{M}}, a_{\mathtt{M}\leftarrow k}$ are not defined.

Compilation procedure $\widetilde{\Pi} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \Pi)$: The compilation procedure $\mathsf{Obf}()$ consists of several steps.
Step 1: Generating parameters. A set of parameters will be generated:

$$K_A \leftarrow \mathsf{PRF.Setup}(1^\lambda)$$
$$(\mathsf{pp}_{\mathsf{Acc,st}}, \hat{w}_{\mathsf{st},0}, \hat{store}_{\mathsf{st},0}) \leftarrow \mathsf{Acc.Setup}(m)$$
$$(\mathsf{pp}_{\mathsf{Acc,com}}, \hat{w}_{\mathsf{com},0}, \hat{store}_{\mathsf{com},0}) \leftarrow \mathsf{Acc.Setup}(m)$$
$$(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{TItr.Setup}(T)$$

Step 2: Generating stateful algorithms $\widetilde{F}$. Based on the parameters T, $\mathsf{pp}_{\mathsf{Acc,st}}$, $\mathsf{pp}_{\mathsf{Acc,com}}$, $\mathsf{pp}_{\mathsf{Itr}}$, K_A generated above, as well as the program F, we define the program $\widehat{F}$ in Algorithm 8. Here $\widehat{F}$ executes internal programs F_{branch} (Algorithm 9), which in turn executes F, or F_{combine} (Algorithm 10) depending on its input.

Similar to the program $\widehat{F}$ (Algorithm 1) in the construction of $\mathsf{Ci}\mathcal{O}$-RAM, F_{branch} first verifies its input, performs the actual computation of F, and authenticates its output. The communication commands of F are interpreted as access commands in the obfuscated program, and will be accumulated to the corresponding memory accumulator. The difference is that here updating the accumulator is deferred to the combine stage of $\widehat{F}$ defined in F_{combine}, and the obfuscated CPU state $\widetilde{\mathsf{st}}$ is never signed or verified by signature.

Then, m copies of F_{combine} will be executed multiple rounds so as to combine the m newly accumulated value into a common digest. At the first iteration of F_{combine}, each pair of neighboring CPUs form a group to combine their accumulated access commands into a common value in the parent node. Then, each pair of neighboring groups will form a larger group to combine their values into a common parent node. This process will continue until a common root node is reached, so that the program $\widehat{F}$ will resume to the branch stage.

The compilation procedure then computes an obfuscation of the program $\widehat{F}$. That is, $\widetilde{F} \leftarrow \mathsf{i}\mathcal{O}.\mathsf{Gen}(\widehat{F})$.

Algorithm 8: $\widehat{F}$
// for simplicity, we drop the subscripts from $\widetilde{a}^{\text{in}}_{id_{\mathsf{cpu}}\leftarrow \mathtt{M}}$ and $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow id_{\mathsf{cpu}}}$, and use $\widetilde{a}^{\text{in}}$ and $\widetilde{a}^{\text{out}}$ respectively

```
  Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\text{in}}$
1 if $\mathsf{st}^{\text{in}} = (\mathtt{halt}, \cdot)$ then
2     Output $\mathtt{Reject}$;
3 else if $\mathsf{root\_node} \neq \bot$ then
4     Compute $(\widetilde{\mathsf{st}}^{\text{out}}, \widetilde{a}^{\text{out}}) = F_{\mathsf{branch}}(\widetilde{\mathsf{st}}^{\text{in}}, \widetilde{a}^{\text{in}})$;
5 else
6     Compute $(\widetilde{\mathsf{st}}^{\text{out}}, \widetilde{a}^{\text{out}}) = F_{\mathsf{combine}}(\widetilde{\mathsf{st}}^{\text{in}}, \widetilde{a}^{\text{in}})$;
7 Output $(\widetilde{\mathsf{st}}^{\text{out}}, \widetilde{a}^{\text{out}})$;
```

Step 3: Generating the initial configuration $(\widetilde{\mathsf{mem}}^0, \{\widetilde{\mathsf{st}}_k^0\}_{k=1}^m)$. Recall that $c_k^0 = \bot$. Based on given states $\mathsf{st}_1^0, \ldots, \mathsf{st}_m^0$, the compilation procedure computes the initial configuration for the complied computation system as follows.

– For each $j \in \{1, \ldots, m\}$, it computes iteratively:

$$\pi_j \leftarrow \mathsf{Acc.PrepWrite}(\mathsf{pp}_{\mathsf{Acc,st}}, \hat{store}_{\mathsf{st},j-1}, j-1)$$
$$\hat{w}_{\mathsf{st},j} \leftarrow \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc,st}}, \hat{w}_{\mathsf{st},j-1}, j-1, \mathsf{st}_j^0, \pi_j)$$
$$\hat{store}_{\mathsf{st},j} \leftarrow \mathsf{Acc.WriteStore}(\mathsf{pp}_{\mathsf{Acc,st}}, \hat{store}_{\mathsf{st},j-1}, j-1, \mathsf{st}_j^0)$$

Set $w_{\mathsf{st}}^0 := \hat{w}_{\mathsf{st},m}$, and $store_{\mathsf{st}}^0 := \hat{store}_{\mathsf{st},m}$.
Set $w_{\mathsf{com}}^0 = \bot$ and $store_{\mathsf{com}}^0 = \hat{store}_{\mathsf{com},0}$ (w_{com}^0 is computed similarly as w_{st}^0. However, since $\mathsf{com}_j^0 = \bot$ for all CPU j, $w_{\mathsf{com}}^0 = \bot$.)

– Compute $\mathsf{root\_node}^0 = (t, \mathsf{root\_index}, w_{\mathsf{st}}^0, w_{\mathsf{com}}^0, v^0, \sigma^0)$ where $t = 0$, $\mathsf{root\_index} = \epsilon$, and w_{st}^0, w_{com}^0,v^0 are computed above. σ^0 is computed as follows:

$$r_A \leftarrow \mathsf{PRF}(K_A, 0)$$
$$(\mathrm{sk}^0, \mathrm{vk}^0) \leftarrow \mathsf{Spl.Setup}(1^\lambda; r_A)$$
$$\sigma^0 \leftarrow \mathsf{Spl.Sign}(\mathrm{sk}^0, (t, \mathsf{root\_index}, w_{\mathsf{st}}^0, w_{\mathsf{com}}^0, v^0))$$

Algorithm 9: F_{branch}
// for simplicity, we drop the subscripts from $\widetilde{a}^{\text{in}}_{id_{\mathsf{cpu}}\leftarrow\mathtt{M}}$ and $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow id_{\mathsf{cpu}}}$, and use $\widetilde{a}^{\text{in}}$ and $\widetilde{a}^{\text{out}}$ respectively

```
   Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\text{in}} = (\mathsf{com}^{\text{in}}, \pi^{\text{in}}_{\mathsf{st}}, \pi^{\text{in}}_{\mathsf{com}})$
   Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  Parse $\mathsf{root\_node}$ as $(t, \mathsf{root\_index}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}, v^{\text{in}}, \sigma^{\text{in}})$;
2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$;
4  Let $m^{\text{in}} = (t, \mathsf{root\_index}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}, v^{\text{in}})$;
5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 0$ then output $\mathtt{Reject}$;
6  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\text{in}}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}), \pi^{\text{in}}_{\mathsf{st}}) = 0$ then output $\mathtt{Reject}$;
7  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\text{in}}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\text{in}}), \pi^{\text{in}}_{\mathsf{com}}) = 0$ then output $\mathtt{Reject}$;
8  Compute $(\mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}) \leftarrow F(id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}, \mathsf{com}^{\text{in}})$;
9  Compute $v^{\text{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}, \mathsf{com}^{\text{in}}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}))$;
10 if $\mathsf{st}^{\text{out}} = \mathtt{Reject}$ then
11     Output $\mathtt{Reject}$;
12 else
13     Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$;
14     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$;
15     Let $m^{\text{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}, v^{\text{out}})$ and $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}})$;
16     Let $\mathsf{node}^{\text{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}, v^{\text{out}}, \sigma^{\text{out}})$;
17     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{out}}, id_{\mathsf{cpu}}, \bot), \widetilde{a}^{\text{out}} = \mathsf{node}^{\text{out}}$;
```

– Now we compute the initial configuration as

$$\widetilde{\mathsf{mem}}^0 = (store^0_{\mathsf{st}}, store^0_{\mathsf{com}})$$
$$\widetilde{\mathsf{st}}^0_j = (\mathsf{st}^0_j, j, \mathsf{root\_node}^0)$$

Final step. Finally the compilation procedure returns the value $\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \{\widetilde{\mathsf{st}}^0_k\}^m_{k=1}), \widetilde{F})$ as output.

Evaluation algorithm $\mathsf{conf} := \mathsf{Eval}(\widetilde{\Pi})$**:** Upon receiving an obfuscated system $\widetilde{\Pi}$, the evaluator runs Algorithm 11 and returns at the halting time t^* the result

$$(\widetilde{\mathsf{mem}}^{t^*}, \{\widetilde{\mathsf{st}}^{t^*}_k, \widetilde{a}^{t^*}_{\mathtt{M}\leftarrow k}\}^m_{k=1}).$$

– For each $1 \le k \le m$, parse:

$$\widetilde{\mathsf{st}}^{t^*}_k = (\mathsf{st}^{t^*}_k, k, \cdot)$$
$$\widetilde{a}^{t^*}_{\mathtt{M}\leftarrow k} = (\mathsf{com}^{t^*}_k)$$

– Return $\mathsf{conf} = \{\mathsf{st}^{t^*}_k, c^{t^*}_k = \mathsf{com}^{t^*}_k\}^m_{k=1}$.

Efficiency Let m be the number of CPUs, $|F|$ be the description size of program F, n/m be the size of each initial states st^0_k for $k \in [m]$, computation system Π proceeds with time bound T. We first note the circuit size of $\widehat{F}$ is $|F| + O(\log m)$, where $\log m$ is the amount of hardwired information in some hybrid programs that required in security proof. Please refer to Appendix B.3.2 for details. Assuming $\mathsf{i}\mathcal{O}$ is a circuit obfuscator with circuit size $|\mathsf{i}\mathcal{O}(C)| \le \mathsf{poly}|C|$ for given circuit C. Our $\mathsf{Ci}\mathcal{O}$ for mPRAM has following complexity:

– Compilation time is $O(\mathsf{poly}(|F|) + n)$.

```
Algorithm 10: $F_{\mathsf{combine}}$
// for simplicity, we drop the subscripts from $\widetilde{a}^{\text{in}}_{id_{\mathsf{cpu}}\leftarrow \mathtt{M}}$ and $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow id_{\mathsf{cpu}}}$, and use $\widetilde{a}^{\text{in}}$ and $\widetilde{a}^{\text{out}}$ respectively

Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\text{in}} = (\mathsf{node}_1, \mathsf{node}_2)$
Data  : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$;
10     Let $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $\mathsf{Spl.Verify}(\mathrm{vk}_{A,\zeta}, m_\zeta, \sigma_\zeta) = 0$ then output Reject;
12 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2}, \mathsf{parent\_index})$;
13 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2}, \mathsf{parent\_index})$;
14 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
15 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$;
16 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$;
17 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
18 Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
19 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
20 if $\mathsf{parent\_index} = \epsilon$ then
21     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\text{out}} = \perp$;
22 else
23     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\text{out}} = \mathsf{parent\_node}$;
```

Algorithm 11: Eval: Evaluator of $\widetilde{\Pi}$

```
Input : $\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \{\widetilde{\mathsf{st}}_k^0\}_{k=1}^m), \widetilde{F})$
1  Let $z_{max} \leftarrow \lceil \log(m) \rceil$ be the length of $m$ in binary;
2  for $1 \le t \le T$ do
3      Parse $\widetilde{\mathsf{mem}}^{t-1} = (store_{\mathsf{st}}^{t-1}, store_{\mathsf{com}}^{t-1})$;
4      for $1 \le k \le m$ do
5          Compute $(\cdot, \pi_{\mathsf{st},k}^{t-1}) \leftarrow \mathsf{PrepRead}(\mathsf{pp}_{\mathsf{Acc,st}}, store_{\mathsf{st}}^{t-1}, k)$;
6          Compute $(\mathsf{com}_k^{t-1}, \pi_{\mathsf{com},k}^{t-1}) \leftarrow \mathsf{PrepRead}(\mathsf{pp}_{\mathsf{Acc,com}}, store_{\mathsf{com}}^{t-1}, \mathsf{src}(t-1,k))$;
7          Let $\widetilde{a}_k^{\mathrm{in}} \leftarrow (\mathsf{com}_k^{t-1}, \pi_{\mathsf{st},k}^{t-1}, \pi_{\mathsf{com},k}^{t-1})$ ;
8          Evaluate $(\widetilde{\mathsf{st}}_k^{t,z_{max}}, \widetilde{a}_k^{\mathrm{out}}) \leftarrow \widetilde{F}(\widetilde{\mathsf{st}}_k^{t-1}, \widetilde{a}_k^{\mathrm{in}})$ ;          // Evaluate $F_{\mathsf{branch}}$
9          Parse $\widetilde{a}_k^{\mathrm{out}} = (\cdot, \cdot, \mathsf{st}_k^{t,z_{max}}, \mathsf{com}_k^t, \cdot, \cdot)$;
10         Let $store_{\mathsf{st}}^t[k] \leftarrow \mathsf{st}_k^{t,z_{max}}$, which stores $\mathsf{st}_k^{t,z_{max}}$ in the $k$-th cell in $store_{\mathsf{st}}^t$;
11         Let $store_{\mathsf{com}}^t[k] \leftarrow \mathsf{com}_k^t$, which stores $\mathsf{com}_k^t$ in the $k$-th cell in $store_{\mathsf{com}}^t$;
12         Let $\mathsf{node}_k^t \leftarrow \widetilde{a}_k^{\mathrm{out}}$;
13     if all $\widetilde{\mathsf{st}}_k^{t,z_{max}}$ is halt state then
14         Let $\widetilde{\mathsf{mem}}^t \leftarrow (store_{\mathsf{st}}^t, store_{\mathsf{com}}^t)$;
15         for $1 \le k \le m$ do let $\widetilde{\mathsf{st}}_k^t \leftarrow \widetilde{\mathsf{st}}_k^{t,z_{max}}$;
16         for $1 \le k \le m$ do let $\widetilde{a}_{\mathtt{M}\leftarrow k}^t \leftarrow \mathsf{com}_k^t$;
17         return $(\widetilde{\mathsf{mem}}^t, \{\widetilde{\mathsf{st}}_k^t, \widetilde{a}_{\mathtt{M}\leftarrow k}^t\}_{k=1}^m)$;
18     for $z \leftarrow z_{max} - 1$ to $0$ do
19         foreach $k \in [m]$ do
20             Represent $k-1$ in a binary string $s(k-1)$ of length $z_{max}$;
21             Let $k_z$ be the prefix $z$ bits of $s(k-1)$;
22             Let $k_z \| b$ be the binary string that $k_z$ concatenates bit $b$;
23         for $1 \le k \le m$ do
24             Let $\widetilde{a}_k^{\mathrm{in}} \leftarrow (\mathsf{node}_{k_z\|0}^t, \mathsf{node}_{k_z\|1}^t)$;
25             Evaluate $(\widetilde{\mathsf{st}}_k^{t,z}, \widetilde{a}_k^{\mathrm{out}}) \leftarrow \widetilde{F}(\widetilde{\mathsf{st}}_k^{t,z+1}, \widetilde{a}_k^{\mathrm{in}})$ ;          // Evaluate $F_{\mathsf{combine}}$
26             Let $\mathsf{node}_{k_z}^t \leftarrow \widetilde{a}_k^{\mathrm{out}}$;
27     Let $\widetilde{\mathsf{mem}}^t \leftarrow (store_{\mathsf{st}}^t, store_{\mathsf{com}}^t)$ ;          // Input to the next iteration
28     Let $\widetilde{\mathsf{st}}_k^t \leftarrow \widetilde{\mathsf{st}}_k^{t,0}$;
```

- Compilation size is $O(\mathsf{poly}(|F|) + n)$.
- Parallel evaluation time is $O(T \cdot \mathsf{poly}(|F|))$.
- Evaluation space is $O(m)$, which corresponds to keeping the CPU states of F during branch and combine.

Theorem 6.6. *Assuming* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* TItr *is a secure topological iterator,* Acc *is a secure accumulator,* Spl *is a secure splittable signature scheme; then* $\mathsf{Ci}\mathcal{O}$ *is a secure computation-trace indistinguishability obfuscation with respect to* $\mathcal{P}_{\mathrm{mPRAM}}$.

Proof can be found in Appendix B.3.

6.6 Construction for $\mathsf{Ci}\mathcal{O}$-PRAM

Finally, we construct $\mathsf{Ci}\mathcal{O}$ in the (standard) PRAM (PRAM) model. A PRAM consists of m CPUs running simultaneously with random access to a shared memory, but without communication with each others. Formally, the class of distributed computation for PRAM, denoted by $\mathcal{P}_{\mathrm{PRAM}}$, is defined as follows:

Definition 6.7 (PRAM Computation Class). $\mathcal{P}_{\mathrm{PRAM}}$ *is a class of distributed computation systems for PRAM with* m *agents (a.k.a. CPU)* $1, \ldots, m$ *and a shared memory* $\mathtt{M}$ *where*
- *the terminating time* t^* *is bounded by* T*;*
- *the communication between agents are not allowed, i.e.,* $c^t_{j\leftarrow k} := \perp$ *for all* $t \in [t^*]$ *and for all* $j, k \in [m]$*;*
- *the memory size* $|\mathsf{mem}|$ *is bounded by* S*;*
- *for all* $k \in [m]$*, the state size* $|\mathsf{st}_k|$ *and the communication buffer sizes* $|a_{k\leftarrow \mathtt{M}}|$ *and* $|a_{\mathtt{M}\leftarrow k}|$ *are bounded by* $\mathsf{poly}\log(T)$*;*
- *for all* $k \in [m]$*, the initial access commands are restricted to* $a^0_{k\leftarrow \mathtt{M}} := \perp$ *and* $a^0_{\mathtt{M}\leftarrow k} := \perp$*;*
- *for all* $k \in [m]$*, the initial states are restricted to* $\mathsf{st}^0_k := \perp$.

Our construction of $\mathsf{Ci}\mathcal{O}$-PRAM is very similar to that of $\mathsf{Ci}\mathcal{O}$ for mPRAM except that F_{branch} now also takes as input a bit read from the memory with its proof, and outputs a bit to be written to some memory location. Thus, as for $\mathsf{Ci}\mathcal{O}$-RAM, the evaluator in addition maintains an accumulator for storing the actual memory content. Correspondingly, instead of executing F directly, F_{branch} executes another program called F_{check} which encapsulates F and the oblivious update mechanism described above.

We next describe in details our scheme $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{Obf}, \mathsf{Eval}\}$ in the PRAM model. The compilation procedure $\mathsf{Ci}\mathcal{O}.\mathsf{Obf}$ can transform a given computation system $\Pi \in \mathcal{P}_{\mathrm{PRAM}}$ into an obfuscated computation system $\widetilde{\Pi}$, where

$$\Pi = (\mathsf{mem}^0, F), \text{ and}$$
$$\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0), \widetilde{F}).$$

Compilation procedure $\widetilde{\Pi} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \Pi)$: We provide the details of the compilation procedure $\mathsf{Obf}()$ which consists of several steps as follows.

<u>*Step 1: Generating parameters.*</u> The compilation procedure computes the following parameters for the obfuscated computation system:

$$K_A \leftarrow \mathsf{PRF.Setup}(1^\lambda)$$
$$(\mathsf{pp}_{\mathsf{Acc,mem}}, \hat{w}_{\mathsf{mem},0}, \hat{store}_{\mathsf{mem},0}) \leftarrow \mathsf{Acc.Setup}(m)$$
$$(\mathsf{pp}_{\mathsf{Acc,st}}, \hat{w}_{\mathsf{st},0}, \hat{store}_{\mathsf{st},0}) \leftarrow \mathsf{Acc.Setup}(m)$$
$$(\mathsf{pp}_{\mathsf{Acc,com}}, \hat{w}_{\mathsf{com},0}, \hat{store}_{\mathsf{com},0}) \leftarrow \mathsf{Acc.Setup}(S)$$
$$(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{TItr.Setup}(T)$$

Step 2: Generating stateful algorithms $\widetilde{F}$. Based on the parameters T, $\mathsf{pp}_{\mathsf{Acc,mem}}$, $\mathsf{pp}_{\mathsf{Acc,st}}$, $\mathsf{pp}_{\mathsf{Acc,com}}$, $\mathsf{pp}_{\mathsf{Itr}}$, K_A generated above, as well as program F, we define the program $\widehat{F}$ in Algorithm 12. $\widehat{F}$ executes internal programs F_{branch} (Algorithm 13) or F_{combine} (Algorithm 14) depending on its input. Now F_{branch} in turn executes F_{check} defined in Algorithm 7, where $F_{\mathsf{check}} = \mathsf{AccCompile}(F, \mathsf{Acc.OUpdate}\{\mathsf{pp}_{\mathsf{Acc,mem}}\})$.

The compilation procedure then computes an obfuscation of the program $\widehat{F}$. That is, $\widetilde{F} \leftarrow \mathsf{i}\mathcal{O}.\mathsf{Gen}(\widehat{F})$.

Algorithm 12: $\widehat{F}$, which is identical to its mPRAM counterpart (Algorithm 8)
// for simplicity, we drop the subscripts from $\widetilde{a}^{\mathrm{in}}_{id_{\mathsf{cpu}}\leftarrow \mathtt{M}}$ and $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow id_{\mathsf{cpu}}}$, and use $\widetilde{a}^{\mathrm{in}}$ and $\widetilde{a}^{\mathrm{out}}$ respectively

```
   Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\mathrm{in}}$
 1 if $\mathsf{st}^{\mathrm{in}} = (\mathtt{halt}, \cdot)$ then
 2     Output $\mathtt{Reject}$;
 3 else if $\mathsf{root\_node} \neq \perp$ then
 4     Compute $(\widetilde{\mathsf{st}}^{\mathrm{out}}, \widetilde{a}^{\mathrm{out}}) = F_{\mathsf{branch}}(\widetilde{\mathsf{st}}^{\mathrm{in}}, \widetilde{a}^{\mathrm{in}})$;
 5 else
 6     Compute $(\widetilde{\mathsf{st}}^{\mathrm{out}}, \widetilde{a}^{\mathrm{out}}) = F_{\mathsf{combine}}(\widetilde{\mathsf{st}}^{\mathrm{in}}, \widetilde{a}^{\mathrm{in}})$;
 7 Output $(\widetilde{\mathsf{st}}^{\mathrm{out}}, \widetilde{a}^{\mathrm{out}})$;
```

Algorithm 13: F_{branch}
// for simplicity, we drop the subscripts from $\widetilde{a}^{\mathrm{in}}_{id_{\mathsf{cpu}}\leftarrow \mathtt{M}}$ and $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow id_{\mathsf{cpu}}}$, and use $\widetilde{a}^{\mathrm{in}}$ and $\widetilde{a}^{\mathrm{out}}$ respectively

```
    Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\mathrm{in}} = (b^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, \pi^{\mathrm{in}}_{\mathsf{mem}}, \pi^{\mathrm{in}}_{\mathsf{st}}, \pi^{\mathrm{in}}_{\mathsf{com}})$
    Data  : $\mathsf{pp}_{\mathsf{Acc,mem}}, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
 1  Parse $\mathsf{root\_node}$ as $(t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}}, \sigma^{\mathrm{in}})$;
 2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$;
 3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$;
 4  Let $m^{\mathrm{in}} = (t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}})$;
 5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 0$ then output $\mathtt{Reject}$;
 6  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\mathrm{in}}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{st}}) = 0$ then output $\mathtt{Reject}$;
 7  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\mathrm{in}}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{com}}) = 0$ then output $\mathtt{Reject}$;
 8  $(\mathsf{st}^{\mathrm{out}}, (\mathsf{com}^{\mathrm{out}}, \mathsf{loc}^{\mathrm{out}}, b^{\mathrm{out}})) \leftarrow F_{\mathsf{check}}(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, (b^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, \pi^{\mathrm{in}}_{\mathsf{mem}}))$;
 9  Compute $v^{\mathrm{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}))$;
 10 if $\mathsf{st}^{\mathrm{out}} = \mathtt{Reject}$ then
 11     Output $\mathtt{Reject}$;
 12 else
 13     Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$;
 14     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$;
 15     Let $m^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}})$ and $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
 16     Let $\mathsf{node}^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}}, \sigma^{\mathrm{out}})$;
 17     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\mathrm{out}} = (\mathsf{node}^{\mathrm{out}}, \mathsf{loc}^{\mathrm{out}}, b^{\mathrm{out}})$;
```

Step 3: Generating the initial configuration $(\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0)$. Recall that initial memory accesses are empty: $a^0_{k\leftarrow \mathtt{M}} = \perp$, $a^0_{\mathtt{M}\leftarrow k} = \perp$. Based on mem^0 and $\mathsf{st}^0_k = \perp$ for all $k \in [m]$, the compilation procedure computes the initial configuration for the complied computation system as follows.

Algorithm 14: F_{combine}, which is identical to its mPRAM counterpart (Algorithm 10)
// for simplicity, we drop the subscripts from $\widetilde{a}^{\text{in}}_{id_{\mathsf{cpu}}\leftarrow\mathtt{M}}$ and $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow id_{\mathsf{cpu}}}$, and use $\widetilde{a}^{\text{in}}$ and $\widetilde{a}^{\text{out}}$ respectively

```
   Input  : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\text{in}} = (\mathsf{node}_1, \mathsf{node}_2)$
   Data   : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  for $\zeta = 1, 2$ do Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$;
10     Let $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $\mathsf{Spl.Verify}(\mathrm{vk}_{A,\zeta}, m_\zeta, \sigma_\zeta) = 0$ then output Reject;
12 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2}, \mathsf{parent\_index})$;
13 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2}, \mathsf{parent\_index})$;
14 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
15 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$;
16 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$;
17 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
18 Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
19 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
20 if $\mathsf{parent\_index} = \epsilon$ then
21     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\text{out}} = \perp$;
22 else
23     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\text{out}} = \mathsf{parent\_node}$;
```

- For each $j \in \{1, \ldots, |\mathsf{mem}^0|\}$, it computes iteratively:

$$\begin{aligned} \hat{store}_{\mathsf{mem},j} &\leftarrow \mathsf{Acc.WriteStore}(\mathsf{pp}_{\mathsf{Acc},\mathsf{mem}}, \hat{store}_{\mathsf{mem},j-1,}, j, \mathsf{mem}^0[j]) \\ \pi_j &\leftarrow \mathsf{Acc.PrepWrite}(\mathsf{pp}_{\mathsf{Acc},\mathsf{mem}}, \hat{store}_{\mathsf{mem},j-1}, j) \\ \hat{w}_j &\leftarrow \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc},\mathsf{mem}}, \hat{w}_{j-1}, j, x_j, \pi_j) \end{aligned}$$

  Set $w^0_{\mathsf{mem}} := \hat{w}_{|\mathsf{mem}^0|}$, and $store^0_{\mathsf{mem}} := \hat{store}_{|\mathsf{mem}^0|}$.
- Set $w^0_{\mathsf{st}} := \perp$ and $store^0_{\mathsf{st}} := \hat{store}_{\mathsf{st},0}$, where $\hat{store}_{\mathsf{st},0}$ is initialized with value $\perp$ as the initial state in all m cells.
- Set $w^0_{\mathsf{com}} := \perp$ and $store^0_{\mathsf{com}} := \hat{store}_{\mathsf{com},0}$, where $\hat{store}_{\mathsf{com},0}$ is initialized with value $\perp$ as no communication in all m cells.
- Set $\mathsf{root\_node}^0 = (t, \mathsf{root\_index}, w^0_{\mathsf{st}}, w^0_{\mathsf{com}}, v^0, \sigma^0)$ where $t = 0$, $\mathsf{root\_index} = \epsilon$; and w^0_{st}, w^0_{com},v^0 are computed above; and σ^0 is computed as follows:

$$\begin{aligned} r_A &\leftarrow \mathsf{PRF}(K_A, 0) \\ (\mathrm{sk}^0, \mathrm{vk}^0) &\leftarrow \mathsf{Spl.Setup}(1^\lambda; r_A) \\ \sigma^0 &\leftarrow \mathsf{Spl.Sign}(\mathrm{sk}^0, (0, \mathsf{root\_index}, w^0_{\mathsf{st}}, w^0_{\mathsf{com}}, v^0)) \end{aligned}$$

- $\mathsf{st}^0 = ((\mathsf{st}^0_\Pi, \mathsf{st}^0_{\mathsf{Acc}}), d_{\mathsf{Acc}}, b, w^0, \mathsf{loc}) = ((\perp, \perp), \textsc{Read}, \perp, w^0_{\mathsf{mem}}, \perp)$ where w^0_{mem} is computed above.
- $buff^0[1] = \ldots = buff^0[m] = (\perp, \perp)$
- Now we can define the initial configuration to be

$$\begin{aligned} \widetilde{\mathsf{mem}}^0 &= (store^0_{\mathsf{mem}}, store^0_{\mathsf{st}}, store^0_{\mathsf{com}}, buff^0) \\ \widetilde{\mathsf{st}}^0 &= (\mathsf{root\_node}^0, \perp, \mathsf{st}^0, \mathsf{com}^0). \end{aligned}$$

<u>*Final step.*</u> Finally, the compilation procedure returns the value $\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0), \widetilde{F})$ as output.

Evaluation algorithm $\mathsf{conf} := \mathsf{Eval}(\widetilde{\Pi})$**:** Upon receiving an obfuscated system $\widetilde{\Pi}$, the evaluator parse $\widetilde{\Pi} = (\widetilde{\mathsf{mem}}^0, \widetilde{\mathsf{st}}^0)$, where $\widetilde{\mathsf{st}}^0 = (\mathsf{root\_node}^0, \perp, \mathsf{st}^0, \mathsf{com}^0)$. It sets $\widetilde{\mathsf{st}}^0_k = (\mathsf{root\_node}^0, k, \mathsf{st}^0, \mathsf{com}^0)$ for $k = 1$ to m. It then runs Algorithm 11 and carries out the result

$$(\widetilde{\mathsf{mem}}^{t^*}, \{\widetilde{\mathsf{st}}^{t^*}_k, \widetilde{a}^{t^*}_{\mathtt{M}\leftarrow k}\}^m_{k=1})$$

at the halting time t^*.
- For $1 \le k \le m$, parse:

$$\begin{aligned} \widetilde{\mathsf{mem}}^{t^*} &= (store^{t^*}_{\mathsf{mem}}, store^{t^*}_{\mathsf{st}}, store^{t^*}_{\mathsf{com}}, buff^{t^*}) \\ \widetilde{\mathsf{st}}^{t^*}_k &= (\mathsf{st}^{t^*}_k, k, \cdot) \\ \widetilde{a}^{t^*}_{\mathtt{M}\leftarrow k} &= (\mathsf{node}^{t^*}_k, \mathsf{loc}^{t^*}_k, b^{t^*}_k) \end{aligned}$$

- For $1 \le k \le m$, let:

$$\begin{aligned} a^{t^*}_{\mathtt{M}\leftarrow k} &= (\mathsf{loc}^{t^*}_k, b^{t^*}_k) \\ a^{t^*}_{k\leftarrow \mathtt{M}} &= \perp. \end{aligned}$$

  Let $\mathsf{mem}^{t^*} = store^{t^*}_{\mathsf{mem}}$.
- Return $\mathsf{conf} = (\{\mathsf{st}^{t^*}_k, a^{t^*}_{\mathtt{M}\leftarrow k}, a^{t^*}_{k\leftarrow \mathtt{M}}\}^m_{k=1}, \mathsf{mem}^{t^*})$.

Algorithm 15: Eval: Evaluator of $\widetilde{\Pi}$

```
Input : $\widetilde{\Pi} = ((\widetilde{\mathsf{mem}}^0, \{\widetilde{\mathsf{st}}_k^0\}_{k=1}^m), \widetilde{F})$
1 Let $z_{max} \leftarrow \lceil \log(m) \rceil$ be the length of $m$ in binary;
2 for $1 \le t \le T$ do
3     Parse $\widetilde{\mathsf{mem}}^{t-1} = (store^{t-1}_{\mathsf{mem}}, store^{t-1}_{\mathsf{st}}, store^{t-1}_{\mathsf{com}}, buff^{t-1})$;
4     for $1 \le k \le m$ do
5         Compute $(\cdot, \pi^{t-1}_{\mathsf{st},k}) \leftarrow \mathsf{PrepRead}(\mathsf{pp}_{\mathsf{Acc,st}}, store^{t-1}_{\mathsf{st}}, k)$;
6         Compute $(\mathsf{com}^{t-1}_k, \pi^{t-1}_{\mathsf{com},k}) \leftarrow \mathsf{PrepRead}(\mathsf{pp}_{\mathsf{Acc,com}}, store^{t-1}_{\mathsf{com}}, \mathsf{src}(t-1,k))$;
7         Parse $buff^{t-1}[k] = (b^{\mathrm{in}}_k, \pi^{\mathrm{in}}_{\mathsf{mem},k})$;
8         Let $\widetilde{a}^{\mathrm{in}}_k \leftarrow (b^{\mathrm{in}}_k, \mathsf{com}^{t-1}_k, \pi^{\mathrm{in}}_{\mathsf{mem},k}, \pi^{t-1}_{\mathsf{st},k}, \pi^{t-1}_{\mathsf{com},k})$ ;
9         Evaluate $(\widetilde{\mathsf{st}}^{t,z_{max}}_k, \widetilde{a}^{\mathrm{out}}_k) \leftarrow \widetilde{F}(\widetilde{\mathsf{st}}^{t-1}_k, \widetilde{a}^{\mathrm{in}}_k)$ ;        // Evaluate $F_{\mathsf{branch}}$
10        Parse $\widetilde{a}^{\mathrm{out}}_k = (\mathsf{node}^t_k, \mathsf{loc}^t_k, b^t_k)$;
11        Parse $\mathsf{node}^t_k = (\cdot, \cdot, \mathsf{st}^{t,z_{max}}_k, \mathsf{com}^t_k, \cdot, \cdot)$;
12        Let $store^t_{\mathsf{st}}[k] \leftarrow \mathsf{st}^{t,z_{max}}_k$, which stores $\mathsf{st}^{t,z_{max}}_k$ in the $k$-th cell in $store^t_{\mathsf{st}}$;
13        Let $store^t_{\mathsf{com}}[k] \leftarrow \mathsf{com}^t_k$, which stores $\mathsf{com}^t_k$ in the $k$-th cell in $store^t_{\mathsf{com}}$;
14    if $\mathsf{loc}^t_k = \perp \ \forall 1 \le k \le m$ then
15        Let $store^t_{\mathsf{mem}} \leftarrow store^{t-1}_{\mathsf{mem}}$;
16        for $1 \le k \le m$ do let $buff^t[k] = (\perp, \perp)$;
17    else
18        for $1 \le k \le m$ do
19            Compute $buff^t[k] = (b^{\mathrm{in}}_k, \pi^{\mathrm{in}}_{\mathsf{mem},k}) \leftarrow \mathsf{PrepRead}(\mathsf{pp}_{\mathsf{Acc,mem}}, store^{t-1}_{\mathsf{mem}}, \mathsf{loc}^t_k)$;
20        Let $store^{t,0}_{\mathsf{mem}} \leftarrow store^{t-1}_{\mathsf{mem}}$;
21        for $1 \le k \le m$ do compute $store^{t,k}_{\mathsf{mem}} \leftarrow \mathsf{WriteStore}(\mathsf{pp}_{\mathsf{Acc,mem}}, store^{t,k-1}_{\mathsf{mem}}, (\mathsf{loc}^t_k, b^t_k))$;
22        Let $store^t_{\mathsf{mem}} \leftarrow store^{t,m}_{\mathsf{mem}}$;
23    if all $\widetilde{\mathsf{st}}^{t,z_{max}}_k$ is halt state then
24        Let $\widetilde{\mathsf{mem}}^t \leftarrow (store^t_{\mathsf{mem}}, store^t_{\mathsf{st}}, store^t_{\mathsf{com}}, buff^t)$;
25        for $1 \le k \le m$ do let $\widetilde{\mathsf{st}}^t_k \leftarrow \widetilde{\mathsf{st}}^{t,z_{max}}_k$;
26        for $1 \le k \le m$ do let $\widetilde{a}^t_{\mathtt{M}\leftarrow k} \leftarrow (\mathsf{node}^t_k, \mathsf{loc}^t_k, b^t_k)$;
27        return $(\widetilde{\mathsf{mem}}^t, \{\widetilde{\mathsf{st}}^t_k, \widetilde{a}^t_{\mathtt{M}\leftarrow k}\}_{k=1}^m)$;
28    for $z \leftarrow z_{max} - 1$ to $0$ do
29        foreach $k \in [m]$ do
30            Represent $k-1$ in a binary string $s(k-1)$ of length $z_{max}$;
31            Let $k_z$ be the prefix $z$ bits of $s(k-1)$;
32            Let $k_z \| b$ be the binary string that $k_z$ concatenates bit $b$;
33        for $1 \le k \le m$ do
34            Let $\widetilde{a}^{\mathrm{in}}_k \leftarrow (\mathsf{node}^t_{k_z\|0}, \mathsf{node}^t_{k_z\|1})$;
35            Evaluate $(\widetilde{\mathsf{st}}^{t,z}_k, \widetilde{a}^{\mathrm{out}}_k) \leftarrow \widetilde{F}(\widetilde{\mathsf{st}}^{t,z+1}_k, \widetilde{a}^{\mathrm{in}}_k)$ ;        // Evaluate $F_{\mathsf{combine}}$
36            Let $\mathsf{node}^t_{k_z} \leftarrow \widetilde{a}^{\mathrm{out}}_k$;
37    Let $\widetilde{\mathsf{mem}}^t \leftarrow (store^t_{\mathsf{mem}}, store^t_{\mathsf{st}}, store^t_{\mathsf{com}}, buff^t)$ ;    // Input to the next iteration
38    Let $\widetilde{\mathsf{st}}^t_k \leftarrow \widetilde{\mathsf{st}}^{t,0}_k$;
```

Efficiency Let m be the number of CPUs, $|F|$ be the description size of program F, n be the description size of initial memory mem^0, computation system Π proceeds with time and space bound T and S. We first note the circuit size of $\widehat{F}$ is $|F| + O(\log m)$, where $\log m$ is the amount of hardwired information required in security proof (similar to Appendix B.3.2). Assume that $\mathsf{i}\mathcal{O}$ is a circuit obfuscator with circuit size $|\mathsf{i}\mathcal{O}(C)| \leq \mathsf{poly}|C|$ for given circuit C. Our $\mathsf{Ci}\mathcal{O}$-PRAM has following complexity:

- Compilation time is $\tilde{O}(\mathsf{poly}(|F|) + n)$.
- Compilation size is $\tilde{O}(\mathsf{poly}(|F|) + n)$.
- Parallel evaluation time is $\tilde{O}(T \cdot \mathsf{poly}(|F|))$.
- Evaluation space is $\tilde{O}(m + S)$, where m term is to keep CPU states of F while branch and combine, and S term is needed by F intrinsically.

Theorem 6.8. *Assuming* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* TItr *is a secure topological iterator,* Acc *is a secure positional accumulator, and* Spl *is a secure splittable signature scheme;* $\mathsf{Ci}\mathcal{O}$ *is a secure computation-trace indistinguishability obfuscation with respect to* $\mathcal{P}_{\mathrm{PRAM}}$.

The proof sketch can be found in Appendix B.4.

7 Constructing $\mathcal{RE}$ in the RAM Model ($\mathcal{RE}$-RAM)

In this section, we showcase the power of our fully succinct $\mathsf{Ci}\mathcal{O}$-RAM by constructing the first fully succinct randomized encoding in the RAM model. A randomized encoding of a computation instance (P, x) hides everything except its output $y = P(x)$ and runtime t^*. This requires hiding both the memory content and the access pattern of the computation. The formal definition can be found in Appendix A.2. At a high level, our construction is a fairly natural one: We use public-key encryption to hide the memory content (including the input), and use oblivious RAM to hide the access pattern. We then use $\mathsf{Ci}\mathcal{O}$ to obfuscate the compiled computation instance. Namely, our $\mathcal{RE}$ encoding algorithm outputs $\widetilde{\Pi} = \mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$, where Π_{hide} is a computation instance defined by P_{hide} and x_{hide}. P_{hide} is a $\mathcal{PKE}$ and ORAM compiled version of P, and x_{hide} is an encrypted version of x. Namely, P_{hide} outputs encrypted CPU states and memory contents at each time step, and uses ORAM to compile its memory access (with randomness supplied by PRF for succinctness).

Intuitively, if $\mathcal{PKE}$ and ORAM are secure, then the computation should be hidden. However, note that, the decryption keys need to be hardwired in P_{hide} to evaluate $P(x)$. As $\mathsf{Ci}\mathcal{O}$ does not hide anything explicitly, it is not clear whether we can use the security of $\mathcal{PKE}$ and ORAM at all. In particular, it is unlikely that we can use the security of ORAM, since it only hides the access pattern when the CPU state and the memory contents are hidden from the adversary. Indeed, hiding the access pattern is the major technical challenge to prove the security of our $\mathcal{RE}$ construction. Next we provide an overview of our main ideas.

Basic version: $\mathcal{RE}$ for oblivious RAM computation To demonstrate the ideas in our full-fledged construction, we start with the simpler case of $\mathcal{RE}$ for *oblivious* RAM computation where the given RAM computation instance Π defined by (P, x) has oblivious access pattern. Namely, we assume that there is a public access function $\mathsf{ap}(t)$ that predicts the memory access at each time step t, which is given to the simulator. Thus, we only need to hide the CPU state and the memory content in each step of the computation, without using oblivious RAM to hide the access pattern.

For this simpler case, we can directly use techniques developed by [KLW15] to hide the memory content using public-key encryption. In fact, the construction of machine-hiding encoding for TM in [KLW15] can be modified in a straightforward way to yield $\mathcal{RE}$ for oblivious RAM computation based on $\mathsf{i}\mathcal{O}$ for circuits. Our $\mathsf{Ci}\mathcal{O}$-based construction presented below can be viewed as a modularization and simplification of their construction through our $\mathsf{Ci}\mathcal{O}$ notion.

Recall that our construction is of the form $\mathcal{RE}.\mathsf{Encode}(P, x) = \mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$, where Π_{hide} is defined by P_{hide} and x_{hide}. Here, we only use $\mathcal{PKE}$ to compile P, and denote the compiled program by $P_{\mathcal{PKE}}$ instead of P_{hide}. We also denote encrypted input by $x_{\mathcal{PKE}}$ instead of x_{hide}. At a high level, $P_{\mathcal{PKE}}$ emulates P step by step, but instead of outputting the CPU state and the memory content in the clear, $P_{\mathcal{PKE}}$ outputs an encrypted version of them. $P_{\mathcal{PKE}}$ also expects encrypted CPU states and memory contents as input, and emulates P on the decryption of the input. A key idea here (following [KLW15]) is to encrypt each message (either a CPU state of a memory cell) using different keys, and generate these keys (as well as encryption randomness) using puncturable PRF (PPRF), which allows us to use a standard puncturing argument (extended to work with $\mathsf{Ci}\mathcal{O}$ instead of $\mathsf{i}\mathcal{O}$) to move to a hybrid where semantic security holds for a particular message so that we can "erase" the message.

In the detailed proof, we prove the security by a sequence of hybrids that "erase" the computation *backward in time*, which leads to a simulated encoding $\mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{Sim}})$ where all ciphertexts generated by P_{Sim} as well as in x_{Sim} are replaced by encryption of a special `dummy` symbol. More precisely, P_{Sim} simulates the access pattern using the public access function ap at each time step $t < t^*$, simply ignores the input and outputs encryption of `dummy` (for both CPU state and memory content), and outputs y at time step $t = t^*$.

Full solution: $\mathcal{RE}$ for general RAM computation We now turn to our full solution, and deal with the main challenge of hiding access pattern. As mentioned, our approach is a natural one, where we use oblivious RAM (ORAM) compiler to hide the access pattern. Recall that an ORAM compiler compiles a RAM program by replacing each memory access by a *probabilistic* procedure OACCESS which accesses memory in a way that hides the access pattern. Given a computation instance Π defined by (P, x), we first compile P using an ORAM compiler with randomness supplied by puncturable PRF. Let P_{ORAM} be the compiled program. We also initiate the ORAM memory by inserting the input x. Let x_{ORAM} be the resulting memory. We then compile $(P_{\mathsf{ORAM}}, x_{\mathsf{ORAM}})$ using $\mathcal{PKE}$ in the same way as in the basic version above. Namely, we use PPRF to generate multiple keys, and use each key to encrypt a single message, including the input x_{ORAM}. Denote the resulting instance by $(P_{\mathsf{hide}}, x_{\mathsf{hide}})$. Our randomized encoding of computation instance (P, x) is now $\widetilde{\Pi} = \mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$, where Π_{hide} is defined by P_{hide} and x_{hide}.

Note that ORAM security only holds when the adversary does not learn any content of the computation. Given the fact that $\mathsf{Ci}\mathcal{O}$ does not hide anything explicitly, it is unlikely that we can use the security of ORAM in a black-box way. In a previous seminal work [CHJV15], Canetti et al. provide a novel solution to this problem, and prove the security via a sequence of hybrids that "erase" the computation *forward* in time. Unfortunately, their solution incurs dependency on the space complexity of the RAM program, and thus is not fully succinct.

To solve this problem, we rely on the specific ORAM construction of [CP13] (referred to as CP-ORAM hereafter), and develop a puncturing technique to reason about the simulation. As in our basic version, we prove the security by a sequence of hybrids that "erase" the computation *backward in time*. At a very high level, to move from the i-th hybrid to the $(i-1)$-th hybrid, i.e., erase the computation at i-th time step, we "puncture" ORAM at time step i (i.e., the i-th memory access), which enables us to replace the access pattern by a simulated one at this time step. We can then move to the $(i-1)$-th hybrid by replacing the access pattern, erasing the memory content and computation, and undoing the "puncturing."

Puncturing the ORAM access pattern cleanly could be very subtle. Note that the access pattern at time step i is generated at the latest time step t' that access the same memory location as time step i; This last access time t' can be much smaller than i, so the puncturing may cause global changes in the computation. Thus, moving to the punctured hybrid, i.e., the $(i-1)$-th hybrid in the previous paragraph, requires a sequence of sub-hybrids that modifies the computation step by step. We therefore further introduce an auxiliary "*partially puncturing*" technique to achieve this goal.

This completes the overview of our main ideas. In the detailed proof in Appendix B.5, we will elaborate the above ideas.

Section Outline The remaining of this section will be organized as follows. We will first list all required building blocks in Section 7.1 and then review the CP-ORAM in Section 7.2. Next we provide the $\mathcal{RE}$ construction details in Section 7.3, and finally, prove the security in Appendix B.5.

7.1 Building Blocks

In our $\mathcal{RE}$ construction in Section 7.3, we will use several building blocks:

- Public-key encryption scheme $\mathcal{PKE} = \mathcal{PKE}.\{\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt}\}$ with IND-CPA security. Here we use $\ell_1 = \ell_1(\lambda)$ bits of randomness in $\mathcal{PKE}.\mathsf{Gen}$, and $\ell_2 = \ell_2(\lambda)$ bits of randomness in $\mathcal{PKE}.\mathsf{Encrypt}$ respectively; we let $\ell_{\mathsf{rnd}} = \ell_1 + \ell_2$, and assume the ciphertext length in $\mathcal{PKE}.\mathsf{Encrypt}$ is ℓ_3.
- Puncturable PRF scheme $\mathsf{PPRF} = \mathsf{PPRF}.\{\mathsf{Setup}, \mathsf{Puncture}, \mathsf{Eval}\}$ with key space $\mathcal{K}$, punctured key space $\mathcal{K}_{\mathsf{punct}}$, domain $[T] \cup ([T] \times [\log n])$, and range $\{0,1\}^{\ell_{\mathsf{rnd}}}$.
- Computation-trace indistinguishability obfuscation scheme in the RAM model, $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{Obf}, \mathsf{Eval}\}$.
- The oblivious RAM complier by Chung and Pass [CP13].

The computation-trace indistinguishability obfuscation for RAM has been introduced and constructed in Section 5.2. In the next subsection, we will briefly review the oblivious RAM compilation technique in [CP13].

We also use several primitives in vector form (Section 7.3), and they are defined here for completeness. Bold face symbols, such as $\mathbf{pk}$, $\mathbf{sk}$, $\mathbf{r}$, and $\mathbf{dummy}$, denote vectors, and a vector $\mathbf{v}$ concatenated with index i in square brackets $\mathbf{v}[i]$ denotes the i-th element in $\mathbf{v}$.

The public key encryption scheme is generalized to vector form as follows:

- $\mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r})$: The key generating algorithm takes as input the security parameter λ and a vector of randomness $\mathbf{r}$. It outputs vector of pairs of public and secret keys $(\mathbf{pk}, \mathbf{sk})$, where $(\mathbf{pk}[i], \mathbf{sk}[i]) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda, \mathbf{r}[i])$ for each i.
- $\mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}, \mathbf{m}; \mathbf{r})$: The encrypting algorithm takes as input a vector of public keys $\mathbf{pk}$, a vector of messages $\mathbf{m}$, and a vector of randomness $\mathbf{r}$. It outputs a vector of ciphertext $\mathbf{c}$, where $\mathbf{c}[i] = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}[i], \mathbf{m}[i], \mathbf{r}[i])$ for each i.
- $\mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}, \mathbf{c})$: The decrypting algorithm takes as input a vector of secret keys $\mathbf{sk}$ and a vector of ciphertext $\mathbf{c}$. It outputs a vector of messages $\mathbf{m}$, where $\mathbf{m}[i] = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}[i], \mathbf{c}[i])$ for each i.

The puncturable PRF scheme is generalized to vector form as follows, and we did not extend the puncturing procedure $\mathsf{Puncture}$.

- $\mathsf{PRF}(K, \mathbf{x})$: The pseudorandom function takes as input the key K and a vector $\mathbf{x}$. It outputs vector of pseudorandom numbers $\mathbf{r}$, where $\mathbf{r}[i] = \mathsf{PRF}(K, \mathbf{x}[i])$ for each i.
- $\mathsf{PPRF}.\mathsf{Eval}(K\{x'\}, \mathbf{x})$: The pseudorandom function takes as input the punctured key $K\{x'\}$ and a vector $\mathbf{x}$. It outputs vector of pseudorandom numbers $\mathbf{r}$, where $\mathbf{r}[i] = \mathsf{PPRF}.\mathsf{Eval}(K\{x'\}, \mathbf{x}[i])$ for each i.

7.2 Recap: The CP-ORAM

As mentioned in the section summary above, we use ORAM techniques to hide the access pattern. Our construction is essentially based on the ORAM compilation technique by Chung and Pass [CP13]. For short, we call it CP-ORAM. We believe that our construction can also be based on all existing tree-based ORAM compilation techniques [SCSL11]. Below, we give an overview of CP-ORAM. Please refer to [CP13] for more details. For better presentation, we will use both function program and next-step program to describe the programs that used in our construction.

In CP-ORAM (as well as all tree-based ORAM), the memory is stored in a complete binary tree (called ORAM tree), where each node of the tree is associated with a bucket that can store a few memory blocks. A position map Pos is used to record where each memory block is stored in the tree, where a block b is stored in a node somewhere along a path from the root to the leave indexed by $\mathsf{Pos}[b]$. Each memory block b in the ORAM tree also stores its index b and position map value $\mathsf{Pos}[b]$ as meta data. Each memory access (say, to block b) is performed by OACCESS, which (i) reads the position map value $pos = \mathsf{Pos}[b]$ and refresh $\mathsf{Pos}[b]$ to a random value, (ii) fetches and remove the block b from the path, (iii) updates the block content and puts it back to the root, and (iv) performs a flush operation along another random path pos' to move the blocks down along pos' (subject to the condition that each block is stored in the path specified by their position map value). At a high level, the security follows by the fact that the position map values are uniform and hidden from the adversary, and thus the access pattern of each OACCESS is simply a uniformly random path, which is trivial to simulate.

This completes the basic version of CP-ORAM which can be found in Section 7.2.1. The position map is large in the above basic version, but it is recursively outsourced to lower level ORAM structures to reduce its size in the full-fledged version. See Section 7.2.2 for more details.

7.2.1 Basic version: ORAM with $\Theta(n)$ registers

We here present the basic version of CP-ORAM. Consider memory be an array with n cells. The CP complier can transform a given program P into a new program P_o, which replaces the memory access instructions by the oblivious memory access algorithm OACCESS. More concretely, each memory access command READ(loc) and WRITE(loc, val) is replaced by corresponding commands OACCESS(loc, $\perp$) and OACCESS(loc, val) respectively which will be specified shortly. The new program P_o has the same registers as P and additionally has n/α registers for storing a *position map* Pos, plus a polylogarithmic number of additional *work* registers used by OACCESS, where $\alpha \geq 2$ is a constant to ensure that the position map is smaller than the memory size. In its external memory, P_o will maintain a complete binary tree Γ of depth $\log(n/\alpha)$. We index nodes in the tree by a binary string of length at most tree depth $\log(n/\alpha)$, where the root is indexed by the empty string ϵ, and each node indexed by γ has left and right children indexed $\gamma 0$ and $\gamma 1$, respectively. Each memory cell at location loc will be associated with a random leaf *pos* in the tree, specified by the position map Pos; as we shall see shortly, the memory cell loc will be stored at one of the nodes on the path from the root ϵ to the leaf *pos*. We assign a *block* of α consecutive memory cells to the same leaf; thus any memory cell loc corresponding to block $b = \lfloor \mathsf{loc}/\alpha \rfloor$ will be associated with leaf *pos* $= \mathsf{Pos}(b)$.

Each node in the tree is associated with a *bucket* which stores (at most) K tuples (b, pos, v), where v is the memory content of block b and *pos* is the leaf associated with the block b, and $K \in \omega(\log n) \cap \mathsf{poly}\log(n)$ is a parameter that will determine the security of the ORAM (thus each bucket stores $K(\alpha + 2)$ words). A bucket may store 0 to K valid tuples, and each empty slot in the bucket is denoted by `Empty` $= (\perp, \perp, \perp)$. We assume that all registers and memory cells are initialized with a special symbol $\perp$, and all buckets are initialized with `Empty`.

The following is a specification of the OACCESS(loc, val) procedure:

Update Position Map: Pick a uniformly random leaf $pos' \leftarrow [n/\alpha]$.

Fetch: Let $b = \lfloor \mathsf{loc}/\alpha \rfloor$ be the block containing memory cell loc (in the original database), and let $i = \mathsf{loc} \bmod \alpha$ be loc's component within the block b. We first look up the position of the block b using the position map: $pos = \mathsf{Pos}(b)$ and let $\mathsf{Pos}(b) = pos'$; if $pos = \perp$, then choose a uniformly random leaf $pos \leftarrow [n/\alpha]$.

Next, traverse the data tree from the root to the leaf *pos*, making exactly one READ and one WRITE operation for the memory bucket associated with each of the nodes along the path. More precisely, we read the memory content once, and then we either write it back (unchanged), or we simply "erase it" (writing $\perp$) so as to implement the following task: Search for a tuple of the form (b, pos, v) for the desired b, pos in any of the nodes during the traversal; if such a tuple is found, remove it from its place in the tree and set v to the found value, and otherwise set $v = \perp$. Finally, return the i-th component of v as the output of the OACCESS(loc, val) operation.

Put Back: If val is not $\perp$ (which means this is a WRITE), let v' be the string v but the i-th component is set to val. Otherwise, let $v' = v$. Add the tuple (b, pos', v') to the root ϵ of the tree. If there is not enough space left in the root bucket, abort and output `overflow`.

Flush: Pick a uniformly random leaf $pos'' \leftarrow [n/\alpha]$ and traverse the tree from the root to the leaf pos'', making exactly one READ and one WRITE operation for every memory cell associated with the nodes along the path so as to implement the following task: "Push down" each tuple (b, pos, v) read in the nodes traversed as far as possible along the path to pos'' while ensuring that the tuple is still on the path to its associated leaf *pos* (that is, the tuple ends up in the node γ = longest common prefix of *pos* and pos''.) Note that this operation can be performed trivially as long as the CPU has sufficiently many work registers to load two whole buckets into memory; since the bucket size is polylogarithmic, this is possible. If at any point some bucket is about to overflow, abort and output `overflow`.

We overloaded the second parameter of $\textsc{OAccess}$ to replace both $\textsc{Read}$ and $\textsc{Write}$ with the special symbol $\perp$ in the "Put Back" steps. Note that with all input including $\mathsf{val} = \perp$, $\textsc{OAccess}$ always outputs the original memory content of the cell loc; this feature will be useful in the "full-fledged" construction.

7.2.2 The full-fledged construction: ORAM with $\mathsf{poly}\log$ registers

The full-fledged construction of the $\mathsf{CP\text{-}ORAM}$ proceeds as above, except that instead of storing the position map in registers in the CPU, we now recursively store them in another ORAM (which only needs to operate on n/α memory cells, but still using buckets that store K tuples). Recall that each invocation of $\textsc{OAccess}$ requires reading one position in the position map and updating its value to a random leaf. That is, we need to perform a *single* recursive $\textsc{OAccess}$ call (recall that $\textsc{OAccess}$ updates the value in a memory cell, and returns the old value) to emulate the position map.

At the base of the recursion, when the position map is of constant size, we use the trivial ORAM construction which simply stores the position map in the CPU registers.

7.2.3 Notations for $\mathsf{CP\text{-}ORAM}$ compilation

We use $\mathsf{CP\text{-}ORAM}.\{\mathsf{Compile}, \mathsf{Eval}\}$ to denote the $\mathsf{CP\text{-}ORAM}$ scheme described above,where $\mathsf{Compile}$ and Eval denote the compilation and evaluation algorithms respectively. As mentioned before, the $\mathsf{CP\text{-}ORAM}$ can compile a given RAM program P into a new RAM program P_o by replacing the memory access instructions with the oblivious memory access algorithm $\textsc{OAccess}$. Formally, we write

$$P_o = \mathsf{CP\text{-}ORAM}.\mathsf{Compile}(P, \textsc{OAccess}).$$

To feed such ORAM on our $\mathcal{RE}$ construction, we slightly change the presentation of the oblivious access pattern algorithm in [CP13], $\textsc{OAccess}\{K_N\}$, in Algorithm 16. The involved randomness is produced by invoking PRF with a key K_N and the time parameter $\mathfrak{t}$. For simplicity, we use $\textsc{OAccess}$ instead of $\textsc{OAccess}\{K_N\}$ if not specified. Other detailed routines are abstracted as follows:

- $\textsc{Path}(d, pos)$ outputs the path $\mathbf{I}$ from root ϵ to leaf pos in the d-th ORAM tree Γ_d, where d is the recursion level of $\textsc{OAccess}$.
- $\textsc{FetchAndUpdate}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, pos, newpos)$ performs the "Fetch" and "Put Back" steps.
- $\textsc{Flush}(\mathbf{B}_{\mathsf{flush}}, pos'')$ performs "Flush" steps from tree root to leaf pos''. We remark that there is a negligible probability (w.r.t. bucket size K) that overflow occurs at this step. For simplicity, we just assume that $\textsc{OAccess}$ is supplied with a "good" randomness.[20]

We remark that $\textsc{OAccess}$ initializes the memory during the compilation, and the ORAM compiled program is still a RAM program. Therefore the evaluation algorithm Eval is a standard evaluation algorithm for RAM programs. In addition, we can construct a simulation algorithm $\textsc{SimOAccess}$ to generate statistically indistinguishable memory access pattern, and define a simulated RAM program $P_{o,sim}$ based on $\textsc{SimOAccess}$, and thus we write

$$P_{o,sim} = \mathsf{CP\text{-}ORAM}.\mathsf{Compile}(\textsc{SimOAccess}).$$

Note that the simulated access pattern is independent of the original program P.

[20]If the honest encoding overflowed, then an adversary could distinguish that from a simulated one. However, the overall probability to distinguish the honest is still negligible because the probability to overflow is negligible with good randomness.

Algorithm 16: $\text{OACCESS}\{K_N\}$: the recursive ORAM accessing function

```
Input  : $\mathsf{t}, d, \mathsf{loc}, \mathsf{val}$
Output : $oldval$
Data   : $K_N, \alpha, \mathit{MaxDepth}$ (Memory size $S = \alpha^{\mathit{MaxDepth}}$)
1  if $d \geq \mathit{MaxDepth}$ then
2      return $0$
3  Pick leaf $newpos$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \texttt{FetchR}))$ ;   // Update position map
4  $pos \leftarrow \text{OACCESS}(d+1, \lfloor \mathsf{loc}/\alpha \rfloor, newpos)$ ;   // Fetch
5  $\mathbf{I}_{\mathsf{fetch}} \leftarrow \text{PATH}(d, pos)$;
6  $\mathbf{B}_{\mathsf{fetch}} \leftarrow \text{READ}(\mathbf{I}_{\mathsf{fetch}})$;
7  $(\mathbf{B}^{\text{out}}_{\mathsf{fetch}}, oldval) \leftarrow \text{FETCHANDUPDATE}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, pos, newpos)$;
8  $\text{WRITE}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}^{\text{out}}_{\mathsf{fetch}})$;
9  Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \texttt{FlushR}))$ ;   // Flush
10 $\mathbf{I}_{\mathsf{flush}} \leftarrow \text{PATH}(d, pos'')$;
11 $\mathbf{B}_{\mathsf{flush}} \leftarrow \text{READ}(\mathbf{I}_{\mathsf{flush}})$;
12 $\mathbf{B}^{\text{out}}_{\mathsf{flush}} \leftarrow \text{FLUSH}(\mathbf{B}_{\mathsf{flush}}, pos'')$;
13 $\text{WRITE}(\mathbf{I}_{\mathsf{flush}}, \mathbf{B}^{\text{out}}_{\mathsf{flush}})$;
14 return $oldval$;
```

7.2.4 ORAM **compilation of a computation system**

Next, we describe how to use CP-ORAM to compile a computation system. We recall the notations above and for a given next-step program, and write

$$F_o = \text{CP-ORAM.Compile}(F, \text{OACCESS}).$$

Given a RAM computation system $\Pi = ((\mathsf{mem}^0, \mathsf{st}^0), F)$, we compile it into Π_o as follows. First, the compilation runs OACCESS to initialize mem^0_o for each non-empty memory cell in mem^0, and sets $\mathsf{st}^0_o = \texttt{Init}$. Then the compilation transforms next-step program F into a new next-step program F_o. Finally it outputs $\Pi_o := ((\mathsf{mem}^0_o, \mathsf{st}^0_o), F_o)$. We abuse the notations again, and write

$$\Pi_o = \text{CP-ORAM.Compile}(\Pi, \text{OACCESS}).$$

Similarly, based on SIMOACCESS, we can define $F_{o,sim}$ and write

$$F_{o,sim} = \text{CP-ORAM.Compile}(\text{SIMOACCESS})$$

Complied next-step program F_o For readability, we present F_o as a stateful, next-step program that reads or writes a complete ORAM tree path (rather than one memory cell) in each round, while the locations of memory cells on this path are denoted with a vector $\mathbf{I}$, as follows:

$$(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}).$$

Here F_o takes as input a round counter t of the given program F, a state st^{in}, a vector of input locations $\mathbf{I}^{\text{in}}$, a vector of input values $\mathbf{B}^{\text{in}}$, and outputs a state st^{out}, a vector of output locations $\mathbf{I}^{\text{out}}$, and a vector of output values $\mathbf{B}^{\text{out}}$. Note that the efficiency does not suffer too much in this vector notation because any path in the ORAM tree has length $\log(n)$, and it is straightforward to transform it to a cell-wise function. The ORAM compiled program has a multiplicative overhead q_o in computation time. We denote the time counter of program F by t and denote the time counter of F_o by t, where $\mathsf{t} = \lceil t/q_o \rceil$. In the remaining of this section, both time metrics are used when simulating access patterns.

We further abuse the notation of the READ and WRITE operations in OACCESS such that they now work on vector of locations $\mathbf{I}$ and values $\mathbf{B}$. In particular, F_o has the following output cases depending on the input memory command of a round:

1. Neither READ nor WRITE command: outputs both $\mathbf{I}$ and $\mathbf{B}$ as an empty set.
2. READ command: outputs $\mathbf{I}$ as a vector of locations to read and $\mathbf{B}$ as an empty set.
3. WRITE command: outputs $\mathbf{I}$ as a vector of locations to write and $\mathbf{B}$ as a vector of values to write.

7.3 Construction for $\mathcal{RE}$-RAM

A randomized encoding of a computation instance (P, x) hides everything about the computation instance except its output $y = P(x)$ and runtime t^*. This requires hiding both the memory content and the access pattern of the computation. We follow a natural construction idea: we use public-key encryptions to hide the memory content (including the input) and oblivious RAM to hide the access pattern, and then use $\mathsf{Ci}\mathcal{O}$ to obfuscate the compiled computation instance. Namely, our $\mathcal{RE}$ encoding algorithm outputs $\mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$ as the encoding, where Π_{hide} is defined by P_{hide} and x_{hide}: P_{hide} is a $\mathcal{PKE}$ and ORAM compiled version of P, and x_{hide} is an encrypted version of x. Namely, P_{hide} outputs encrypted CPU states and memory contents at each time step, and uses ORAM to compile its memory access.

More concretely, our construction of $\mathcal{RE}$ in the RAM model is split into four major steps: **(i)** given a RAM program P and its input x, we interpret it as a RAM computation Π; **(ii)** we compile Π into Π_o using CP-ORAM compiler to hide the access pattern; **(iii)** we transform Π_o into Π_e, which further hides the memory content in the computation system; and **(iv)** we obfuscate Π_e into ENC using $\mathsf{Ci}\mathcal{O}$-RAM. Formally, we construct our $\mathcal{RE} = \mathcal{RE}.\{\mathsf{Encode}, \mathsf{Decode}\}$ for the RAM program P and input x as follows:

Encoding algorithm ENC $\leftarrow \mathcal{RE}.\mathsf{Encode}(P, x, 1^\lambda)$: The encoding algorithm takes the following steps to generate the encoding ENC.

– Upon receiving the description of RAM program P and an input value x, first, the encoding algorithm transforms them into a computation system. It represents P into a next-step program F, and stores x into the memory, i.e., sets $\mathsf{mem}^0 := x$. Then it sets $\mathsf{st}^0 := \mathtt{Init}$, and defines the following computation system in the RAM model
$$\Pi = ((\mathsf{mem}^0, \mathsf{st}^0),\ F).$$

– Second, the encoding algorithm hides the access pattern in the computation system. It randomly chooses puncturable PRF key $K_N \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$. Then it runs the CP-ORAM compilation described in Section 7.2, i.e., $\Pi_o = \mathsf{CP\text{-}ORAM.Compile}(\Pi, \text{OACCESS}\{K_N\})$ and obtains
$$\Pi_o = ((\mathsf{mem}_o^0, \mathsf{st}_o^0),\ F_o).$$

– Third, the encoding algorithm further hides the CPU state and the memory content. That is, it transforms Π_o into
$$\Pi_e = ((\mathsf{mem}_e^0, \mathsf{st}_e^0),\ F_e).$$

Here the encoding algorithm randomly chooses puncturable PRF key $K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$, and generates an initial configuration of the encrypted version of memory and CPU state as follows:

To initialize memory mem_e^0, the encoding algorithm parses mem_o^0 as ORAM trees $\{\Gamma\}$, and for each Γ it further parses all paths $\mathbf{I}$ from root to leaf. For each path $(\mathbf{I}, \mathbf{B})$ with its index $\mathbf{I}$ and buckets $\mathbf{B}$, the encoding algorithm computes

$$(\mathbf{r}_1^0, \mathbf{r}_2^0) = \mathsf{PRF}(K_E, (\mathbf{lw}^0, h(\mathbf{I}))) \text{ where } \mathbf{lw}^0 = \mathbf{0},$$
$$(\mathbf{pk}^0, \mathbf{sk}^0) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^0),$$
$$\mathbf{B}[i] = \begin{cases} \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}^0[i], \mathbf{B}[i]), & \text{if } \mathbf{B}[i] \text{ stores any valid block} \\ \mathbf{B}[i], & \text{otherwise,} \end{cases}$$

where h is a function to compute the "height" of elements in vector $\mathbf{I}$. That is, for any vector $\mathbf{I}$ of length $|\mathbf{I}|$, define $h(\mathbf{I}) = (1, 2, \ldots, |\mathbf{I}|)$. For each non-empty encrypted bucket $\boxed{\mathbf{B}}[i]$, store $(\boxed{\mathbf{B}}[i], 0)$ to its corresponding location $\mathbf{I}[i]$ in mem_e^0, which are the encrypted ORAM trees. Because many buckets are empty and never touched while OACCESS initializes mem_o^0, we represent mem_o^0 and mem_e^0 in sparse arrays for efficiency, where only those non-empty buckets are stored and processed with encryption. Therefore, the encoding time and space of mem_e^0 are both efficient.

In addition, the encoding algorithm computes $(r_3^0, r_4^0) = \mathsf{PRF}(K_E, 0)$, $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^0)$, and $\boxed{\mathsf{st}}^0 \leftarrow \mathcal{PKE}.\mathsf{Encrypt}(pk_{\mathsf{st}}, \mathsf{st}_o^0)$. Then it sets $\mathsf{st}_e^0 = (\boxed{\mathsf{st}}^0, 0)$.

The encoding algorithm then upgrades F_o, into a more sophisticated next-step program F_e which decrypts its inputs, performs the computation of Π_o, and encrypts its outputs. Please refer to Algorithm 17 for more details of F_e. Because there are non-encrypted empty buckets in mem_e^0, the procedure $\mathcal{PKE}.\mathsf{Decrypt}$ to decrypt $\boxed{\mathbf{B}}^{\text{in}}$ in F_e is augmented to ignore any empty bucket in mem_e^0, that is for each i

$$\mathbf{B}^{\text{in}}[i] = \begin{cases} \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}[i], \boxed{\mathbf{B}}^{\text{in}}[i]) & \text{if } \boxed{\mathbf{B}}^{\text{in}}[i] \neq \text{ empty bucket} \\ \boxed{\mathbf{B}}^{\text{in}}[i] & \text{otherwise.} \end{cases}$$

This technique is applied to eliminate the dependency of memory size S from the complexity of encoding size and time, and we summarize it in Table 4.

Note that F_e (Algorithm 17) abuses the notation of PRF, $\mathcal{PKE}.\mathsf{Gen}$, $\mathcal{PKE}.\mathsf{Encrypt}$, $\mathcal{PKE}.\mathsf{Decrypt}$, which computes on a vector of inputs and returns a vector of outputs. Please refer to Section 7.1 for formal description.

– Finally, the encoding algorithm computes ENC $\leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \Pi_e)$ and outputs ENC.

Observation	**Technique to encode input efficiently**	**Corresponding shorthand in program F_e**
Sparse input data in ORAM tree	Encrypt only those buckets with data	For each encrypted bucket $\boxed{\mathbf{B}}^{\text{in}}$, decrypt ciphertext except empty bucket

Table 4: Techniques to improve encoding efficiency

Decoding algorithm $y \leftarrow \mathcal{RE}.\mathsf{Decode}(\textbf{ENC}, 1^\lambda, T, S)$: Upon receiving the encoding ENC, the decoding algorithm executes $\mathsf{Ci}\mathcal{O}.\mathsf{Eval}(\text{ENC})$. If the decoding algorithm does not terminate in T steps, it outputs $y := \perp$. Otherwise, if it terminates at step t^*, and obtains $\left(\widetilde{\mathsf{mem}}^{t^*}, \widetilde{\mathsf{st}}^{t^*}\right)$ where $\widetilde{\mathsf{st}}^{t^*} = (\mathtt{halt}, y)$ then it outputs y.

It is straightforward to verify the correctness of the above construction. Next, we describe the efficiency and then present a theorem for its security.

Efficiency Let $|F|$ be the description size of program F, n be the size of input x, F computes on x with time and space bound T and S. Assuming $\mathsf{Ci}\mathcal{O}$ has compilation time $O(\mathsf{poly}(|F|) + n \log S)$, compilation size $O(\mathsf{poly}(|F|) + n)$, evaluation time $\tilde{O}(T \cdot \mathsf{poly}(|F|))$, and evaluation space proportional to S. Observing only constant amount of information is hardwired throughout our security proof (Appendix B.5), our $\mathcal{RE}$ has following complexity:

– Encoding time is $\tilde{O}(\mathsf{poly}|F| + n)$.
– Encoding size is $\tilde{O}(\mathsf{poly}|F| + n)$.
– Decoding time is $\tilde{O}(T \cdot \mathsf{poly}(|F|))$.
– Decoding space is $\tilde{O}(S)$.

Security We now state the following theorem that the randomized encoding scheme $\mathcal{RE}$ described above is secure. Please refer to Appendix A.2 for the security definition of randomized encoding scheme.

Theorem 7.1. *Let $\mathcal{PKE}$ be an IND-CPA secure public key encryption scheme,* $\mathsf{Ci}\mathcal{O}$ *be a computation-trace indistinguishability obfuscation scheme in the RAM model,* PRF *be a secure puncturable PRF scheme; then* $\mathcal{RE}$ *is a secure randomized encoding scheme.*

The security proof can be found in Appendix B.5.

Algorithm 17: F_e

```
   Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\boxed{\mathsf{st}}^{\mathrm{in}}, t)$,  $\tilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, (\boxed{\mathbf{B}}^{\mathrm{in}}, \mathbf{lw}^{\mathrm{in}}))$
   Data  : $T, K_E, K_N$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$ ;          // $q_o$ is the ORAM compilation overhead
2  Compute $(\mathbf{r}_1^{\mathrm{in}}, \mathbf{r}_2^{\mathrm{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{in}}, h(\mathbf{I}^{\mathrm{in}})))$ ;   // For any vector $\mathbf{I}$ of length $|\mathbf{I}|$, define $h(\mathbf{I}) = (1, 2, \ldots, |\mathbf{I}|)$
3  Compute $(\mathbf{pk}^{\mathrm{in}}, \mathbf{sk}^{\mathrm{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{in}})$;
4  Compute $\mathbf{B}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\mathrm{in}}, \boxed{\mathbf{B}}^{\mathrm{in}})$;
5  Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
6  Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
7  Compute $\mathsf{st}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\mathrm{in}})$;

8  Compute $(\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;

9  Set $\mathbf{lw}^{\mathrm{out}} = (t, \ldots, t)$;
10 Compute $(\mathbf{r}_1^{\mathrm{out}}, \mathbf{r}_2^{\mathrm{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{out}}, h(\mathbf{I}^{\mathrm{out}})))$;
11 Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{out}})$;
12 Compute $\boxed{\mathbf{B}}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\mathrm{out}}; \mathbf{r}_2^{\mathrm{out}})$;
13 if $\mathsf{st}^{\mathrm{out}} \neq (\mathtt{halt}, \cdot)$ then
14     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
15     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
16     Compute $\boxed{\mathsf{st}}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\mathrm{out}}; r_4^t)$;
17 else
18     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = \mathsf{st}^{\mathrm{out}}$
19 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\boxed{\mathsf{st}}^{\mathrm{out}}, t+1)$,  $\tilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{out}}, (\boxed{\mathbf{B}}^{\mathrm{out}}, \mathbf{lw}^{\mathrm{out}}))$ ;
```

8 Constructing $\mathcal{RE}$ in the PRAM Model ($\mathcal{RE}$-PRAM)

In this section, we describe our construction of randomized encoding for PRAM ($\mathcal{RE}$-PRAM). As $\mathcal{RE}$-RAM, the main goal of $\mathcal{RE}$-PRAM is to hide states and memory access pattern. Recall the construction of $\mathcal{RE}$-RAM in Section 7 based on $\mathsf{Ci}\mathcal{O}$-RAM, tree-based ORAM, and $\mathcal{PKE}$; naturally, we use $\mathsf{Ci}\mathcal{O}$-PRAM, tree-based oblivious PRAM (OPRAM), and $\mathcal{PKE}$ as building blocks to achieve $\mathcal{RE}$ simulation security.

At a high level, our $\mathcal{RE}$-PRAM construction works as follows.

– We first use OPRAM compiler to hide the access pattern. Given a computation instance Π defined by (P, x), we compile P using an OPRAM compiler with randomness supplied by a puncturable PRF. Let P_{OPRAM} be the compiled program. We also initiate the OPRAM memory by inserting the input x. Let x_{OPRAM} be the resulting memory.
– We then compile $(P_{\mathsf{OPRAM}}, x_{\mathsf{OPRAM}})$ using $\mathcal{PKE}$ in the same way as in the RAM version above. Namely, we use PPRF to generate multiple keys, and use each key to encrypt a single message, including the input x_{OPRAM}. Denote the resulting instance by $(P_{\mathsf{hide}}, x_{\mathsf{hide}})$.
– The randomized encoding of computation instance Π is $\widetilde{\Pi} = \mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$, where $\mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$ is defined by $(P_{\mathsf{hide}}, x_{\mathsf{hide}})$.

To build $\mathcal{RE}$-PRAM, we use the oblivious PRAM compiler by Boyle, Chung, and Pass [BCP16] (BCP-OPRAM) and the other building blocks which are identical to those in Section 7.1. The security proof of the $\mathcal{RE}$-PRAM construction also follows identical steps, where we prove the security by a sequence of hybrids that erases the computation backward in time, and argue simulation of access patterns by generalizing the puncturing ORAM argument to puncturing BCP-OPRAM. However, there are two natural issues in the arguments of generalization: (**i**) As the OPACCESS algorithm of BCP-OPRAM is more complicated, we need to be slightly careful in defining the simulated encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$. (**ii**) To avoid dependency on the number m of CPUs, we need to gradually handle a single CPU at a time in the hybrids to puncture OPRAM.

Section Outline We will review BCP-OPRAM compilation technique in the next subsection. The construction of $\mathcal{RE}$-PRAM and its proof sketch will be shown in Section 8.2 and Appendix B.6.

8.1 Recap: The BCP-OPRAM

For hiding access pattern, the security of our $\mathcal{RE}$-PRAM must rely heavily on OPRAM as a building block. We first briefly review BCP's OPACCESS [BCP16], and then show its *puncturability* property similar to that of OACCESS where the randomness is generated from a PPRF.

The BCP OPRAM Construction The OPRAM compiler, on input $m, n \in \mathbb{N}$ and an m-processor PRAM program P with memory size n, outputs a program P' that is identical to P, except that each $\mathsf{access}(r, v)$ operation is replaced by a sequence of operations defined by subroutine OPACCESS(r, v), which is described as follows.

The OPACCESS procedure begins with m CPUs, each requesting data cell r (within some block b) and some action to be taken (either $\perp$ denoting a read, or v denoting rewriting cell r with value v). The primary challenges in implementing oblivious parallel data accesses within the tree-based ORAM structure ([SCSL11, CP13]) are in handling collisions between processor accesses, and in reinserting data to the ORAM (and flushing data down the tree) in parallel. OPACCESS addresses these challenges by the following sequence of tasks:

1. **Resolve Conflict:**
   - Choose one representative CPU per requested data block b (in the real database). This representative will perform the real data fetch and computation on b in later steps, while the other CPUs will simply make "dummy" accesses of the ORAM structure.
   - Aggregate all CPU instructions to take place on each requested block b.
2. **Read/Write Position Map:**
   - Each representative CPU: Sample a fresh random leaf id ℓ'. Perform a (recursive) read/write access command on the position map database $\ell \leftarrow$ OPACCESS(b_i, ℓ') to fetch the current position map value ℓ and rewrite it with the newly sampled value ℓ'.
   - Each dummy CPU: Perform a dummy access to an arbitrary cell in the position map database, say the first. (Recall that the position map database is itself protected by a layer of ORAM). That is, execute $\ell \leftarrow$ OPACCESS$(1, \emptyset)$, and ignore the read value ℓ.
3. **Look Up Current Memory Values:** Each representative CPU fetches memory from ORAM database nodes corresponding to accessing its desired data block b (i.e., the collection of buckets down the relevant path in the ORAM tree) and copies the values into local memory. We denote this representative CPU by $\mathsf{rep}(b)$. Non-chosen (dummy) CPUs choose a random path ℓ (independent of the position map above) and make analogous dummy data fetches along the path to ℓ, ignoring all read values. Recall that simultaneous data *reads* do not yield conflicts.
4. **Remove Old Data:** Consider the paths down the ORAM tree accessed in the previous step.

  - Aggregate instructions across CPUs accessing the same "buckets" of memory on the server side. Each representative CPU $\mathsf{rep}(b)$ begins with the instruction of "remove block b if it occurs" and dummy CPUs hold the empty instruction. (Aggregation is as before, but at bucket level instead of the block level).
  - For each bucket to be modified, the CPU with the *smallest* id from those who wish to modify it executes the aggregated block-removal instructions for the bucket.

5. **Insert Updated Data into Database in *Parallel*:** All CPUs execute a parallel insertion procedure into the ORAM database at the appropriate level (corresponding to the number of active CPUs) in order to insert the updated data tuples (b, ℓ', v') with new leaf node ℓ' as sampled in Step 1 and new value v' into the bucket along the path to ℓ'.

6. **Flush the ORAM Database:** In parallel, each CPU initiates an independent flush of the ORAM tree. (Recall that this corresponds to selecting a random path down the tree, and pushing all data blocks in this path as far as they will go). To implement the simultaneous flush commands, as before, commands are aggregated across CPUs for each bucket to be modified, and the CPU with the *smallest* id performs the corresponding aggregated set of commands. (For example, all CPUs will wish to access the root node in their flush; the aggregation of all corresponding commands to the root node data will be executed by the lowest-numbered CPU who wishes to access this bucket, which is CPU 1 in this case).

7. **Return Output:** Each representative CPU $\mathsf{rep}(b)$ communicates the *original* value of the data block b to the subset of CPUs that originally requested it.

As a result, the BCP-OPRAM compiler enjoys the same advantages as CP-ORAM by finishing the above tasks. Intuitively, CP-ORAM and BCP-OPRAM must have the same property, puncturability.

Puncturability of BCP-OPRAM In the above OPACCESS, observe that the location to be looked up (in Step 3) only depends on the previous fresh random sample ℓ' (in Step 2). Therefore, if previous random sample ℓ' is information-theoretically hidden, the look up step can be simulated. Specifically, the punctured OPRAM program erases that block blk^* containing ℓ' at that time step t' such that ℓ' is generated by PPRF and does not recursively write ℓ' to the position map. With this punctured OPRAM (and PPRF key punctured at the corresponding point of ℓ'), the step looking for blk^* can be simulated indistinguishably with a uniformly random OPRAM tree path. Hence, BCP-OPRAM achieves puncturability similar to that of ORAM described in Appendix B.5.4.

8.2 Construction for $\mathcal{RE}$-PRAM

A randomized encoding of a computation instance (P, x) hides both the memory content and the access pattern of the computation except for its output $y = P(x)$ and runtime t^*. Conceptually, we follow the same natural idea to use public-key encryption to hide the memory content (including the input) and oblivious PRAM to hide the access pattern, and then use Ci$\mathcal{O}$-PRAM to obfuscate the compiled computation instance. Namely, our $\mathcal{RE}$ encoding algorithm outputs $\mathsf{Ci}\mathcal{O}(\Pi_{\mathsf{hide}})$ as the encoding, where Π_{hide} is defined by $(P_{\mathsf{hide}}, x_{\mathsf{hide}})$, P_{hide} is a $\mathcal{PKE}$ and OPRAM compiled version of P, and x_{hide} is an encrypted version of x. P_{hide} outputs encrypted CPU states and memory contents at each time step, and uses OPRAM to compile its memory access.

Our construction of $\mathcal{RE}$ in the PRAM model is split into four major steps: **(i)** given a PRAM program P and its input x, we interpret it as a PRAM computation Π; **(ii)** we compile Π into Π_o using BCP-OPRAM compiler to hide the access pattern; **(iii)** we transform Π_o into Π_e, which further hides the memory content in the computation system; and finally **(iv)** we obfuscate Π_e into ENC using Ci$\mathcal{O}$-PRAM. Formally, we construct our $\mathcal{RE} = \mathcal{RE}.\{\mathsf{Encode}, \mathsf{Decode}\}$ for the PRAM program P and input x as follows:

Encoding algorithm ENC $\leftarrow \mathcal{RE}.\mathsf{Encode}(P, x, 1^\lambda)$**:** The encoding algorithm takes the following steps to generate the encoding ENC.

- Upon receiving the description of PRAM program P and an input value x, first, the encoding algorithm transforms them into a computation system Π. It represents P into a next-step program F, and stores x into the memory, i.e., sets $\mathsf{mem}^0 := x$. Then it sets $\mathsf{st}_k^0 := \perp$ for all k, $1 \le k \le m$, and defines the following computation system in the PRAM model
$$\Pi = ((\mathsf{mem}^0, \{\mathsf{st}_k^0\}_{k=1}^m),\ F).$$
- Second, the encoding algorithm hides the access pattern in the computation system. It chooses puncturable PRF key $K_N \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$. Then it runs the BCP-OPRAM compilation described in Section 8.1, i.e., $\Pi_o = \mathsf{BCP\text{-}OPRAM.Compile}(\Pi, \text{OPACCESS}\{K_N\})$, and obtains
$$\Pi_o = ((\mathsf{mem}_o^0, \{\mathsf{st}_{o,k}^0\}_{k=1}^m),\ F_o),$$
where $\mathsf{st}_{o,k}^0 = \mathsf{st}_o^0$ such that all CPUs have the same OPRAM state.
- Third, the encoding algorithm further hides the CPU state and the memory content. That is, it transforms Π_o into
$$\Pi_e = ((\mathsf{mem}_e^0, \{\mathsf{st}_{e,k}^0\}_{k=1}^m),\ F_e).$$
Here the encoding algorithm chooses puncturable PRF key $K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$, and generates an initial configuration of the encrypted version of memory and CPU state as follows:

  To initialize memory mem_e^0, the encoding algorithm parses mem_o^0 as trees Γ, and then for each Γ it further parses all paths $\mathbf{I}$ from root to leaf. For each vector $\mathbf{I}$, the encoding algorithm computes
$$(\mathbf{r}_1^0, \mathbf{r}_2^0) = \mathsf{PRF}(K_E, (\mathbf{lw}^0, h(\mathbf{I}))) \text{ where } \mathbf{lw}^0 = \mathbf{0},$$
$$(\mathbf{pk}^0, \mathbf{sk}^0) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^0),$$
$$\boxed{\mathbf{B}}[i] = \begin{cases} \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}^0[i], \mathbf{B}[i]) & \text{if } \mathbf{B}[i] \text{ stores any valid block} \\ \mathbf{B}[i] & \text{otherwise,} \end{cases}$$
where $\mathbf{B}[i]$ denotes the i-th element (which is also a bucket here) in vector $\mathbf{B}$, and h is a function to compute the "height" of elements in vector $\mathbf{I}$. That is, for any vector $\mathbf{I}$ of length $|\mathbf{I}|$, define $h(\mathbf{I}) = (1, 2, \ldots, |\mathbf{I}|)$. For each non-empty $\boxed{\mathbf{B}}$, store $(\boxed{\mathbf{B}}, \mathbf{lw}^0)$ to its corresponding path $\mathbf{I}$ in mem_e^0.

  In addition, the encoding algorithm sets $\mathsf{st}_{e,k}^0 = \mathsf{st}_{o,k}^0$ for all $k \in [m]$. Note that each CPU holds the same non-encrypted $\mathsf{st}_{e,k}^0$ because st_k^0 is only $\perp$ for all k. To work with such initialization, the procedure $\mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \cdot)$ for decryption of states (in F_e) is augmented to ignore non-encrypted special value $\perp$ as follows:
$$\mathsf{st}_{\mathtt{A}}^{\text{in}} = \begin{cases} \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}_{\mathtt{A}}^{\text{in}}) & \text{if } \boxed{\mathsf{st}}_{\mathtt{A}}^{\text{in}} \neq \perp \\ \perp & \text{otherwise.} \end{cases}$$
These techniques are applied to eliminate the dependency of memory size S and number of CPUs m from the complexity of encoding size and time, and we summarize them in Table 5.

  The encoding algorithm then upgrades F_o into a more sophisticated next-step program F_e which decrypts its inputs, performs the computation of Π_o, and encrypts its outputs. Please refer to Algorithm 18 for more details of F_e.
- Finally, the encoding algorithm computes ENC $\leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \Pi_e)$ and outputs ENC.

Decoding algorithm $y \leftarrow \mathcal{RE}.\mathsf{Decode}(\textbf{ENC}, 1^\lambda, T, S)$**:** Upon receiving the encoding ENC, the decoding algorithm executes $\mathsf{Ci}\mathcal{O}.\mathsf{Eval}(\text{ENC})$. If the decoding algorithm does not terminate in T steps, then it outputs $y := \perp$. Otherwise, if it terminates at step t^*, and obtains $\left(\widetilde{\mathsf{mem}}^{t^*}, \widetilde{\mathsf{st}}_1^{t^*}\right)$ where $\widetilde{\mathsf{st}}_1^{t^*} = (\mathtt{halt}, y)$, then it outputs y by cpu1, a special CPU designated to output the result.

Observation	**Technique to encode input efficiently**	**Corresponding shorthand in program F_e**
Input data in ORAM tree structure is sparse	Encrypt only those buckets with data	For each encrypted bucket $\boxed{\mathbf{B}}^{\text{in}}_{\mathtt{A}}$, decrypt ciphertext except empty bucket
All m initial CPU states are the same empty value	Leave the state in plaintext	Decrypt ciphertext $\boxed{\mathsf{st}}^{\text{in}}_{\mathtt{A}}$ except empty state

Table 5: Techniques to improve encoding efficiency

Efficiency Let $|F|$ be the description size of program F, n be the description size of initial memory mem^0, m be the total number of CPUs, T and S be time and space bound. According to $\mathsf{Ci}\mathcal{O}$-PRAM, assume that $\mathsf{Ci}\mathcal{O}$ has compilation time $\tilde{O}(\mathsf{poly}(|F|)+n)$ and compilation size $\tilde{O}(\mathsf{poly}(|F|)+n)$, and parallel evaluation time $\tilde{O}(T \cdot \mathsf{poly}(|F|))$ and evaluation space $\tilde{O}(m+S)$. However, there remains polylogarithmic overhead of OPRAM including computation overhead $\mathsf{poly}\log m \mathsf{poly}\log S$ and space overhead $\omega(\log S)$. Finally, our $\mathcal{RE}$ construction has following complexity:

- Encoding time is $\tilde{O}(\mathsf{poly}(|F|)+n)$.
- Encoding size is $\tilde{O}(\mathsf{poly}(|F|)+n)$.
- Parallel decoding time is $\tilde{O}(T \cdot \mathsf{poly}(|F|))$.
- Decoding space is $\tilde{O}(m+S)$.

Security We state the following theorem that the randomized encoding scheme $\mathcal{RE}$ described above is secure. Please refer to Appendix A.2 for the security definition of randomized encoding schemes.

Theorem 8.1. *Let $\mathcal{PKE}$ be a semantically-secure public key encryption scheme, $\mathsf{Ci}\mathcal{O}$ be a computation-trace indistinguishability obfuscation scheme in the PRAM model, and PRF be a secure puncturable PRF scheme; then $\mathcal{RE}$ is a secure randomized encoding scheme in the PRAM model.*

The proof sketch can be found in Appendix B.6.

9 Extensions

In this section, we extend our results in previous sections to suit for several important scenarios of delegation of computations. One of our major extensions is to let $\mathcal{RE}$ support persistent database (PDB). This can be achieved by first defining and constructing the corresponding variants of $\mathsf{Ci}\mathcal{O}$ with PDB. Next, recall that ordinary $\mathcal{RE}$ only provides input and program privacy, and produces a short output in the clear. For practical scenarios of delegation of computation, other properties such as long output, output hiding, and output verifiability may be desirable. Thus, we will demonstrate how we can obtain these extensions by possibly using other primitives such as encryption and signatures.

9.1 $\mathsf{Ci}\mathcal{O}$ with Persistent Database

In the persistent database setting, we consider an initial memory and a sequence of programs which work on the memory content processed and left over by the previous program. Recall that $\mathsf{Ci}\mathcal{O}$ in some sense forces the evaluator to evaluate an obfuscated program as intended to produce the intended computation trace. In the persistent database setting, we further require that the sequence of programs is executed in the intended order.

9.1.1 Definition

Let a computation system $\Pi \in \mathcal{P}$ be composed of an initial database and many programs written as $\Pi = (\mathsf{mem}^{0,0}, \{F_{\mathsf{sid}}\}_{\mathsf{sid}=1}^{l})$ where sid denotes the session identity and l denotes the total number of programs. Each

Algorithm 18: F_e in $\mathcal{RE}$-PRAM

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}}_{e,\mathtt{A}} = (\mathtt{A}, \mathsf{st}^{\mathrm{in}}_{e,\mathtt{A}}, t), \widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}_{\mathtt{A}}, (\boxed{\mathbf{B}}^{\mathrm{in}}_{\mathtt{A}}, \mathbf{lw}^{\mathrm{in}}_{\mathtt{A}}))$
Data  : $T, K_E, K_N$
Compute $\mathsf{t} = \lceil t/q_o \rceil$;
Compute $(\mathbf{r}^{\mathrm{in}}_1, \mathbf{r}^{\mathrm{in}}_2) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{in}}_{\mathtt{A}}, h(\mathbf{I}^{\mathrm{in}}_{\mathtt{A}})))$;
Compute $(\mathbf{pk}^{\mathrm{in}}, \mathbf{sk}^{\mathrm{in}}) = \mathcal{PKE}.\mathsf{Setup}(1^\lambda; \mathbf{r}^{\mathrm{in}}_1)$;
Compute $\mathbf{B}_{\mathtt{A}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\mathrm{in}}, \boxed{\mathbf{B}}^{\mathrm{in}}_{\mathtt{A}})$;
Compute $(r^{t-1}_3, r^{t-1}_4) = \mathsf{PRF}(K_E, t-1)$;
Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Setup}(1^\lambda; r^{t-1}_3)$;
Parse $\mathsf{st}^{\mathrm{in}}_{e,\mathtt{A}}$ as $(\boxed{\mathsf{st}}^{\mathrm{in}}_{\mathtt{A}} || \mathsf{st}^{\mathrm{in}}_{o,\mathtt{A}})$;
Compute $\mathsf{st}^{\mathrm{in}}_{\mathtt{A}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\mathrm{in}}_{\mathtt{A}})$;
Set $\widehat{\mathsf{st}}^{\mathrm{in}}_{\mathtt{A}} = (\mathsf{st}^{\mathrm{in}}_{\mathtt{A}} || \mathsf{st}^{\mathrm{in}}_{o,\mathtt{A}})$;

Compute $r_N = \mathsf{PRF}(K_N, t)$;
Compute $(\widehat{\mathsf{st}}^{\mathrm{out}}_{\mathtt{A}}, \mathbf{I}^{\mathrm{out}}_{\mathtt{A}}, \mathbf{B}^{\mathrm{out}}_{\mathtt{A}}) = F_o(\mathsf{t}, \mathtt{A}, \widehat{\mathsf{st}}^{\mathrm{in}}_{\mathtt{A}}, \mathbf{I}^{\mathrm{in}}_{\mathtt{A}}, \mathbf{B}^{\mathrm{in}}_{\mathtt{A}}, r_N)$;

Parse $\widehat{\mathsf{st}}^{\mathrm{out}}_{\mathtt{A}}$ as $(\mathsf{st}^{\mathrm{out}}_{\mathtt{A}} || \mathsf{st}^{\mathrm{out}}_{o,\mathtt{A}})$;
Set $\mathbf{lw}^{\mathrm{out}}_{\mathtt{A}} = (t, \ldots, t)$;
Compute $(\mathbf{r}^{\mathrm{out}}_1, \mathbf{r}^{\mathrm{out}}_2) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{out}}_{\mathtt{A}}, h(\mathbf{I}^{\mathrm{out}}_{\mathtt{A}})))$;
Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Setup}(1^\lambda; \mathbf{r}^{\mathrm{out}}_1)$;
Compute $\boxed{\mathbf{B}}^{\mathrm{out}}_{\mathtt{A}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\mathrm{out}}_{\mathtt{A}}; \mathbf{r}^{\mathrm{out}}_2)$;
if $\mathsf{st}^{\mathrm{out}}_{\mathtt{A}} \neq (\mathtt{halt}, \cdot)$ then
    Compute $(r^t_3, r^t_4) = \mathsf{PRF}(K_E, t)$;
    Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Setup}(1^\lambda; r^t_3)$;
    Compute $\boxed{\mathsf{st}}^{\mathrm{out}}_{\mathtt{A}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\mathrm{out}}_{\mathtt{A}}; r^t_4)$;
    Set $\mathsf{st}^{\mathrm{out}}_{e,\mathtt{A}} = (\boxed{\mathsf{st}}^{\mathrm{out}}_{\mathtt{A}} || \mathsf{st}^{\mathrm{out}}_{o,\mathtt{A}})$;
else
    if all agents output $\mathsf{st}^{\mathrm{out}}_{\mathtt{A}} = (\mathtt{halt}, \perp)$ then a special CPU agent returns the output $y$;
    else $\mathtt{A}$ returns $\mathsf{st}^{\mathrm{out}}_{\mathtt{A}}$ as $(\mathtt{halt}, \cdot)$;
Output $\widetilde{\mathsf{st}}^{\mathrm{out}}_{e,\mathtt{A}} = (\mathtt{A}, \mathsf{st}^{\mathrm{out}}_{e,\mathtt{A}}, t+1), \widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{out}}_{\mathtt{A}}, \mathbf{D}^{\mathrm{out}}_{\mathtt{A}})$, where $\mathbf{D}^{\mathrm{out}}_{\mathtt{A}} = (\boxed{\mathbf{B}}^{\mathrm{out}}_{\mathtt{A}}, \mathbf{lw}^{\mathrm{out}}_{\mathtt{A}})$;
```

stateful function F_{sid} has its program and state hardwired. For simplicity, we adopt a convention that the label of the database and the state are set to 1) $(\mathsf{sid}-1, 0)$ at the beginning of session sid, 2) $(\mathsf{sid}-1, i)$ where $i \neq 0$ in the duration of session sid, and finally 3) $(\mathsf{sid}, 0)$ in the termination stage.

Definition 9.1 ($\mathsf{Ci}\mathcal{O}$ with Persistent Database)**.** *A computation-trace indistinguishability obfuscation scheme with persistent database w.r.t. $\mathcal{P}$, denoted by $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{DBCompile}, \mathsf{Obf}, \mathsf{Eval}\}$, is defined as follows:*

Database compilation algorithm $(\widetilde{\mathsf{mem}}^{0,0}, \widetilde{\mathsf{st}}^{0,0}) := \mathsf{DBCompile}(1^\lambda, \mathsf{mem}^{0,0}; \rho)$**:** $\mathsf{DBCompile}()$ *is a probabilistic algorithm which takes as input the security parameter λ, the database* $\mathsf{mem}^{0,0}$*, and some randomness ρ; and returns the complied database and state $(\widetilde{\mathsf{mem}}^{0,0}, \widetilde{\mathsf{st}}^{0,0})$ as output.*

Program compilation algorithm $\widetilde{F}_{\mathsf{sid}} := \mathsf{Obf}(1^\lambda, F_{\mathsf{sid}}; \rho')$**:** $\mathsf{Obf}()$ *is a probabilistic algorithm which takes as input the security parameter λ, the stateful function F_{sid}, and some randomness ρ'; and returns a complied / obfuscated function $\widetilde{F}_{\mathsf{sid}}$ as output.*

Evaluation algorithm $\mathsf{conf} := \mathsf{Eval}(\widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}, \widetilde{\mathsf{st}}^{\mathsf{sid}-1,0}, \widetilde{F}_{\mathsf{sid}})$**:** $\mathsf{Eval}()$ *is a deterministic algorithm which takes as input $(\widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}, \widetilde{\mathsf{st}}^{\mathsf{sid}-1,0}, \widetilde{F}_{\mathsf{sid}})$; and returns a configuration* $\mathsf{conf} = (\widetilde{\mathsf{mem}}^{\mathsf{sid},0}, \widetilde{\mathsf{st}}^{\mathsf{sid},0})$ *as output.*

Correctness *For all F_{sid} with termination time t^*_{sid} and all randomness ρ', let $\widetilde{F}_{\mathsf{sid}} := \mathsf{Obf}(1^\lambda, F_{\mathsf{sid}}; \rho')$; it holds that* $\mathsf{Eval}(\widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}, \widetilde{\mathsf{st}}^{\mathsf{sid}-1,0}, \widetilde{F}_{\mathsf{sid}}) = \mathsf{Conf}\langle \mathsf{mem}^{\mathsf{sid}-1,0}, \mathsf{st}^{\mathsf{sid}-1,0}, F_{\mathsf{sid}}, t^*_{\mathsf{sid}}\rangle$.

Security *For any (not necessarily uniform) PPT distinguisher $\mathcal{D}$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that, for all security parameters $\lambda \in \mathbb{N}$, $\Pi^0, \Pi^1 \in \mathcal{P}$ where $\Pi^b = (\mathsf{mem}^{0,0}, F_1^b, \ldots, F_l^b)$ for $b \in \{0,1\}$ and* $\mathsf{Trace}\langle\Pi^0\rangle = \mathsf{Trace}\langle\Pi^1\rangle$*, it holds that*

$$|\Pr[\mathcal{D}(\mathsf{Obf}(1^\lambda, \Pi^0)) = 1] - \Pr[\mathcal{D}(\mathsf{Obf}(1^\lambda, \Pi^1)) = 1] \leq \mathsf{negl}(\lambda).$$

Efficiency *We require* $\mathsf{DBCompile}$ *and* Obf *runs in time $\tilde{O}(|\mathsf{mem}^{0,0}|)$ and $\tilde{O}(\mathsf{poly}(|F_{\mathsf{sid}}|))$, and efficient* Eval *runs in time $\tilde{O}(t^*_{\mathsf{sid}})$.*

9.1.2 Constructing $\mathsf{Ci}\mathcal{O}$-RAM with persistent database

Construction We construct $\mathsf{Ci}\mathcal{O}$-RAM with persistent database from the ordinary $\mathsf{Ci}\mathcal{O}$-RAM (without persistent database). In general, we still follow the original setting of $\mathsf{Ci}\mathcal{O}$-RAM, but use (sid, t) as timestamp instead. Moreover, a new key K_T, called termination key, is involved in the obfuscated state function and only used at the beginning and end of a program. These three algorithms work as follows.

- Database compilation algorithm $\mathsf{DBCompile}$ is identical to Steps 1 and 3 of $\mathsf{Ci}\mathcal{O}$-RAM (without persistent database). It generates the initial configuration $(\widetilde{\mathsf{mem}}^{0,0}, \widetilde{\mathsf{st}}^{0,0})$ except that $\sigma^{0,0}$ is generated from the (pseudo-)randomness $r_0 \leftarrow \mathsf{PRF}(K_T, 0)$.
- Program compilation algorithm Obf is similar to Step 2 of $\mathsf{Ci}\mathcal{O}$-RAM except additional authentications under K_T for each sid, $1 \leq \mathsf{sid} \leq l$. It generates the obfuscated stateful function (See Algorithm 19). Note that the authentications under K_T are only performed in the beginning and end of a program. This algorithm outputs $\widetilde{F}_{\mathsf{sid}} \leftarrow \mathsf{i}\mathcal{O}.\mathsf{Gen}(\widehat{F}'_{\mathsf{sid}})$.
- Evaluation algorithm $\mathsf{Eval}(\widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}, \widetilde{\mathsf{st}}^{\mathsf{sid}-1,0}, \widetilde{F}_{\mathsf{sid}})$ is identical to Evaluation algorithm of $\mathsf{Ci}\mathcal{O}$-RAM. It outputs $(\widetilde{\mathsf{mem}}^{\mathsf{sid},0}, \widetilde{\mathsf{st}}^{\mathsf{sid},0})$ for the next session.

Algorithm 19: $\widehat{F}'_{\mathsf{sid}}$ in Ci$\mathcal{O}$-RAM with persistent database

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = ((\mathsf{sid}, t), \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}}), \ldots$
Data  : $\ldots, K_T$

1 if sid is correct and $(\mathsf{sid}, t)$ is the beginning of the session sid then
2     Compute $r_{\mathsf{sid}-1} = \mathsf{PRF}(K_T, \mathsf{sid}-1)$ and $(\mathrm{sk}_{\mathsf{sid}-1}, \mathrm{vk}_{\mathsf{sid}-1}, \mathrm{vk}_{\mathsf{sid}-1,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{\mathsf{sid}-1})$;
3     if $\mathsf{Spl.Verify}(\mathrm{vk}_{\mathsf{sid}-1}, (\mathsf{sid}-1, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}), \sigma^{\mathrm{in}}) = 0$ then output Reject;
4     Set $\mathsf{st}^{\mathrm{in}} =$ Init;

5 ...// Lines 1 to 16, Algorithm 1

6 if $\mathsf{st}^{\mathrm{out}}$ returns halt for termination then
7     Compute $r_{\mathsf{sid}} = \mathsf{PRF}(K_T, \mathsf{sid})$ and $(\mathrm{sk}_{\mathsf{sid}}, \mathrm{vk}_{\mathsf{sid}}, \mathrm{vk}_{\mathsf{sid},\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{\mathsf{sid}})$;
8     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}_{\mathsf{sid}}, (\mathsf{sid}, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}))$;
9     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = ((\mathsf{sid}, 0), \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$ // no database access
```

Security Sketch Recall that the computation system Π consists of an initial memory and a sequence of programs. Although we cannot directly use the security of Ci$\mathcal{O}$-RAM, using the pebble game analogy, we can go through the hybrid argument that is quite similar to Ci$\mathcal{O}$-RAM without persistent database.

Conceptually, we can view the computation paths of the sequence of programs as a single large computation path. The proof strategy is modified as follows: Recall that in the security proof of Ci$\mathcal{O}$-RAM without persistent database, we move the check-point from $t = 1$ to $t = t^*$ through hybrid argument. In the persistent database setting, the technique of moving from the timestamp (sid, t) to $(\mathsf{sid}, t+1)$ is identical to that in the setting without PDB.

The only difference here is that we need to move from the termination time $(\mathsf{sid}, t^*_{\mathsf{sid}})$ of session sid to the beginning $(\mathsf{sid}+1, 0)$ of the next session. For this, we can use the same technique as before to switch between the type A and B termination key K_T. We note that the purpose of K_T is to introduce keys which are independent of the termination time of the programs. It is otherwise conceptually the same as the type A key K_A used to sign the internal states.

A special conceptual point to note is that, in some intermediate hybrids, the enforcement of the accumulator or iterator is required to enforce the *whole history* from the initiation to the current timestamp.

9.1.3 Constructing Ci$\mathcal{O}$-PRAM with persistent database

Construction Following the same technique and conventions above, we construct Ci$\mathcal{O}$-PRAM with persistent database from full-fledged Ci$\mathcal{O}$-PRAM. In our construction of Ci$\mathcal{O}$-PRAM with persistent database, database compilation DBCompile, program compilation Obf, and evaluation algorithm Eval works as those in Ci$\mathcal{O}$-RAM with persistent database respectively, except for the obfuscated stateful function (See Algorithm 20). Note that once all CPUs terminate in session sid, the stateful function $\widehat{F}'_{\mathsf{sid}}$ only takes the cpu1's state to generate the signature for connecting the next session.

Security Sketch As for Ci$\mathcal{O}$-RAM with persistent database, the enforcement of the accumulator or iterator is required to enforce the whole history from the initiation to the current timestamp. We can use the same proof technique illustrated by the pebble game to go through the hybrid argument.

9.2 $\mathcal{RE}$ with Persistent Database

As in the ordinary setting without persistent database, after obtaining Ci$\mathcal{O}$ which forces the obfuscated program to be executed as intended, we can extend it to $\mathcal{RE}$ so as to provide input and program privacy. In the persistent

Algorithm 20: $\widehat{F}'_{\mathsf{sid}}$ in Ci$\mathcal{O}$-PRAM with persistent database

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{sid}, \mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \ldots$
Data  : $\ldots, K_T$

1  Parse root_node as before // extract $t$ from root_node
2  if sid is correct and $(\mathsf{sid}, t)$ is the beginning of the session sid then
3      Compute $r_{\mathsf{sid}-1} = \mathsf{PRF}(K_T, \mathsf{sid})$ and $(\mathrm{sk}_{\mathsf{sid}-1}, \mathrm{vk}_{\mathsf{sid}-1}, \mathrm{vk}_{\mathsf{sid}-1,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{\mathsf{sid}-1})$;
4      if $\mathsf{Spl.Verify}(\mathrm{vk}_{\mathsf{sid}-1}, (\mathsf{sid}-1, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}), \sigma^{\text{in}}) = 0$ then output Reject;
5      Set $\mathsf{st}^{\text{in}} =$ Init;

6  ... // Branch and Combine of Ci$\mathcal{O}$-PRAM

7  if all CPUs enter halt for termination then
8      Set $\mathsf{st}^{\text{out}}$ as cpu1's state;
9      // Let cpu1's final state be the initial state of the next session
10     Computes $r_{\mathsf{sid}} = \mathsf{PRF}(K_T, \mathsf{sid})$ and $(\mathrm{sk}_{\mathsf{sid}}, \mathrm{vk}_{\mathsf{sid}}, \mathrm{vk}_{\mathsf{sid},\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_{\mathsf{sid}})$;
11     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}_{\mathsf{sid}}, (\mathsf{sid}, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}_{\mathsf{st}}, w^{\text{out}}_{\mathsf{com}}))$;
12     Generate $\mathsf{root\_node} = (t, \mathsf{Root}, w^{\text{out}}_{\mathsf{st}}, w^{\text{out}}_{\mathsf{com}}, v^{\text{out}}, \sigma^{\text{out}})$;
13     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{sid}, \mathsf{st}^{\text{out}}, \mathsf{root\_node})$ ;
```

database setting, we wish to protect the privacy of the entire sequence of inputs and programs, while allowing the output of each program in the sequence to be learnt by the decoder in the clear.

9.2.1 Definition

Definition 9.2 ($\mathcal{RE}$ with Persistent Database)**.** *A randomized encoding scheme $\mathcal{RE}$ with persistent database consists of algorithms $\mathcal{RE} = \mathcal{RE}.\{\mathsf{DBInit}, \mathsf{Encode}, \mathsf{Decode}\}$ described below.*

- $\mathcal{RE}.\mathsf{DBEncode}(\mathsf{mem}^{0,0}, 1^\lambda) \to \widetilde{\mathsf{mem}}^{0,0}$*: The database compilation algorithm* DBEncode *is a probabilistic algorithm which takes as input the security parameter* 1^λ *and a database* $\mathsf{mem}^{0,0}$*. It outputs a compiled database* $\widetilde{\mathsf{mem}}^{0,0}$.
- $\mathcal{RE}.\mathsf{Encode}(P_{\mathsf{sid}}, x_{\mathsf{sid}}, 1^\lambda) \to \mathrm{ENC}_{\mathsf{sid}}$*: The encoding algorithm* Encode *is a probabilistic algorithm which takes as input the security parameter* 1^λ*, the description of a RAM program* P_{sid} *with time bound* T *and space bound* S*, and an input* x_{sid}*. It outputs an encoding* $\mathrm{ENC}_{\mathsf{sid}}$.
- $\mathcal{RE}.\mathsf{Decode}(\mathrm{ENC}_{\mathsf{sid}}, \widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}, 1^\lambda, T, S) \to (y_{\mathsf{sid}}, \widetilde{\mathsf{mem}}^{\mathsf{sid},0})$*: The decoding algorithm* Decode *is a deterministic algorithm which takes as input the security parameter* 1^λ*, time bound* T *and space bound* S*, an encoding* $\mathrm{ENC}_{\mathsf{sid}}$*, and a compiled database* $\widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}$*. It outputs* $y_{\mathsf{sid}} = P_{\mathsf{sid}}(x_{\mathsf{sid}})$ *or* $\perp$*, and a compiled database* $\widetilde{\mathsf{mem}}^{\mathsf{sid},0}$.

Correctness *A randomized encoding scheme $\mathcal{RE}$ is said to be* correct *if*

$$\Pr[\widetilde{\mathsf{mem}}^{0,0} \leftarrow \mathcal{RE}.\mathsf{DBEncode}(\mathsf{mem}^{0,0}, 1^\lambda); \mathrm{ENC}_{\mathsf{sid}} \leftarrow \mathcal{RE}.\mathsf{Encode}(P_{\mathsf{sid}}, x_{\mathsf{sid}}, 1^\lambda);$$
$$(y_{\mathsf{sid}}, \widetilde{\mathsf{mem}}^{\mathsf{sid},0}) \leftarrow \mathcal{RE}.\mathsf{Decode}(\mathrm{ENC}_{\mathsf{sid}}, \widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}, 1^\lambda, T, S) \ : \ y_{\mathsf{sid}} = P_{\mathsf{sid}}(x_{\mathsf{sid}})\ \forall \mathsf{sid}, 1 \le \mathsf{sid} \le l] = 1.$$

Security *A randomized encoding scheme $\mathcal{RE}$ with persistent database is said to be* hiding *if for all PPT adversary $\mathcal{A}$, time l, database $\mathsf{mem}^{0,0}$, program P_{sid} with time bound T and space bound S, input value x_{sid}, and output value $y_{\mathsf{sid}} = P_{\mathsf{sid}}(x_{\mathsf{sid}})$ for $\mathsf{sid} \ge 0$ that generated at termination time t^*_{sid}, there exists a PPT* *simulator $\mathcal{S}$ such that*

$$\begin{aligned}
&|\Pr[\widetilde{\mathsf{mem}}^{0,0} \leftarrow \mathcal{S}(1^{|\mathsf{mem}^{0,0}|}, 1^\lambda); \mathrm{ENC}_{\mathsf{sid}} \leftarrow \mathcal{S}(1^{|P_{\mathsf{sid}}|}, 1^{|x_{\mathsf{sid}}|}, t^*_{\mathsf{sid}}, y_{\mathsf{sid}}, 1^\lambda, T, S) : \\
&\quad \mathcal{A}(1^\lambda, \widetilde{\mathsf{mem}}^{0,0}, \{\mathrm{ENC}_{\mathsf{sid}}\}^l_{\mathsf{sid}=1}) = 1] \\
&- \Pr[\widetilde{\mathsf{mem}}^{0,0} \leftarrow \mathcal{RE}.\mathsf{DBEncode}(\mathsf{mem}^{0,0}, 1^\lambda); \mathrm{ENC}_{\mathsf{sid}} \leftarrow \mathcal{RE}.\mathsf{Encode}(P_{\mathsf{sid}}, x_{\mathsf{sid}}, 1^\lambda) : \\
&\quad \mathcal{A}(1^\lambda, \widetilde{\mathsf{mem}}^{0,0}, \{\mathrm{ENC}_{\mathsf{sid}}\}^l_{\mathsf{sid}=1}) = 1]| \leq \mathsf{negl}(\lambda).
\end{aligned}$$

Efficiency *We require* DBEncode *and* Encode *runs in time* $\tilde{O}(|\mathsf{mem}^{0,0}|)$ *and* $\tilde{O}(\mathsf{poly}(|P_{\mathsf{sid}}|) + |x_{\mathsf{sid}}|)$, *and efficient* Decode *runs in time* $\tilde{O}(t^*_{\mathsf{sid}})$.

9.2.2 Constructing $\mathcal{RE}$ with Persistent Database

Construction The construction of $\mathcal{RE}$ with PDB relies on the same technique to build $\mathcal{RE}$ from Ci$\mathcal{O}$ without PDB. As in Section 7, we use public-key encryption to hide the CPU state and the memory content, use oblivious RAM or PRAM to hide the access pattern, and finally use Ci$\mathcal{O}$-RAM or PRAM with PDB to obfuscate the compiled programs. The $\mathcal{RE}$ with PDB construction works as follows.

- $\mathcal{RE}$.DBEncode: It first compiles database $\mathsf{mem}^{0,0}$ to $(\widetilde{\mathsf{mem}}^{0,0}_o, \tilde{\mathsf{st}}^{0,0}_o)$ by ORAM or OPRAM compiler, then generates encryption of $(\widetilde{\mathsf{mem}}^{0,0}_e, \tilde{\mathsf{st}}^{0,0}_e)$ by $\mathcal{PKE}$. Finally, it outputs $(\widetilde{\mathsf{mem}}^{0,0}_c, \tilde{\mathsf{st}}^{0,0}_c)$ by DBCompile of Ci$\mathcal{O}$ with PDB.
- $\mathcal{RE}$.Encode: Unlike in ordinary $\mathcal{RE}$ where the input is written to the memory, we embed both the program P_{sid} and the input x_{sid} into a stateful function F_{sid}. It compiles the stateful function F_{sid} to $F_{\mathsf{sid},o}$ by ORAM or OPRAM compiler, and then generates $F_{\mathsf{sid},e}$ which includes decryption and encryption, except that at $t = 0$, $F_{\mathsf{sid},e}$ accepts the plaintext output generated by the previous program without performing decryption. We note that now the last write time used for decryption is in the format $\mathsf{lw} = (\mathsf{sid}, t)$. Finally, it outputs $\mathrm{ENC}_{\mathsf{sid}} = \mathsf{Obf}(F_{\mathsf{sid},e})$ by Obf of Ci$\mathcal{O}$.
- $\mathcal{RE}$.Decode: It executes $\mathsf{Eval}((\widetilde{\mathsf{mem}}^{\mathsf{sid}-1,0}_c, \tilde{\mathsf{st}}^{\mathsf{sid}-1,0}_c), \mathrm{ENC}_{\mathsf{sid}})$.

Security Sketch As in the security proof of $\mathcal{RE}$ without PDB, we wish to prove that if $\mathcal{PKE}$ and ORAM are secure, then the computation should be hidden. As before, we go through the hybrid argument backward in time, i.e., from the termination time of the last program, to the beginning of the last program, then the second last program, etc. Within a single program, the technique to move backward is identical to that in the setting without PDB. The only difference is at the beginning of a program. Instead of a ciphertext state, the initial state is hardwired, since the output of the previous program is a plaintext. This is possible since all intermediate outputs are given to the simulator.

9.3 $\mathcal{RE}$ with Output Hiding

In some applications, a client might want to delegate a computation to a server while ensuring that the latter does not learn anything about the input, the program, and the output. Observe that this extension of $\mathcal{RE}$ is somewhat similar to fully homomorphic encryption ($\mathcal{FHE}$) as both $\mathcal{FHE}$ and $\mathcal{RE}$ with output hiding hide the input and output, and allow arbitrary computation on the input. However, $\mathcal{RE}$ is a symmetric key primitive, and it additionally hides the program functionality. On the other hand, $\mathcal{FHE}$ is a public key encryption scheme which does not protect program privacy.

Definition 9.3 ($\mathcal{RE}$ with output hiding)**.** *A randomized encoding scheme with output hiding $\mathcal{RE}_{\mathsf{ohiding}}$, denoted by $\mathcal{RE}_{\mathsf{ohiding}} = \mathcal{RE}_{\mathsf{ohiding}}.\{\mathsf{Encode}, \mathsf{Decode}, \mathsf{Decrypt}\}$, is defined as follows.*

- $\mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Encode}(P, x, 1^\lambda) \to (\mathrm{ENC}, \mathrm{sk})$*: The encoding algorithm* Encode *is a probabilistic algorithm which takes as input the security parameter 1^λ, the description of a RAM or PRAM program P with time bound T and space bound S, and an input x. It outputs an encoding* ENC *and a private key* sk.

- $\mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S) \to c$: *The decoding algorithm* Decode *is a deterministic algorithm which takes as input the security parameter* 1^λ*, time bound* T*, space bound* S*, and an encoding* ENC. *It outputs ciphertext* c*, or* $\bot$.
- $\mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Decrypt}(\mathrm{sk}, c) \to y$: *The output decrypting algorithm* Decrypt *is a deterministic algorithm which takes as input the private key* sk *and the ciphertext* c*. It outputs the plaintext* y*.*

For efficiency*, we require that* Encode *runs in time* $\tilde{O}(\mathsf{poly}(|P|) + |x|)$ *and* Decrypt *runs in time* $\tilde{O}(1)$*, and efficient* Decode *runs in time* $\tilde{O}(T)$*. That is, a client can efficiently* Encode *a computation and* Decrypt *the ciphertext output* c*, and a server carries out* Decode *in time comparable to the original unsecured computation.*

Correctness *A scheme* $\mathcal{RE}_{\mathsf{ohiding}}$ *is said to be* correct *if*

$$\Pr[(\textsc{enc}, \mathrm{sk}) \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Encode}(P, x, 1^\lambda); c \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S);$$
$$y \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Decrypt}(\mathrm{sk}, c) \;:\; y = P(x)] = 1.$$

Hiding *A scheme* $\mathcal{RE}_{\mathsf{ohiding}}$ *is said to have* output hiding *if for all PPT adversary* $\mathcal{A}$*, program* P *with time bound* T *and space bound* S*, input value* x*, and output value* $y = P(x)$ *that generated at termination time* t^**, there exists a PPT simulator* Sim *such that*

$$|\Pr[\textsc{enc} \leftarrow \mathsf{Sim}(1^{|P|}, 1^{|x|}, 1^{|y|}, t^*, 1^\lambda, T, S) : \mathcal{A}(1^\lambda, \textsc{enc}) = 1]$$
$$- \Pr[(\textsc{enc}, \mathrm{sk}) \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Encode}(P, x, 1^\lambda) : \mathcal{A}(1^\lambda, \textsc{enc}) = 1]| \le \mathsf{negl}(\lambda).$$

Construction Let $\mathcal{RE} = \mathcal{RE}.\{\mathsf{Encode}, \mathsf{Decode}\}$ be a randomized encoding scheme. Let $\mathcal{SKE} = \mathcal{SKE}.\{\mathsf{Gen}, \mathsf{Encrypt}, \mathsf{Decrypt}\}$ be a symmetric key encryption scheme. The randomized encoding scheme with output hiding, $\mathcal{RE}_{\mathsf{ohiding}} = \mathcal{RE}_{\mathsf{ohiding}}.\{\mathsf{Encode}, \mathsf{Decode}, \mathsf{Decrypt}\}$, is constructed as follows:

- $\textsc{enc} \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Encode}(P, x, 1^\lambda)$:
  - Compute $\mathrm{sk} \leftarrow \mathcal{SKE}.\mathsf{Gen}(1^\lambda)$.
  - Sample $\rho \leftarrow \{0,1\}^\lambda$.
  - Compute $\textsc{enc} \leftarrow \mathcal{RE}.\mathsf{Encode}(P', (x, \rho), 1^\lambda)$, where P' is defined in Algorithm 21.
  - Return $(\textsc{enc}, \mathrm{sk})$.
- $c \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S)$: Compute $c \leftarrow \mathcal{RE}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S)$.
- $y \leftarrow \mathcal{RE}_{\mathsf{ohiding}}.\mathsf{Decrypt}(\mathrm{sk}, c)$: Return $y \leftarrow \mathcal{SKE}.\mathsf{Decrypt}(\mathrm{sk}, c)$.

Algorithm 21: P'

```
  Input : $(x, \rho)$
  Data  : $P, \mathrm{sk}$
1 Compute $y \leftarrow P(x)$;
2 Compute $c \leftarrow \mathcal{SKE}.\mathsf{Encrypt}(\mathrm{sk}, y; \rho)$;
3 Output $c$;
```

Security The security follows directly from the security of $\mathcal{RE}$ and $\mathcal{SKE}$.

9.4 $\mathcal{RE}$ with Verifiability, and Verifiable Encoding ($\mathcal{VE}$)

In this extension, we consider adding *verifiability* to $\mathcal{RE}$. We call it a verifiable randomized encoding ($\mathcal{VRE}$). Intuitively, to achieve verifiability, the encoding process first generates a signing key and verification key, then uses Ci$\mathcal{O}$ to obfuscate a program which signs the output of the program being encoded using the signing key.

Observe that although such a construction is non-black-box, it is mostly orthogonal to the construction of $\mathcal{RE}$. On the other hand, an encoding with verifiability but without privacy, which we call a verifiable encoding ($\mathcal{VE}$), is already useful in some delegation scenarios. Thus, it makes sense to consider $\mathcal{VE}$ as a stand-alone extension from $\mathsf{Ci}\mathcal{O}$.

More explicitly, we consider a verifiable encoding $\mathcal{VE}$ which encodes a program P and an input x into an encoding $\textsc{Enc}$, which can be decoded by the decoder to produce the computation result $y = P(x)$ and a proof π proving the correctness of the computation. The encoding algorithm also outputs a public verification key vk, with which any public verifier can check the correctness of y by verifying the proof π. This directly implies a two-message publicly-verifiable delegation scheme in the corresponding computation model.

9.4.1 Verifiable Encoding ($\mathcal{VE}$)

Formally, a verifiable encoding scheme $\mathcal{VE}$ consists of algorithms $\mathcal{VE} = \mathcal{VE}.\{\mathsf{Encode}, \mathsf{Decode}, \mathsf{Verify}\}$ described below.

- $\mathcal{VE}.\mathsf{Encode}(P, x, 1^\lambda) \to (\textsc{Enc}, \mathrm{vk})$: The encoding algorithm Encode is a probabilistic algorithm which takes as input the security parameter 1^λ, the description of a RAM / PRAM program P with time bound T and space bound S, and an input x. It outputs an encoding $\textsc{Enc}$ and a verification key vk.
- $\mathcal{VE}.\mathsf{Decode}(\textsc{Enc}, 1^\lambda, T, S) \to (y, \pi)$: The decoding algorithm Decode is a deterministic algorithm which takes as input the security parameter 1^λ, time bound T, space bound S, and an encoding $\textsc{Enc}$. It outputs $y = P(x)$ or $\perp$, and a proof π.
- $\mathcal{VE}.\mathsf{Verify}(\mathrm{vk}, \pi, y) \to b$: The verification algorithm Verify is a deterministic algorithm which takes as input a verification key vk, a proof π and an output of the computation y. It outputs a bit $b = 0$ or 1.

Correctness A verifiable randomized encoding scheme $\mathcal{VE}$ is said to be correct if

$$\begin{aligned}\Pr[&(\textsc{Enc}, \mathrm{vk}) \leftarrow \mathcal{VE}.\mathsf{Encode}(P, x, 1^\lambda); (y, \pi) \leftarrow \mathcal{VE}.\mathsf{Decode}(\textsc{Enc}, 1^\lambda, T, S);\\ &b \leftarrow \mathcal{VE}.\mathsf{Verify}(\mathrm{vk}, \pi, y) : y = P(x) \wedge b = 1] = 1.\end{aligned}$$

Verifiability A verifiable randomized encoding scheme $\mathcal{VE}$ is said to be verifiable if for all PPT adversary $\mathcal{A}$

$$\begin{aligned}\Pr[&(\textsc{Enc}, \mathrm{vk}) \leftarrow \mathcal{VE}.\mathsf{Encode}(P, x, 1^\lambda); (\widetilde{y}, \widetilde{\pi}) \leftarrow \mathcal{A}(1^\lambda, T, S, \textsc{Enc}, \mathrm{vk});\\ &b \leftarrow \mathcal{VE}.\mathsf{Verify}(\mathrm{vk}, \widetilde{\pi}, \widetilde{y}) : \widetilde{y} \neq P(x) \wedge b = 1] \leq \mathsf{negl}(\lambda).\end{aligned}$$

Efficiency We require Encode runs in time $\tilde{O}(\mathsf{poly}(|P|) + |x|)$ and Verify runs in time $\tilde{O}(\ell^{\mathrm{out}})$, and efficient Decode runs in time $\tilde{O}(T)$, where $\ell^{\mathrm{out}} = |y|$ is the length of output.

9.4.2 Building Blocks

Our construction uses several building blocks listed as follows:

- A signature scheme $\mathsf{SIG} = \mathsf{SIG}.\{\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify}\}$.
- A computation-trace indistinguishability obfuscation scheme $\mathsf{Ci}\mathcal{O} = \mathsf{Ci}\mathcal{O}.\{\mathsf{Obf}, \mathsf{Eval}\}$, for RAM or PRAM computation.

9.4.3 The Construction

We define our $\mathcal{VE} = \mathcal{VE}.\{\mathsf{Encode}, \mathsf{Decode}, \mathsf{Verify}\}$ for the program P and input x as follows:

Encoding algorithm $(\textsc{enc}, \mathrm{vk}) \leftarrow \mathcal{VE}.\mathsf{Encode}(P, x, 1^\lambda)$**:** For input (P, x), the encoding algorithm represents it as $\Pi = ((\mathsf{mem}^0, \mathsf{st}^0), F)$ for RAM program P or $\Pi = (\mathsf{mem}^0, F)$ for PRAM program P with x written to mem^0 sequentially. The encoding algorithm randomly chooses r_1, r_2, r_3, and computes $(\mathrm{sk}, \mathrm{vk}) = \mathsf{SIG.Gen}(1^\lambda; r_1)$. It then further compiles F into a program $\widehat{F}$ defined in Algorithm 22.

Let $\widehat{\Pi} = ((\mathsf{mem}^0, \mathsf{st}_1^0, \ldots, \mathsf{st}_m^0), \widehat{F})$, it computes $\textsc{enc} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \widehat{\Pi})$. Finally, it outputs $(\textsc{enc}, \mathrm{vk})$.

Decoding algorithm $(y, \pi) \leftarrow \mathcal{VE}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S)$**:** It executes $\mathsf{Ci}\mathcal{O}.\mathsf{Eval}(\textsc{enc})$ to obtain the configuration $\left(\hat{\mathsf{st}}_1^{t^*} = ((\mathtt{halt}, y), \sigma), \hat{\mathsf{st}}_2^{t^*}, \ldots, \hat{\mathsf{st}}_m^{t^*}, \mathsf{mem}^{t^*}\right)$ upon termination, and outputs $(y, \pi) = (y, \sigma)$.

Verification algorithm $b \leftarrow \mathcal{VE}.\mathsf{Verify}(\mathrm{vk}, y, \sigma)$**:** The verification algorithm outputs $b = \mathsf{SIG.Verify}(\mathrm{vk}, y, \sigma)$.

Theorem 9.4. *Let* $\mathsf{Ci}\mathcal{O}$ *be an indistinguishability obfuscation for computation in the RAM / PRAM model, and* SIG *be a secure signature scheme; then* $\mathcal{VE}$ *is a secure verifiable encoding scheme.*

The proof can be found in Appendix B.7.

```
Algorithm 22: $\widehat{F}$                         // this program is used in $\mathcal{VE}$
   Input  : $\hat{\mathsf{st}}^{in} = (\mathsf{st}^{in}, t), a^{in}$
   Data   : $T, r_1, r_2, \mathrm{sk}$
 1 Compute $(\mathsf{st}^{out}, a^{out}) = F(\mathsf{st}^{in}, a^{in})$;
 2 if $\mathsf{st}^{out} \neq (\mathtt{halt}, \cdot)$ then
 3     Set $\hat{\mathsf{st}}^{out} = (\mathsf{st}^{out}, t+1)$;
 4 else
 5     Parse $\mathsf{st}^{out} = (\mathtt{halt}, y)$;
 6     if $y = \perp$ then
 7         Set $\hat{\mathsf{st}}^{out} = \mathsf{st}^{out}$;
 8     else
 9         Compute $(\mathrm{sk}, \mathrm{vk}) = \mathsf{SIG.Gen}(1^\lambda; r_1)$;
10         Compute $\sigma = \mathsf{SIG.Sign}(\mathrm{sk}, y; r_2)$;
11         Set $\hat{\mathsf{st}}^{out} = (\mathsf{st}^{out}, \sigma)$;
12         Set $a^{out} = \perp$;
13 Output $\hat{\mathsf{st}}^{out}, a^{out}$;
```

9.5 $\mathcal{RE}$ and $\mathcal{VE}$ with Long Output

Recall that in the definitions of $\mathcal{RE}$ and its extensions (including $\mathcal{VE}$), we always consider a program P and input x such that $y = P(x)$ is of a fixed short length. However, in practical applications, a program might produce an output of length which is long and possibly variable.

To support long output, our main strategy is to let the main program write its output into a specified area of the memory. In the following, we first consider the simpler case of $\mathcal{RE}$ with output hiding. In this case, the decoder simply returns the ciphertexts stored in the specified area. Next, for the more complicated case of $\mathcal{RE}$ without output hiding, $\mathcal{RE}$ with long output can be constructed using $\mathcal{RE}$ with persistent database. The idea is to encode a sequence of short programs which reads, decrypts, and outputs a short portion of the specified area. Finally, for $\mathcal{VE}$ which only provides verifiability but without any privacy, our strategy is very similar to that of $\mathcal{RE}$ with output hiding. Concretely, the main program writes its output, which is in plaintext, along with a signature into a specified area of the memory. Note that in $\mathcal{VE}$ the memory content is not encrypted. Thus, the decoder simply returns the plaintexts stored in the specified area.

$\mathcal{RE}$ **with Long Output with Output Hiding** In this setting, instead of the output y, a position-length pair is put in the termination state. The decoder then simply returns the specified portion of the memory to the encoder. In the case where verifiability (of the ciphertexts) is required, the main program signs the long sequence of ciphertexts using the hash-then-sign paradigm, so that a single short signature can be appended to the end of the sequence. Concretely, the main program maintains a hash tree which compresses the long sequence of output ciphertexts into a short digest stored in the root of the tree. The root is then signed to authenticate the entire tree. Notice that in this setting, the size of the encoding is independent of the output size. This corresponds to the fact that the simulator can simply simulate the encrypted output sequence by a sequence of random values, so that no hardwiring of the long output is needed.

$\mathcal{RE}$ **with Long Output without Output Hiding** In the following, we let l be an upper bound of the output length of the program P with input x. $\mathcal{RE}$ with long output without output hiding can be achieved by first writing the sequence of ciphertext into the memory as specified above, then encoding a sequence of l short programs which reads, decrypts, and outputs a short portion of the specified area. In the case where verifiability (of the plaintexts) is required, we simply add into these short programs some additional lines of code for signing. We note that the number of encodings l depends on the output length of the program. This is due to the fact that the entire sequence of outputs must be hardwired to the simulated encoding in the security proof. Thus, $\mathcal{RE}$ with output hiding not only provides output privacy, but also produces shorter encodings.

To see why the dependency on l is necessary, we consider a program which functions as a pseudo random generator (PRG) which has short input and long output. Suppose that there exists secure $\mathcal{RE}$ with long output without output hiding, which produces encoding of (P, x) with length independent of the length of $y = P(x)$. By the security of $\mathcal{RE}$, there exists a simulator for $\mathcal{RE}$ which produces upon input y a simulated encoding with length independent of the length of y. We wish to construct a distinguisher which distinguishes PRG from a random function. Suppose in the security game of PRG the challenger return an output y upon a query x. We then pass y to the simulator of $\mathcal{RE}$. If the chosen function is a PRG, then the simulated encoding has length independent of y. Otherwise, if the chosen function is a random function, then either the simulated encoding has length dependent on y, or the decoding of the simulated encoding produces result different from y with non-negligible probability. In either case, we can distinguish a PRG from the random function.

$\mathcal{VE}$ **with Long Output** Similar to $\mathcal{RE}$ with output hiding, the main program signs the long sequence of outputs using the hash-then-sign paradigm, so that a single short signature can be appended to the end of the sequence. Then, instead of the output y, a position-length pair is put in the termination state. Finally, the decoder returns the specified portion of the memory to the encoder.

9.6 Application: Searchable Symmetric Encryption ($\mathcal{SSE}$)

In the previous sections, we show how $\mathcal{RE}$-RAM and PRAM can be extended to support a wide range of properties, including persistent database (Section 9.2), output hiding (Section 9.3), long output (Section 9.5), and verifiability (Section 9.4). Different combinations of these properties are useful for different scenarios of outsourced computation. In particular, as a direct application of $\mathcal{RE}$ with all the above extensions, we consider a very powerful searchable symmetric encryption ($\mathcal{SSE}$) scheme with almost all desirable properties considered in the $\mathcal{SSE}$ literature.

Roughly, $\mathcal{SSE}$ allows a client to outsource the storage of his or her encrypted data to a semi-honest (possibly malicious) server, while retaining the server's ability to query over the encrypted data without learning the plaintext data. The query can be as general as data modification, (conjunctive / fuzzy) keyword search, or essentially any function over the plaintext data. To query over the encrypted data, the client uses its private key to transform its query into a trapdoor, which is sent to the server. With the help of the trapdoor, the server possibly updates the encrypted database and returns the encrypted query results to the client.

Ideally, an $\mathcal{SSE}$ scheme is considered secure if the encrypted data and queries do not reveal any information about the plaintext data and query results[21] respectively. It is commonly believed that such security requirements can be achieved using ORAM. In reality, typical $\mathcal{SSE}$ schemes which do not rely on ORAM leak some information such as the search and access patterns as a trade-off for efficiency.

Using $\mathcal{RE}$ with persistent database, we can naturally encode our plaintext data into an encrypted database which will then be stored in a cloud server. Then, with the support of long output and output hiding, an encoded query can be processed by the server to return a long sequence of ciphertext, which can be decrypted to obtain the results of the query. Moreover, by the succinctness of our $\mathcal{RE}$ construction, the query complexity is preserved up to a logarithmic factor.

In terms of security, note that by the security of $\mathcal{RE}$, the server only learns the sizes of the database and the query results. The security of this $\mathcal{SSE}$ scheme is thus not only much stronger than most of the existing schemes which leak search and access patterns, but also achieves two very desirable properties named forward privacy and backward privacy. Forward privacy means that a previously issued trapdoor for a query is not useful for querying newly added data. Similarly, backward privacy means that a trapdoor is not useful for querying deleted data. In addition, with the verifiability extension, the correctness of the query results can be verified. It is also worth mentioning that while most $\mathcal{SSE}$ schemes are proven secure in the random oracle model, our construction is secure in the standard model.

[21]Consider the query result returned by the server is in an encrypted form which only the client can decrypt.

A Preliminaries

Notations Let λ be the security parameter. Let poly be any polynomial. Let negl be any negligible function.

A.1 Models of Computation

A.1.1 Random-Access Machines (RAM)

A random-access machine (RAM) consists of a CPU with a local register st of size $\log n$ and an external memory $\mathsf{mem} \in \{0,1\}^n$, where $n = \mathsf{poly}(\lambda)$. A RAM program P with random-access to mem takes as input $x \in \{0,1\}^{\ell_{\mathsf{input}}}$, where $\ell_{\mathsf{input}} \leq n$, and outputs $y = P(x)$ as the result of the computation. During the computation, the CPU may access the memory multiple times using $\textsc{Read}$ or $\textsc{Write}$ operations:

– $\textsc{Read}(\mathsf{loc})$: upon receiving a memory address loc, return the value $\mathsf{mem}[\mathsf{loc}]$.
– $\textsc{Write}(\mathsf{loc}, \mathsf{val})$: upon receiving a memory address loc and a value val, set $\mathsf{mem}[\mathsf{loc}] := \mathsf{val}$.

In this work, we use both functional program and next-step program to represent the RAM program, and we represent the above functional program P as a series of executions of a small next-step program F which executes a single CPU step:

$$(\mathsf{st}^{\text{out}}, \mathsf{loc}^{\text{out}}, \mathsf{val}^{\text{out}}) = F(\mathsf{st}^{\text{in}}, \mathsf{loc}^{\text{in}}, \mathsf{val}^{\text{in}}).$$

At each time step t, the CPU-step circuit takes as input an input state st^{in}, a location loc^{in}, and a value $\mathsf{val}^{\text{in}} = \mathsf{mem}[\mathsf{loc}^{\text{in}}]$ read from the memory, and outputs an output state st^{out}, a location $\mathsf{loc}^{\text{out}}$ to be accessed, and a value $\mathsf{val}^{\text{out}}$.

By convention, at the first step (i.e., step 0), the next-step program is executed with $\mathsf{loc}^{\text{in}} = \perp$ and $\mathsf{val}^{\text{in}} = \perp$. At each step, a copy of the next-step program is executed. If F issues a $\textsc{Write}$ memory operation with $\mathsf{loc}^{\text{out}}$ and $\mathsf{val}^{\text{out}}$ specified, then the value $\mathsf{val}^{\text{out}}$ will be written to $\mathsf{mem}[\mathsf{loc}^{\text{out}}]$, and the evaluator sets $\mathsf{loc}^{\text{in}} = \perp$ and $\mathsf{val}^{\text{in}} = \perp$ for the next step. Else if F issues a $\textsc{Read}$ memory operation with loc^{in} specified and $\mathsf{val}^{\text{out}} = \perp$, then the evaluator sets $\mathsf{loc}^{\text{in}} = \mathsf{loc}^{\text{out}}$, and the location loc^{in} is read by setting $\mathsf{val}^{\text{in}} = \mathsf{mem}[\mathsf{loc}^{\text{in}}]$ for the next step.

There are two ways to define the output of the computation. The first approach is to interpret the output state of the last CPU-step circuit as the output of the computation, which limits the size of the output to $\log n$. The second approach is to interpret a pre-defined region of the external memory mem as the output of the computation. *For simplicity, we adopt in this work the first definition, but note that the second definition can also be adopted.*

A.1.2 Parallel RAM (PRAM)

A parallel random-access machine (PRAM) consists of m CPUs, each with local memory register of size $\log n$, sharing an external memory $\mathsf{mem} \in \{0,1\}^n$, where $n = \mathsf{poly}(\lambda)$. A RAM is simply a PRAM with $m = 1$. A PRAM program P has random-access to mem, takes as input x and outputs $y = P(x)$ as the result of the computation. In general, a PRAM program utilizes a dynamic number of CPUs in each time step. In a simpler variant, it is assumed that the program always uses all the m CPUs.

Similar to a RAM program, a PRAM program can be represented by a series of executions of the next-step program F, but with the additional ability to execute m copies in parallel at each time step. For each CPU $k \in [m]$, F computes a time step with its k-th copy of state and memory operation, and an additional argument k denoting which CPU it is computing. That is, for each $k \in [m]$,

$$(\mathsf{st}_k^{\text{out}}, \mathsf{loc}_k^{\text{out}}, \mathsf{val}_k^{\text{out}}) = F(k, \mathsf{st}_k^{\text{in}}, \mathsf{loc}_k^{\text{in}}, \mathsf{val}_k^{\text{in}}).$$

The conflicts in read and write locations are resolved according to either the exclusive read exclusive write (EREW), concurrent read exclusive write (CREW), or concurrent read concurrent write (CRCW) strategy. For simpler analysis, we always assume that a *PRAM program P follows the CREW rule*, so that there must not be any conflicting writes. We further assume for simplicity (but equivalently) that all m CPUs read and write synchronously and alternatively, which yields a two-fold (parallel) time overhead because any CPU can at least issue a dummy access and defer the actual access to the next iteration.

Without loss of generality, the input x is stored in a pre-defined region of the external memory mem, and all initial states are the same value $\perp$ for all CPUs. The output of the computation is the output state of the last CPU-step circuit of a specific CPU, which is defined similarly as that for RAM programs. All CPUs halt at the same time with a state $\mathsf{st} = (\texttt{halt}, \cdot)$. There is a special CPU $\mathsf{cpu}1$ which always halts with result y by outputting $\mathsf{st} = (\texttt{halt}, y)$ while all other CPUs output $\mathsf{st} = (\texttt{halt}, \perp)$.

In some occasions, we will assume additionally (but equivalently) that the CPUs can communicate with each other directly. Roughly speaking, such communication can be simulated by accessing the shared memory. We will explain the details when needed in Sections 6 and 8.

A.1.3 Memoryless PRAM (mPRAM)

A simpler variant of PRAM is the memoryless PRAM (denoted by mPRAM), which consists of m CPUs, each with local memory register of size $\log n$, but without external memory. However, there are synchronous communications transmitting constant size messages between CPUs. Their communication pattern is assumed to be oblivious and, at each time step, each CPU only receives one message from one CPU and sends one message to one other CPU.

Similar to the standard PRAM program, an mPRAM program can be represented by a series of executions of the next-step circuit, but with the additional ability to execute multiple copies of the circuit at a time step, corresponding to the number of CPUs used in that time step. Unlike in PRAM, the input and output are both stored in the corresponding initial and final CPU states. We will explain the details when needed in Section 6.

Memoryless PRAM is strictly weaker than the standard PRAM, which can emulate mPRAM with memory size $m < n$ and emulate each communication by writing and reading memory cells.

A.2 Randomized Encoding ($\mathcal{RE}$)

Randomized encoding scheme $\mathcal{RE}$ was originally introduced by Ishai and Kushilevitz [IK00]. Recently, Bitansky et al. and Canetti et al. studied $\mathcal{RE}$ in the TM/RAM models [BGL+15, CHJV15]. Here we state the definition of $\mathcal{RE}$ in the RAM model, as follows. We can similarly define $\mathcal{RE}$ in the PRAM model.

A randomized encoding scheme $\mathcal{RE}$ consists of algorithms $\mathcal{RE} = \mathcal{RE}.\{\mathsf{Encode}, \mathsf{Decode}\}$ described below.

- $\mathcal{RE}.\mathsf{Encode}(P, x, 1^\lambda) \to \textsc{enc}$: The encoding algorithm Encode is a probabilistic algorithm which takes as input the security parameter 1^λ, the description of a RAM program P with time bound T and space bound S, and an input x. It outputs an encoding $\textsc{enc}$.
- $\mathcal{RE}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S) \to y$: The decoding algorithm Decode is a deterministic algorithm which takes as input the security parameter 1^λ, time bound T and space bound S, and an encoding $\textsc{enc}$. It outputs $y = P(x)$ or $\perp$.

Correctness A randomized encoding scheme $\mathcal{RE}$ is said to be *correct* if

$$\Pr[\textsc{enc} \leftarrow \mathcal{RE}.\mathsf{Encode}(P, x, 1^\lambda); y \leftarrow \mathcal{RE}.\mathsf{Decode}(\textsc{enc}, 1^\lambda, T, S) \ : \ y = P(x)] = 1.$$

Hiding A randomized encoding scheme $\mathcal{RE}$ is said to be *hiding* if for all PPT adversary $\mathcal{A}$, program P with time bound T and space bound S, input value x, and output value $y = P(x)$ that generated at termination time t^*, there exists a PPT simulator Sim such that

$$\begin{aligned} &| \Pr[\widetilde{\textsc{enc}} \leftarrow \mathsf{Sim}(1^{|P|}, 1^{|x|}, t^*, y, 1^\lambda, T, S) : \mathcal{A}(1^\lambda, \widetilde{\textsc{enc}}) = 1] \\ &- \Pr[\textsc{enc} \leftarrow \mathcal{RE}.\mathsf{Encode}(P, x, 1^\lambda) : \mathcal{A}(1^\lambda, \textsc{enc}) = 1]| \leq \mathsf{negl}(\lambda). \end{aligned}$$

Efficiency We require Encode runs in time $\tilde{O}(\mathsf{poly}(|P|)+|x|)$, and efficient Decode runs in time $\tilde{O}(t^*)$. That is, a client can efficiently encode (P, x), and a server carries out evaluation in time comparable to the original unsecured computation.

A.3 Building Blocks

A.3.1 Iterators

An iterator [KLW15] is a cryptographic data structure which maintains a small iterator state regardless of the number of messages iterated. Although it is impossible for a small iterator state to uniquely identify a sequence of iterated messages, a secure iterator guarantees that normally generated public-parameters are computationally indistinguishable from specially constructed "enforcing" parameters, which ensures a particular iterator state to be obtainable only by iterating a specific message to another specific iterator state. Such a localized property can be achieved information-theoretically by fixing the enforcement ahead of time.

Syntax An iterator Itr with message space $\mathcal{M}_\lambda = \{0,1\}^{\mathsf{poly}(\lambda)}$ and state space $\mathcal{S}_\lambda$ consists of three algorithms - $\mathsf{Itr}.\{\mathsf{Setup}, \mathsf{SetupEnforceIterate}, \mathsf{Iterate}\}$, defined below.

- $\mathsf{Itr.Setup}(1^\lambda, T)$: The setup algorithm takes as input the security parameter λ (in unary), and an integer bound T (in binary) on the number of iterations. It outputs public parameters $\mathsf{pp}_{\mathsf{Itr}}$ and an initial state $v^0 \in \mathcal{S}_\lambda$.
- $\mathsf{Itr.SetupEnforceIterate}(1^\lambda, T, \mathbf{m})$: The enforced setup algorithm takes as input the security parameter λ (in unary), an integer bound T (in binary), and a vector of messages $\mathbf{m} = (m^1, \ldots, m^k)$. It outputs public parameters $\mathsf{pp}_{\mathsf{Itr}}$ and an initial state $v^0 \in \mathcal{S}_\lambda$.
- $\mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, m)$: The iterate algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Itr}}$, a state v^{in}, and a message $m \in \mathcal{M}_\lambda$. It outputs a state $v^{\mathrm{out}} \in \mathcal{S}_\lambda$.

For presentational convenience, we use the notation $\mathsf{Itr.Iterate}^j(\mathsf{pp}_{\mathsf{Itr}}, v^0, (m^1, \ldots, m^j))$ to denote v^j where $v^j \leftarrow \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{j-1}, m^j)$ for all $j \in [k]$.

Security Let $\mathsf{Itr} = \mathsf{Itr}.\{\mathsf{Setup}, \mathsf{SetupEnforceIterate}, \mathsf{Iterate}\}$ be an iterator with message space $\mathcal{M}_\lambda$ and state space $\mathcal{S}_\lambda$. We require the following notions of security.

Definition A.1 (Indistinguishability of Setup)**.** *An iterator* Itr *is said to satisfy indistinguishability of Setup phase if any PPT adversary* $\mathcal{A}$*'s advantage in the security game* ***Exp-Setup-Itr***$(1^\lambda, \mathsf{Itr}, \mathcal{A})$ *is at most negligible in* λ*, where* ***Exp-Setup-Itr*** *is defined as follows.*

Exp-Setup-Itr$(1^\lambda, \mathsf{Itr}, \mathcal{A})$

- *The adversary* $\mathcal{A}$ *chooses a bound* $N \in \Theta(2^\lambda)$ *and sends it to the challenger.*
- $\mathcal{A}$ *sends* $\mathbf{m}$ *to the challenger, where* $\mathbf{m} = (m^1, \ldots, m^k) \in (\mathcal{M}_\lambda)^k$.
- *The challenger chooses a bit* b*. If* $b = 0$*, the challenger outputs* $(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{Itr.Setup}(1^\lambda, T)$*. Else, it outputs* $(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{Itr.SetupEnforceIterate}(1^\lambda, T, \mathbf{m})$ *where* $\mathbf{m} = (m^1, \ldots, m^k) \in (\mathcal{M}_\lambda)^k$.
- $\mathcal{A}$ *sends a bit* b'.

$\mathcal{A}$ *wins the security game if* $b = b'$.

Definition A.2 (Enforcing)**.** *Consider any* $\lambda \in \mathbb{N}, T \in \Theta(2^\lambda), \mathbf{m} = (m^1, \ldots, m^k) \in (\mathcal{M}_\lambda)^k$*. Let* $(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{SetupEnforceIterate}(1^\lambda, T, \mathbf{m})$ *and* $v^j = \mathsf{Itr.Iterate}^j(\mathsf{pp}_{\mathsf{Itr}}, v^0, (m^1, \ldots, m^k))$ *for all* $j \in [k]$*. We say* $\mathsf{Itr} = \mathsf{Itr}.\{\mathsf{Setup}, \mathsf{SetupEnforceIterate}, \mathsf{Iterate}\}$ *is* enforcing *if*

$$v^k = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v', m') \Rightarrow (v', m') = (v^{k-1}, m^k).$$

Note that the enforcing property is an information-theoretic property.

A.3.2 Positional Accumulators

A positional accumulator [KLW15] is a cryptographic data structure which maintains a relatively large storage with a short accumulator value. The accumulator is designed in such a way that, given the last accumulator value and some new modification to the storage, a new accumulator value can be computed efficiently. While the accumulator value does not contain all the information about the storage, a "helper" algorithm allows the (untrusted) storage party who is maintaining the full storage to help the (restricted) computation party that has the accumulator value recover any data stored in arbitrary location. A positional accumulator for message space $\mathcal{M}_\lambda$ consists of the following algorithms.

Syntax

- $\mathsf{Acc.Setup}(1^\lambda, S)$: The setup algorithm takes as input the security parameter λ (in unary), and an integer bound S (in binary) on the number of iterations. It outputs public parameters $\mathsf{pp}_{\mathsf{Acc}}$, an initial accumulator value w^0, and an initial storage value $store^0$.
- $\mathsf{Acc.SetupEnforceRead}(1^\lambda, S, (m_1, \mathsf{index}_1), \dots, (m_k, \mathsf{index}_k), \mathsf{index}^*)$: The setup enforce-read algorithm takes as input the security parameter λ (in unary), an integer bound S (in binary) representing the maximum number of values that can be stored, and a vector of symbol-index pairs where each index is in $\{0, \dots, S-1\}$, and an additional index^* also in $\{0, \dots, S-1\}$. It outputs public parameters $\mathsf{pp}_{\mathsf{Acc}}$, an initial accumulator value w^0, and an initial storage value $store^0$.
- $\mathsf{Acc.SetupEnforceWrite}(1^\lambda, S, (m_1, \mathsf{index}_1), \dots, (m_k, \mathsf{index}_k))$: The setup enforce-write algorithm takes as input the security parameter λ (in unary), an integer bound S (in binary) representing the maximum number of values that can be stored, and a vector of symbol-index pairs where each index is in $\{0, \dots, S-1\}$. It outputs public parameters $\mathsf{pp}_{\mathsf{Acc}}$, an initial accumulator value w^0, and an initial storage value $store^0$.
- $\mathsf{Acc.PrepRead}(\mathsf{pp}_{\mathsf{Acc}}, store^{\mathrm{in}}, \mathsf{index})$: The prep-read algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Acc}}$, a storage value $store^{\mathrm{in}}$, and an $\mathsf{index} \in \{0, \dots, S-1\}$. It outputs a symbol m (that can be $\emptyset$) and a value π.
- $\mathsf{Acc.PrepWrite}(\mathsf{pp}_{\mathsf{Acc}}, store^{\mathrm{in}}, \mathsf{index})$: The prep-write algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Acc}}$, a storage value $store^{\mathrm{in}}$, and an $\mathsf{index} \in \{0, \dots, S-1\}$. It outputs an auxiliary value aux.
- $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, m_{read}, \mathsf{index}, \pi)$: The verify-read algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Acc}}$, a, accumulator value w^{in}, a symbol m_{read}, an $\mathsf{index} \in \{0, \dots, S-1\}$, and a value π. It outputs True or False.
- $\mathsf{Acc.WriteStore}(\mathsf{pp}_{\mathsf{Acc}}, store^{\mathrm{in}}, \mathsf{index}, m)$: The write-store algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Acc}}$, a storage value $store^{\mathrm{in}}$, an $\mathsf{index} \in \{0, \dots, S-1\}$, and a symbol m. It outputs a storage value $store^{\mathrm{out}}$.
- $\mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, m_{write}, \mathsf{index}, \pi)$: The update algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Acc}}$, an accumulator value w^{in}, a symbol m_{write}, an $\mathsf{index} \in \{0, \dots, S-1\}$, and an auxiliary value aux. It outputs an accumulator value w^{out} or Reject.
- $\mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc}}, h_1, h_2, \mathsf{index})$: The update algorithm takes as input the public parameters $\mathsf{pp}_{\mathsf{Acc}}$, two hashes $h_1, h_2 \in \{0,1\}^\ell$, and an $\mathsf{index} \in \{0,1\}^{<\lceil \log S \rceil}$. It outputs another hash value $h^{\mathrm{out}} \in \{0,1\}^\ell$, which must be consistent with the output of $\mathsf{Acc.Update}$ after iterating over the whole storage $store$.

Security Let $\mathsf{Acc} = \mathsf{Acc}.\{\mathsf{Setup}, \mathsf{SetupEnforceRead}, \mathsf{SetupEnforceWrite}, \mathsf{PrepRead}, \mathsf{PrepWrite}, \mathsf{VerifyRead}, \mathsf{WriteStore}, \mathsf{Update}\}$ be an accumulator with message space $\mathcal{M}_\lambda$ and state space $\mathcal{S}_\lambda$. We require the following notions of security.

Definition A.3 (Indistinguishability of Read-Setup). *A positional accumulator* Acc *is said to satisfy indistinguishability of Read-Setup phase if any PPT adversary $\mathcal{A}$'s advantage is negligible at most in λ for winning the security game **Exp-Setup-Read**$(1^\lambda, \mathsf{Itr}, \mathcal{A})$, where **Exp-Setup-Read** is defined as follows.*

Exp-Setup-Read$(1^\lambda, \mathsf{Acc}, \mathcal{A})$

- *The adversary $\mathcal{A}$ chooses a bound $S \in \Theta(2^\lambda)$ and sends it to challenger.*
- *$\mathcal{A}$ sends k messages $m^1, \dots, m^k \in \mathcal{M}_\lambda$, and $k+1$ indexes $\mathsf{index}^1, \dots, \mathsf{index}^k, \mathsf{index}^* \in \{0, \dots, S-1\}$.*

- *The challenger chooses a bit b. The challenger outputs, if $b = 0$, $(\mathsf{pp}_{\mathsf{Acc}}, w^0, store^0) \leftarrow \mathsf{Acc.Setup}(1^\lambda, S)$; else, $(\mathsf{pp}_{\mathsf{Acc}}, w^0, store^0) \leftarrow \mathsf{Acc.SetupEnforceRead}(1^\lambda, S, (m^1, \mathsf{index}^1), \ldots, (m^k, \mathsf{index}^k), \mathsf{index}^*)$.*
- *$\mathcal{A}$ sends a bit b'.*

$\mathcal{A}$ wins the security game if $b = b'$.

Definition A.4 (Indistinguishability of Write-Setup)**.** *A positional accumulator* Acc *is said to satisfy indistinguishability of Write-Setup phase if any PPT adversary $\mathcal{A}$'s advantage is at most negligible in λ for winning the security game **Exp-Setup-Write**$(1^\lambda, \mathsf{Itr}, \mathcal{A})$, where **Exp-Setup-Write** is defined as follows.*

***Exp-Setup-Write**$(1^\lambda, \mathsf{Acc}, \mathcal{A})$*

- *The adversary $\mathcal{A}$ chooses a bound $S \in \Theta(2^\lambda)$ and sends it to challenger.*
- *$\mathcal{A}$ sends k messages $m^1, \ldots, m^k \in \mathcal{M}_\lambda$, and k indexes $\mathsf{index}^1, \ldots, \mathsf{index}^k \in \{0, \ldots, S-1\}$.*
- *The challenger chooses a bit b. If $b = 0$, the challenger outputs $(\mathsf{pp}_{\mathsf{Acc}}, w^0, store^0) \leftarrow \mathsf{Acc.Setup}(1^\lambda, S)$. Else, it outputs $(\mathsf{pp}_{\mathsf{Acc}}, w^0, store^0) \leftarrow \mathsf{Acc.SetupEnforceWrite}(1^\lambda, S, (m^1, \mathsf{index}^1), \ldots, (m^k, \mathsf{index}^k))$.*
- *$\mathcal{A}$ sends a bit b'.*

$\mathcal{A}$ wins the security game if $b = b'$.

Definition A.5 (Read-Enforcing)**.** *Consider any $\lambda \in \mathbb{N}, S \in \Theta(2^\lambda)$, $m^1, \ldots, m^k \in \mathcal{M}_\lambda$, $\mathsf{index}^1, \ldots, \mathsf{index}^k \in \{0, \ldots, S-1\}$, and any $\mathsf{index}^* \in \{0, \ldots, S-1\}$.*

Let $(\mathsf{pp}_{\mathsf{Acc}}, w^0, store^0) \leftarrow \mathsf{Acc.SetupEnforceRead}(1^\lambda, S, (m^1, \mathsf{index}^1), \ldots, (m^k, \mathsf{index}^k), \mathsf{index}^*)$.

For all $j \in [k]$, we define $store^j$ iteratively as $store^j := \mathsf{WriteStore}(\mathsf{pp}_{\mathsf{Acc}}, store^{j-1}, \mathsf{index}^j, m^j)$.

We similarly define aux^j and w^j iteratively as $aux^j := \mathsf{PrepWrite}(\mathsf{pp}_{\mathsf{Acc}}, store^{j-1}, \mathsf{index}^j)$ and $w^j := \mathsf{Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{j-1}, m^j, \mathsf{index}^j, aux^j)$.

Acc *is said to be* read-enforcing *if* $\mathsf{VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^k, m, \mathsf{index}^*, \pi) = 1$, *we have either $\mathsf{index}^* \notin \{\mathsf{index}^1, \ldots, \mathsf{index}^k\}$ and $m = \emptyset$, or $m = m^i$ for the largest $i \in [k]$ such that $\mathsf{index}^i = \mathsf{index}^*$.*

Note that this is an information-theoretic property. We are requiring that for all other symbols m, values of π that would cause VerifyRead to output 1 at index^* do not exist.

Definition A.6 (Write-Enforcing)**.** *Consider any $\lambda \in \mathbb{N}, S \in \Theta(2^\lambda)$, $m^1, \ldots, m^k \in \mathcal{M}_\lambda$, $\mathsf{index}^1, \ldots, \mathsf{index}^k \in \{0, \ldots, S-1\}$.*

Let $(\mathsf{pp}_{\mathsf{Acc}}, w^0, store^0) \leftarrow \mathsf{Acc.SetupEnforceWrite}(1^\lambda, S, (m^1, \mathsf{index}^1), \ldots, (m^k, \mathsf{index}^k))$.

For all $j \in [k]$, we define $store^j$ iteratively as $store^j := \mathsf{WriteStore}(\mathsf{pp}_{\mathsf{Acc}}, store^{j-1}, \mathsf{index}^j, m^j)$.

We similarly define aux^j and w^j iteratively as $aux^j := \mathsf{PrepWrite}(\mathsf{pp}_{\mathsf{Acc}}, store^{j-1}, \mathsf{index}^j)$ and $w^j := \mathsf{Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{j-1}, m^j, \mathsf{index}^j, aux^j)$.

Acc *is said to be* write-enforcing *if* $\mathsf{Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{k-1}, m^k, \mathsf{index}^k, aux) = w^{\text{out}} \neq \texttt{Reject}$ *for any aux, then $w^{\text{out}} = w^k$.*

Note that this is an information-theoretic property: we are requiring that an aux value producing an accumulated value other than w^k or `Reject` does not exist.

A.3.3 Splittable Signatures

Splittable signatures [KLW15] are normal signatures with additional algorithms and properties. In particular, the following keys are introduced:

- "All but one" keys function normally except for a particular message m^*.
- "One" keys function only for a particular message m^*.
- Reject-verification keys reject all signatures when used for verification.

The security requirement of splittable signatures is weaker than that of normal signatures in the sense that no signing oracle is provided to the adversary. This weaker requirement is sufficient for our applications and enables us to argue the indistinguishability between different types of verification keys.

Syntax A splittable signature scheme Spl for message space $\mathcal{M}_\lambda$ consists of the following algorithms:

- Setup: The setup algorithm is a probabilistic algorithm that takes as input the security parameter λ and outputs a signing key sk, a verification key vk, and reject-verification key $\mathrm{vk}_{\mathrm{rej}}$.
- Sign: The signing algorithm is a deterministic algorithm that takes as input a signing key sk and a message $m \in \mathcal{M}_\lambda$. It outputs a signature σ.
- Verify: The verification algorithm is a deterministic algorithm that takes as input a verification key vk, signature σ, and a message m. It outputs either 0 or 1.
- Split: The splitting algorithm is probabilistic. It takes as input a secret key sk and a message $m^* \in \mathcal{M}_\lambda$. It outputs a signature $\sigma_{\mathrm{one}} \leftarrow \mathsf{Sign}(\mathrm{sk}, m^*)$, a one-message verification key $\mathrm{vk}_{\mathrm{one}}$, an all-but-one signing key $\mathrm{sk}_{\mathrm{abo}}$, and an all-but-one verification key $\mathrm{vk}_{\mathrm{abo}}$.
- $\mathsf{AboSign}$: The all-but-one signing algorithm is deterministic. It takes as input an all-but-one signing key $\mathrm{sk}_{\mathrm{abo}}$ and a message m, and outputs a signature σ.

Correctness Let $(\mathrm{sk}, \mathrm{vk}, \mathrm{vk}_{\mathrm{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$. Let $m^* \in \mathcal{M}_\lambda$ be any message and $(\sigma_{\mathrm{one}}, \mathrm{vk}_{\mathrm{one}}, \mathrm{sk}_{\mathrm{abo}}, \mathrm{vk}_{\mathrm{abo}}) \leftarrow \mathsf{Spl.Split}(\mathrm{sk}, m^*)$. We require the following correctness properties:

1. For all $m \in \mathcal{M}_\lambda$, $\mathsf{Spl.Verify}(\mathrm{vk}, m, \mathsf{Spl.Sign}(\mathrm{sk}, m)) = 1$.
2. For all $m \in \mathcal{M}_\lambda$, $m \neq m^*$, $\mathsf{Spl.Sign}(\mathrm{sk}, m) = \mathsf{Spl.AboSign}(\mathrm{sk}_{\mathrm{abo}}, m)$.
3. For all σ, $\mathsf{Spl.Verify}(\mathrm{vk}, m^*, \sigma) = \mathsf{Spl.Verify}(\mathrm{vk}, m^*, \sigma)$.
4. For all $m \neq m^*$ and σ, $\mathsf{Spl.Verify}(\mathrm{vk}, m, \sigma) = \mathsf{Spl.Verify}(\mathrm{vk}_{\mathrm{abo}}, m, \sigma)$.
5. For all $m \neq m^*$ and σ, $\mathsf{Spl.Verify}(\mathrm{vk}_{\mathrm{one}}, m, \sigma) = 0$.
6. For all σ, $\mathsf{Spl.Verify}(\mathrm{vk}_{\mathrm{abo}}, m^*, \sigma) = 0$.
7. For all σ and all $m \in \mathcal{M}_\lambda$, $\mathsf{Spl.Verify}(\mathrm{vk}_{\mathrm{rej}}, m, \sigma) = 0$.

Security We will now define the security notions for splittable signature schemes. Each security notion is defined in terms of a security game between a challenger and an adversary $\mathcal{A}$.

Definition A.7 ($\mathrm{vk}_{\mathrm{rej}}$ indistinguishability)**.** *A splittable signature scheme* Spl *is said to be* $\mathrm{vk}_{\mathrm{rej}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

Exp-$\mathrm{vk}_{\mathrm{rej}}(1^\lambda, \mathsf{Spl}, \mathcal{A})$

- *The challenger computes* $(\mathrm{sk}, \mathrm{vk}, \mathrm{vk}_{\mathrm{rej}}) \leftarrow \mathsf{Setup}$*. It chooses a bit* $b \in \{0,1\}$*. If* $b = 0$*, the challenger sends* vk *to* $\mathcal{A}$*. Else, it sends* $\mathrm{vk}_{\mathrm{rej}}$ *to* $\mathcal{A}$*.*
- $\mathcal{A}$ *sends a bit* b'*.*

$\mathcal{A}$ *wins if* $b = b'$*.*

Definition A.8 ($\mathrm{vk}_{\mathrm{one}}$ indistinguishability)**.** *A splittable signature scheme* Spl *is said to be* $\mathrm{vk}_{\mathrm{one}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

Exp-$\mathrm{vk}_{\mathrm{one}}(1^\lambda, \mathsf{Spl}, \mathcal{A})$

- $\mathcal{A}$ *sends a message* $m^* \mathcal{M}_\lambda$*.*
- *The challenger computes* $(\mathrm{sk}, \mathrm{vk}, \mathrm{vk}_{\mathrm{rej}}) \leftarrow \mathsf{Setup}$*, and computes* $(\sigma_{\mathrm{one}}, \mathrm{vk}_{\mathrm{one}}, \mathrm{sk}_{\mathrm{abo}}, \mathrm{vk}_{\mathrm{abo}}) \leftarrow \mathsf{Split}(\mathrm{sk}, m^*)$*. It chooses a bit* $b \in \{0,1\}$*. If* $b = 0$*, the challenger sends* $(\sigma_{\mathrm{one}}, \mathrm{vk}_{\mathrm{one}})$ *to* $\mathcal{A}$*. Else, it sends* $(\sigma_{\mathrm{one}}, \mathrm{vk})$ *to* $\mathcal{A}$*.*
- $\mathcal{A}$ *sends a bit* b'*.*

$\mathcal{A}$ *wins if* $b = b'$*.*

Definition A.9 ($\mathrm{vk_{abo}}$ indistinguishability)**.** *A splittable signature scheme* Spl *is said to be* $\mathrm{vk_{abo}}$ *indistinguishable if any PPT adversary* $\mathcal{A}$ *has negligible advantage in the following security game:*

Exp-$\mathrm{vk_{abo}}(1^\lambda, \mathsf{Spl}, \mathcal{A})$
- *$\mathcal{A}$ sends a message $m^* \in \mathcal{M}_\lambda$.*
- *The challenger computes* $(\mathrm{sk}, \mathrm{vk}, \mathrm{vk_{rej}}) \leftarrow \mathsf{Setup}$*, and* $(\sigma_{\mathrm{one}}, \mathrm{vk_{one}}, \mathrm{sk_{abo}}, \mathrm{vk_{abo}}) \leftarrow \mathsf{Split}(\mathrm{sk}, m^*)$*. It chooses a bit $b \in \{0,1\}$. If $b = 0$, the challenger sends* $(\mathrm{sk_{abo}}, \mathrm{vk_{abo}})$ *to $\mathcal{A}$. Else, it sends* $(\mathrm{sk_{abo}}, \mathrm{vk})$ *to $\mathcal{A}$.*
- *$\mathcal{A}$ sends a bit b'.*

$\mathcal{A}$ wins if $b = b'$.

Definition A.10 (Splitting indistinguishability)**.** *A splittable signature scheme* Spl *is said to be splitting indistinguishable if any PPT adversary $\mathcal{A}$ has negligible advantage in the following security game:*

Exp-$\mathrm{vk_{abo}}(1^\lambda, \mathsf{Spl}, \mathcal{A})$
- *$\mathcal{A}$ sends a message $m^* \in \mathcal{M}_\lambda$.*
- *The challenger computes* $(\mathrm{sk}, \mathrm{vk}, \mathrm{vk_{rej}}) \leftarrow \mathsf{Setup}(1^\lambda)$*,* $(\mathrm{sk}', \mathrm{vk}', \mathrm{vk}'_{\mathrm{rej}}) \leftarrow \mathsf{Setup}(1^\lambda)$*, and computes* $(\sigma_{\mathrm{one}}, \mathrm{vk_{one}}, \mathrm{sk_{abo}}, \mathrm{vk_{abo}}) \leftarrow \mathsf{Split}(\mathrm{sk}, m^*)$*,* $(\sigma'_{\mathrm{one}}, \mathrm{vk}'_{\mathrm{one}}, \mathrm{sk}'_{\mathrm{abo}}, \mathrm{vk}'_{\mathrm{abo}}) \leftarrow \mathsf{Split}(\mathrm{sk}', m^*)$*. It chooses a bit $b \in \{0,1\}$. If $b = 0$, the challenger sends* $(\sigma_{\mathrm{one}}, \mathrm{vk_{one}}, \mathrm{sk_{abo}}, \mathrm{vk_{abo}})$ *to $\mathcal{A}$. Else, if $b = 1$, it sends* $(\sigma'_{\mathrm{one}}, \mathrm{vk}'_{\mathrm{one}}, \mathrm{sk_{abo}}, \mathrm{vk_{abo}})$ *to $\mathcal{A}$.*
- *$\mathcal{A}$ sends a bit b'.*

$\mathcal{A}$ wins if $b = b'$.

A.3.4 Indistinguishability Obfuscation for Circuits

Definition A.11 (Indistinguishability Obfuscation for Circuits [GGH+13, SW14])**.** *Let $\mathcal{C} = \{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ be a family of polynomial-size circuits. Let $i\mathcal{O}$ be a uniform PPT algorithm that takes as input the security parameter λ, a circuit $C \in \mathcal{C}_\lambda$ and outputs a circuit C'. $i\mathcal{O}$ is called an indistinguishability obfuscator for a circuit class $\{\mathcal{C}_\lambda\}_{\lambda \in \mathbb{N}}$ if it satisfies the following conditions:*
- *(Preserving Functionality.) For all security parameters $\lambda \in \mathbb{N}$, for all circuits $C \in \mathcal{C}_\lambda$, for all inputs x, we have that $C'(x) = C(x)$ where $C' \leftarrow i\mathcal{O}(1^\lambda, C)$.*
- *(Indistinguishability of Obfuscation.) For any (not necessarily uniform) PPT distinguisher $\mathcal{B} = (Samp, \mathcal{D})$, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that the following holds: if for all security parameters $\lambda \in \mathbb{N}$,* $\Pr[\forall x, C_0(x) = C_1(x) : (C_0; C_1; \sigma) \leftarrow Samp(1^\lambda)] > 1 - \mathsf{negl}(\lambda)$*, then*

$$|\Pr[\mathcal{D}(\sigma, i\mathcal{O}(1^\lambda, C_0)) = 1 : (C_0; C_1; \sigma) \leftarrow Samp(1^\lambda)] - \\ \Pr[\mathcal{D}(\sigma, i\mathcal{O}(1^\lambda, C_1)) = 1 : (C_0; C_1; \sigma) \leftarrow Samp(1^\lambda)] \le \mathsf{negl}(\lambda).$$

  In addition, we require the efficiency of the input circuit to be preserved.
- *(Preserving Efficiency.) For all security parameters $\lambda \in \mathbb{N}$, for all circuits $C \in \mathcal{C}_\lambda$, we have that $|C'| =$* $\mathsf{poly}(\lambda)|C|$ *where $C' \leftarrow i\mathcal{O}(1^\lambda, C)$.*

A.3.5 Puncturable Pseudorandom Functions

Puncturable pseudorandom functions [BW13, BGI14, KPTZ13, SW14] have proven to be very powerful since introduced. Here we review the definition.

Syntax A function $\mathsf{PRF} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is a puncturable pseudorandom function if there is an additional key space $\mathcal{K}_{\mathsf{punct}}$ and three polynomial time algorithms $\mathsf{PPRF.Setup}$, $\mathsf{PPRF.Puncture}$, and $\mathsf{PPRF.Eval}$ as follows:
- $\mathsf{PPRF.Setup}(1^\lambda)$ is a probabilistic algorithm that takes the security parameter λ as input, and outputs a description of the key space $\mathcal{K}$, the punctured key space $\mathcal{K}_{\mathsf{punct}}$, and the function PRF.
- $\mathsf{PPRF.Puncture}(K, x)$ is a probabilistic algorithm that takes as input a PRF key $K \in \mathcal{K}$ and $x \in \mathcal{X}$, and outputs a key $K\{x\} \in \mathcal{K}_{\mathsf{punct}}$.

– $\mathsf{PPRF.Eval}(K\{x\}, x')$ is a deterministic algorithm that takes as input a punctured key $K\{x\} \in \mathcal{K}_{\mathsf{punct}}$ and $x' \in \mathcal{X}$. Let $K \in \mathcal{K}$, $x \in \mathcal{X}$ and $K\{x\} \leftarrow \mathsf{PPRF.Puncture}(K, x)$. For correctness, we require:

$$\mathsf{PPRF.Eval}(K\{x\}, x') = \begin{cases} \mathsf{PRF}(K, x') & \text{if } x' \neq x \\ \bot & \text{otherwise} \end{cases}$$

For convenience, we simply write $\mathsf{PRF}(K\{x\}, x')$ to denote $\mathsf{PPRF.Eval}(K\{x\}, x')$ when the context is clear.

Security We now define the selective security for puncturable PRFs.

Definition A.12 (Selective Security)**.** *We say a puncturable PRF scheme* PPRF *is* selectively secure *if for all probabilistic polynomial time adversaries $\mathcal{A}$, its advantage* $\mathsf{Adv}_{\mathcal{A},\mathsf{PPRF}}(\lambda)$ *in the following security game is negligible in λ:*

Challenge Phase *$\mathcal{A}$ sends a challenge $x^* \in \mathcal{X}$. The challenger chooses uniformly at random a PRF key $K \leftarrow \mathcal{K}$ and a bit $b \leftarrow \{0,1\}$. It computes $K\{x^*\} \leftarrow$* $\mathsf{PPRF.Puncture}(K, x^*)$*. If $b = 0$, the challenger sets $y =$* $\mathsf{PRF}(K, x^*)$*, else it chooses uniformly at random $y \leftarrow \mathcal{Y}$. It sends $K\{x^*\}$, y to $\mathcal{A}$.*

Guess Phase *$\mathcal{A}$ outputs a guess b' of b.*

$\mathcal{A}$ wins if $b = b'$. The advantage of $\mathcal{A}$ is defined to be $\mathsf{Adv}_{\mathcal{A},\mathsf{PPRF}}(\lambda) = \Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}$.

B Security Proofs

B.1 Proof of Theorem 5.2 (Security for $\mathsf{Ci}\mathcal{O}$-RAM)

Proof. Let $\mathsf{Adv}^x_{\mathcal{A}}$ be the advantage of the adversary $\mathcal{A}$ in the hybrid $\mathbf{Hyb}_x$. We first define the first-layer hybrids $\mathbf{Hyb}_i$ for $i \in \{0, 1\}$.

$\mathbf{Hyb}_i$ for $i \in \{0,1\}$ In this hybrid, the challenger outputs an obfuscation computation system of Π^i where stateful algorithm $\widehat{F}^i$ is defined in Algorithm 23.

Let us assume the obfuscated computation system terminates at time $t^* < T$. We argue that $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$. To show this, we define the second-layer hybrids $\mathbf{Hyb}_{0,1}$, $\mathbf{Hyb}_{0,2}$, $\mathbf{Hyb}_{0,3}$ and $\mathbf{Hyb}_{0,4}$. We also define important third-layer hybrids $\mathbf{Hyb}_{0,2,i}$ and $\mathbf{Hyb}_{0,2',i}$ for $0 \leq i < t^*$.

$\mathbf{Hyb}_{0,0}$ This hybrid is identical to $\mathbf{Hyb}_0$ in the first layer.

$\mathbf{Hyb}_{0,1}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,1}$ defined in Algorithm 25. This program is similar to $\widehat{F}^0$ except that it has PRF key K_B hardwired, accepts both 'A' and 'B' type signatures for $t < t^*$. The type of the outgoing signature follows the type of the incoming signature. Also, if the incoming signature is 'B' type and $t < t^*$, then the program uses F^1 to compute the output.

$\mathbf{Hyb}_{0,2}$ This hybrid is identical to $\mathbf{Hyb}_{0,2,0}$ defined below.

$\mathbf{Hyb}_{0,2,i}$ In this hybrid, based on the initial configuration $(\mathsf{mem}^0, \mathsf{st}^0, a^0_{\mathtt{A}\leftarrow\mathtt{M}} = \bot, a^0_{\mathtt{M}\leftarrow\mathtt{A}} = \bot)$, the challenger computes m^i as follows:

Then the challenger outputs an obfuscation of $\widehat{F}^{0,2,i}$ defined in Algorithm 26. This program is similar to $\widehat{F}^{0,1}$ except that it accepts 'B' type signatures only for inputs corresponding to $i+1 \leq t \leq t^*-1$. It also has the correct output message m^i for step i hardwired. For $i+1 \leq t \leq t^*-1$, the type of the outgoing signature follows the type of the incoming signature. At $t = i$, it outputs an 'A' type signature if $m^{\mathrm{out}} = m^i$, or outputs 'B' type signature otherwise.

$\mathbf{Hyb}_{0,2',i}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,2',i}$ defined in Algorithm 27. This program is similar to $\widehat{F}^{0,2,i}$ except that it accepts 'B' type signatures only for inputs corresponding to $i+2 \leq t \leq t^*-1$. It also has the correct input message m^i for step $i+1$ hardwired. For $i+2 \leq t \leq t^*-1$, the type of the outgoing signature follows the type of the incoming signature. At $t = i+1$, it outputs an 'A' type signature if $m^{\mathrm{in}} = m^i$, or outputs 'B' type signature otherwise.

$\mathbf{Hyb}_{0,3}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,3}$ defined in Algorithm 28. This program is similar to $\widehat{F}^{0,2',t^*-1}$, except that it does not output 'B' type signatures.

$\mathbf{Hyb}_{0,4}$ In this hybrid, the challenger outputs the obfuscation of $\widehat{F}^{0,4}$ defined in Algorithm 29. This program outputs `Reject` for all $t > t^*$ including the case when the signature is a valid 'A' type signature.

Analysis In the remaining of this subsection, we prove Lemmas B.1, B.9, B.23, B.33, and B.37.

By Lemma B.1, we have $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^{0,1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$. Since $\widehat{F}^{0,1}$ and $\widehat{F}^{0,2,0}$ have identical functionality, we have $|\mathsf{Adv}^{0,1}_{\mathcal{A}} - \mathsf{Adv}^{0,2,0}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$. By Lemma B.9, we have $|\mathsf{Adv}^{0,2,i}_{\mathcal{A}} - \mathsf{Adv}^{0,2',i}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$ for $0 \leq i \leq t^*-1$. By Lemma B.23, we have $|\mathsf{Adv}^{0,2',i}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i+1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$ for $0 \leq i \leq t^*-2$. By Lemma B.33, we have $|\mathsf{Adv}^{0,2',t^*-1}_{\mathcal{A}} - \mathsf{Adv}^{0,3}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$. By Lemma B.37, we have $|\mathsf{Adv}^{0,3}_{\mathcal{A}} - \mathsf{Adv}^{0,4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$. Summarizing the above, we have $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^{0,4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Symmetrically, we can show that $|\mathsf{Adv}^1_{\mathcal{A}} - \mathsf{Adv}^{0,4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$. Finally, we can conclude that $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$, which completes the proof. ☐

Algorithm 23: $\widehat{F}^i$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$,  $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$;
4  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$;
5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 0$ then output Reject;

6  Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^i(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$;
7  if $\mathsf{st}^{\mathrm{out}} =$ Reject then output Reject;

8  $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
9  if $w^{\mathrm{out}} =$ Reject then output Reject;
10 Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
11 if $v^{\mathrm{out}} =$ Reject then output Reject;
12 Compute $r'_A = \mathsf{PRF}(K_A, t)$;
13 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$;
14 Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
15 Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;

16 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$,    $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 24: This algorithm is used in $\mathbf{Hyb}_{0,2,i}$

```
1  for $j \in \{1, \ldots, i\}$ do
2      Compute $(\mathsf{st}^j, a^j_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{j-1}, a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}})$ ;      // $a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{j-1}, \mathbf{B}^{j-1})$
3      $(a^j_{\mathtt{M}\leftarrow\mathtt{A}}, \pi^j) \leftarrow \mathsf{Acc.PrepRead}(\mathsf{pp}_{\mathsf{Acc}}, store^j, \mathbf{I}^j)$
4      $w^j \leftarrow \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{j-1}, a^j_{\mathtt{M}\leftarrow\mathtt{A}}, \pi^j)$
5      $store^j \leftarrow \mathsf{Acc.WriteStore}(\mathsf{pp}_{\mathsf{Acc}}, store^{j-1}, {}^{j}_{,\mathtt{M}\leftarrow\mathtt{A}}, a^j_{\mathtt{M}\leftarrow\mathtt{A}})$
6      $v^j \leftarrow \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{j-1}, (\mathsf{st}^{j-1}, w^{j-1}, \mathbf{I}^{j-1}))$
7  Set $m^i = (v^i, \mathsf{st}^i, w^i, \mathbf{I}^i)$;
```

B.1.1 From $\mathbf{Hyb}_{0,0}$ to $\mathbf{Hyb}_{0,1}$:

Lemma B.1. *Let $i\mathcal{O}$ be a secure indistinguishability obfuscator,* PRF *be a selectively secure puncturable PRF, and* Spl *be a secure splittable signature scheme; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^{0,1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. We define third layer hybrids $\mathbf{Hyb}_{0,0,i}$ where $i \in \{0, 1, \ldots, t^*\}$.

$\mathbf{Hyb}_{0,0,i}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}_{0,0,i}$ defined in Algorithm 30. This program is similar to $\widetilde{F}_0$ except that it has PRF key K_B hardwired, accepts both 'A' and 'B' type signatures for $i < t \leq t^*$. The type of the outgoing signature follows the type of the incoming signature.

```
Algorithm 25: $\widehat{F}^{0,1}$
    Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
    Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, \underline{K_B}$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $\underline{r_B = \mathsf{PRF}(K_B, t-1)}$;
3  Compute $(\text{sk}_A, \text{vk}_A, \text{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $\underline{(\text{sk}_B, \text{vk}_B, \text{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)}$;
4  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\underline{\alpha = \text{‘-’}}$ ;
5  $\underline{\textbf{if } \mathsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1 \textbf{ then } \text{set } \alpha = \text{‘A’}}$ ;
6  $\underline{\textbf{if } \alpha = \text{‘-’} \textbf{ and } t > t^* \textbf{ then } \text{output } \texttt{Reject};}$
7  $\underline{\textbf{if } \alpha \neq \text{‘A’} \textbf{ and } \mathsf{Spl.Verify}(\text{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1 \textbf{ then } \text{set } \alpha = \text{‘B’}}$ ;
8  $\underline{\textbf{if } \alpha = \text{‘-’} \textbf{ then } \text{output } \texttt{Reject};}$

9  if $\alpha$ = ‘B’ then
10     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
11 else
12     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
13 if $\mathsf{st}^{\text{out}} = \texttt{Reject}$ then output Reject;

14 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
15 if $w^{\text{out}} = \texttt{Reject}$ then output Reject;
16 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
17 if $v^{\text{out}} = \texttt{Reject}$ then output Reject;
18 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $\underline{r'_B = \mathsf{PRF}(K_B, t)}$;
19 Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $\underline{(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)}$;
20 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
21 Compute $\underline{\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_\alpha, m^{\text{out}})}$;

22 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,  $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 26: $\widehat{F}^{0,2,i}$

```
Input  : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data   : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, \underline{m^i}$

if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ '-' ;
if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A' ;
if $\alpha =$ '-' and $(t > t^*$ $\underline{\textbf{or } t \leq i}$ $)$ then output Reject;
if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B' ;
if $\alpha =$ '-' then output Reject;

if $\alpha =$ 'B' $\underline{\textbf{or } t \leq i}$ then
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
else
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;

$w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
if $w^{\text{out}} =$ Reject then output Reject;
Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
if $v^{\text{out}} =$ Reject then output Reject;
Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
if $\underline{t = i \textbf{ and } m^{\text{out}} = m^i}$ then
    $\underline{\text{Compute } \sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}})}$;
else if $\underline{t = i \textbf{ and } m^{\text{out}} \neq m^i}$ then
    $\underline{\text{Compute } \sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\text{out}})}$;
else
    $\underline{\text{Compute } \sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\text{out}})}$;

Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 27: $\widehat{F}^{0,2',i}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A} \leftarrow \mathtt{M}} = (a^{\text{in}}_{\mathtt{A} \leftarrow \mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A} \leftarrow \mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m^i$

if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ '-' ;
if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A' ;
if $\alpha =$ '-' and $(t > t^*$ $\underline{\textbf{or } t \leq i+1}$ $)$ then output Reject;
if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B' ;
if $\alpha =$ '-' then output Reject;

if $\alpha =$ 'B' $\underline{\textbf{or } t \leq i+1}$ then
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M} \leftarrow \mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A} \leftarrow \mathtt{M}})$
else
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M} \leftarrow \mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A} \leftarrow \mathtt{M}})$
if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;

$w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
if $w^{\text{out}} =$ Reject then output Reject;
Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
if $v^{\text{out}} =$ Reject then output Reject;
Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
if $\underline{t = i+1 \textbf{ and } m^{\text{in}} = m^i}$ then
    $\underline{\text{Compute } \sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}})}$;
else if $\underline{t = i+1 \textbf{ and } m^{\text{in}} \neq m^i}$ then
    $\underline{\text{Compute } \sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\text{out}})}$;
else
    Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\text{out}})$;

Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,  $\widetilde{a}^{\text{out}}_{\mathtt{M} \leftarrow \mathtt{A}} = a^{\text{out}}_{\mathtt{M} \leftarrow \mathtt{A}}$;
```

Algorithm 28: $\widehat{F}^{0,3}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, t^*$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
3  Compute $(\text{sk}_A, \text{vk}_A, \text{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\text{sk}_B, \text{vk}_B, \text{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ ;
5  if $\mathsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 0$ then output Reject;

6  if $t \leq t^*$ then
7      Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
8  else
9      Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
10 if $\mathsf{st}^{\text{out}} = \texttt{Reject}$ then output Reject;

11 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
12 if $w^{\text{out}} = \texttt{Reject}$ then output Reject;
13 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
14 if $v^{\text{out}} = \texttt{Reject}$ then output Reject;
15 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
16 Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
17 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
18 if $t = t^*$ then
19     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_B, m^{\text{out}})$;
20 else
21     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_A, m^{\text{out}})$;

22 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,    $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 29: $\widehat{F}^{0,4}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, t^*$

1  if $t > t^*$ then output Reject;
2  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
3  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
4  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
5  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ ;
6  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 0$ then output Reject;

7  if $t \leq t^*$ then
8      Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
9  if $\mathsf{st}^{\text{out}} = $ Reject then output Reject;

10 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
11 if $w^{\text{out}} = $ Reject then output Reject;
12 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
13 if $v^{\text{out}} = $ Reject then output Reject;
14 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
15 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
16 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
17 if $t = t^*$ then
18     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\text{out}})$;
19 else
20     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}})$;

21 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

We observe that hybrids $\mathbf{Hyb}_{0,0,0}$ and $\mathbf{Hyb}_{0,1}$ are identical. In addition, hybrids $\mathbf{Hyb}_{0,0,t^*}$ and $\mathbf{Hyb}_{0,0}$ are functionally identical, since the difference between these two hybrids is a dummy code which has never been executed. Therefore, it suffices to show that $\mathbf{Hyb}_{0,0,i}$ and $\mathbf{Hyb}_{0,0,i-1}$ are computationally indistinguishable for $0 \le i \le t^*$, which is implied by Lemma B.2. □

Algorithm 30: $\widehat{F}_{0,0,i}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, \underline{K_B}$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ '-' ;
5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A' ;
6  if $\alpha =$ '-' and ($t > t^*$ \underline{or $t \le i$}) then output Reject;
7  if $\alpha =$ '-' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B' ;
8  if $\alpha =$ '-' then output Reject;

9  if $\alpha =$ 'B' then
10     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
11 else
12     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
13 if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;

14 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
15 if $w^{\text{out}} =$ Reject then output Reject;
16 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
17 if $v^{\text{out}} =$ Reject then output Reject;
18 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
19 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
20 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
21 Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\text{out}})$;

22 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Lemma B.2. *Let $i\mathcal{O}$ be a secure indistinguishability obfuscator,* PRF *be a selectively secure puncturable PRF, and* Spl *be a secure splittable signature scheme; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}^{0,0,i}_{\mathcal{A}} - \mathsf{Adv}^{0,0,i-1}_{\mathcal{A}}| \le \mathsf{negl}(\lambda)$.

Proof. We define fourth-layer hybrids $\mathbf{Hyb}_{0,0,i,a}, \ldots, \mathbf{Hyb}_{0,0,i,f}$.

$\mathbf{Hyb}_{0,0,i,a}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,0,i,a}$ defined in Algorithm 31. This program is similar to $\widehat{F}^{0,0,i}$ except that when $t = i$, it verifies the signature using $\mathrm{vk}_{B,\text{rej}}$ if it is not accepted as 'A' type signature.

$\mathbf{Hyb}_{0,0,i,b}$ In this hybrid, the challenger first punctures the PRF key K_B on input $i-1$ by computing $K_B\{i-1\} \leftarrow \mathsf{PRF.Puncture}(K_B, i-1)$. Next, it computes $r_C = \mathsf{PRF}(K_B, i-1)$ and $(\mathrm{sk}_C, \mathrm{vk}_C, \mathrm{vk}_{C,\text{rej}}) =$

$\mathsf{Spl.Setup}(1^\lambda; r_C)$. It outputs an obfuscation of $\widehat{F}^{0,0,i,b}$ defined in Algorithm 32. This program is similar to $\widehat{F}^{0,0,i,a}$ except that it has $K_B\{i-1\}$ and $\mathrm{vk}_{C,\mathrm{rej}}$ hardwired, and when $t = i$, it replaces $\mathrm{vk}_{B,\mathrm{rej}}$ by $\mathrm{vk}_{C,\mathrm{rej}}$.

$\mathbf{Hyb}_{0,0,i,c}$ This hybrid is similar to $\mathbf{Hyb}_{0,0,i,b}$, except that r_C is now chosen uniformly at random from $\{0,1\}^\lambda$.

$\mathbf{Hyb}_{0,0,i,d}$ This hybrid is similar to $\mathbf{Hyb}_{0,0,i,c}$, except that vk_C instead of $\mathrm{vk}_{C,\mathrm{rej}}$ is hardwired to the program.

$\mathbf{Hyb}_{0,0,i,e}$ This hybrid is similar to $\mathbf{Hyb}_{0,0,i,d}$, except that $r_C = \mathsf{PRF}(K_B, i-1)$ is now pseudorandom.

$\mathbf{Hyb}_{0,0,i,f}$ This hybrid is identical to $\mathbf{Hyb}_{0,0,i-1}$.

Analysis In the remaining we prove the following claims:

Claim B.3. *Let $i\mathcal{O}$ be a secure indistinguishability obfuscator; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i,a}| \leq \mathsf{negl}(\lambda)$.

Proof. Observe that $\widehat{F}^{0,0,i}$ and $\widehat{F}^{0,0,i,a}$ have identical functionality. Therefore $\mathbf{Hyb}_{0,0,i}$ and $\mathbf{Hyb}_{0,0,i,a}$ are computationally indistinguishable under the assumption that $i\mathcal{O}$ is a secure indistinguishability obfuscation scheme. □

Claim B.4. *Let $i\mathcal{O}$ be a secure indistinguishability obfuscator; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i,a} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i,b}| \leq \mathsf{negl}(\lambda)$.

Proof. Note that the only difference between $\widehat{F}^{0,0,i,a}$ and $\widehat{F}^{0,0,i,b}$ is that the latter uses a punctured PRF key $K_B\{i-1\}$ to compute the verification key for time $t-1$ and the signing key for time t. For verification, the functionality is preserved since $\mathrm{vk}_{C,\mathrm{rej}}$ is hardwired to the circuit. For signing, 'B' type key is never used to sign at time $t = i-1$. Therefore $\mathbf{Hyb}_{0,0,i,a}$ and $\mathbf{Hyb}_{0,0,i,b}$ are computationally indistinguishable. □

Claim B.5. *Let PRF be a selectively secure puncturable PRF; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i,b} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i,c}| \leq \mathsf{negl}(\lambda)$.

Proof. Since both $\widehat{F}^{0,0,i,b}$ and $\widehat{F}^{0,0,i,c}$ depend only on $K_B\{i-1\}$, by the security of indistinguishability obfuscation, the value of $\mathsf{PRF}(K_B, i-1)$ can be replaced by a random value. Therefore $\mathbf{Hyb}_{0,0,i,b}$ and $\mathbf{Hyb}_{0,0,i,c}$ are computationally indistinguishable under the assumption that PRF is a selectively secure puncturable PRF. □

Claim B.6. *Let Spl be a splittable signature scheme which satisfies $\mathrm{vk}_{\mathrm{rej}}$ indistinguishability (Definition A.7); then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i,c} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i,d}| \leq \mathsf{negl}(\lambda)$.

Proof. Note that sk_C is not hardwired in either $\widehat{F}^{0,0,i,c}$ or $\widehat{F}^{0,0,i,d}$. Based on the $\mathrm{vk}_{\mathrm{rej}}$ indistinguishability property of splittable signature scheme Spl, given only vk_C or $\mathrm{vk}_{C,\mathrm{rej}}$, the two hybrids are computationally indistinguishable. □

Claim B.7. *Let PRF be a selectively secure puncturable PRF; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i,d} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i,e}| \leq \mathsf{negl}(\lambda)$.

Proof. Since both $\widehat{F}^{0,0,i,d}$ and $\widehat{F}^{0,0,i,e}$ depend only on $K_B\{i-1\}$, by the security of indistinguishability obfuscation, the random value can be switched back to $\mathsf{PRF}(K_B, i-1)$. Therefore $\mathbf{Hyb}_{0,0,i,d}$ and $\mathbf{Hyb}_{0,0,i,e}$ are computationally indistinguishable. □

Claim B.8. *Let $i\mathcal{O}$ be a secure indistinguishability obfuscator; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i,e} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i,f}| \leq \mathsf{negl}(\lambda)$.

Proof. Observe that $\widehat{F}^{0,0,i,e}$ and $\widehat{F}^{0,0,i,f}$ have identical functionality. $\square$

To conclude, we have for all PPT $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i-1}| \leq \mathsf{negl}(\lambda)$ as required.

$\square$

Algorithm 31: $\widehat{F}^{0,0,i,a}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A \leftarrow M}} = (a^{\text{in}}_{\mathtt{A \leftarrow M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A \leftarrow M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B$

if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
$\underline{\text{Set } \mathrm{vk} = \mathrm{vk}_{B,\mathrm{rej}}}$;
Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ '-' ;
if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A' ;
if $\alpha =$ '-' and ($t > t^*$ $\underline{\textbf{or } t \leq i-1}$) then output Reject;
$\underline{\textbf{if } \alpha = \text{'-'} \textbf{ and } t = i \textbf{ and } \mathsf{Spl.Verify}(\mathrm{vk}, m^{\text{in}}, \sigma^{\text{in}}) = 0 \textbf{ then output Reject;}}$
if $\alpha =$ '-' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B' ;
if $\alpha =$ '-' then output Reject;

if $\alpha =$ 'B' then
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M \leftarrow A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A \leftarrow M}})$
else
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M \leftarrow A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A \leftarrow M}})$
if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;

$w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
if $w^{\text{out}} =$ Reject then output Reject;
Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
if $v^{\text{out}} =$ Reject then output Reject;
Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\text{out}})$;

Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,  $\widetilde{a}^{\text{out}}_{\mathtt{M \leftarrow A}} = a^{\text{out}}_{\mathtt{M \leftarrow A}}$;
```

B.1.2 From $\mathbf{Hyb}_{0,2,i}$ to $\mathbf{Hyb}_{0,2',i}$

Lemma B.9. *Let $1 \leq i < t^*$. Assume $\mathsf{i}\mathcal{O}$ is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* Spl *is a secure splittable signature scheme, and* Acc *is a secure positional accumulator scheme; then for any PPT adversary $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i}| \leq \mathsf{negl}(\lambda)$.*

Proof. We define fourth layer hybrids $\mathbf{Hyb}_{0,2,i,0}$, $\mathbf{Hyb}_{0,2,i,1}$, $\ldots$, $\mathbf{Hyb}_{0,2,i,13}$. The first hybrid corresponds to $\mathbf{Hyb}_{0,2,i}$, and the last one corresponds to $\mathbf{Hyb}_{0,2',i}$.

$\mathbf{Hyb}_{0,2,i,0}$ This hybrid corresponds to $\mathbf{Hyb}_{0,2,i}$.

Algorithm 32: $\widehat{F}^{0,0,i,b}$

```
Input : $\widetilde{\mathsf{st}}^{\rm in} = (t, \mathsf{st}^{\rm in}, v^{\rm in}, w^{\rm in}, \sigma^{\rm in})$,  $\widetilde{a}^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\rm in})$ where $a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\rm in}, \mathbf{B}^{\rm in})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, \underline{K_B\{i-1\}, \mathrm{vk}_{C,\mathrm{rej}}}$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\rm in}, \mathbf{I}^{\rm in}, \mathbf{B}^{\rm in}, \pi^{\rm in}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$;
4  if $\underline{t \neq i}$ then
5      $\underline{\text{Compute } r_B = \mathsf{PRF}(K_B\{i-1\}, t-1)}$;
6      $\underline{\text{Compute } (\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)}$;
7      $\underline{\text{Set } \mathrm{vk} = \mathrm{vk}_{B,\mathrm{rej}}}$;
8  else
9      $\underline{\text{Set } \mathrm{vk} = \mathrm{vk}_{C,\mathrm{rej}}}$;
10 Set $m^{\rm in} = (v^{\rm in}, \mathsf{st}^{\rm in}, w^{\rm in}, \mathbf{I}^{\rm in})$ and $\alpha =$ '-' ;
11 if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\rm in}, \sigma^{\rm in}) = 1$ then set $\alpha =$ 'A' ;
12 if $\alpha =$ '-' and $(t > t^*$ $\underline{\text{or } t \leq i-1}$ $)$ then output Reject;
13 $\underline{\text{if } \alpha = \text{'-' and } t = i \text{ and } \mathsf{Spl.Verify}(\mathrm{vk}, m^{\rm in}, \sigma^{\rm in}) = 0 \text{ then output Reject;}}$
14 if $\alpha =$ '-' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\rm in}, \sigma^{\rm in}) = 1$ then set $\alpha =$ 'B' ;
15 if $\alpha =$ '-' then output Reject;

16 if $\alpha =$ 'B' then
17     Compute $(\mathsf{st}^{\rm out}, a^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\rm in}, a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}})$
18 else
19     Compute $(\mathsf{st}^{\rm out}, a^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\rm in}, a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}})$
20 if $\mathsf{st}^{\rm out} =$ Reject then output Reject;

21 $w^{\rm out} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\rm in}, \mathbf{B}^{\rm out}, \pi^{\rm in})$;
22 if $w^{\rm out} =$ Reject then output Reject;
23 Compute $v^{\rm out} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\rm in}, (\mathsf{st}^{\rm in}, w^{\rm in}, \mathbf{I}^{\rm in}))$;
24 if $v^{\rm out} =$ Reject then output Reject;
25 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $\underline{r'_B = \mathsf{PRF}(K_B\{i-1\}, t)}$;
26 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
27 Set $m^{\rm out} = (v^{\rm out}, \mathsf{st}^{\rm out}, w^{\rm out}, \mathbf{I}^{\rm out})$;
28 Compute $\sigma^{\rm out} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\rm out})$;

29 Output $\widetilde{\mathsf{st}}^{\rm out} = (t+1, \mathsf{st}^{\rm out}, v^{\rm out}, w^{\rm out}, \sigma^{\rm out})$,  $\widetilde{a}^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Hyb$_{0,2,i,1}$ In this hybrid, the challenger punctures key K_A, K_B at input i, uses $\mathsf{PRF}(K_A, i)$ and $\mathsf{PRF}(K_B, i)$ to compute $(\mathrm{sk}_C, \mathrm{vk}_C)$ and $(\mathrm{sk}_D, \mathrm{vk}_D)$ respectively. Formally, it computes $K_A\{i\} \leftarrow \mathsf{PRF.Puncture}(K_A, i)$, $r_C = \mathsf{PRF}(K, i)$, $(\mathrm{sk}_C, \mathrm{vk}_C, \mathrm{vk}_{C,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_C)$ and $K_B\{i\} \leftarrow \mathsf{PRF.Puncture}(K_B, i)$, $r_D = \mathsf{PRF}(K, i)$, $(\mathrm{sk}_D, \mathrm{vk}_D, \mathrm{vk}_{D,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_D)$.

In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,2,i,1}$ defined in Algorithm 34. Here $\widehat{F}^{0,2,i,1}$ is identical to $\widehat{F}^{0,2,i}$ defined in Algorithm 26 except that it uses a punctured PRF key $K_A\{i\}$ instead of K_A, and $K_B\{i\}$ instead of K_B.

Hyb$_{0,2,i,2}$ In this hybrid, the challenger chooses r_C, r_D uniformly at random instead of computing them using $\mathsf{PRF}(K_A, i)$ and $\mathsf{PRF}(K_B, i)$. In other words, the secret key/verification key pairs are sampled by $(\mathrm{sk}_C, \mathrm{vk}_C) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$ and $(\mathrm{sk}_D, \mathrm{vk}_D) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$.

Hyb$_{0,2,i,3}$ In this hybrid, the challenger computes constrained signing keys using the $\mathsf{Spl.Split}$ algorithm. As in the previous hybrids, the challenger first computes the i-th message m^i; then, it computes the following: $(\sigma_{C,\mathrm{one}}, \mathrm{vk}_{C,\mathrm{one}}, \mathrm{sk}_{C,\mathrm{abo}}, \mathrm{vk}_{C,\mathrm{abo}}) = \mathsf{Spl.Split}(\mathrm{sk}_C, m^i)$ and $(\sigma_{D,\mathrm{one}}, \mathrm{vk}_{D,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}_{D,\mathrm{abo}}) = \mathsf{Spl.Split}(\mathrm{sk}_D, m^i)$.

In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,2,i,2}$ defined in Algorithm 35. Note that the only difference between $\widehat{F}^{0,2,i,2}$ and $\widehat{F}^{0,2,i,1}$ is that in $\widehat{F}^{0,2,i,1}$, on input corresponding to step i, signs the outgoing message m using sk_C if $m = m^i$, else it signs using sk_D. On the other hand, at step i, $\widehat{F}^{0,2,i,2}$ outputs $\sigma_{C,\mathrm{one}}$ if the outgoing message $m = m^i$, else it signs using $\mathrm{sk}_{C,\mathrm{abo}}$.

Hyb$_{0,2,i,4}$ This hybrid is similar to the previous one, except that the challenger hardwires $\mathrm{vk}_{C,\mathrm{one}}$ in $\widehat{F}^{0,2,i,2}$ instead of vk_C.

Hyb$_{0,2,i,5}$ This hybrid is similar to the previous one, except that the challenger hardwires $\mathrm{vk}_{D,\mathrm{abo}}$ instead of vk_D. As in the previous hybrid, the challenger uses $\mathsf{Spl.Split}$ to compute $(\sigma_{C,\mathrm{one}}, \mathrm{vk}_{C,\mathrm{one}}, \mathrm{sk}_{C,\mathrm{abo}}, \mathrm{vk}_{C,\mathrm{abo}})$ and $(\sigma_{D,\mathrm{one}}, \mathrm{vk}_{D,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}_{D,\mathrm{abo}})$ from sk_C and sk_D respectively.

Hyb$_{0,2,i,6}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,2,i,3}$ defined in Algorithm 36. This program performs extra checks before computing the signature. In particular, the program additionally checks if the input corresponds to step $i+1$. If so, it checks whether $m^{\mathrm{in}} = m^i$ or not, and accordingly outputs either 'A' or 'B' type signature.

Hyb$_{0,2,i,7}$ In this hybrid, the challenger makes the accumulator "read enforcing". It computes the first i number of "correct inputs" for the accumulator. Based on the initial configuration, we obtain $\left(\{(j, \mathsf{mem}^0[j])\}_{j=1}^{|\mathsf{mem}^0|}\right)$; then we run the following algorithm to obtain $\{a^j_{\mathtt{M}\leftarrow\mathtt{A}}\}_{j=0}^{i}$.

Algorithm 33: This algorithm is for **Hyb**$_{0,2,i,7}$

```
1 for $j \in \{1, \dots, i\}$ do
2     Compute $(\mathsf{st}^j, a^j_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{j-1}, a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}})$ ;    // $a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{j-1}, \mathbf{B}^{j-1})$
3     $(\mathsf{mem}^j, a^j_{\mathtt{A}\leftarrow\mathtt{M}}) \leftarrow \mathsf{access}(\mathsf{mem}^{j-1}, a^j_{\mathtt{M}\leftarrow\mathtt{A}})$ ;    // $a^j_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^j, \mathbf{B}^j)$
```

Let $\ell = |\mathsf{mem}^0|$. Now we set

$$\mathbf{enf} = \Big((1, \mathsf{mem}^0[1]), \dots, (\ell, \mathsf{mem}^0[\ell]), (\mathbf{I}^0, \mathbf{B}^0), \dots, (\mathbf{I}^{i-1}, \mathbf{B}^{i-1})\Big)$$

Finally, the challenger computes $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0) \leftarrow \mathsf{Acc.SetupEnforceRead}(1^\lambda; T, \mathbf{enf}, \mathbf{I}^i)$.

Hyb$_{0,2,i,8}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,2,i,4}$ defined in Algorithm 37. This program runs F^1 instead of F^0, if on the $(i+1)$-st step, the input signature 'A' verifies. Note that the accumulator is "read enforced" in this hybrid.

Hyb$_{0,2,i,9}$ In this hybrid, the challenger uses normal setup for the accumulator related parameters; that is, it computes $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0) \leftarrow \mathsf{Acc.Setup}(1^\lambda; T)$. The remaining steps are exactly identical to the corresponding ones in the previous hybrid.

Hyb$_{0,2,i,10}$ In this hybrid, the challenger computes $(\sigma_{C,\text{one}}, \text{vk}_{C,\text{one}}, \text{sk}_{C,\text{abo}}, \text{vk}_{C,\text{abo}}) = \mathsf{Spl.Split}(\text{sk}_C, m^i)$, but does not compute $(\text{sk}_D, \text{vk}_D)$. Instead, it outputs an obfuscation of $\widehat{F}^{0,2,i,4}$ hardwiring $K_A\{i\}, K_B\{i\}$, $\sigma_{C,\text{one}}, \text{vk}_{C,\text{one}}, \text{sk}_{C,\text{abo}}, \text{vk}_{C,\text{abo}}, m^i$. Note that the hardwired keys for verification/signing (that is, $\sigma_{C,\text{one}}$, $\text{vk}_{C,\text{one}}$, $\text{sk}_{C,\text{abo}}$, $\text{vk}_{C,\text{abo}}$) are all derived from the same signing key sk_C, whereas in the previous hybrid, the first two components were derived from sk_C while the next two from sk_D.

Hyb$_{0,2,i,11}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,2,i,5}$ defined in Algorithm 35.

Hyb$_{0,2,i,12}$ In this hybrid, the challenger chooses the randomness r_C used to compute $(\text{sk}_C, \text{vk}_C)$ pseudorandomly; that is, it sets $r_C = \mathsf{PRF}(K_A, i)$.

Hyb$_{0,2,i,13}$ This corresponds to **Hyb**$_{0,2',i}$.

Analysis In the remaining we prove the following claims.

Claim B.10. *Let* $i\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,1}| \leq \mathsf{negl}(\lambda)$.

Proof. The only difference between **Hyb**$_{0,2,i}$ and **Hyb**$_{0,2,i,1}$ is that **Hyb**$_{0,2,i}$ uses program $\widehat{F}^{0,2,i}$, while **Hyb**$_{0,2,i,1}$ uses $\widehat{F}^{0,2,i,1}$. From the correctness of puncturable PRFs, it follows that both programs have identical functionality for $t \neq i$. For $t = i$, the two programs have identical functionality because $(\text{sk}_C, \text{vk}_C)$ and $(\text{sk}_D, \text{vk}_D)$ are correctly computed using $\mathsf{PRF}(K_A, i)$ and $\mathsf{PRF}(K_B, i)$ respectively. Therefore, by the security of $i\mathcal{O}$, it follows that the obfuscations of the two programs are computationally indistinguishable. □

Claim B.11. *Let* PRF *be a selectively secure puncturable PRF; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,1} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,2}| \leq \mathsf{negl}(\lambda)$.

Proof. We will construct an intermediate experiment **Hyb**, where r_C is chosen uniformly at random, while $r_D = \mathsf{PRF}(K_B, i)$. Now, if an adversary can distinguish between **Hyb**$_{0,2,i,1}$ and **Hyb**, then we can construct a reduction algorithm that breaks the security of PRF. The reduction algorithm sends i as the challenge, and receives $K_A\{i\}, r$. It then uses r to compute $(\text{sk}_C, \text{vk}_C) = \mathsf{Spl.Setup}(1^\lambda; r)$. Depending on whether r is truly random or not, $\mathcal{B}$ simulates either hybrid **Hyb** or **Hyb**$_{0,2,i,1}$. Clearly, if $\mathcal{A}$ can distinguish between **Hyb**$_{0,2,i,1}$ and **Hyb** with advantage non-negl, then $\mathcal{B}$ breaks the PRF security with advantage non-negl. □

Claim B.12. *Let* $i\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,2} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,3}| \leq \mathsf{negl}(\lambda)$.

Proof. The correctness property of Spl ensures that $\widehat{F}^{0,2,i,1}$ and $\widehat{F}^{0,2,i,2}$ have identical functionality. □

Claim B.13. *Let* Spl *be a secure splittable signature scheme which satisfies* vk_{one} *indistinguishability (Definition A.8); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,3} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,4}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose there exists an adversary $\mathcal{A}$ such that $|\mathsf{Adv}^{0,2,i,3}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,4}_{\mathcal{A}}| = \mathsf{non\text{-}negl}$; we can construct a reduction algorithm $\mathcal{B}$ that breaks the $\mathrm{vk}_{\mathrm{one}}$ indistinguishability of splittable signature scheme Spl. Upon receiving m^i from $\mathcal{B}$, the challenger chooses $(\mathrm{sk}_C, \mathrm{vk}_C, \mathrm{vk}_{C,\mathrm{rej}}) \leftarrow \mathsf{Spl.Setup}(1^\lambda)$, $(\sigma_{C,\mathrm{one}}, \mathrm{vk}_{C,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}_{D,\mathrm{abo}})$ and receives (σ, vk), where $\sigma = \sigma_{C,\mathrm{one}}$ and $\mathrm{vk} = \mathrm{vk}_C$ or $\mathrm{vk}_{C,\mathrm{one}}$. It chooses the remaining components (including $\mathrm{sk}_{D,\mathrm{abo}}$ and vk_D) and computes $\widehat{F}^{0,2,i,2}$ where $(T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A\{i\}, K_B\{i\}, \sigma_{C,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}, \mathrm{vk}_D, m^i)$ is hardwired.

Now, note that $\mathcal{B}$ perfectly simulates either $\mathbf{Hyb}_{0,2,i,3}$ or $\mathbf{Hyb}_{0,2,i,4}$, depending on whether the challenge message was $(\sigma_{C,\mathrm{one}}, \mathrm{vk}_C)$ or $(\sigma_{C,\mathrm{one}}, \mathrm{vk}_{C,\mathrm{one}})$. □

Claim B.14. *Let* Spl *be a secure splittable signature scheme which satisfies* $\mathrm{vk}_{\mathrm{abo}}$ *indistinguishability (Definition A.9); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2,i,4}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,5}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. This proof is similar to the previous one. Suppose there exists an adversary A such that $|\mathsf{Adv}^{0,2,i,4}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,5}_{\mathcal{A}}| = \mathsf{non\text{-}negl}$; then there exists a reduction algorithm $\mathcal{B}$ that breaks the $\mathrm{vk}_{\mathrm{abo}}$ security of Spl with advantage $\mathsf{non\text{-}negl}$. In this case, the reduction algorithm uses the challenger's output to set up $\mathrm{sk}_{D,\mathrm{abo}}$ and vk, which is either vk_D or $\mathrm{vk}_{D,\mathrm{abo}}$. □

Claim B.15. *Let* $i\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2,i,5}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,6}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. Let P_0 be $\widehat{F}^{0,2,i,2}$ and P_1 be $\widehat{F}^{0,2,i,3}$ respectively with identically computed constants $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}$, $K_A\{i\}, K_B\{i\}, \sigma_{C,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}_{C,\mathrm{one}}, \mathrm{vk}_{D,\mathrm{abo}}, m^i$.

It suffices to show that P_0 and P_1 have identical functionality. Note that the only inputs where P_0 and P_1 can possibly differ correspond to step $i+1$. Fix any input in step $i+1$. Let us consider two cases:

- $m^{\mathrm{in}} = m^i$. In this case, using the correctness properties of Spl we can argue that for both programs, $\alpha =$ 'A' . Now, P_0 outputs $\mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$, while P_1 is hardwired to output $\mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$. Therefore, both programs have the same output in this case.
- $m^{\mathrm{in}} \neq m^i$. Here, we use the correctness properties of Spl to argue that $\alpha \neq$ 'A' , and conclude that $\alpha =$ 'B' . P_1 is hardwired to output $\mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$, while P_0 outputs $\mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$.

□

Claim B.16. *Let* Acc *be a positional accumulator which satisfies indistinguishability of read setup (Definition A.3); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2,i,6}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,7}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose there exists an adversary $\mathcal{A}$ such that $|\mathsf{Adv}^{0,2,i,6}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,7}_{\mathcal{A}}| = \mathsf{non\text{-}negl}$. We will construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the read setup indistinguishability of Acc. Here $\mathcal{B}$ computes the first i tuples to be accumulated. It computes $(\mathbf{B}^j, \mathbf{I}^j)$ for $j \leq i$ as described in $\mathbf{Hyb}_{0,2,i,7}$, and sends $(\mathbf{B}^j, \mathbf{I}^j)$ for $j < i$, and $\mathbf{I}^i$ to the challenger, and receives $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0)$. $\mathcal{B}$ uses these components to compute the encoding. Note that the remaining steps are identical in both hybrids, and therefore, $\mathcal{B}$ can simulate them perfectly. Finally, using $\mathcal{A}$'s guess, $\mathcal{B}$ guesses whether the setup was normal or read-enforced.

□

Claim B.17. *Let* $i\mathcal{O}$ *be a secure indistinguishability obfuscator, and* F^0 *and* F^1 *be functionally equivalent; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2,i,7}_{\mathcal{A}} - \mathsf{Adv}^{0,2,i,8}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. Let P_0 be $\widehat{F}^{0,2,i,3}$ and P_1 be $\widehat{F}^{0,2,i,4}$ respectively with identically computed constants $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}$, $K_A\{i\}, K_B\{i\}, \sigma_{C,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}_C, \mathrm{vk}_D, m^i$.

We need to show that P_0 and P_1 have identical functionality. Note that in this case, F^0 and F^1 are used in $\widehat{F}^{0,2,i,3}$ and in $\widehat{F}^{0,2,i,4}$ to compute the output respectively. Based on the assumption that F^0 and F^1 are functionally equivalent, now the only difference could be in the case where $t = i+1$. If $\mathsf{Spl.Verify}(\mathrm{vk}_{C,\mathrm{one}}, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ and $\mathsf{st}^{\mathrm{out}} = \mathtt{Reject}$, the two programs could have different functionality. Next we argue this case cannot happen.

From the correctness of Spl, we have that if $\mathsf{Spl.Verify}(\mathrm{vk}_{C,\mathrm{one}}, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$, then $m^{\mathrm{in}} = m^i$. As a result, $w^{\mathrm{in}} = w^i$, $\mathbf{I}^{\mathrm{in}} = \mathbf{I}^i$, $\mathsf{st}^{\mathrm{in}} = \mathsf{st}^i$. So, $(\mathbf{B}^{\mathrm{in}} = \perp$ or $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, \mathbf{B}^{\mathrm{in}}, w^i, \mathbf{I}^i, \pi) = 1) \Rightarrow \mathbf{B}^{\mathrm{in}} = \mathbf{B}^i$, which implies $\mathsf{st}^{\mathrm{out}} = \mathsf{st}^{i+1}$. However, $\mathsf{st}^{i+1} \neq \mathtt{Reject}$. Hence, $t = i+1$ and $\mathsf{Spl.Verify}(\mathrm{vk}_{C,\mathrm{one}}, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ and $\mathsf{st}^{\mathrm{out}} = \mathtt{Reject}$ cannot take place. □

Claim B.18. *Let* Acc *be a positional accumulator which satisfies indistinguishability of read setup (Definition A.3); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,8} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,9}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is similar to that for Claim B.16. □

Claim B.19. *Let* Spl *be a secure splittable signature scheme which satisfies splitting indistinguishability (Definition A.10); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,9} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,10}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose there exists an adversary $\mathcal{A}$ such that $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,9} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,10}| = \mathsf{non\text{-}negl}$. We will construct an algorithm B that uses $\mathcal{A}$ to break the splitting indistinguishability of Spl. $\mathcal{B}$ first receives as input from the challenger a tuple $(\sigma_{\mathrm{one}}, \mathrm{vk}_{\mathrm{one}}, \mathrm{sk}_{\mathrm{abo}}, \mathrm{vk}_{\mathrm{abo}})$, where either all components are derived from the same secret key, or the first two are from one secret key, and the last two from another secret key. Using this tuple, $\mathcal{B}$ can define the constants required for $\widehat{F}^{0,2,i,4}$. It computes $K_A\{i\}, K_B\{i\}, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, m^i$ as described in hybrid $\mathbf{Hyb}_{0,2,i,9}$ and hardwires $\sigma_{\mathrm{one}}, \mathrm{vk}_{\mathrm{one}}, \mathrm{sk}_{\mathrm{abo}}, \mathrm{vk}_{\mathrm{abo}}$ in the program. In this way, $\mathcal{B}$ can simulate either $\mathbf{Hyb}_{0,2,i,9}$ or $\mathbf{Hyb}_{0,2,i,10}$, and therefore, use $\mathcal{A}$'s advantage to break the splitting indistinguishability. □

Claim B.20. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,10} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,11}| \leq \mathsf{negl}(\lambda)$.

Proof. This claim follows from correctness properties of Spl. Note that the programs $\widehat{F}^{0,2,i,4}$ and $\widehat{F}^{0,2,i,5}$ can possibly differ only if $t = i + 1$. We argue that in this case, the two programs are identical as follows:

First, if signatures verify and $\mathsf{st}^{\mathrm{in}} = \mathtt{Reject}$, then both programs will output $\mathtt{Reject}$.

Second, if $\mathsf{Spl.Verify}(\mathrm{vk}_{C,\mathrm{one}}, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$, and $\mathsf{st}^{\mathrm{out}} \neq \mathtt{Reject}$, then both programs will output $\mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$.

Third, if $\mathsf{Spl.Verify}(\mathrm{vk}_{C,\mathrm{one}}, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 0$ but $\mathsf{Spl.Verify}(\mathrm{vk}_{C,\mathrm{abo}}, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$, then both programs will output $\mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$.

Finally, if signatures do not verify at both steps, then both programs will output $\mathtt{Reject}$.

□

Claim B.21. *Let* PRF *be a selectively secure puncturable PRF; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,11} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,12}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is similar to that for Claim B.11. □

Claim B.22. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,i,12} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i,13}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is similar to that for Claim B.10. □

□

B.1.3 From $\mathbf{Hyb}_{0,2',i}$ to $\mathbf{Hyb}_{0,2,i+1}$

Lemma B.23. *Let* $1 \leq i < t^*$. *Assume* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* Itr *is a secure iterator scheme, and* Acc *is a secure positional accumulator scheme; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i} - \mathsf{Adv}_{\mathcal{A}}^{0,2,i+1}| \leq \mathsf{negl}(\lambda)$.

Proof. We define fourth layer hybrids $\mathbf{Hyb}_{0,2',i,0}, \mathbf{Hyb}_{0,2',i,1}, \ldots, \mathbf{Hyb}_{0,2',i,8}$. The first hybrid corresponds to $\mathbf{Hyb}_{0,2',i}$, and the last one corresponds to $\mathbf{Hyb}_{0,2,i+1}$.

Algorithm 34: $\widehat{F}^{0,2,i,1}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A\{i\}, K_B\{i\}, \mathrm{sk}_C, \mathrm{sk}_D, \mathrm{vk}_C, \mathrm{vk}_D, m^i$
1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
2  if $t \neq i + 1$ then
3      Compute $r_A = \mathsf{PRF}(K_A\{i\}, t-1), r_B = \mathsf{PRF}(K_B\{i\}, t-1)$;
4      Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
5  else
6      Set $\mathrm{vk}_A = \mathrm{vk}_C, \mathrm{vk}_B = \mathrm{vk}_D$;
7  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ '-' ;
8  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A' ;
9  if $\alpha =$ '-' and $(t > t^*$ or $t \leq i$ ) then output Reject;
10 if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B' ;
11 if $\alpha =$ '-' then output Reject;

12 if $\alpha =$ 'B' or $t \leq i$ then
13     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
14 else
15     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
16 if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;

17 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
18 if $w^{\text{out}} =$ Reject then output Reject;
19 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
20 if $v^{\text{out}} =$ Reject then output Reject;
21 if $t \neq i$ then
22     Set $r'_A = \mathsf{PRF}(K_A\{i\}, t), r'_B = \mathsf{PRF}(K_B\{i\}, t)$;
23     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24 else
25     Set $\mathrm{sk}'_A = \mathrm{sk}_C, \mathrm{sk}'_B = \mathrm{sk}_D$;
26 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
27 if $t = i$ and $m^{\text{out}} = m^i$ then
28     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}})$;
29 else if $t = i$ and $m^{\text{out}} \neq m^i$ then
30     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\text{out}})$;
31 else
32     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\text{out}})$;
33 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 35: $\widehat{F}^{0,2,i,2}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$,  $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A\{i\}, K_B\{i\}, \underline{\sigma_{C,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}}, \mathrm{vk}_C, \mathrm{vk}_D, m^i$
1  if Acc.VerifyRead($\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output Reject;
2  if $t \neq i+1$ then
3      Compute $r_A = \mathsf{PRF}(K_A\{i\}, t-1)$, $r_B = \mathsf{PRF}(K_B\{i\}, t-1)$;
4      Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
5  else
6      Set $\mathrm{vk}_A = \mathrm{vk}_C$, $\mathrm{vk}_B = \mathrm{vk}_D$;
7  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$ and $\alpha =$ '-' ;
8  if Spl.Verify($\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A' ;
9  if $\alpha =$ '-' and ($t > t^*$ or $t \leq i$ ) then output Reject;
10 if $\alpha \neq$ 'A' and Spl.Verify($\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B' ;
11 if $\alpha =$ '-' then output Reject;

12 if $\alpha =$ 'B' or $t \leq i$ then
13     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
14 else
15     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
16 if $\mathsf{st}^{\mathrm{out}} =$ Reject then output Reject;

17 $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
18 if $w^{\mathrm{out}} =$ Reject then output Reject;
19 Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
20 if $v^{\mathrm{out}} =$ Reject then output Reject;
21 if $t \neq i$ then
22     Set $r'_A = \mathsf{PRF}(K_A\{i\}, t)$, $r'_B = \mathsf{PRF}(K_B\{i\}, t)$;
23     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24 else
25     $\underline{\text{Set } \mathrm{sk}'_A = \sigma_{C,\mathrm{one}}, \mathrm{sk}'_B = \mathrm{sk}_{D,\mathrm{abo}}}$;
26 Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
27 if $t = i$ and $m^{\mathrm{out}} = m^i$ then
28     $\underline{\text{Compute } \sigma^{\mathrm{out}} = \sigma_{C,\mathrm{one}}}$;
29 else if $t = i$ and $m^{\mathrm{out}} \neq m^i$ then
30     $\underline{\text{Compute } \sigma^{\mathrm{out}} = \mathsf{Spl.AboSign}(\mathrm{sk}'_{D,\mathrm{abo}}, m^{\mathrm{out}})}$;
31 else
32     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
33 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$,   $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 36: $\widehat{F}^{0,2,i,3}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$, $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A\{i\}, K_B\{i\}, \sigma_{C,\mathrm{one}}, \mathrm{sk}_{D,\mathrm{abo}}, \mathrm{vk}_C, \mathrm{vk}_D, m^i$
1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output Reject;
2  if $t \neq i+1$ then
3      Compute $r_A = \mathsf{PRF}(K_A\{i\}, t-1)$, $r_B = \mathsf{PRF}(K_B\{i\}, t-1)$;
4      Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
5  else
6      Set $\mathrm{vk}_A = \mathrm{vk}_C$, $\mathrm{vk}_B = \mathrm{vk}_D$;
7  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$ and $\alpha =$ '-' ;
8  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A' ;
9  if $\alpha =$ '-' and ($t > t^*$ or $t \leq i$ ) then output Reject;
10 if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B' ;
11 if $\alpha =$ '-' then output Reject;
12 if $\alpha =$ 'B' or $t \leq i$ then
13     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
14 else
15     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
16 if $\mathsf{st}^{\mathrm{out}} =$ Reject then output Reject;
17 $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
18 if $w^{\mathrm{out}} =$ Reject then output Reject;
19 Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
20 if $v^{\mathrm{out}} =$ Reject then output Reject;
21 if $t \neq i$ then
22     Set $r'_A = \mathsf{PRF}(K_A\{i\}, t)$, $r'_B = \mathsf{PRF}(K_B\{i\}, t)$;
23     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24 else
25     Set $\mathrm{sk}'_A = \sigma_{C,\mathrm{one}}$, $\mathrm{sk}'_B = \mathrm{sk}_{D,\mathrm{abo}}$;
26 Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
27 if $t = i$ and $m^{\mathrm{out}} = m^i$ then
28     Compute $\sigma^{\mathrm{out}} = \sigma_{C,\mathrm{one}}$;
29 else if $t = i$ and $m^{\mathrm{out}} \neq m^i$ then
30     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.AboSign}(\mathrm{sk}'_{D,\mathrm{abo}}, m^{\mathrm{out}})$;
31 else if $t = i+1$ and $m^{\mathrm{in}} = m^i$ then
32     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
33 else if $t = i+1$ and $m^{\mathrm{in}} \neq m^i$ then
34     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
35 else
36     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
37 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$, $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 37: $\widehat{F}^{0,2,i,4}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A\{i\}, K_B\{i\}, \sigma_{C,\text{one}}, \text{sk}_{D,\text{abo}}, \text{vk}_C, \text{vk}_D, m^i$
1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
2  if $t \neq i+1$ then
3      Compute $r_A = \mathsf{PRF}(K_A\{i\}, t-1)$, $r_B = \mathsf{PRF}(K_B\{i\}, t-1)$;
4      Compute $(\text{sk}_A, \text{vk}_A, \text{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\text{sk}_B, \text{vk}_B, \text{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
5  else
6      Set $\text{vk}_A = \text{vk}_C$, $\text{vk}_B = \text{vk}_D$;
7  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ ‘-’ ;
8  if $\mathsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ ‘A’ ;
9  if $\alpha =$ ‘-’ and ($t > t^*$ or $t \leq i$ ) then output Reject;
10 if $\alpha \neq$ ‘A’ and $\mathsf{Spl.Verify}(\text{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ ‘B’ ;
11 if $\alpha =$ ‘-’ then output Reject;
12 if $\alpha =$ ‘B’ or $t \leq i+1$ then
13     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
14 else
15     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
16 if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;
17 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
18 if $w^{\text{out}} =$ Reject then output Reject;
19 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
20 if $v^{\text{out}} =$ Reject then output Reject;
21 if $t \neq i$ then
22     Set $r'_A = \mathsf{PRF}(K_A\{i\}, t)$, $r'_B = \mathsf{PRF}(K_B\{i\}, t)$;
23     Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24 else
25     Set $\text{sk}'_A = \sigma_{C,\text{one}}$, $\text{sk}'_B = \text{sk}_{D,\text{abo}}$;
26 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
27 if $t = i$ and $m^{\text{out}} = m^i$ then
28     Compute $\sigma^{\text{out}} = \sigma_{C,\text{one}}$;
29 else if $t = i$ and $m^{\text{out}} \neq m^i$ then
30     Compute $\sigma^{\text{out}} = \mathsf{Spl.AboSign}(\text{sk}'_{D,\text{abo}}, m^{\text{out}})$;
31 else if $t = i+1$ and $m^{\text{in}} = m^i$ then
32     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_A, m^{\text{out}})$;
33 else if $t = i+1$ and $m^{\text{in}} \neq m^i$ then
34     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_B, m^{\text{out}})$;
35 else
36     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_\alpha, m^{\text{out}})$;
37 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 38: $\widehat{F}^{0,2,i,5}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,   $\widetilde{a}^{\text{in}}_{\mathtt{A \leftarrow M}} = (a^{\text{in}}_{\mathtt{A \leftarrow M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A \leftarrow M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A\{i\}, K_B\{i\}, \underline{\mathrm{sk}_C}, \mathrm{vk}_C, m^i$
if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
if $t \neq i + 1$ then
    Compute $r_A = \mathsf{PRF}(K_A\{i\}, t-1)$, $r_B = \mathsf{PRF}(K_B\{i\}, t-1)$;
    Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
else
    $\underline{\text{Set } \mathrm{vk}_A = \mathrm{vk}_C;}$
Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ '-' ;
if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A' ;
if $\alpha =$ '-' and ($t > t^*$ $\underline{\textbf{or } t \leq i+1}$ ) then output Reject;
if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B' ;
if $\alpha =$ '-' then output Reject;
if $\alpha =$ 'B' or $t \leq i + 1$ then
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M \leftarrow A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A \leftarrow M}})$
else
    Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M \leftarrow A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A \leftarrow M}})$
if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;
$w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
if $w^{\text{out}} =$ Reject then output Reject;
Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
if $v^{\text{out}} =$ Reject then output Reject;
if $t \neq i$ then
    Set $r'_A = \mathsf{PRF}(K_A\{i\}, t)$, $r'_B = \mathsf{PRF}(K_B\{i\}, t)$;
    Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
else
    $\underline{\text{Set } \mathrm{sk}'_A = \mathrm{sk}_C;}$
Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
if $t = i$ then
    $\underline{\text{Compute } \sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}});}$
else if $t = i + 1$ and $m^{\text{in}} = m^i$ then
    Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\text{out}})$;
else if $t = i + 1$ and $m^{\text{in}} \neq m^i$ then
    Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\text{out}})$;
else
    Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\text{out}})$;
Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,   $\widetilde{a}^{\text{out}}_{\mathtt{M \leftarrow A}} = a^{\text{out}}_{\mathtt{M \leftarrow A}}$;
```

Hyb$_{0,2',i,0}$ This hybrid corresponds to **Hyb**$_{0,2',i}$.

Hyb$_{0,2',i,1}$ In this hybrid, the challenger makes the accumulator "read enforcing".

Based on the initial configuration, we first obtain $\left(\{(j, \mathsf{mem}^0[j])\}_{j=1}^{|\mathsf{mem}^0|}\right)$. Let $\ell = |\mathsf{mem}^0|$. It computes the first $\ell + i$ "correct inputs" for the accumulator; then we run the following algorithm to obtain $\{a^j_{\mathtt{M}\leftarrow\mathtt{A}}\}_{j=0}^{i}$.

Algorithm 39: This algorithm is for **Hyb**$_{0,2',i,1}$

```
1 for $j \in \{1, \dots, i\}$ do
2     Compute $(\mathsf{st}^j, a^j_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{j-1}, a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}})$ ;        // $a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{j-1}, \mathbf{B}^{j-1})$
3     $(\mathsf{mem}^j, a^j_{\mathtt{A}\leftarrow\mathtt{M}}) \leftarrow \mathsf{access}(\mathsf{mem}^{j-1}, a^j_{\mathtt{M}\leftarrow\mathtt{A}})$ ;        // $a^j_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^j, \mathbf{B}^j)$
```

Now we set

$$\mathbf{enf} = \Big((1, \mathsf{mem}^0[1]), \dots, (\ell, \mathsf{mem}^0[\ell]), (\mathbf{I}^0, \mathbf{B}^0), \dots, (\mathbf{I}^{i-1}, \mathbf{B}^{i-1})\Big).$$

Finally, the challenger computes $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0) \leftarrow \mathsf{Acc.SetupEnforceRead}(1^\lambda; T, \mathbf{enf}, \mathbf{I}^i)$.

Hyb$_{0,2',i,2}$ In this hybrid, the challenger uses program $\widehat{F}^{0,2',i,2}$ (defined in Algorithm 42), which is similar to $\widehat{F}^{0,2,i}$. However, in addition to checking if $m^i = m^{\text{in}}$, it also checks if $(v^{\text{out}}, \mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}) = (v^{i+1}, \mathsf{st}^{i+1}, \mathbf{I}^{i+1})$.

Hyb$_{0,2',i,3}$ In this experiment, the challenger uses normal setup instead of "read enforced" setup for the accumulator.

Hyb$_{0,2',i,4}$ In this hybrid, the challenger "write enforces" the accumulator. As in **Hyb**$_{0,2',i,1}$, based on the initial configuration, we first obtain $\left(\{(j, \mathsf{mem}^0[j])\}_{j=1}^{|\mathsf{mem}^0|}\right)$. But now it computes the first $\ell + i + 1$ "correct inputs" for the accumulator. We run the following algorithm to obtain $\{a^j_{\mathtt{M}\leftarrow\mathtt{A}}\}_{j=0}^{i+1}$.

Algorithm 40: This algorithm is for **Hyb**$_{0,2',i,4}$

```
1 for $j \in \{1, \dots, i\}$ do
2     Compute $(\mathsf{st}^j, a^j_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{j-1}, a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}})$ ;        // $a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{j-1}, \mathbf{B}^{j-1})$
3     $(\mathsf{mem}^j, a^j_{\mathtt{A}\leftarrow\mathtt{M}}) \leftarrow \mathsf{access}(\mathsf{mem}^{j-1}, a^j_{\mathtt{M}\leftarrow\mathtt{A}})$ ;        // $a^j_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^j, \mathbf{B}^j)$
```

Now we set

$$\mathbf{enf} = \Big((1, \mathsf{mem}^0[1]), \dots, (\ell, \mathsf{mem}^0[\ell]), (\mathbf{I}^0, \mathbf{B}^0), \dots, (\mathbf{I}^{i}, \mathbf{B}^{i})\Big)$$

Finally, the challenger computes $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0) \leftarrow \mathsf{Acc.SetupEnforceRead}(1^\lambda; T, \mathbf{enf})$.

Hyb$_{0,2',i,5}$ In this experiment, the challenger outputs an obfuscation of $\widehat{F}^{0,2',i,5}$ in Algorithm 43, which is very similar to $\widehat{F}^{0,2',i,2}$. However, on input where $t = i + 1$, before computing signature, it also checks if $w^{\text{out}} = w^{i+1}$. Therefore, it checks whether $m^{\text{in}} = m^i$ and $m^{\text{out}} = m^{i+1}$.

Hyb$_{0,2',i,6}$ This experiment is similar to the previous one, except that the challenger uses normal setup for accumulator instead of "enforcing write".

Hyb$_{0,2',i,7}$ This experiment is similar to the previous one, except that the challenger uses enforced setup for iterator instead of normal setup. It first computes $\mathsf{pp}_{\mathsf{Acc}}, w^0, \mathit{store}^0$ as in the previous hybrid. Next, it computes the first $i+1$ "correct messages" for the iterator.

Based on the initial configuration $\mathsf{mem}^0, \mathsf{st}^0, a^0_{\mathtt{A}\leftarrow\mathtt{M}} = \bot, a^0_{\mathtt{M}\leftarrow\mathtt{A}} = \bot)$, the challenger computes $\mathbf{enf} = \Big((\mathsf{st}^0, w^0, \mathbf{I}^0), (\mathsf{st}^1, w^1, \mathbf{I}^1), \ldots, (\mathsf{st}^i, w^i, \mathbf{I}^i)\Big)$ as follows:

Algorithm 41: This algorithm is for **Hyb**$_{0,2',i,7}$

```
1 for $j \in \{1, \ldots, i+1\}$ do
2     Compute $(\mathsf{st}^j, a^j_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{j-1}, a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}})$ ;        // $a^{j-1}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{j-1}, \mathbf{B}^{j-1})$
3     $(a^j_{\mathtt{M}\leftarrow\mathtt{A}}, \pi^j) \leftarrow \mathsf{Acc.PrepRead}(\mathsf{pp}_{\mathsf{Acc}}, \mathit{store}^j, \mathbf{I}^j)$
4     $w^j \leftarrow \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{j-1}, a^j_{\mathtt{M}\leftarrow\mathtt{A}}, \pi^j)$
5     $\mathit{store}^j \leftarrow \mathsf{Acc.WriteStore}(\mathsf{pp}_{\mathsf{Acc}}, \mathit{store}^{j-1}, a^j_{\mathtt{M}\leftarrow\mathtt{A}})$
```

Then the challenger computes $(\mathsf{pp}_{\mathsf{Itr}}, v^0) \leftarrow \mathsf{Itr.SetupEnforceIterate}(1^\lambda; T, \mathbf{enf})$.

Hyb$_{0,2',i,8}$ In this experiment, the challenger outputs an obfuscation of $\widehat{F}^{0,2',i,8}$ in Algorithm 44, which is similar to $\widehat{F}^{0,2',i,5}$, except that it only checks if $m^{\mathrm{out}} = m^{i+1}$.

Hyb$_{0,2',i,9}$ This corresponds to **Hyb**$_{0,2,i+1}$. The only difference between this experiment and the previous one is that this uses normal setup for iterator.

Analysis.

Claim B.24. *Let* Acc *be a positional accumulator which satisfies indistinguishability of read setup (Definition A.3); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2',i}_{\mathcal{A}} - \mathsf{Adv}^{0,2',i,1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is very similar to that for Claim B.16. □

Claim B.25. *Let* Acc *be a positional accumulator which is read-enforcing (Definition A.5), and* $i\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2',i,1}_{\mathcal{A}} - \mathsf{Adv}^{0,2',i,2}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. In order to prove the claim, it suffices to show that $P_0 = \widehat{F}^{0,2',i}$ and $P_1 = \widehat{F}^{0,2,i,b}$ are functionally equivalent. These two programs are functionally identical iff $m^{\mathrm{in}} = m^i \Rightarrow (v^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}) = (v^{i+1}, \mathbf{I}^{i+1}, \mathsf{st}^{i+1})$, which is implied by the read-enforcing property of the accumulator. □

Claim B.26. *Let* Acc *be a positional accumulator which satisfies indistinguishability of read setup (Definition A.3); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2',i,2}_{\mathcal{A}} - \mathsf{Adv}^{0,2',i,3}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is very similar to that for Claim B.16. □

Claim B.27. *Let* Acc *be a positional accumulator which satisfies indistinguishability of write setup (Definition A.4); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,2',i,3}_{\mathcal{A}} - \mathsf{Adv}^{0,2',i,4}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose there exists an adversary $\mathcal{A}$ such that $|\mathsf{Adv}^{0,2',i,3}_{\mathcal{A}} - \mathsf{Adv}^{0,2',i,4}_{\mathcal{A}}| = \mathsf{non\text{-}negl}$. We will construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the write setup indistinguishability of Acc. Here $\mathcal{B}$ computes the first $\ell_{\mathsf{input}} + i + 1$ tuples to be accumulated, i.e., $\mathbf{enf}$. It then sends $\mathbf{enf}$ to the challenger, and receives $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \widehat{store}_0)$. Note that the remaining steps are identical in both hybrids, and therefore, $\mathcal{B}$ can simulate them perfectly. Finally, using $\mathcal{A}$'s guess, $\mathcal{B}$ guesses whether the setup was normal or write-enforced. □

Claim B.28. *Let* Acc *be a positional accumulator which is write-enforcing (Definition A.6), and* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i,4} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i,5}| \leq \mathsf{negl}(\lambda)$.

Proof. In order to prove the claim, it suffices to show that $\widehat{F}^{0,2,i,b}$ and $\widehat{F}^{0,2,i,c}$ are functionally equivalent. These two programs are functionally identical iff $m^{\text{in}} = m^i$ and $(v^{\text{out}}, \mathbf{I}^{\text{out}}, \mathsf{st}^{\text{out}}) = (v^{i+1}, \mathbf{I}^{i+1}, \mathsf{st}^{i+1}) \Rightarrow w^{\text{out}} = w^{i+1}$, which is implied by the read-enforcing property of the accumulator. □

Claim B.29. *Let* Acc *be a positional accumulator which satisfies indistinguishability of write setup (Definition A.4); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i,5} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i,6}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is very similar to that for Claim B.27.

□

Claim B.30. *Let* Itr *be an iterator which satisfies indistinguishability of setup (Definition A.1); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i,6} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i,7}| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose there exists an adversary $\mathcal{A}$ such that $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i,6} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i,7}| = \mathsf{non\text{-}negl}$. We will construct an algorithm $\mathcal{B}$ that uses $\mathcal{A}$ to break the setup indistinguishability of Itr. Here $\mathcal{B}$ computes the first $i+1$ tuples to be iterated on, i.e., $\mathbf{enf} = \Big((\mathsf{st}^0, w^0, \mathbf{I}^0), (\mathsf{st}^1, w^1, \mathbf{I}^1), \ldots, (\mathsf{st}^i, w^i, \mathbf{I}^i)\Big)$. It then sends $\mathbf{enf}$ to the challenger, and receives $(\mathsf{pp}_{\mathsf{Itr}}, v_0)$. Note that the remaining steps are identical in both hybrids, and therefore, $\mathcal{B}$ can simulate them perfectly. Finally, using $\mathcal{A}$'s guess, $\mathcal{B}$ guesses whether the setup was normal or enforced.

□

Claim B.31. *Let* Itr *be an iterator which is enforcing (Definition A.2), and* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i,7} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i,8}| \leq \mathsf{negl}(\lambda)$.

Proof. In order to prove the claim, it suffices to show that $P_0 = \widehat{F}^{0,2',i,5}$ and $P_1 = \widehat{F}^{0,2',i,8}$ are functionally equivalent. Note that the only difference between P_0 and P_1 is that, in P_0 we check if $(m^{\text{in}} = m^i)$ and $(m^{\text{out}} = m^{i+1})$, while in P_1 we only check if $(m^{\text{out}} = m^{i+1})$. Therefore, we need to show that $m^{\text{out}} = m^{i+1} \Rightarrow m^{\text{in}} = m^i$. This follows directly from the enforcing property of the iterator. □

Claim B.32. *Let* Itr *be an iterator which satisfies indistinguishability of setup (Definition A.1); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',i,8} - \mathsf{Adv}_{\mathcal{A}}^{0,2',i,9}| \leq \mathsf{negl}(\lambda)$.

Proof. The proof is very similar to that for Claim B.30.

□

□

B.1.4 From $\mathbf{Hyb}_{0,2',t^*-1}$ to $\mathbf{Hyb}_{0,3}$

Lemma B.33. *Let* $i\mathcal{O}$ *be a secure indistinguishability obfuscator, and* Acc *be a secure positional accumulator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2,t^*-1} - \mathsf{Adv}_{\mathcal{A}}^{0,3}| \leq \mathsf{negl}(\lambda)$.

$\mathbf{Hyb}_{0,2',t^*-1,1}$ In this hybrid, the challenger makes the accumulator "read enforcing".

Based on the initial configuration, we first obtain $\Big(\{(j, \mathsf{mem}^0[j])\}_{j=1}^{|\mathsf{mem}^0|}\Big)$. Let $\ell = |\mathsf{mem}^0|$. It computes the first t^*-1 "correct inputs" for the accumulator; then we run the following algorithm to obtain $\{a^j_{\mathtt{M}\leftarrow\mathtt{A}}\}_{j=0}^{t^*-1}$.

Now we set

$$\mathbf{enf} = \Big((1, \mathsf{mem}^0[1]), \ldots, (\ell, \mathsf{mem}^0[\ell]), (\mathbf{I}^0, \mathbf{B}^0), \ldots, (\mathbf{I}^{t^*-2}, \mathbf{B}^{t^*-2})\Big)$$

Finally, the challenger computes $(\mathsf{pp}_{\mathsf{Acc}}, \hat{w}_0, \hat{store}_0) \leftarrow \mathsf{Acc.SetupEnforceRead}(1^\lambda; T, \mathbf{enf}, \mathbf{I}^{t^*-1})$.

Algorithm 42: $\widehat{F}^{0,2',i,2}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$,  $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m^i, v^{i+1}, \mathsf{st}^{i+1}, \mathbf{I}^{i+1}$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$ and $\alpha =$ '-' ;
5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A' ;
6  if $\alpha =$ '-' and $(t > t^*$ or $t \leq i+1$ ) then output Reject;
7  if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B' ;
8  if $\alpha =$ '-' then output Reject;

9  if $\alpha =$ 'B' or $t \leq i+1$ then
10     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
11 else
12     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
13 if $\mathsf{st}^{\mathrm{out}} =$ Reject then output Reject;

14 $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
15 if $w^{\mathrm{out}} =$ Reject then output Reject;
16 Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
17 if $v^{\mathrm{out}} =$ Reject then output Reject;
18 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
19 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
20 Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
21 if $t = i+1$ and $(m^{\mathrm{in}} = m^i$ and $(v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}) = (v^{i+1}, \mathsf{st}^{i+1}, \mathbf{I}^{i+1}))$ then
22     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
23 else if $t = i+1$ and $(m^{\mathrm{in}} \neq m^i$ or $(v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}) \neq (v^{i+1}, \mathsf{st}^{i+1}, \mathbf{I}^{i+1}))$ then
24     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
25 else
26     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;

27 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$,  $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 43: $\widehat{F}^{0,2',i,5}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (t, \mathsf{st}^{\text{in}}, v^{\text{in}}, w^{\text{in}}, \sigma^{\text{in}})$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\text{in}})$ where $a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m^i, \underline{m^{i+1}}$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}}, \pi^{\text{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
3  Compute $(\text{sk}_A, \text{vk}_A, \text{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\text{sk}_B, \text{vk}_B, \text{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Set $m^{\text{in}} = (v^{\text{in}}, \mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}})$ and $\alpha =$ ‘-’ ;
5  if $\mathsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ ‘A’ ;
6  if $\alpha =$ ‘-’ and $(t > t^*$ or $t \leq i+1$ ) then output Reject;
7  if $\alpha \neq$ ‘A’ and $\mathsf{Spl.Verify}(\text{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ ‘B’ ;
8  if $\alpha =$ ‘-’ then output Reject;

9  if $\alpha =$ ‘B’ or $t \leq i+1$ then
10     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
11 else
12     Compute $(\mathsf{st}^{\text{out}}, a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\text{in}}, a^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
13 if $\mathsf{st}^{\text{out}} =$ Reject then output Reject;

14 $w^{\text{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\text{in}}, \mathbf{B}^{\text{out}}, \pi^{\text{in}})$;
15 if $w^{\text{out}} =$ Reject then output Reject;
16 Compute $v^{\text{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (\mathsf{st}^{\text{in}}, w^{\text{in}}, \mathbf{I}^{\text{in}}))$;
17 if $v^{\text{out}} =$ Reject then output Reject;
18 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
19 Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
20 Set $m^{\text{out}} = (v^{\text{out}}, \mathsf{st}^{\text{out}}, w^{\text{out}}, \mathbf{I}^{\text{out}})$;
21 if $t = i+1$ and $\underline{(m^{\text{in}} = m^i \text{ and } m^{\text{out}} = m^{i+1})}$ then
22     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_A, m^{\text{out}})$;
23 else if $t = i+1$ and $\underline{(m^{\text{in}} \neq m^i \text{ or } m^{\text{out}} \neq m^{i+1})}$ then
24     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_B, m^{\text{out}})$;
25 else
26     Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_\alpha, m^{\text{out}})$;

27 Output $\widetilde{\mathsf{st}}^{\text{out}} = (t+1, \mathsf{st}^{\text{out}}, v^{\text{out}}, w^{\text{out}}, \sigma^{\text{out}})$,  $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 44: $\widehat{F}^{0,2',i,8}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$,  $\widetilde{a}^{\mathrm{in}}_{\mathtt{A \leftarrow M}} = (a^{\mathrm{in}}_{\mathtt{A \leftarrow M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A \leftarrow M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, \underline{m^{i+1}}$

1  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output Reject;
2  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$ and $\alpha =$ '-' ;
5  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A' ;
6  if $\alpha =$ '-' and ($t > t^*$ or $t \leq i+1$ ) then output Reject;
7  if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B' ;
8  if $\alpha =$ '-' then output Reject;

9  if $\alpha =$ 'B' or $t \leq i+1$ then
10     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M \leftarrow A}}) \leftarrow F^1(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A \leftarrow M}})$
11 else
12     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M \leftarrow A}}) \leftarrow F^0(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A \leftarrow M}})$
13 if $\mathsf{st}^{\mathrm{out}} =$ Reject then output Reject;

14 $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
15 if $w^{\mathrm{out}} =$ Reject then output Reject;
16 Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
17 if $v^{\mathrm{out}} =$ Reject then output Reject;
18 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
19 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
20 Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
21 if $t = i+1$ and $\underline{(m^{\mathrm{out}} = m^{i+1})}$ then
22     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
23 else if $t = i+1$ and $\underline{(m^{\mathrm{out}} \neq m^{i+1})}$ then
24     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
25 else
26     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;

27 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$,   $\widetilde{a}^{\mathrm{out}}_{\mathtt{M \leftarrow A}} = a^{\mathrm{out}}_{\mathtt{M \leftarrow A}}$;
```

Algorithm 45: This algorithm is for $\mathbf{Hyb}_{0,2',t^*-1,1}$

```
1 for $j \in \{1, \ldots, t^*-1\}$ do
2     Compute $(\mathsf{st}^j, a^j_{\mathtt{M \leftarrow A}}) \leftarrow F^0(\mathsf{st}^{j-1}, a^{j-1}_{\mathtt{A \leftarrow M}})$ ;      // $a^{j-1}_{\mathtt{A \leftarrow M}} = (\mathbf{I}^{j-1}, \mathbf{B}^{j-1})$
3     $(\mathsf{mem}^j, a^j_{\mathtt{A \leftarrow M}}) \leftarrow \mathsf{access}(\mathsf{mem}^{j-1}, a^j_{\mathtt{M \leftarrow A}})$ ;      // $a^j_{\mathtt{M \leftarrow A}} = (\mathbf{I}^j, \mathbf{B}^j)$
```

Hyb$_{0,2',t^*-1,2}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,3}$.

Hyb$_{0,2',t^*-1,3}$ In this hybrid, the challenger uses $\mathsf{Acc.Setup}$ instead of using $\mathsf{Acc.SetupEnforceRead}$.

Claim B.34. *Let* Acc *be an accumulator which satisfies indistinguishability of read setup (Definition A.3); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',t^*-1} - \mathsf{Adv}_{\mathcal{A}}^{0,2',t^*-1,1}| \leq \mathsf{negl}(\lambda)$.

Proof. This proof is similar to that for Claim B.16. □

Claim B.35. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',t^*-1,1} - \mathsf{Adv}_{\mathcal{A}}^{0,2',t^*-1,2}| \leq \mathsf{negl}(\lambda)$.

Proof. This proof is similar to that for Claim B.17. □

Claim B.36. *Let* Acc *be an accumulator which satisfies indistinguishability of read setup (Definition A.3); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,2',t^*-1,2} - \mathsf{Adv}_{\mathcal{A}}^{0,2',t^*-1,3}| \leq \mathsf{negl}(\lambda)$.

Proof. This proof is similar to that for Claim B.16. □

B.1.5 From Hyb$_{0,3}$ to Hyb$_{0,4}$

Lemma B.37. *Let* $i\mathcal{O}$ *be a secure indistinguishability obfuscator,* PRF *be a selectively secure puncturable PRF, and* Spl *be a secure splitting signature scheme; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3} - \mathsf{Adv}_{\mathcal{A}}^{0,4}| \leq \mathsf{negl}(\lambda)$.

Proof. We will define $T - t^* + 1$ hybrids, and show they are computationally indistinguishable.

Hyb$_{0,3,i}$ In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,3,i}$ defined in Algorithm 46 for $t^* \leq i \leq T$.

Clearly, programs $\widehat{F}_{0,3}$ and $\widehat{F}_{0,3,t^*}$ are functionally identical, and therefore **Hyb**$_{0,3}$ and **Hyb**$_{0,3,t^*}$ are computationally indistinguishable. In addition, hybrids **Hyb**$_{0,3,T}$ and **Hyb**$_{0,4}$ are functionally identical, since the difference between these two hybrids is a dummy code which has never been executed. In order to show that **Hyb**$_{0,3,i}$ and **Hyb**$_{0,3,i+1}$ are computationally indistinguishable, we define intermediate hybrid experiments **Hyb**$_{0,3,i,a}$, **Hyb**$_{0,3,i,a}$, $\ldots$, **Hyb**$_{0,3,i,f}$ as follows. Note that **Hyb**$_{0,3,i,a}$ corresponds to **Hyb**$_{0,3,i}$ and **Hyb**$_{0,3,i,f}$ corresponds to **Hyb**$_{0,3,i+1}$.

Hyb$_{0,3,i,a}$ This hybrid corresponds to **Hyb**$_{0,3,i}$.

Hyb$_{0,3,i,b}$ In this hybrid, the challenger first punctures the PRF key K_A on input i by computing $K_A\{i\} \leftarrow \mathsf{PRF.Puncture}(K_A, i)$. Next, it computes $r_C = \mathsf{PRF}(K_A, i)$ and $(\mathrm{sk}_C, \mathrm{vk}_C, \mathrm{vk}_{C,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_C)$. It outputs an obfuscation of $\widehat{F}^{0,3,i,b}$ defined in Algorithm 47. This program is similar to $\widehat{F}^{0,3,i,a}$ except that it has $K_A\{i\}$ and $\mathrm{vk}_{C,\mathrm{rej}}$ hardwired, and when $t = i$, it replaces $\mathrm{vk}_{A,\mathrm{rej}}$ by $\mathrm{vk}_{C,\mathrm{rej}}$.

Hyb$_{0,3,i,c}$ This hybrid is similar to **Hyb**$_{0,3,i,b}$, except that r_C is now chosen uniformly at random from $\{0,1\}^\lambda$.

Hyb$_{0,3,i,d}$ This hybrid is similar to **Hyb**$_{0,3,i,c}$, except that $\mathrm{vk}_{C,\mathrm{rej}}$ is hardwired to the program.

Hyb$_{0,3,i,e}$ This hybrid is similar to **Hyb**$_{0,3,i,d}$, except that $r_C = \mathsf{PRF}(K_B, i)$ is now pseudorandom.

$\mathbf{Hyb}_{0,3,i,f}$ This hybrid corresponds to $\mathbf{Hyb}_{0,3,i+1}$.

Claim B.38. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i,a}| \leq \mathsf{negl}(\lambda)$.

Proof. Observe that $\widehat{F}^{0,3,i}$ and $\widehat{F}^{0,3,i,a}$ have identical functionality. Therefore $\mathbf{Hyb}_{0,3,i}$ and $\mathbf{Hyb}_{0,3,i,a}$ are computationally indistinguishable under the assumption that $\mathsf{i}\mathcal{O}$ is a secure indistinguishability obfuscation scheme. ☐

Claim B.39. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i,a} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i,b}| \leq \mathsf{negl}(\lambda)$.

Proof. Note that the only difference between $\widehat{F}^{0,3,i,a}$ and $\widehat{F}^{0,3,i,b}$ is that the latter uses a punctured PRF key $K_A\{i\}$ to compute the verification key for time $t-1$ and the signing key for time t. For verification, the functionality is preserved since $\mathrm{vk}_{C,\mathrm{rej}}$ is hardwired to the circuit. For signing, 'B' type key is never used to sign at time $t = i$. Therefore $\mathbf{Hyb}_{0,3,i,a}$ and $\mathbf{Hyb}_{0,3,i,b}$ are computationally indistinguishable. ☐

Claim B.40. *Let* PRF *be a selectively secure puncturable PRF; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i,b} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i,c}| \leq \mathsf{negl}(\lambda)$.

Proof. Since both $\widehat{F}^{0,3,i,b}$ and $\widehat{F}^{0,3,i,c}$ depend only on $K_A\{i\}$, by the security of indistinguishability obfuscation, the value of $\mathsf{PRF}(K_A, i)$ can be replaced by a random value. Therefore $\mathbf{Hyb}_{0,3,i,b}$ and $\mathbf{Hyb}_{0,3,i,c}$ are computationally indistinguishable under the assumption that PRF is selectively secure puncturable PRF. ☐

Claim B.41. *Let* Spl *be a splittable signature scheme which satisfies* $\mathrm{vk}_{\mathrm{rej}}$ *indistinguishability (Definition A.7); then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i,c} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i,d}| \leq \mathsf{negl}(\lambda)$.

Proof. Note that sk_C is not hardwired in either $\widehat{F}^{0,3,i,c}$ or $\widehat{F}^{0,3,i,d}$. Based on the $\mathrm{vk}_{\mathrm{rej}}$ indistinguishability property of splittable signature scheme Spl, given only vk_C or $\mathrm{vk}_{C,\mathrm{rej}}$, the two hybrids are computationally indistinguishable. ☐

Claim B.42. *Let* PRF *be a selectively secure puncturable PRF; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i,d} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i,e}| \leq \mathsf{negl}(\lambda)$.

Proof. Since both $\widehat{F}^{0,3,i,d}$ and $\widehat{F}^{0,3,i,e}$ depend only on $K_B\{i\}$, by the security of indistinguishability obfuscation, the random value can be switched back to $\mathsf{PRF}(K_B, i)$. Therefore $\mathbf{Hyb}_{0,3,i,d}$ and $\mathbf{Hyb}_{0,3,i,e}$ are computationally indistinguishable. ☐

Claim B.43. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i,e} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i,f}| \leq \mathsf{negl}(\lambda)$.

Proof. Observe that $\widehat{F}^{0,3,i,e}$ and $\widehat{F}^{0,3,i,f}$ have identical functionality. Therefore $\mathbf{Hyb}_{0,3,i,e}$ and $\mathbf{Hyb}_{0,3,i,f}$ are computationally indistinguishable under the assumption that $\mathsf{i}\mathcal{O}$ is a secure indistinguishability obfuscation scheme. ☐

Claim B.44. *Let* $\mathsf{i}\mathcal{O}$ *be a secure indistinguishability obfuscator; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i,f} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i+1}| \leq \mathsf{negl}(\lambda)$.

Proof. Observe that $\widehat{F}^{0,3,i,f}$ and $\widehat{F}^{0,3,i+1}$ have identical functionality. Therefore $\mathbf{Hyb}_{0,3,i,f}$ and $\mathbf{Hyb}_{0,3,i+1}$ are computationally indistinguishable under the assumption that $\mathsf{i}\mathcal{O}$ is a secure indistinguishability obfuscation scheme. ☐

To conclude, we have for all PPT $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,3,i} - \mathsf{Adv}_{\mathcal{A}}^{0,3,i+1}| \leq \mathsf{negl}(\lambda)$ as required.

☐

Algorithm 46: $\widehat{F}^{0,3,i}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (t, \mathsf{st}^{\mathrm{in}}, v^{\mathrm{in}}, w^{\mathrm{in}}, \sigma^{\mathrm{in}})$,  $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\mathrm{in}})$ where $a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, t^*$

1  if $t^* < t \leq i$ then output $\mathtt{Reject}$;
2  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}}, \pi^{\mathrm{in}}) = 0$ then output $\mathtt{Reject}$;
3  Compute $r_A = \mathsf{PRF}(K_A, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
4  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
5  Set $m^{\mathrm{in}} = (v^{\mathrm{in}}, \mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}})$ ;
6  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 0$ then output $\mathtt{Reject}$;

7  if  $t \leq t^*$ then
8      Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
9  else
10     Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}})$
11 if $\mathsf{st}^{\mathrm{out}} = \mathtt{Reject}$ then output $\mathtt{Reject}$;

12 $w^{\mathrm{out}} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\mathrm{in}}, \mathbf{B}^{\mathrm{out}}, \pi^{\mathrm{in}})$;
13 if $w^{\mathrm{out}} = \mathtt{Reject}$ then output $\mathtt{Reject}$;
14 Compute $v^{\mathrm{out}} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (\mathsf{st}^{\mathrm{in}}, w^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}))$;
15 if $v^{\mathrm{out}} = \mathtt{Reject}$ then output $\mathtt{Reject}$;
16 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
17 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
18 Set $m^{\mathrm{out}} = (v^{\mathrm{out}}, \mathsf{st}^{\mathrm{out}}, w^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}})$;
19 if $t = t^*$ then
20     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
21 else
22     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;

23 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (t+1, \mathsf{st}^{\mathrm{out}}, v^{\mathrm{out}}, w^{\mathrm{out}}, \sigma^{\mathrm{out}})$,   $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

Algorithm 47: $\widehat{F}^{0,3,i,b}$

```
Input : $\widetilde{\mathsf{st}}^{\rm in} = (t, \mathsf{st}^{\rm in}, v^{\rm in}, w^{\rm in}, \sigma^{\rm in})$,  $\widetilde{a}^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}} = (a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}}, \pi^{\rm in})$ where $a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\rm in}, \mathbf{B}^{\rm in})$
Data  : $T, \mathsf{pp}_{\mathsf{Acc}}, \mathsf{pp}_{\mathsf{Itr}}, \underline{K_A\{i\}}, K_B, t^*, \underline{\mathrm{vk}}$

1  if $t^* < t \le i$ then output $\mathtt{Reject}$;
2  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc}}, w^{\rm in}, \mathbf{I}^{\rm in}, \mathbf{B}^{\rm in}, \pi^{\rm in}) = 0$ then output $\mathtt{Reject}$;
3  if $\underline{t \ne i+1}$ then
4      Compute $r_A = \mathsf{PRF}(K_A\{i\}, t-1)$, $r_B = \mathsf{PRF}(K_B, t-1)$;
5      Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\rm rej}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$, $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\rm rej}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
6  else
7      $\underline{\text{Set } \mathrm{vk}_A = \mathrm{vk}}$;
8  Set $m^{\rm in} = (v^{\rm in}, \mathsf{st}^{\rm in}, w^{\rm in}, \mathbf{I}^{\rm in})$ ;
9  if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\rm in}, \sigma^{\rm in}) = 0$ then output $\mathtt{Reject}$;

10 if  $t \le t^*$ then
11     Compute $(\mathsf{st}^{\rm out}, a^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^1(\mathsf{st}^{\rm in}, a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}})$
12 else
13     Compute $(\mathsf{st}^{\rm out}, a^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}}) \leftarrow F^0(\mathsf{st}^{\rm in}, a^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}})$
14 if $\mathsf{st}^{\rm out} = \mathtt{Reject}$ then output $\mathtt{Reject}$;

15 $w^{\rm out} = \mathsf{Acc.Update}(\mathsf{pp}_{\mathsf{Acc}}, w^{\rm in}, \mathbf{B}^{\rm out}, \pi^{\rm in})$;
16 if $w^{\rm out} = \mathtt{Reject}$ then output $\mathtt{Reject}$;
17 Compute $v^{\rm out} = \mathsf{Itr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\rm in}, (\mathsf{st}^{\rm in}, w^{\rm in}, \mathbf{I}^{\rm in}))$;
18 if $v^{\rm out} = \mathtt{Reject}$ then output $\mathtt{Reject}$;
19 Compute $r'_A = \mathsf{PRF}(K_A, t)$, $r'_B = \mathsf{PRF}(K_B, t)$;
20 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\rm rej}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$, $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\rm rej}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
21 Set $m^{\rm out} = (v^{\rm out}, \mathsf{st}^{\rm out}, w^{\rm out}, \mathbf{I}^{\rm out})$;
22 if $t = t^*$ then
23     Compute $\sigma^{\rm out} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\rm out})$;
24 else
25     Compute $\sigma^{\rm out} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\rm out})$;

26 Output $\widetilde{\mathsf{st}}^{\rm out} = (t+1, \mathsf{st}^{\rm out}, v^{\rm out}, w^{\rm out}, \sigma^{\rm out})$,    $\widetilde{a}^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}} = a^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}}$;
```

B.2 Proof of Lemma 6.3 (Security for Topological Iterators)

Proof. To prove this lemma, we define $\mathbf{Hyb}_0, \ldots, \mathbf{Hyb}_3$.

$\mathbf{Hyb}_0$ In this experiment, the challenger always sends the normal setup, $(\mathsf{pp}_{\mathsf{Itr}}, v) \leftarrow \mathsf{TItr.Setup}(1^\lambda, N)$, to $\mathcal{A}$.

$\mathbf{Hyb}_1$ In this experiment, the challenger computes the normal setup $\mathsf{TItr.Setup}(1^\lambda, N)$ and the enforced setup with the sink point ct^* hardwired $(\mathsf{pp}_{\mathsf{Itr}}\{\mathsf{ct}^*, K\{v_{l,\mathsf{sink}}, v_{r,\mathsf{sink}}, m_{\mathsf{sink}}\}\}, v) \leftarrow \mathsf{TItr.SetupEnf}(1^\lambda, N, \mathbf{DAG})$, and it sends one of the two to $\mathcal{A}$.

$\mathbf{Hyb}_2$ In this experiment, the challenger computes the normal setup $\mathsf{TItr.Setup}(1^\lambda, N)$ and the enforced setup with the sink point ct^* (encrypted with a fresh randomness) hardwired by using the $\mathsf{TItr.SetupEnf}$ algorithm, i.e., $(\mathsf{pp}_{\mathsf{Itr}}\{\mathsf{ct}^*, K\{v_{l,\mathsf{sink}}, v_{r,\mathsf{sink}}, m_{\mathsf{sink}}\}\}, v) \leftarrow \mathsf{TItr.SetupEnf}(1^\lambda, N, \mathbf{DAG})$, and it sends one of the two to $\mathcal{A}$.

$\mathbf{Hyb}_3$ In this experiment, the challenger computes the normal setup $\mathsf{TItr.Setup}(1^\lambda, N)$ and the enforced setup and it sends one of the two to $\mathcal{A}$.

Claim B.45. *$\mathbf{Hyb}_3$ has only negligible advantage over $\mathbf{Hyb}_0$.*

Proof. Given that $\mathsf{TItr.SetupEnf}$ requires each node n has a *unique* message value $m_n \in \mathcal{M}_\lambda$, $\mathsf{progEnforce}$ only differs from prog at the input $(v_{l,\mathsf{sink}}, v_{r,\mathsf{sink}}, m_{\mathsf{sink}})$. Thus we focused on the sink point in following hybrid steps. Assuming $\mathsf{i}\mathcal{O}$ is a secure indistinguishable obfuscator, $\mathbf{Hyb}_0 \approx \mathbf{Hyb}_1$ because $\mathsf{pp}_{\mathsf{Itr}}$ and $\mathsf{pp}_{\mathsf{Itr}}\{\mathsf{ct}^*,$ $K\{v_{l,\mathsf{sink}},\ v_{r,\mathsf{sink}},\ m_{\mathsf{sink}}\}\}$ are functionally equivalent. $\mathbf{Hyb}_1 \approx \mathbf{Hyb}_2$ by the selective security of puncturable PRF (because the only difference is the randomness at the punctured point). $\mathbf{Hyb}_2 \approx \mathbf{Hyb}_3$ by the semantic security of $\mathcal{PKE}$. □

The adversary $\mathcal{A}$ has no advantage in $\mathbf{Hyb}_0$, and has only negligible advantage in $\mathbf{Hyb}_3$ (which is the **Exp-Setup-Itr** game) over $\mathbf{Hyb}_0$. Therefore, any PPT adversary $\mathcal{A}$ has only negligible advantage in the **Exp-Setup-Itr** game. □

B.3 Proof of Theorem 6.6 (Security for $\mathsf{Ci}\mathcal{O}$-mPRAM)

We now prove the security for our $\mathsf{Ci}\mathcal{O}$ in the mPRAM model. The proof idea is essentially similar to that for $\mathsf{Ci}\mathcal{O}$-RAM. We first allow the program to accept 'B' type signatures, to do this we can straightly use the sequence of hybrids backward from time $i = t^*$ to 0. Then, we slowly switch F^0 by F^1 from $t = 0$ to $t = t^*$. Recall that now both F^0 and F^1 include the branch and combine stages. The branch stage is taken care of using the same techniques in the proof of $\mathsf{Ci}\mathcal{O}$-RAM. In particular, the main challenge here is the combine stage. We will show how this is tackled by hardwiring $\log m$ amount of information in the intermediate steps of the combine stage.

Proof. To show the contradiction, we suppose the theorem statement is false, and then there exists a security parameter 1^λ, computation systems Π^0, Π^1 with the identical computation trace, and a PPT adversary $\mathcal{A}$ such that $|\Pr[\mathcal{A}(1^\lambda, \widetilde{\Pi}^\beta) = 1] - \frac{1}{2}|$ is non-negligible.

Before proceeding to the proof, we first define some notations and conventions used in the following proof.

- We use $\widehat{F}' = F_1 \boxplus F_2$ to denote a program which is identical to $\widehat{F}$ except that F_{branch} in $\widehat{F}$ is replaced by F_1 and F_{combine} is replaced by F_2.
- Unless specified, the challenger in each hybrid replaces $\widehat{F}$ by some $\widehat{F}'$, so that the computation system $\widehat{\Pi}$ defined by $\widehat{F}$ is replaced by another computation system $\widehat{\Pi}'$ that is defined by $\widehat{F}'$ and outputs the obfuscated computation $\widetilde{\Pi}' \leftarrow \mathsf{Obf}(1^\lambda, \widehat{\Pi}'; \rho)$.

We first define the first-layer hybrids $\mathbf{Hyb}_\beta$ for $\beta \in \{0, 1\}$.

Hyb$_\beta$ In this hybrid, the challenger replaces $\widehat{F}$ by $\widehat{F}^\beta = F^\beta_{\mathsf{branch}} \boxplus F_{\mathsf{combine}}$, where $F^\beta_{\mathsf{branch}}$ is similar to F_{branch} except that F is replaced by F^β.

Let us assume that the program $\widehat{F}^\beta$ terminates at time $t^* < T$. To argue that $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$, we define the second- and third-layer hybrids $\mathbf{Hyb}_{0,0}$, $\mathbf{Hyb}_{0,0,i}$, $\mathbf{Hyb}_{0,1}$, $\mathbf{Hyb}_{0,1,i}$, $\mathbf{Hyb}_{0,2}$, and $\mathbf{Hyb}_{0,3}$ for $0 \leq i \leq t^* - 1$.

$\mathbf{Hyb}_{0,0}$ This hybrid is identical to $\mathbf{Hyb}_0$ in the first layer.

$\mathbf{Hyb}_{0,0,i}$ In this hybrid, the challenger replaces $\widehat{F}$ by $\widehat{F}^{0,0,i} = F^{0,0,i}_{\mathsf{branch}} \boxplus F^{0,0,i}_{\mathsf{combine}}$ defined in Algorithms 48 and 49. $\widehat{F}^{0,0,i}$ is similar to $\widehat{F}^{0,0}$ except the following differences.
- $F^{0,0,i}_{\mathsf{branch}}$ uses F^1 to compute the output if the incoming signature is 'B' type.
- At $i+1 \leq t \leq t^* - 1$, $F^{0,0,i}_{\mathsf{branch}}$ and $F^{0,0,i}_{\mathsf{combine}}$ accept 'B' type signatures.
- At $i+1 \leq t \leq t^* - 1$, $F^{0,0,i}_{\mathsf{branch}}$ and $F^{0,0,i}_{\mathsf{combine}}$ follow the type of the incoming signature to generate the type of the outgoing signature.

$\mathbf{Hyb}_{0,1}$ In this hybrid, the challenger replaces $\widehat{F}$ by $\widehat{F}^{0,1} = F^{0,1}_{\mathsf{branch}} \boxplus F^{0,1}_{\mathsf{combine}}$ defined in Algorithms 50 and 51. This program is similar to $\widehat{F}^0$ except that it has PRF key K_B hardwired, accepts both 'A' and 'B' type signatures at time $t < t^*$. The type of the outgoing signature follows the type of the incoming signature. In addition, if the incoming signature is 'B' type and $t < t^*$, then $F^{0,1}_{\mathsf{branch}}$ uses F^1 to compute the output.

$\mathbf{Hyb}_{0,1,i}$ In this hybrid, the challenger replaces $\widehat{F}$ by $\widehat{F}^{0,1,i} = F^{0,1,i}_{\mathsf{branch}} \boxplus F^{0,1,i}_{\mathsf{combine}}$ defined in Algorithms 52 and 53. $\widehat{F}^{0,1,i}$ is similar to $\widehat{F}^{0,1}$ except the following differences.
- At $t \leq i$, $F^{0,1,i}_{\mathsf{branch}}$ uses F^1 to compute the output; otherwise, uses F^0.
- At $t = i$, $F^{0,1,i}_{\mathsf{branch}}$ has the correct input message m_i hardwired.
- At $t = i$, $F^{0,1,i}_{\mathsf{branch}}$ outputs an 'A' type signature if $m_{\mathrm{out}} = m_i$, 'B' type otherwise.
- At $i+1 \leq t \leq t^* - 1$, $F^{0,1,i}_{\mathsf{branch}}$ and $F^{0,1,i}_{\mathsf{combine}}$ accept 'B' type signatures.
- At $i+1 \leq t \leq t^* - 1$, $F^{0,1,i}_{\mathsf{branch}}$ and $F^{0,1,i}_{\mathsf{combine}}$ follow the type of the incoming signature to generate the type of the outgoing signature.

$\mathbf{Hyb}_{0,2}$ This hybrid is similar to $\mathbf{Hyb}_{0,1,t^*-1}$ except that $F^{0,2}_{\mathsf{branch}}$ does not hardwire the input message m_{t^*-1}.

$\mathbf{Hyb}_{0,3}$ This hybrid is similar to $\mathbf{Hyb}_{0,2}$ except that $F^{0,3}_{\mathsf{branch}}$ uses F^1 to compute the output if $t > t^*$ instead.

Analysis Let $\mathsf{Adv}^z_{\mathcal{A}}$ be the advantage of adversary $\mathcal{A}$ in $\mathbf{Hyb}_z$.

Lemma B.46. *Assuming* i$\mathcal{O}$ *is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* TItr *is an topological iterator satisfying Definitions 6.1 and 6.2,* Acc *is a secure accumulator,* Spl *is a secure splittable signature scheme; then for any PPT adversary $\mathcal{A}$,* $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^{0,1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof.
- $\mathbf{Hyb}_0$ is identical to $\mathbf{Hyb}_{0,0}$.
- $\mathbf{Hyb}_{0,0}$ is identical to $\mathbf{Hyb}_{0,0,t^*-1}$.
- $\mathbf{Hyb}_{0,0,i} \approx \mathbf{Hyb}_{0,0,i-1}$ for $1 \leq i \leq t^*$ will be proven in Appendix B.3.1.
- $\mathbf{Hyb}_{0,0,0}$ is identical to $\mathbf{Hyb}_{0,1}$.

□

Lemma B.47. *Assuming* i$\mathcal{O}$ *is a secure indistinguishability obfuscator,* $|\mathsf{Adv}^{0,1}_{\mathcal{A}} - \mathsf{Adv}^{0,2}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof.
- $\mathbf{Hyb}_{0,1} \approx \mathbf{Hyb}_{0,1,0}$ since the programs $\widehat{F}^{0,1}$ and $\widehat{F}^{0,1,0}$ are functionally identical.

- $\mathbf{Hyb}_{0,1,i} \approx \mathbf{Hyb}_{0,1,i+1}$ for $0 \leq i \leq t^* - 1$ will be proven in Appendix B.3.2.
- $\mathbf{Hyb}_{0,1,t^*-1} \approx \mathbf{Hyb}_{0,2}$ since the programs $\widehat{F}^{0,1,t^*-1}$ and $\widehat{F}^{0,2}$ are functionally identical.

□

Lemma B.48. *Assuming* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* $|\mathsf{Adv}^{0,2}_{\mathcal{A}} - \mathsf{Adv}^{0,3}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. The programs $\widehat{F}^{0,2}$ and $\widehat{F}^{0,3}$ are functionally identical, since they run neither F^0 nor F^1 at time $t > t^*$. □

Lemma B.49. *Assuming* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* TItr *is an topological iterator satisfying Definitions 6.1 and 6.2,* Acc *is a secure positional accumulator,* Spl *is a secure splittable signature scheme; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}^{0,3}_{\mathcal{A}} - \mathsf{Adv}^{1}_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

The proof of this lemma is identical to the previous proof technique of $\mathsf{Ci}\mathcal{O}$-RAM. □

Algorithm 48: $F^{0,0,i}_{\mathsf{branch}}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\mathrm{in}} = (\mathsf{com}^{\mathrm{in}}, \pi^{\mathrm{in}}_{\mathsf{st}}, \pi^{\mathrm{in}}_{\mathsf{com}})$
Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  Parse $\mathsf{root\_node}$ as $(t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}}, \sigma^{\mathrm{in}})$;
2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$ and $r_B = \mathsf{PRF}(K_B, (t, \mathsf{root\_index}))$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Let $\alpha =$ '-' and $m^{\mathrm{in}} = (t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}})$;
5  if $t \leq i$ then
6      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A';
7      Else, output Reject;
8  else
9      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A';
       if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B';
10     if $\alpha =$ '-' then output Reject;
11 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\mathrm{in}}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{st}}) = 0$ then output Reject;
12 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\mathrm{in}}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{com}}) = 0$ then output Reject;
13 if $\alpha =$ 'B' then
14     Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^1(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
15 else
16     Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^0(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
17 Compute $v^{\mathrm{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}))$;
18 if $\mathsf{st}^{\mathrm{out}} =$ Reject then
19     Output Reject;
20 else
21     Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$ and $r'_B = \mathsf{PRF}(K_B, (t+1, \mathtt{A}))$;
22     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
23     Let $m^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}})$;
24     Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
25     Let $\mathsf{node}^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}}, \sigma^{\mathrm{out}})$;
26     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\mathrm{out}} = \mathsf{node}^{\mathrm{out}}$;
```

Algorithm 49: $F^{0,0,i}_{\mathsf{combine}}$

```
   Input : $\widetilde{\mathsf{st}}^{\rm in} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\rm in} = (\mathsf{node}_1, \mathsf{node}_2)$
   Data  : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$ and $r_{B,\zeta} = \mathsf{PRF}(K_B, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\mathrm{sk}_{B,\zeta}, \mathrm{vk}_{B,\zeta}, \mathrm{vk}_{B,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta$ = '-' and $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $t \leq i$ then
12         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
13         else output Reject;
14     else
15         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
           if $\alpha_\zeta \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'B';
16         if $\alpha_\zeta$ = '-' then output Reject;
17 if $\alpha_1$ = 'A' and $\alpha_2$ = 'A' then set $\alpha$ = 'A';else set $\alpha$ = 'B';
18 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2})$;
19 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2})$;
20 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
21 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$ and $r'_B = \mathsf{PRF}(K_B, (t, \mathsf{parent\_index}))$;
22 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
23 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
24 if $t \leq i$ then
25     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
26 else
27     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
28 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
29 if $\mathsf{parent\_index} = \epsilon$ then
30     Output $\widetilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\rm out} = \perp$;
31 else
32     Output $\widetilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\rm out} = \mathsf{parent\_node}$;
```

Algorithm 50: $F^{0,1}_{\textsf{branch}}$

```
Input : $\widetilde{\textsf{st}}^{\text{in}} = (\textsf{st}^{\text{in}}, id_{\textsf{cpu}}, \textsf{root\_node}), \widetilde{a}^{\text{in}} = (\textsf{com}^{\text{in}}, \pi^{\text{in}}_{\textsf{st}}, \pi^{\text{in}}_{\textsf{com}})$
Data  : $\textsf{pp}_{\textsf{Acc,st}}, \textsf{pp}_{\textsf{Acc,com}}, \textsf{pp}_{\textsf{Itr}}, K_A$
1  Parse root_node as $(t, \textsf{root\_index}, w^{\text{in}}_{\textsf{st}}, w^{\text{in}}_{\textsf{com}}, v^{\text{in}}, \sigma^{\text{in}})$;
2  Let $r_A = \textsf{PRF}(K_A, (t, \textsf{root\_index}))$ and $r_B = \textsf{PRF}(K_B, (t, \textsf{root\_index}))$;
3  Compute $(\text{sk}_A, \text{vk}_A, \text{vk}_{A,rej}) = \textsf{Spl.Setup}(1^\lambda; r_A)$ and $(\text{sk}_B, \text{vk}_B, \text{vk}_{B,\text{rej}}) = \textsf{Spl.Setup}(1^\lambda; r_B)$;
4  Let $\alpha$ = '-' and $m^{\text{in}} = (t, \textsf{root\_index}, w^{\text{in}}_{\textsf{st}}, w^{\text{in}}_{\textsf{com}}, v^{\text{in}})$;
5  if $\textsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha$ = 'A';
6  if $\alpha$ = '-' and $t > t^*$ then output Reject;
7  if $\alpha \neq$ 'A' and $\textsf{Spl.Verify}(\text{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha$ = 'B';
8  if $\alpha$ = '-' then output Reject;
9  if $\textsf{Acc.VerifyRead}(\textsf{pp}_{\textsf{Acc,st}}, w^{\text{in}}_{\textsf{st}}, (id_{\textsf{cpu}}, \textsf{st}^{\text{in}}), \pi^{\text{in}}_{\textsf{st}}) = 0$ then output Reject;
10 if $\textsf{Acc.VerifyRead}(\textsf{pp}_{\textsf{Acc,com}}, w^{\text{in}}_{\textsf{com}}, (\textsf{src}(t, id_{\textsf{cpu}}), \textsf{com}^{\text{in}}), \pi^{\text{in}}_{\textsf{com}}) = 0$ then output Reject;
11 if $\alpha$ = 'B' then
12     Compute $(\textsf{st}^{\text{out}}, \textsf{com}^{\text{out}}) \leftarrow F^1(id_{\textsf{cpu}}, \textsf{st}^{\text{in}}, \textsf{com}^{\text{in}})$;
13 else
14     Compute $(\textsf{st}^{\text{out}}, \textsf{com}^{\text{out}}) \leftarrow F^0(id_{\textsf{cpu}}, \textsf{st}^{\text{in}}, \textsf{com}^{\text{in}})$;
15 Compute $v^{\text{out}} = \textsf{TItr.Iterate}(\textsf{pp}_{\textsf{Itr}}, v^{\text{in}}, (t+1, id_{\textsf{cpu}}, \textsf{st}^{\text{in}}, \textsf{com}^{\text{in}}, w^{\text{in}}_{\textsf{st}}, w^{\text{in}}_{\textsf{com}}))$;
16 if $\textsf{st}^{\text{out}}$ = Reject then
17     Output Reject;
18 else
19     Let $r'_A = \textsf{PRF}(K_A, (t+1, id_{\textsf{cpu}}))$ and $r'_B = \textsf{PRF}(K_B, (t+1, \texttt{A}))$;
20     Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,rej}) = \textsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,\text{rej}}) = \textsf{Spl.Setup}(1^\lambda; r'_B)$;
21     Let $m^{\text{out}} = (t+1, id_{\textsf{cpu}}, \textsf{st}^{\text{out}}, \textsf{com}^{\text{out}}, v^{\text{out}})$;
22     Compute $\sigma^{\text{out}} = \textsf{Spl.Sign}(\text{sk}'_\alpha, m^{\text{out}})$;
23     Let $\textsf{node}^{\text{out}} = (t+1, id_{\textsf{cpu}}, \textsf{st}^{\text{out}}, \textsf{com}^{\text{out}}, v^{\text{out}}, \sigma^{\text{out}})$;
24     Output $\widetilde{\textsf{st}}^{\text{out}} = (\textsf{st}^{\text{out}}, id_{\textsf{cpu}}, \perp), \widetilde{a}^{\text{out}} = \textsf{node}^{\text{out}}$;
```

Algorithm 51: $F^{0,1}_{\mathsf{combine}}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\mathrm{in}} = (\mathsf{node}_1, \mathsf{node}_2)$
Data  : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$ and $r_{B,\zeta} = \mathsf{PRF}(K_B, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\mathrm{sk}_{B,\zeta}, \mathrm{vk}_{B,\zeta}, \mathrm{vk}_{B,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta =$ '-' and $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'A';
12     if $\alpha_\zeta \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'B';
13     if $\alpha_\zeta =$ '-' then output Reject;
14 if $\alpha_1 =$ 'A' and $\alpha_2 =$ 'A' then set $\alpha =$ 'A';
15 else set $\alpha =$ 'B';
16 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2})$;
17 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2})$;
18 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
19 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$ and $r'_B = \mathsf{PRF}(K_B, (t, \mathsf{parent\_index}))$;
20 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
21 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
22 Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
23 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
24 if $\mathsf{parent\_index} = \epsilon$ then
25     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\mathrm{out}} = \perp$;
26 else
27     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\mathrm{out}} = \mathsf{parent\_node}$;
```

Algorithm 52: $F^{0,1,i}_{\mathsf{branch}}$

```
Input : $\widetilde{\mathsf{st}}^{\rm in} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \mathsf{root\_node})$, $\widetilde{a}^{\rm in} = (\mathsf{com}^{\rm in}, \pi^{\rm in}_{\mathsf{st}}, \pi^{\rm in}_{\mathsf{com}})$
Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m_{i,\mathsf{Root}}$
1  Parse root_node as $(t, \mathsf{root\_index}, w^{\rm in}_{\mathsf{st}}, w^{\rm in}_{\mathsf{com}}, v^{\rm in}, \sigma^{\rm in})$;
2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$ and $r_B = \mathsf{PRF}(K_B, (t, \mathsf{root\_index}))$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Let $\alpha =$ '-' and $m^{\rm in} = (t, \mathsf{root\_index}, w^{\rm in}_{\mathsf{st}}, w^{\rm in}_{\mathsf{com}}, v^{\rm in})$;
5  if $t \leq i$ then
6      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\rm in}, \sigma^{\rm in}) = 1$ then set $\alpha =$ 'A'; else output Reject;
7  else
8      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\rm in}, \sigma^{\rm in}) = 1$ then set $\alpha =$ 'A';
9      if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\rm in}, \sigma^{\rm in}) = 1$ then set $\alpha =$ 'B';
10     if $\alpha =$ '-' then output Reject;
11 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\rm in}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\rm in}), \pi^{\rm in}_{\mathsf{st}}) = 0$ then output Reject;
12 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\rm in}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\rm in}), \pi^{\rm in}_{\mathsf{com}}) = 0$ then output Reject;
13 if $t \leq i$ or $\alpha =$ 'B' then
14     Compute $(\mathsf{st}^{\rm out}, \mathsf{com}^{\rm out}) \leftarrow F^1(id_{\mathsf{cpu}}, \mathsf{st}^{\rm in}, \mathsf{com}^{\rm in})$;
15 else
16     Compute $(\mathsf{st}^{\rm out}, \mathsf{com}^{\rm out}) \leftarrow F^0(id_{\mathsf{cpu}}, \mathsf{st}^{\rm in}, \mathsf{com}^{\rm in})$;
17 Compute $v^{\rm out} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\rm in}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\rm in}, \mathsf{com}^{\rm in}, w^{\rm in}_{\mathsf{st}}, w^{\rm in}_{\mathsf{com}}))$;
18 if $\mathsf{st}^{\rm out} =$ Reject then
19     Output Reject;
20 else
21     Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$ and $r'_B = \mathsf{PRF}(K_B, (t+1, \mathtt{A}))$;
22     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
23     Let $m^{\rm out} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\rm out}, \mathsf{com}^{\rm out}, v^{\rm out})$;
24     if $t = i$ and $m^{\rm in} = m_{i,\mathsf{Root}}$ then compute $\sigma^{\rm out} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\rm out})$;
       else if $t = i$ and $m^{\rm in} \neq m_{i,\mathsf{Root}}$ then compute $\sigma^{\rm out} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\rm out})$;
       else compute $\sigma^{\rm out} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\rm out})$;
25     Let $\mathsf{node}^{\rm out} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\rm out}, \mathsf{com}^{\rm out}, v^{\rm out}, \sigma^{\rm out})$;
26     Output $\widetilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm out}, id_{\mathsf{cpu}}, \perp)$, $\widetilde{a}^{\rm out} = \mathsf{node}^{\rm out}$;
```

Algorithm 53: $F^{0,1,i}_{\mathsf{combine}}$

```
Input : $\tilde{\mathsf{st}}^{\rm in} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \bot), \tilde{a}^{\rm in} = (\mathsf{node}_1, \mathsf{node}_2)$
Data  : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$ and $r_{B,\zeta} = \mathsf{PRF}(K_B, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\mathrm{sk}_{B,\zeta}, \mathrm{vk}_{B,\zeta}, \mathrm{vk}_{B,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta =$ '-' and $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $t \leq i$ then
12         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'A'; else output Reject;
13     else
14         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'A';
15         if $\alpha_\zeta \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'B';
16         if $\alpha_\zeta =$ '-' then output Reject;
17 if $\alpha_1 =$ 'A' and $\alpha_2 =$ 'A' then
18     set $\alpha =$ 'A'
19 else
20     set $\alpha =$ 'B'
21 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2})$;
22 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2})$;
23 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
24 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$ and $r'_B = \mathsf{PRF}(K_B, (t, \mathsf{parent\_index}))$;
25 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
26 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
27 if $t \leq i$ then
28     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
29 else
30     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
31 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
32 if $\mathsf{parent\_index} = \epsilon$ then
33     Output $\tilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \mathsf{parent\_node})$, $\tilde{a}^{\rm out} = \bot$;
34 else
35     Output $\tilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \bot)$, $\tilde{a}^{\rm out} = \mathsf{parent\_node}$;
```

B.3.1 From $\mathbf{Hyb}_{0,0,i}$ to $\mathbf{Hyb}_{0,0,i-1}$:

Lemma B.50. *Assuming* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* TItr *is an topological iterator satisfying Definitions 6.1 and 6.2,* Acc *is a secure accumulator,* Spl *is a secure splittable signature scheme; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i-1}| \leq \mathsf{negl}(\lambda)$.

Proof. To argue $|\mathsf{Adv}_{\mathcal{A}}^{0,0,i} - \mathsf{Adv}_{\mathcal{A}}^{0,0,i-1}| \leq \mathsf{negl}(\lambda)$, we define a sequence of fourth-layer hybrids $\mathbf{Hyb}_{0,0,i,j}$ where j is indexed by the node index via pre-order. For example, in Figure 3, we consider 4 CPUs, the order of hybrids is ($\mathbf{Hyb}_{0,0,i,\epsilon}$, $\mathbf{Hyb}_{0,0,i,0}$, $\mathbf{Hyb}_{0,0,i,00}$, $\mathbf{Hyb}_{0,0,i,01}$, $\mathbf{Hyb}_{0,0,i,1}$, $\mathbf{Hyb}_{0,0,i,10}$, $\mathbf{Hyb}_{0,0,i,11}$).

$\mathbf{Hyb}_{0,0,i,j}$ This hybrid similar to $\mathbf{Hyb}_{0,0,i}$ except of the following:

- At time $t = i$, if $id_{\mathsf{cpu}} \geq \mathsf{min\text{-}cpu}(j)$, $F_{\mathsf{branch}}^{0,0,i,j}$ follows the type of the incoming signature to generate the type of the outgoing signature.
- At time $t = t + 1$, if $\mathsf{parent\_index} \geq j$, $F_{\mathsf{combine}}^{0,0,i,j}$ only accepts 'A' type signatures.

Finally, from $\mathbf{Hyb}_{0,0,i,j}$ to $\mathbf{Hyb}_{0,0,i,j+1}$, we can directly apply KLW proof technique as in the proof of Lemma B.2. □

B.3.2 From $\mathbf{Hyb}_{0,1,i}$ to $\mathbf{Hyb}_{0,1,i+1}$:

Lemma B.51. *Assuming* $\mathsf{i}\mathcal{O}$ *is a secure indistinguishability obfuscator,* PRF *is a selectively secure puncturable PRF,* TItr *is an topological iterator satisfying Definitions 6.1 and 6.2,* Acc *is an accumulator,* Spl *is a secure splittable signature scheme; then for any PPT adversary* $\mathcal{A}$, $|\mathsf{Adv}_{\mathcal{A}}^{0,1,i} - \mathsf{Adv}_{\mathcal{A}}^{0,1,i+1}| \leq \mathsf{negl}(\lambda)$.

Proof. To argue $|\mathsf{Adv}_{\mathcal{A}}^{0,1,i} - \mathsf{Adv}_{\mathcal{A}}^{0,1,i+1}| \leq \mathsf{negl}(\lambda)$, we define a sequence of fourth-layer hybrids $\mathbf{Hyb}_{0,1,i,j(\text{type})}$ where j is indexed by the node index via post-order, and (type) specifies the type of node j.

Concretely, we define the fourth-layer hybrids $\mathbf{Hyb}_{0,1,i,j(\mathsf{left\text{-}leaf})}$, $\mathbf{Hyb}_{0,1,i,j(\mathsf{right\text{-}leaf})}$, $\mathbf{Hyb}_{0,1,i,j(\mathsf{internal})}$, and $\mathbf{Hyb}_{0,1,i,j(\mathsf{intermediate})}$. See Figure 3 as an example.

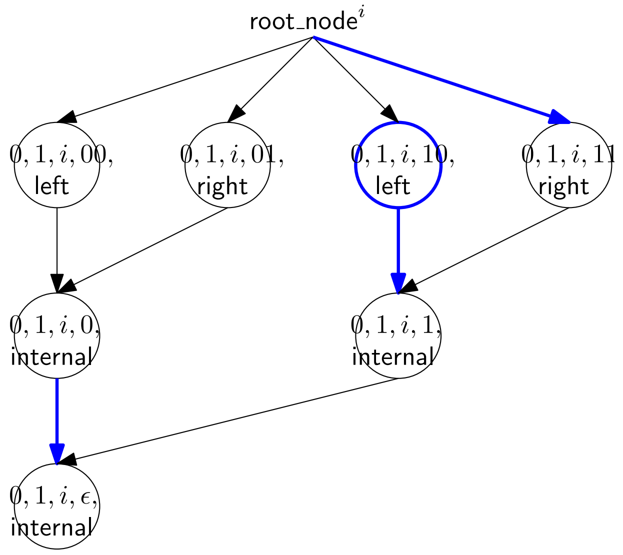

Figure 3: The sequence of hybrids with 4 CPUs: Each node is a computation step, which also has a corresponding hybrid on it. Each arrow is the input/output to be hardwired by some hybrids. Hybrids are proceeded by: $\mathbf{Hyb}_{0,1,i,00,(\mathsf{left\text{-}leaf})}$, $\mathbf{Hyb}_{0,1,i,01,(\mathsf{right\text{-}leaf})}$, $\mathbf{Hyb}_{0,1,i,0,(\mathsf{internal})}$, $\mathbf{Hyb}_{0,1,i,10,(\mathsf{left\text{-}leaf})}$, $\mathbf{Hyb}_{0,1,i,11,(\mathsf{right\text{-}leaf})}$, $\mathbf{Hyb}_{0,1,i,1,(\mathsf{internal})}$, $\mathbf{Hyb}_{0,1,i,\epsilon,(\mathsf{internal})}$. As an example, $\mathbf{Hyb}_{0,1,i,10,(\mathsf{left\text{-}leaf})}$ is circled in thick blue line $(0,1,i,10)$, and its hardwired information are shown in thick blue arrows.

Algorithm 54: $F^{0,0,i,j}_{\mathsf{branch}}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\mathrm{in}} = (\mathsf{com}^{\mathrm{in}}, \pi^{\mathrm{in}}_{\mathsf{st}}, \pi^{\mathrm{in}}_{\mathsf{com}})$
Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
Parse root_node as $(t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}}, \sigma^{\mathrm{in}})$;
Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$ and $r_B = \mathsf{PRF}(K_B, (t, \mathsf{root\_index}))$;
Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
Let $\alpha =$ '-' and $m^{\mathrm{in}} = (t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}})$;
if $t \le i$ then
    if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A';
    else output Reject;
else
    if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A';
    if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B';
    if $\alpha =$ '-' then output Reject;
if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\mathrm{in}}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{st}}) = 0$ then output Reject;
if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\mathrm{in}}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{com}}) = 0$ then output Reject;
if $\alpha =$ 'B' then
    Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^1(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
else
    Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^0(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
Compute $v^{\mathrm{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}))$;
if $\mathsf{st}^{\mathrm{out}} =$ Reject then
    Output Reject;
else
    Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$ and $r'_B = \mathsf{PRF}(K_B, (t+1, \mathtt{A}))$;
    Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
    Let $m^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}})$;
    if $\underline{t = i}$ then
        if $id_{\mathsf{cpu}} \ge \mathsf{min\text{-}cpu}(j)$ then compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
        else compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
    else
        Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
    Let $\mathsf{node}^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}}, \sigma^{\mathrm{out}})$;
    Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, id_{\mathsf{cpu}}, \bot), \widetilde{a}^{\mathrm{out}} = \mathsf{node}^{\mathrm{out}}$;
```

Algorithm 55: $F^{0,0,i,j}_{\mathsf{combine}}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\text{in}} = (\mathsf{node}_1, \mathsf{node}_2)$
Data  : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$ and $r_{B,\zeta} = \mathsf{PRF}(K_B, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\mathrm{sk}_{B,\zeta}, \mathrm{vk}_{B,\zeta}, \mathrm{vk}_{B,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta$ = '-' and $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $t \leq i$ or $(\underline{t = i + 1 \text{ and } \mathsf{parent\_index} \geq j})$ then
12         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
13         else output Reject;
14     else
15         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
16         else output Reject;
17         if $\alpha_\zeta \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'B';
18         if $\alpha_\zeta$ = '-' then output Reject;
19 if $\alpha_1$ = 'A' and $\alpha_2$ = 'A' then set $\alpha$ = 'A';
20 else set $\alpha$ = 'B';
21 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2})$;
22 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2})$;
23 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
24 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$ and $r'_B = \mathsf{PRF}(K_B, (t, \mathsf{parent\_index}))$;
25 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
26 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
27 if $t \leq i$ then
28     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
29 else
30     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
31 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
32 if $\mathsf{parent\_index} = \epsilon$ then
33     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\text{out}} = \perp$;
34 else
35     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\text{out}} = \mathsf{parent\_node}$;
```

Hyb$_{0,1,i,j(\textbf{left-leaf})}$**:** j is a left leaf node. In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,1,i,j(\text{left-leaf})} = F_{\text{branch}}^{0,1,i,j(\text{left-leaf})} \boxplus F_{\text{combine}}^{0,1,i,j(\text{left-leaf})}$ defined in Algorithms 56 and 57. $\widehat{F}^{0,1,i,j(\text{left-leaf})}$ is similar to $\widehat{F}^{0,1,i}$ except the following differences.

– At $t = i$, $F_{\text{branch}}^{0,1,i,j(\text{left-leaf})}$ has the correct input message $m_{i,\text{root\_index}}$ and an agent j's output message $m_{i,j}$ hardwired.
– At $t = i$, if $m^{\text{in}} = m_i$ and $\mathtt{A} > j$, $F_{\text{branch}}^{0,1,i,j(\text{left-leaf})}$ outputs an 'A' type signature.
If $m^{\text{in}} \neq m_i$ and $\mathtt{A} > j$, outputs a 'B' type signature.
If $m^{\text{out}} = m_{i,j}$ and $\mathtt{A} = j$, outputs an 'A' type signature.
If $m^{\text{out}} \neq m_{i,j}$ and $\mathtt{A} = j$, outputs a 'B' type signature.
– $F_{\text{combine}}^{0,1,i,j(\text{left-leaf})}$ hardwires a set of indices $\mathbf{C}_{i,j}$ and the corresponding set of output message $\mathbf{M}_{i,j}$.
– $F_{\text{combine}}^{0,1,i,j(\text{left-leaf})}$ accepts 'B' type signatures only for inputs if $i + 2 \leq t \leq t^* - 1$ or ($\text{parent\_index} > j$ and $t = i + 1$).
– For $\text{parent\_index} \in \mathbf{C}_{i,j}$ at time $t = i + 1$, $F_{\text{combine}}^{0,1,i,j(\text{left-leaf})}$ checks whether $m' = m_{\text{parent\_index}} \in \mathbf{M}_{i,j}$ or not. If $m' = m_{\text{parent\_index}}$, outputs an 'A' type signature; otherwise, outputs 'B' type.

Hyb$_{0,1,i,j(\textbf{right-leaf})}$**:** j is a right leaf node. In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,1,i,j(\text{right-leaf})} = F_{\text{branch}}^{0,1,i,j(\text{right-leaf})} \boxplus F_{\text{combine}}^{0,1,i,j(\text{right-leaf})}$ where $F_{\text{branch}}^{0,1,i,j(\text{right-leaf})}$ is defined in Algorithm 58. In particular, $F_{\text{combine}}^{0,1,i,j(\text{right-leaf})}$ is functionally identical to $F_{\text{combine}}^{0,1,i,j-1(\text{left-leaf})}$ since hardwired $(\mathbf{C}_{i,j}, \mathbf{M}_{i,j})$ in $F_{\text{combine}}^{0,1,i,j(\text{right-leaf})}$ is identical to the hardwired $(\mathbf{C}_{i,j-1}, \mathbf{M}_{i,j-1})$ in $F_{\text{combine}}^{0,1,i,j-1(\text{left-leaf})}$. $\widehat{F}^{0,1,i,j(\text{right-leaf})}$ is similar to $\widehat{F}^{0,1,i}$ except the following differences.

– At $t = i$, $F_{\text{branch}}^{0,1,i,j(\text{right-leaf})}$ has the correct input message $m_{i,\text{root\_index}}$ and agents $j-1$ and j's output message $m_{i,j-1}$ and $m_{i,j}$ hardwired.
– At $t = i$, if $m^{\text{in}} = m_{i,\text{root\_index}}$ and $\mathtt{A} > j$, $F_{\text{branch}}^{0,1,i,j(\text{right-leaf})}$ outputs an 'A' type signature.
If $m^{\text{in}} \neq m_{i,\text{root\_index}}$ and $\mathtt{A} > j$, outputs a 'B' type signature.
If $m_{\text{out}} = m_{i,j}$ and $\mathtt{A} = j$, outputs an 'A' type signature.
If $m_{\text{out}} \neq m_{i,j}$ and $\mathtt{A} = j$, outputs a 'B' type signature.
If $m_{\text{out}} = m_{i,j-1}$ and $\mathtt{A} = j - 1$, outputs an 'A' type signature.
If $m_{\text{out}} \neq m_{i,j-1}$ and $\mathtt{A} = j - 1$, outputs a 'B' type signature.

Hyb$_{0,1,i,j(\textbf{internal})}$**:** j is an internal node. In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,1,i,j(\text{internal})} = F_{\text{branch}}^{0,1,i,j(\text{internal})} \boxplus F_{\text{combine}}^{0,1,i,j(\text{internal})}$ defined in Algorithms 59 and 60. $\widehat{F}^{0,1,i,j(\text{internal})}$ is similar to $\widehat{F}^{0,1,i}$ except the following differences.

– At $t = i$, if $m^{\text{in}} = m_{i,\text{root\_index}}$ and $\mathtt{A} > \text{max-cpu}(j)$, $F_{\text{branch}}^{0,1,i,j(\text{internal})}$ outputs an 'A' type signature.
If $m^{\text{in}} \neq m_{i,\text{root\_index}}$ and $\mathtt{A} > \text{max-cpu}(j)$, outputs a 'B' type signature.
– $F_{\text{combine}}^{0,1,i,j(\text{internal})}$ hardwires an output message $m_{i,j}$, a set of indices $\mathbf{C}_{i,j}$ and the corresponding set of output message $\mathbf{M}_{i,j}$.
– $F_{\text{combine}}^{0,1,i,j(\text{internal})}$ accepts 'B' type signatures only for inputs if $i + 2 \leq t \leq t^* - 1$ or ($\text{parent\_index} > j$ and $t = i + 1$).
– For $\text{parent\_index} \in \mathbf{C}_{i,j}$ at time $t = i + 1$, $F_{\text{combine}}^{0,1,i,j(\text{internal})}$ checks whether $m' = m_{\text{parent\_index}} \in \mathbf{M}_{i,j}$ or not. If $m' = m_{\text{parent\_index}}$, outputs an 'A' type signature; otherwise, outputs 'B' type.
– For $\text{parent\_index} = j$ at time $t = i + 1$, $F_{\text{combine}}^{0,1,i,j(\text{internal})}$ checks whether $m' = m_{i,j}$ or not. If $m' = m_{i,j}$, outputs an 'A' type signature; otherwise, outputs 'B' type.

Hyb$_{0,1,i,j(\textbf{intermediate})}$**:** j is an internal node. In this hybrid, the challenger outputs an obfuscation of $\widehat{F}^{0,1,i,j(\text{intermediate})} = F_{\text{branch}}^{0,1,i,j(\text{intermediate})} \boxplus F_{\text{combine}}^{0,1,i,j(\text{intermediate})}$. $F_{\text{branch}}^{0,1,i,j(\text{intermediate})}$ is functionally identical to

$F_{\textsf{branch}}^{0,1,i,j(\textsf{internal})}$, and $F_{\textsf{combine}}^{2,i,j(\textsf{intermediate})}$ is defined in Algorithm 61. This hybrid is similar to $\mathbf{Hyb}_{0,1,i,j(\textsf{internal})}$ except that

- $F_{\textsf{combine}}^{0,1,i,j(\textsf{intermediate})}$ hardwires two input messages $(m_{i,j,1}, m_{i,j,2})$
- For $\textsf{parent\_index} = j$ at time $t = i+1$, $F_{\textsf{combine}}^{0,1,i,j(\textsf{intermediate})}$ outputs 'A' type signature if $m_1 = m_{i,j,1}$ and $m_2 = m_{i,j,2}$, or outputs 'B' type if not.

Conclusively, we define $\mathbf{Hyb}_{0,1,i,\textsf{root\_index}^*}$: in $\widehat{F}^{0,1,i,\textsf{root\_index}^*}$, $F_{\textsf{branch}}^{0,1,i,\textsf{root\_index}^*}$ is similar to $F_{\textsf{branch}}^{0,1,i+1}$ except for no hardwired information, and $F_{\textsf{combine}}^{0,1,i,\textsf{root\_index}^*}$ is identical to $F_{\textsf{combine}}^{0,1,i,\textsf{root\_index}(\textsf{internal})}$. Note that $\widehat{F}^{0,1,i,\textsf{root\_index}(\textsf{internal})}$ and $\widehat{F}^{0,1,i,\textsf{root\_index}^*}$ are functionally equivalent, which implies $\mathbf{Hyb}_{0,1,i,\textsf{root\_index}(\textsf{internal})} \approx \mathbf{Hyb}_{0,1,i,\textsf{root\_index}^*}$. Then, we conclude $\mathbf{Hyb}_{0,1,i,\textsf{root\_index}^*} \approx \mathbf{Hyb}_{0,1,i+1}$ by KLW proof technique. In general, we can directly apply it as in the proof of Lemma B.9 to prove the argument from any case $\mathbf{Hyb}_{0,1,i,j}$ to $\mathbf{Hyb}_{0,1,i,j+1}$ (with/without $\mathbf{Hyb}_{0,1,i,j+1(\textsf{intermediate})}$).

Instantiation Consider the following sequence of hybrids which corresponds to $\mathbf{Hyb}_{0,1,i,10}$, $\mathbf{Hyb}_{0,1,i,11}$, $\mathbf{Hyb}_{0,1,i,1}$, and $\mathbf{Hyb}_{0,1,i,\epsilon}$ in Figure 3.

$$\mathbf{Hyb}_{0,1,i,j(\textsf{left-leaf})} \approx \mathbf{Hyb}_{0,1,i,j+1(\textsf{right-leaf})} \approx \mathbf{Hyb}_{0,1,i,j+2(\textsf{internal})} \approx \mathbf{Hyb}_{0,1,i,j+3(\textsf{internal})}$$

To prove the indistinguishability of the hybrids in the above sequence, we further claim the sequences below:

- $\mathbf{Hyb}_{0,1,i,j(\textsf{left-leaf})} \approx \mathbf{Hyb}_{0,1,i,j+1(\textsf{right-leaf})}$
- $\mathbf{Hyb}_{0,1,i,j+1(\textsf{right-leaf})} \approx \mathbf{Hyb}_{0,1,i,j+2(\textsf{intermediate})} \approx \mathbf{Hyb}_{0,1,i,j+2(\textsf{internal})}$
- $\mathbf{Hyb}_{0,1,i,j+2(\textsf{internal})} \approx \mathbf{Hyb}_{0,1,i,j+3(\textsf{intermediate})} \approx \mathbf{Hyb}_{0,1,i,j+3(\textsf{internal})}$

The indistinguishability of the hybrids in each of the above sequence can be proven by the KLW proof techniques as in the proof of Lemma B.9. Note that we only need to hardwire $O(\log m)$ messages in $\mathbf{Hyb}_{0,1,i,j}$ according to the above argument. □

B.4 Proof Sketch of Theorem 6.8 (Security for $\mathsf{Ci}\mathcal{O}$-PRAM)

Compared to the $\mathsf{Ci}\mathcal{O}$ construction in the memoryless PRAM model, there are two major differences in the construction in the standard PRAM model. Firstly, we need to verify memory inputs, which can be a value that read from memory, or a proof of the path to the writing location, against the memory accumulator. Secondly, we need to compute the memory accumulator (digest) by running oblivious algorithm $\mathsf{OUpdate}$ several rounds. Both differences have a similar hybridizing strategy described in previous sections, and we introduce a series of hybrids by the computation time, which is the same as that for RAM.

Hybrids to replace F^0 with F^1 are applied iteratively as follows.

Verification of Memory Input To illustrate these hybrids, let time i be a read round for both F^0 and F^1, where both program takes no input from memory and outputs a read command. Let $\widehat{F}^i$ be a hybrid program that runs F^1 if $t < i$ with the memory digest value $w_{\textsf{mem}}^{i-1}$ hardwired. Because there is no memory input, those hybrids from program $\widehat{F}^i$ to $\widehat{F}^{i+1}$ is identical to the combine tree described in Appendix B.3.2, which replaces F^0 by F^1 at $t = i$.

Because time i is a read step and does not change memory digest value, $\widehat{F}^{i+1}$ also hardwires $w_{\textsf{mem}}^{i-1}$. The next round $i+1$ must be a write round which has an input read from memory and outputs a write command. At time $i+1$, $\widehat{F}^{i+1}$ verifies memory inputs just as that of RAM programs, so the security proof is directly identical to that of the RAM program (Appendix B.1) except we need a series of hybrids for each CPU agent $\mathtt{A} \in [m]$. Given the fact that the digest value is correct at time $i+1$, the hybrids are to setup read enforcement for the memory input at time $i+1$ and to argue that read value is information-theoretically correct if it is verified by the accumulator. At this step, we can safely replace the PRAM program from F^0 to F^1 by the security of $\mathsf{i}\mathcal{O}$, and then we have the next hybrid program $\widehat{F}^{i+1,\textsf{OUpdate}}$ by the combine tree described in Appendix B.3.2.

Algorithm 56: $F_{\mathsf{branch}}^{0,1,i,j}\text{(left-leaf)}$

```
    Input : $\tilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \tilde{a}^{\mathrm{in}} = (\mathsf{com}^{\mathrm{in}}, \pi^{\mathrm{in}}_{\mathsf{st}}, \pi^{\mathrm{in}}_{\mathsf{com}})$
    Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m_{i,\mathsf{Root}}, m_{i,j}$
 1  Parse root_node as $(t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}}, \sigma^{\mathrm{in}})$;
 2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$ and $r_B = \mathsf{PRF}(K_B, (t, \mathsf{root\_index}))$;
 3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
 4  Let $\alpha$ = '-' and $m^{\mathrm{in}} = (t, \mathsf{root\_index}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}, v^{\mathrm{in}})$;
 5  if $t \leq i$ then
 6      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha$ = 'A';
 7      else output Reject;
 8  else
 9      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha$ = 'A';
10      if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha$ = 'B';
11      if $\alpha$ = '-' then output Reject;
12  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\mathrm{in}}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{st}}) = 0$ then output Reject;
13  if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\mathrm{in}}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\mathrm{in}}), \pi^{\mathrm{in}}_{\mathsf{com}}) = 0$ then output Reject;
14  if $t \leq i$ or $\alpha$ = 'B' then
15      Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^1(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
16  else
17      Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^0(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
18  Compute $v^{\mathrm{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, w^{\mathrm{in}}_{\mathsf{st}}, w^{\mathrm{in}}_{\mathsf{com}}))$;
19  if $\mathsf{st}^{\mathrm{out}}$ = Reject then
20      Output Reject;
21  else
22      Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$ and $r'_B = \mathsf{PRF}(K_B, (t+1, \mathtt{A}))$;
23      Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,rej}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24      Let $m^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}})$;
25      if $t = i$ then
26          if $id_{\mathsf{cpu}} > j$ and $m^{\mathrm{in}} = m_{i,\mathsf{Root}}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
            else if $id_{\mathsf{cpu}} > j$ and $m^{\mathrm{in}} \neq m_{i,\mathsf{Root}}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
            else if $id_{\mathsf{cpu}} = j$ and $m^{\mathrm{out}} = m_{i,j}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
            else if $id_{\mathsf{cpu}} = j$ and $m^{\mathrm{out}} \neq m_{i,j}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
27          else $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
28      else
29          Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$
30      Let $\mathsf{node}^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}}, \sigma^{\mathrm{out}})$;
31      Output $\tilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, id_{\mathsf{cpu}}, \perp), \tilde{a}^{\mathrm{out}} = \mathsf{node}^{\mathrm{out}}$;
```

Algorithm 57: $F_{\mathsf{combine}}^{0,1,i,j(\text{left-leaf})}$

```
Input : $\widetilde{\mathsf{st}}^{\rm in} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\rm in} = (\mathsf{node}_1, \mathsf{node}_2)$
Data  : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, (\mathbf{C}_{i,j}, \mathbf{M}_{i,j})$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$ and $r_{B,\zeta} = \mathsf{PRF}(K_B, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\mathrm{sk}_{B,\zeta}, \mathrm{vk}_{B,\zeta}, \mathrm{vk}_{B,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta$ = '-' and $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $t \leq i$ or ($t = i + 1$ and $\mathsf{parent\_index} \leq j$) then
12         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
13         else output Reject;
14     else
15         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
16         if $\alpha_\zeta \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'B';
17         if $\alpha_\zeta$ = '-' then output Reject;
18 if $\alpha_1$ = 'A' and $\alpha_2$ = 'A' then set $\alpha$ = 'A';
19 else set $\alpha$ = 'B';
20 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2})$;
21 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2})$;
22 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
23 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$ and $r'_B = \mathsf{PRF}(K_B, (t, \mathsf{parent\_index}))$;
24 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
25 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
26 if $t \leq i$ then
27     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
28 if $t = i + 1$ then
29     if $\mathsf{parent\_index} = \mathsf{index}'$ and $m' = m_{\mathsf{index}'}$ for $\mathsf{index}' \in \mathbf{C}_{i,j}$ and $m_{\mathsf{index}'} \in \mathbf{M}_{i,j}$ then
30         compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
31     else if $\mathsf{parent\_index} = \mathsf{index}'$ and $m' \neq m_{\mathsf{index}'}$ for $\mathsf{index}' \in \mathbf{C}_{i,j}$ and $m_{\mathsf{index}'} \in \mathbf{M}_{i,j}$ then
32         compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m')$;
33     else compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
34 else
35     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
36 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
37 if $\mathsf{parent\_index} = \epsilon$ then
38     Output $\widetilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\rm out} = \perp$;
39 else
40     Output $\widetilde{\mathsf{st}}^{\rm out} = (\mathsf{st}^{\rm in}, id_{\mathsf{cpu}}, \perp), \widetilde{a}^{\rm out} = \mathsf{parent\_node}$;
```

Algorithm 58: $F_{\mathsf{branch}}^{0,1,i,j(\text{right-leaf})}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\mathrm{in}} = (\mathsf{com}^{\mathrm{in}}, \pi_{\mathsf{st}}^{\mathrm{in}}, \pi_{\mathsf{com}}^{\mathrm{in}})$
Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m_{i,\mathsf{Root}}, m_{i,j-1}, m_{i,j}$
1  Parse root_node as $(t, \mathsf{root\_index}, w_{\mathsf{st}}^{\mathrm{in}}, w_{\mathsf{com}}^{\mathrm{in}}, v^{\mathrm{in}}, \sigma^{\mathrm{in}})$;
2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$ and $r_B = \mathsf{PRF}(K_B, (t, \mathsf{root\_index}))$;
3  Compute $(\mathrm{sk}_A, \mathrm{vk}_A, \mathrm{vk}_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $(\mathrm{sk}_B, \mathrm{vk}_B, \mathrm{vk}_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Let $\alpha =$ '-' and $m^{\mathrm{in}} = (t, \mathsf{root\_index}, w_{\mathsf{st}}^{\mathrm{in}}, w_{\mathsf{com}}^{\mathrm{in}}, v^{\mathrm{in}})$;
5  if $t \le i$ then
6      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A';
7      else output Reject;
8  else
9      if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'A';
10     if $\alpha \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m^{\mathrm{in}}, \sigma^{\mathrm{in}}) = 1$ then set $\alpha =$ 'B';
11     if $\alpha =$ '-' then output Reject;
12 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st}}^{\mathrm{in}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}), \pi_{\mathsf{st}}^{\mathrm{in}}) = 0$ then output Reject;
13 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com}}^{\mathrm{in}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\mathrm{in}}), \pi_{\mathsf{com}}^{\mathrm{in}}) = 0$ then output Reject;
14 if $t \le i$ or $\alpha =$ 'B' then
15     Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^1(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
16 else
17     Compute $(\mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}) \leftarrow F^0(id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}})$;
18 Compute $v^{\mathrm{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\mathrm{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{in}}, \mathsf{com}^{\mathrm{in}}, w_{\mathsf{st}}^{\mathrm{in}}, w_{\mathsf{com}}^{\mathrm{in}}))$;
19 if $\mathsf{st}^{\mathrm{out}} =$ Reject then
20     Output Reject;
21 else
22     Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$ and $r'_B = \mathsf{PRF}(K_B, (t+1, \mathtt{A}))$;
23     Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24     Let $m^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}})$;
25     if $t = i$ then
26         if $id_{\mathsf{cpu}} > j$ and $m^{\mathrm{in}} = m_{i,\mathsf{Root}}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
           else if $id_{\mathsf{cpu}} > j$ and $m^{\mathrm{in}} \neq m_{i,\mathsf{Root}}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
           else if $id_{\mathsf{cpu}} = j-1$ and $m^{\mathrm{out}} = m_{i,j-1}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
           else if $id_{\mathsf{cpu}} = j-1$ and $m^{\mathrm{out}} \neq m_{i,j-1}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
           else if $id_{\mathsf{cpu}} = j$ and $m^{\mathrm{out}} = m_{i,j}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m^{\mathrm{out}})$;
           else if $id_{\mathsf{cpu}} = j$ and $m^{\mathrm{out}} \neq m_{i,j}$ then $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m^{\mathrm{out}})$;
           else $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$;
27     else
28         Compute $\sigma^{\mathrm{out}} = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m^{\mathrm{out}})$
29     Let $\mathsf{node}^{\mathrm{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\mathrm{out}}, \mathsf{com}^{\mathrm{out}}, v^{\mathrm{out}}, \sigma^{\mathrm{out}})$;
30     Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, id_{\mathsf{cpu}}, \perp)$, $\widetilde{a}^{\mathrm{out}} = \mathsf{node}^{\mathrm{out}}$;
```

Algorithm 59: $F^{0,1,i,j\text{(internal)}}_{\text{branch}}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, id_{\mathsf{cpu}}, \mathsf{root\_node}), \widetilde{a}^{\text{in}} = (\mathsf{com}^{\text{in}}, \pi^{\text{in}}_{\mathsf{st}}, \pi^{\text{in}}_{\mathsf{com}})$
Data  : $\mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m_{i,\mathsf{Root}}$
1  Parse $\mathsf{root\_node}$ as $(t, \mathsf{root\_index}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}, v^{\text{in}}, \sigma^{\text{in}})$;
2  Let $r_A = \mathsf{PRF}(K_A, (t, \mathsf{root\_index}))$ and $r_B = \mathsf{PRF}(K_B, (t, \mathsf{root\_index}))$;
3  Compute $(\text{sk}_A, \text{vk}_A, \text{vk}_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_A)$ and $(\text{sk}_B, \text{vk}_B, \text{vk}_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r_B)$;
4  Let $\alpha =$ '-' and $m^{\text{in}} = (t, \mathsf{root\_index}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}, v^{\text{in}})$;
5  if $t \le i$ then
6      if $\mathsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A';
7      else output Reject;
8  else
9      if $\mathsf{Spl.Verify}(\text{vk}_A, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'A';
10     if $\alpha \ne$ 'A' and $\mathsf{Spl.Verify}(\text{vk}_B, m^{\text{in}}, \sigma^{\text{in}}) = 1$ then set $\alpha =$ 'B';
11     if $\alpha =$ '-' then output Reject;
12 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,st}}, w^{\text{in}}_{\mathsf{st}}, (id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}), \pi^{\text{in}}_{\mathsf{st}}) = 0$ then output Reject;
13 if $\mathsf{Acc.VerifyRead}(\mathsf{pp}_{\mathsf{Acc,com}}, w^{\text{in}}_{\mathsf{com}}, (\mathsf{src}(t, id_{\mathsf{cpu}}), \mathsf{com}^{\text{in}}), \pi^{\text{in}}_{\mathsf{com}}) = 0$ then output Reject;
14 if $t \le i$ or $\alpha =$ 'B' then
15     Compute $(\mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}) \leftarrow F^1(id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}, \mathsf{com}^{\text{in}})$;
16 else
17     Compute $(\mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}) \leftarrow F^0(id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}, \mathsf{com}^{\text{in}})$;
18 Compute $v^{\text{out}} = \mathsf{TItr.Iterate}(\mathsf{pp}_{\mathsf{Itr}}, v^{\text{in}}, (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\text{in}}, \mathsf{com}^{\text{in}}, w^{\text{in}}_{\mathsf{st}}, w^{\text{in}}_{\mathsf{com}}))$;
19 if $\mathsf{st}^{\text{out}} =$ Reject then
20     Output Reject;
21 else
22     Let $r'_A = \mathsf{PRF}(K_A, (t+1, id_{\mathsf{cpu}}))$ and $r'_B = \mathsf{PRF}(K_B, (t+1, \mathtt{A}))$;
23     Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,\text{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
24     Let $m^{\text{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}, v^{\text{out}})$;
25     if $t = i$ then
26         if $\underline{id_{\mathsf{cpu}} > \mathsf{max\text{-}cpu}(j)}$ and $m^{\text{in}} = m_{i,\mathsf{Root}}$ then
27             $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_A, m^{\text{out}})$;
28         else if $\underline{id_{\mathsf{cpu}} > \mathsf{max\text{-}cpu}(j)}$ and $m^{\text{in}} \ne m_{i,\mathsf{Root}}$ then
29             $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_B, m^{\text{out}})$;
30         else $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_\alpha, m^{\text{out}})$;
31     else
32         Compute $\sigma^{\text{out}} = \mathsf{Spl.Sign}(\text{sk}'_\alpha, m^{\text{out}})$;
33     Let $\mathsf{node}^{\text{out}} = (t+1, id_{\mathsf{cpu}}, \mathsf{st}^{\text{out}}, \mathsf{com}^{\text{out}}, v^{\text{out}}, \sigma^{\text{out}})$;
34     Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{out}}, id_{\mathsf{cpu}}, \perp)$, $\widetilde{a}^{\text{out}} = \mathsf{node}^{\text{out}}$;
```

Algorithm 60: $F^{0,1,i,j(\mathsf{internal})}_{\mathsf{combine}}$

```
Input  : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \bot), \widetilde{a}^{\mathrm{in}} = (\mathsf{node}_1, \mathsf{node}_2)$
Data   : $T, \mathsf{pp}_{\mathsf{Acc,st}}, \mathsf{pp}_{\mathsf{Acc,com}}, \mathsf{pp}_{\mathsf{Itr}}, K_A, K_B, m_{i,j}$
1  Parse $\mathsf{node}_\zeta$ as $(t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\mathsf{index}_1$ and $\mathsf{index}_2$ are not siblings then output Reject;
6  Set $\mathsf{parent\_index}$ as the parent of $\mathsf{index}_1$ and $\mathsf{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \mathsf{PRF}(K_A, (t_\zeta, \mathsf{index}_\zeta))$ and $r_{B,\zeta} = \mathsf{PRF}(K_B, (t_\zeta, \mathsf{index}_\zeta))$;
9      Compute $(\mathrm{sk}_{A,\zeta}, \mathrm{vk}_{A,\zeta}, \mathrm{vk}_{A,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\mathrm{sk}_{B,\zeta}, \mathrm{vk}_{B,\zeta}, \mathrm{vk}_{B,\mathrm{rej},\zeta}) = \mathsf{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta =$ '-' and $m_\zeta = (t_\zeta, \mathsf{index}_\zeta, w_{\mathsf{st},\zeta}, w_{\mathsf{com},\zeta}, v_\zeta)$;
11     if $t \leq i$ or ($t = i + 1$ and $\mathsf{parent\_index} \leq j$) then
12         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'A';
13         else output Reject;
14     else
15         if $\mathsf{Spl.Verify}(\mathrm{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'A';
16         if $\alpha_\zeta \neq$ 'A' and $\mathsf{Spl.Verify}(\mathrm{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta =$ 'B';
17         if $\alpha_\zeta =$ '-' then output Reject;
18 if $\alpha_1 =$ 'A' and $\alpha_2 =$ 'A' then set $\alpha =$ 'A';
19 else set $\alpha =$ 'B';
20 Compute $w'_{\mathsf{st}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,st}}, w_{\mathsf{st},1}, w_{\mathsf{st},2})$;
21 Compute $w'_{\mathsf{com}} = \mathsf{Acc.Combine}(\mathsf{pp}_{\mathsf{Acc,com}}, w_{\mathsf{com},1}, w_{\mathsf{com},2})$;
22 Compute $v' = \mathsf{TItr.Iterate2to1}(\mathsf{pp}_{\mathsf{Itr}}, (v_1, v_2), (t, \mathsf{parent\_index}, w_{\mathsf{st},1}, w_{\mathsf{com},1}, w_{\mathsf{st},2}, w_{\mathsf{com},2}))$;
23 Let $r'_A = \mathsf{PRF}(K_A, (t, \mathsf{parent\_index}))$ and $r'_B = \mathsf{PRF}(K_B, (t, \mathsf{parent\_index}))$;
24 Compute $(\mathrm{sk}'_A, \mathrm{vk}'_A, \mathrm{vk}'_{A,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_A)$ and $(\mathrm{sk}'_B, \mathrm{vk}'_B, \mathrm{vk}'_{B,\mathrm{rej}}) = \mathsf{Spl.Setup}(1^\lambda; r'_B)$;
25 Let $m' = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v')$;
26 if $t \leq i$ then Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
27 if $t = i + 1$ then
28     if $\mathsf{parent\_index} = \mathsf{index}'$ for $\mathsf{index}' \in \mathbf{C}_{i,j}$ then
29         if $m' = m_{\mathsf{index}'}$ for $m_{\mathsf{index}'} \in \mathbf{M}_{i,j}$ then compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
           else ($m' \neq m_{\mathsf{index}'}$ for $m_{\mathsf{index}'} \in \mathbf{M}_{i,j}$) compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m')$;
30     else if $\mathsf{parent\_index} = j$ then
31         if $m' = m_{i,j}$ then compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_A, m')$;
32         else compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_B, m')$;
33     else
34         Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
35 else
36     Compute $\sigma' = \mathsf{Spl.Sign}(\mathrm{sk}'_\alpha, m')$;
37 Let $\mathsf{parent\_node} = (t, \mathsf{parent\_index}, w'_{\mathsf{st}}, w'_{\mathsf{com}}, v', \sigma')$;
38 if $\mathsf{parent\_index} = \epsilon$ then Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \mathsf{parent\_node}), \widetilde{a}^{\mathrm{out}} = \bot$ ;
39 else Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{in}}, id_{\mathsf{cpu}}, \bot), \widetilde{a}^{\mathrm{out}} = \mathsf{parent\_node}$ ;
```

Algorithm 61: $F^{0,1,i,j(\text{intermediate})}_{\text{combine}}$

```
   Input : $\widetilde{\text{st}}^{\text{in}} = (\text{st}^{\text{in}}, id_{\text{cpu}}, \perp), \widetilde{a}^{\text{in}} = (\text{node}_1, \text{node}_2)$
   Data  : $T, \text{pp}_{\text{Acc,st}}, \text{pp}_{\text{Acc,com}}, \text{pp}_{\text{Itr}}, K_A, K_B, (m_{i,j,1}, m_{i,j,2})$
1  Parse $\text{node}_\zeta$ as $(t_\zeta, \text{index}_\zeta, w_{\text{st},\zeta}, w_{\text{com},\zeta}, v_\zeta, \sigma_\zeta)$ for $\zeta = 1, 2$;
2  if $t_1 \neq t_2$ then output Reject;
3  else let $t = t_1$;
4  if $t < 1$ then output Reject;
5  if $\text{index}_1$ and $\text{index}_2$ are not siblings then output Reject;
6  Set parent_index as the parent of $\text{index}_1$ and $\text{index}_2$;
7  for $\zeta = 1, 2$ do
8      Let $r_{A,\zeta} = \text{PRF}(K_A, (t_\zeta, \text{index}_\zeta))$ and $r_{B,\zeta} = \text{PRF}(K_B, (t_\zeta, \text{index}_\zeta))$;
9      Compute $(\text{sk}_{A,\zeta}, \text{vk}_{A,\zeta}, \text{vk}_{A,rej,\zeta}) = \text{Spl.Setup}(1^\lambda; r_{A,\zeta})$ and
       $(\text{sk}_{B,\zeta}, \text{vk}_{B,\zeta}, \text{vk}_{B,\text{rej},\zeta}) = \text{Spl.Setup}(1^\lambda; r_{B,\zeta})$;
10     Let $\alpha_\zeta$ = '-' and $m_\zeta = (t_\zeta, \text{index}_\zeta, w_{\text{st},\zeta}, w_{\text{com},\zeta}, v_\zeta)$;
11     if $t \leq i$ or $(t = i + 1$ and $\text{parent\_index} \leq j)$ then
12         if $\text{Spl.Verify}(\text{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
13         else output Reject;
14     else
15         if $\text{Spl.Verify}(\text{vk}_A, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'A';
16         if $\alpha_\zeta \neq$ 'A' and $\text{Spl.Verify}(\text{vk}_B, m_\zeta, \sigma_\zeta) = 1$ then set $\alpha_\zeta$ = 'B';
17         if $\alpha_\zeta$ = '-' then output Reject;
18 if $\alpha_1$ = 'A' and $\alpha_2$ = 'A' then set $\alpha$ = 'A';
19 else set $\alpha$ = 'B';
20 Compute $w'_{\text{st}} = \text{Acc.Combine}(\text{pp}_{\text{Acc,st}}, w_{\text{st},1}, w_{\text{st},2})$;
21 Compute $w'_{\text{com}} = \text{Acc.Combine}(\text{pp}_{\text{Acc,com}}, w_{\text{com},1}, w_{\text{com},2})$;
22 Compute $v' = \text{TItr.Iterate2to1}(\text{pp}_{\text{Itr}}, (v_1, v_2), (t, \text{parent\_index}, w_{\text{st},1}, w_{\text{com},1}, w_{\text{st},2}, w_{\text{com},2}))$;
23 Let $r'_A = \text{PRF}(K_A, (t, \text{parent\_index}))$ and $r'_B = \text{PRF}(K_B, (t, \text{parent\_index}))$;
24 Compute $(\text{sk}'_A, \text{vk}'_A, \text{vk}'_{A,rej}) = \text{Spl.Setup}(1^\lambda; r'_A)$ and $(\text{sk}'_B, \text{vk}'_B, \text{vk}'_{B,rej}) = \text{Spl.Setup}(1^\lambda; r'_B)$;
25 Let $m' = (t, \text{parent\_index}, w'_{\text{st}}, w'_{\text{com}}, v')$;
26 if $t \leq i$ then
27     Compute $\sigma' = \text{Spl.Sign}(\text{sk}'_A, m')$;
28 if $t = i + 1$ then
29     if $\text{parent\_index} = \text{index}'$ for $\text{index}' \in \mathbf{C}_{i,j}$ then
30         if $m' = m_{\text{index}'}$ for $m_{\text{index}'} \in \mathbf{M}_{i,j}$ then compute $\sigma' = \text{Spl.Sign}(\text{sk}'_A, m')$;
           else if $m' \neq m_{\text{index}'}$ for $m_{\text{index}'} \in \mathbf{M}_{i,j}$ then compute $\sigma' = \text{Spl.Sign}(\text{sk}'_B, m')$;
31     else if $\text{parent\_index} = j$ then
32         if $m_1 = m_{i,j,1}$ and $m_2 = m_{i,j,2}$ then compute $\sigma' = \text{Spl.Sign}(\text{sk}'_A, m')$;
33         else compute $\sigma' = \text{Spl.Sign}(\text{sk}'_B, m')$;
34     else
35         Compute $\sigma' = \text{Spl.Sign}(\text{sk}'_\alpha, m')$;
36 else
37     Compute $\sigma' = \text{Spl.Sign}(\text{sk}'_\alpha, m')$;
38 Let parent_node = $(t, \text{parent\_index}, w'_{\text{st}}, w'_{\text{com}}, v', \sigma')$;
39 if parent_index = $\epsilon$ then Output $\widetilde{\text{st}}^{\text{out}} = (\text{st}^{\text{in}}, id_{\text{cpu}}, \text{parent\_node}), \widetilde{a}^{\text{out}} = \perp$ ;
40 else Output $\widetilde{\text{st}}^{\text{out}} = (\text{st}^{\text{in}}, id_{\text{cpu}}, \perp), \widetilde{a}^{\text{out}} = \text{parent\_node}$ ;
```

We continue with those hybrids of $\mathsf{OUpdate}$.

Hardwiring the Next Accumulator Digest Let us consider the second difference after time $i+1$. It is necessary to show the digest computed by $\mathsf{OUpdate}$ is correct, and we need to show the result of $\mathsf{OUpdate}$ can be replaced by a hardwired correct memory digest w^{i+1}_{mem} at the next hybrid program $\widehat{F}^{i+2}$.

Firstly, an observation is that $\mathsf{OUpdate}$ is exactly carried through an oblivious and memoryless PRAM computation, where each CPU agent $\mathtt{A}$ has old digest w^{i-1}_{mem}, write location $\mathsf{loc}_{\mathtt{A}}$, old proof $\pi_{\mathtt{A}}$ to the location, and a bit $b_{\mathtt{A}}$ to write. Therefore, we can apply those hybrids in Section 6.5 and replace $\mathsf{OUpdate}$ with $\mathsf{OUpdate}^{i+1}$ that always outputs the correct new memory digest w^{i+1}_{mem} at time $i+1$. In particular, we design hybrids through F_{branch} and F_{combine}, use the hybrid steps in memoryless PRAM, and then we can use (indistinguishably) $\widehat{F}^{i+1,\mathsf{OUpdate}^{i+1}}$ that hardwires the correct digest value as the result. Finally, we design hybrids from $\widehat{F}^{i+1,\mathsf{OUpdate}^{i+1}}$ to $\widehat{F}^{i+2}$, where the difference is only at the last round of $\mathsf{OUpdate}^{i+1}$. The proof is similar to Lemma B.9, which is to move hardwired digest from the output of $\mathsf{OUpdate}^{i+1}$ to input of $\widehat{F}^{i+2}$.

Now we have a hybridized program $\widehat{F}^{i+2}$ that always has a correct digest value at a read round, and the hybrid can be carried iteratively to replace all F^0 with F^1.

B.5 Proof of Theorem 7.1 (Security for $\mathcal{RE}$-RAM)

In this subsection, we provide the security proof for our $\mathcal{RE}$ scheme. Following the security definition of randomized encoding scheme, in order to prove that our construction achieves the hiding property, we first present in Appendix B.5.1 a simulator which generates a simulated encoding, then we outline in Appendix B.5.2 the main hybrids to prove that the simulated encoding is indistinguishable from the encoding generated in our $\mathcal{RE}$ scheme. As highlighted in technical overview in Section 3, our proof here is highly non-trivial, and we use "backward in time" hybrid argument of KLW [KLW15] in Appendix B.5.3, and introduce "puncturing ORAM" technique in Appendix B.5.4, and more fine-grained "partial puncturing" technique in Appendix B.5.5. In order to present our proof in a better way, we formulate several technical lemmas. In the proof of each technical lemma, we first give the high-level intuitions, and then present the proof details.

B.5.1 Real and Ideal Experiments

Recall that in our construction, the generated encoding is of the form $\mathsf{Ci}\mathcal{O}(P_{\mathsf{hide}}, x_{\mathsf{hide}})$, where P_{hide} is a compiled version of P, and x_{hide} is an encrypted version of x. To prove the security, i.e., hiding, of our $\mathcal{RE}$, we need to obtain, via a sequence of hybrids, a simulated encoding $\mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$ where all encryptions generated by P_{Sim} as well as in x_{Sim} are replaced by encryption of a special `dummy` symbol. More precisely, P_{Sim} simulates the access pattern by applying the known public strategy. At each time step $t < t^*$, it simply ignores the input and outputs encryptions of `dummy` (for both CPU state and memory content), and outputs y at time step $t = t^*$. More concretely, we consider two security experiments, **Real** and **Ideal**:

- in **Real**, the adversary $\mathcal{A}$ is given the encoding ENC which is generated as in the $\mathcal{RE}$ scheme, i.e., $\text{ENC} \leftarrow \mathcal{RE}.\mathsf{Encode}(P, x, 1^\lambda)$.
- in **Ideal**, the adversary $\mathcal{A}$ is given the emulated encoding ENC_{sim} which is generated by a simulator $\mathcal{S}$, i.e., $\text{ENC}_{sim} \leftarrow \mathcal{S}(1^{|P|}, 1^{|x|}, t^*, y, 1^\lambda, T, S)$.

To complete the proof, we now construct such simulator $\mathcal{S}$ to generate a simulated encoding, then we show that a computationally bounded $\mathcal{A}$ cannot distinguish ENC from ENC_{sim} in the next subsection.

Encoding-Simulation algorithm $\text{ENC}_{sim} \leftarrow \mathcal{S}(1^{|P|}, 1^{|x|}, t^*, y, 1^\lambda, T, S)$: The simulator, i.e., the encoding-simulation algorithm takes the following steps to generate the encoding ENC_{sim}.

- The encoding-simulation algorithm first stores dummy information with $|x|$-bits in mem^0_{sim}, and sets $\mathsf{st}^0 := \mathtt{Init}$, and then transforms $(\mathsf{mem}^0_{sim}, \mathsf{st}^0)$ into $(\mathsf{mem}^0_{o,sim}, \mathsf{st}^0_o)$ using $\text{OA}\textsc{ccess}\{K_N\}$ as in the construction. Then it chooses puncturable PRF key $K_{\mathsf{Sim}} \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$, and constructs an access pattern simulation program SIMOACCESS$\{K_{\mathsf{Sim}}\}$ (see Algorithm 62), and further defines $F_{o,sim}$ based on SIMOACCESS. Now the encoding-simulation algorithm defines

$$\Pi_{o,sim} = ((\mathsf{mem}^0_{o,sim}, \mathsf{st}^0_o),\ F_{o,sim}).$$

- The encoding-simulation algorithm transforms $\Pi_{o,sim}$ into

$$\Pi_{e,sim} = ((\mathsf{mem}^0_{e,sim}, \mathsf{st}^0_e),\ F_{e,sim}).$$

  Here the encoding-simulation algorithm chooses puncturable PRF key $K_E \leftarrow \mathsf{PPRF.Setup}(1^\lambda)$, and then compute $\mathfrak{t}^* = \lceil t^*/q_o \rceil$. Then, based on $F_{o,sim}$, $\mathfrak{t}^*$ as well as the corresponding output value y, it generates the next-step program $F_{e,sim}$ as in Algorithm 63.

  The simulation algorithm encrypts $\mathsf{mem}^0_{o,sim}$ into $\mathsf{mem}^0_{e,sim}$ as in the construction. In addition, the encoding of st^0_e is identical to that in the construction.
- Finally, the encoding-simulation algorithm computes $\textsc{enc}_{sim} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(1^\lambda, \Pi_{e,sim})$ and outputs $\textsc{enc}_{sim}$.

Algorithm 62: SIMOACCESS$\{K_{\mathsf{Sim}}\}$: the recursive program simulates the access pattern of OACCESS$\{K_N\}$

```
Input  : $\mathfrak{t}, d$
Output : No return value
Data   : $K_{\mathsf{Sim}}, \alpha, MaxDepth$ (Memory size $S = \alpha^{MaxDepth}$)
1  if $d \geq MaxDepth$ then
2      return;
3  SIMOACCESS$(\mathfrak{t}, d+1)$ ;                                               // Fetch
4  Pick leaf $pos$ at recursion level $d$ based on PRF$(K_{\mathsf{Sim}}, (\mathfrak{t}, d, \texttt{FetchR}))$;
5  $\mathbf{I}_{\mathsf{fetch}} \leftarrow$ PATH$(d, pos)$;
6  $\mathbf{B}_{\mathsf{fetch}} \leftarrow$ READ$(\mathbf{I}_{\mathsf{fetch}})$;
7  WRITE$(\mathbf{I}_{\mathsf{fetch}}, \mathbf{dummy})$;
8  Pick leaf $pos''$ at recursion level $d$ based on PRF$(K_{\mathsf{Sim}}, (\mathfrak{t}, d, \texttt{FlushR}))$ ;    // Flush
9  $\mathbf{I}_{\mathsf{flush}} \leftarrow$ PATH$(d, pos'')$;
10 $\mathbf{B}_{\mathsf{flush}} \leftarrow$ READ$(\mathbf{I}_{\mathsf{flush}})$;
11 WRITE$(\mathbf{I}_{\mathsf{flush}}, \mathbf{dummy})$;
12 return;
```

B.5.2 Proof Outline: From $\mathbf{Hyb}_0$ to $\mathbf{Hyb}_3$

We here provide a roadmap for proving the security of our $\mathcal{RE}$ construction, and then outline the main hybrids.

Proof. Let $\mathbf{Hyb}_0$ be the real security game **Real**, and $\mathbf{Hyb}_3$ be the ideal security game **Ideal**. We will show via multiple layers of hybrids that $\mathbf{Hyb}_0$ is computationally indistinguishable from $\mathbf{Hyb}_3$. In each hybrid, the simulator generates the encoding ENC as in the construction, except that different next-step programs are used. The overview of the intermediate hybrids is shown as follows:

- $\mathbf{Real} = \mathbf{Hyb}_0 \approx \mathbf{Hyb}_1 \approx \mathbf{Hyb}_{2,\mathfrak{t}^*} \approx \ldots \approx \mathbf{Hyb}_{2,0} = \mathbf{Hyb}_3 = \mathbf{Ideal}$
- $\mathbf{Hyb}_{2,i} = \mathbf{Hyb}_{2,i,0,0} \approx \ldots \approx \mathbf{Hyb}_{2,i,0,d_{max}} \approx \ldots \approx \mathbf{Hyb}_{2,i,0',0} = \mathbf{Hyb}_{2,i-1}$
- $\mathbf{Hyb}_{2,i,0,j} \approx \mathbf{Hyb}_{2,i,0,j,1} \approx \mathbf{Hyb}_{2,i,0,j,2} \approx \mathbf{Hyb}_{2,i,0,j,3} \approx \mathbf{Hyb}_{2,i,0,j+1}$
- $\mathbf{Hyb}_{2,i,0,j,1} \approx \mathbf{Hyb}_{2,i,0,j,1,i-1} \approx \mathbf{Hyb}_{2,i,0,j,1',i-1} \approx \ldots \approx \mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos}} \approx \mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}} \approx \mathbf{Hyb}_{2,i,j,2}$
- $\mathbf{Hyb}_{2,i,0,j,1,z} \approx \mathbf{Hyb}_{2,i,0,j,1',z} \approx \mathbf{Hyb}_{2,i,0,j,1,z-1}$

In the first and outermost layer, we define the hybrids $\mathbf{Hyb}_1$ and $\mathbf{Hyb}_{2,i}$ for $0 \leq i \leq \mathfrak{t}^*$.

Algorithm 63: $F_{e,sim}$

```
Input  : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,   $\tilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data   : $T, K_E, \mathfrak{t}^*, y, K_N, K_{\mathsf{Sim}}$
1  Compute $\mathfrak{t} = \lceil t/q_o \rceil$;
2  if $\mathfrak{t} > \mathfrak{t}^*$ then output $\mathtt{Reject}$;
3  if $\mathfrak{t} = \mathfrak{t}^*$ then
4      Set $\widetilde{\mathsf{st}}^{\text{out}} = (\mathtt{halt}, y)$, $\tilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = \perp$;
5      Output $(\widetilde{\mathsf{st}}^{\text{out}}, \tilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}})$;
6  Compute $(\mathbf{I}^*, \mathbf{B}^*) \leftarrow F_{o,sim}(\mathfrak{t})$
7  Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
8  Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^*)))$;
9  Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
10 Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
11 Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^*; \mathbf{r}_2^{\text{out}})$;
12 Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
13 Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathtt{dummy}; r_4^t)$;
14 Set $\mathbf{I}^{\text{out}} = \mathbf{I}^*$;
15 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,   $\tilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

Hyb$_0$ This hybrid is the real security game **Real**.

Hyb$_1$ In this hybrid, next-step program F_e^1 as defined in Algorithm 64, is used. This program is similar to the next-step program F_e in **Real**, except that, at time $\mathfrak{t} = \mathfrak{t}^*$, it outputs the signed correct computation result $y := P(x)$, which is hardwired into the program. At $\mathfrak{t} > \mathfrak{t}^*$, the program outputs Reject.

Hyb$_{2,i}$ In this hybrid, F_e^1 is replaced by $F_e^{2,i}$ defined in Algorithm 65. This program is similar to F_e^1, except that its access pattern at time $\mathfrak{t}$, where $i \le \mathfrak{t} \le \mathfrak{t}^*$ is replaced by a simulated access pattern provided by the SimOAccess defined in Algorithm 62, and the output state is replaced by an encryption of a special symbol dummy.

Hyb$_3$ This hybrid is the ideal security game **Ideal**. In this hybrid, $F_{e,sim}$, defined in Algorithm 63, is used. This program is identical to $F_e^{2,0}$, in which the access pattern in all time steps $\mathfrak{t}$ where $\mathfrak{t} \le \mathfrak{t}^*$ are simulated. The initial memory mem^0 is written with dummy (rather than x) during the encoding process, and mem^0 is the only difference between **Hyb$_{2,0}$** and **Hyb$_3$**.

Analysis In the remaining of this subsection, we complete the proof of the theorem via several lemmas.

From Hyb$_0$ to Hyb$_1$: The only difference between F_e and F_e^1 is that, in F_e^1, the output in time $\mathfrak{t} = \mathfrak{t}^*$ is hardwired and all computations are rejected after halting time $\mathfrak{t}^*$. Therefore, by Theorem 5.2, since Π_e and Π_e^1 have the same computation trace, their encodings are computationally indistinguishable. We remark that rejecting all computations after $\mathfrak{t}^*$ is very useful when arguing $\mathcal{PKE}$ security in following hybrids because it guarantees the private key is never used after time $\mathfrak{t}^*$.

From $\mathbf{Hyb}_1$ to $\mathbf{Hyb}_{2,\mathsf{t}^*}$: The only difference between F_e^1 and F_e^{2,t^*} is that, in F_e^{2,t^*}, the access pattern in time $\mathsf{t} = \mathsf{t}^*$ is simulated. However, since the program terminates at $t = t^*$ (that means, $\mathsf{t} = \mathsf{t}^*$) by outputting the hardwired computation result, the modified part will never be executed. Therefore, by Theorem 5.2, since Π_e^1 and Π_e^{2,t^*} have the same computation trace, their encodings are computationally indistinguishable.

From $\mathbf{Hyb}_{2,i}$ to $\mathbf{Hyb}_{2,i-1}$: This is the most complicated part, which we will defer the discussion to Lemma B.52.

From $\mathbf{Hyb}_{2,0}$ to $\mathbf{Hyb}_3$: The only difference between the two hybrids is that the initial memory mem^0 in $\mathbf{Hyb}_3$ is encryption of `dummy`. Note that the initial memory is never decrypted in $F_{e,sim}$, and we argue its indistinguishability by standard puncturing and $\mathcal{PKE}$ properties. For each non-empty bit b_i in mem^0, replace its corresponding bit $b'_i \in \mathsf{mem}_o^0$ with `dummy` by following hybrids:

- Puncture PRF key K_E at $(0, h_i)$ in $F_{e,sim}$, where h_i is the "height" given by function $h(\cdot)$ to encrypt the bucket B storing b'_i. This does not change the computation trace and is computationally indistinguishable.
- Encrypt B that contains bit b'_i in mem_o^0 with a truly random $\mathcal{PKE}$ public key, which is computationally indistinguishable by the selective security of PPRF.
- Encrypt B that contains `dummy` instead of bit b'_i. The indistinguishability is guaranteed by the IND-CPA security of $\mathcal{PKE}$.

Therefore, $\mathbf{Hyb}_{2,0}$ and $\mathbf{Hyb}_3$ are computationally indistinguishable.

From above, $\mathbf{Hyb}_0$ and $\mathbf{Hyb}_3$ are computationally indistinguishable as required.

□

Algorithm 64: F_e^1

```
   Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\boxed{\mathsf{st}}^{\mathrm{in}}, t)$,   $\widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, (\boxed{\mathbf{B}}^{\mathrm{in}}, \mathbf{lw}^{\mathrm{in}}))$
   Data  : $T, K_E, \underline{\mathfrak{t}^*, y}, K_N$
1  Compute $\mathfrak{t} = \lceil t/q_o \rceil$;
2  $\underline{\textbf{if } \mathfrak{t} > \mathfrak{t}^* \textbf{ then } \text{output } \mathtt{Reject}}$;
3  if $\underline{\mathfrak{t} = \mathfrak{t}^*}$ then
4      $\underline{\text{Set } \widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathtt{halt}, y), \widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = \perp}$;
5      $\underline{\text{Output } (\widetilde{\mathsf{st}}^{\mathrm{out}}, \widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}})}$;
6  Compute $(\mathbf{r}_1^{\mathrm{in}}, \mathbf{r}_2^{\mathrm{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{in}}, h(\mathbf{I}^{\mathrm{in}})))$;
7  Compute $(\mathbf{pk}^{\mathrm{in}}, \mathbf{sk}^{\mathrm{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{in}})$;
8  Compute $\mathbf{B}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\mathrm{in}}, \boxed{\mathbf{B}}^{\mathrm{in}})$;
9  Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
10 Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
11 Compute $\mathsf{st}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\mathrm{in}})$;

12 Compute $(\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) = F_o(\mathfrak{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;

13 Set $\mathbf{lw}^{\mathrm{out}} = (t, \ldots, t)$;
14 Compute $(\mathbf{r}_1^{\mathrm{out}}, \mathbf{r}_2^{\mathrm{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{out}}, h(\mathbf{I}^{\mathrm{out}})))$;
15 Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{out}})$;
16 Compute $\boxed{\mathbf{B}}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\mathrm{out}}; \mathbf{r}_2^{\mathrm{out}})$;
17 Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
18 Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
19 Compute $\boxed{\mathsf{st}}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\mathrm{out}}; r_4^t)$;
20 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\boxed{\mathsf{st}}^{\mathrm{out}}, t+1)$,   $\widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{out}}, (\boxed{\mathbf{B}}^{\mathrm{out}}, \mathbf{lw}^{\mathrm{out}}))$ ;
```

Algorithm 65: $F_e^{2,i}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, K_N, \underline{K_{\mathsf{Sim}}}, i$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      $\ldots$
5  if $\underline{i \leq \mathsf{t} < \mathsf{t}^*}$ then
6      Compute $(\mathbf{I}^*, \mathbf{B}^*) \leftarrow F_{o,sim}(\mathsf{t})$
7      Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
8      Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^*)))$;
9      Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
10     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
11     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^*; \mathbf{r}_2^{\text{out}})$;
12     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
13     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathtt{dummy}; r_4^t)$;
14     Set $\mathbf{I}^{\text{out}} = \mathbf{I}^*$;
15 else
16     Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
17     Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
18     Compute $\mathbf{B}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}})$;
19     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
20     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
21     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;

22     Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;

23     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
24     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
25     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
26     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
27     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
28     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
29     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
30 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,  $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

B.5.3 Backward Erasing: From $\mathbf{Hyb}_{2,i}$ to $\mathbf{Hyb}_{2,i-1}$

We start to elaborate the outline provided in the previous subsection. We prove the security by a sequence of hybrids that "erases" the computation *backward in time*, which leads to a simulated encoding $\texttt{ENC}_{sim} = \mathsf{Ci}\mathcal{O}(P_{\mathsf{Sim}}, x_{\mathsf{Sim}})$ where all encryptions generated by P_{Sim} as well as in x_{Sim} are replaced by encryption of a special $\texttt{dummy}$ symbol. More precisely, P_{Sim} simulates the access pattern using the public access function ap, and at each time step $\mathsf{t} < \mathsf{t}^*$, simply ignores the input and outputs encryptions of $\texttt{dummy}$ (for both CPU state and memory content), and outputs y at time step $\mathsf{t} = \mathsf{t}^*$.

By erasing the computation backward in time, we consider intermediate hybrids $\mathbf{Hyb}_{2,i}$ where the first i time steps of computation are real, and those of the remaining time step are simulated. Namely, $\mathbf{Hyb}_{2,i}$ is a hybrid encoding $\mathsf{Ci}\mathcal{O}(P_{\mathbf{Hyb}_{2,i}}, x_{\mathsf{hide}})$, where $P_{\mathbf{Hyb}_{2,i}}$ acts as P_{hide} for the first i time steps, and acts as P_{Sim} in the remaining time steps. (Note here that $P_{\mathbf{Hyb}_{2,i}}$, P_{hide}, P_{Sim} correspond to next-step programs $F_e^{2,i}$, F_e, $F_{e,sim}$, respectively.)

The main step is to show indistinguishability of $\mathbf{Hyb}_{2,i}$ and $\mathbf{Hyb}_{2,i-1}$, which corresponds to erasing the computation at the i-th time step. Roughly, to argue this, the key observation is that the i-th decryption key is *not* used in the honest evaluation, which allows us to replace the output of the i-th time step by encryption of $\texttt{dummy}$ by a puncturing argument. We can then further remove the computation at the i-th time step readily by $\mathsf{Ci}\mathcal{O}$ security.

In fact, the argument is more involved given the fact that the $\mathsf{CP\text{-}ORAM}$ is used in our construction. Suppose that at time i the program wishes to access a memory block ℓ which is well-defined by the computation. The program must read the position map value $p = pos[\ell]$ at first, and then fetches the block ℓ along the path p in the ORAM tree. However, our $\mathcal{RE}$ construction relies on $\mathsf{CP\text{-}ORAM}$ tree-based structure where the position map is recursively outsourced to d_{max} ORAM trees. Here we divide the sequence of hybrids into two phases. The first one is to simulate the memory accesses from level 0 to d_{max}. The second one is also to simulate the memory accesses as the first phase, but additionally to erase the CPU states from level d_{max} to 0. The two important hybrids corresponding to these two phases are respectively defined as follows.

$\mathbf{Hyb}_{2,i,0,j}$ (the first phase):

- At time $\mathsf{t} = i-1$, $F_e^{2,i,0,j}$ returns real $\mathsf{st}^{\mathrm{out}}$, and outputs real accesses identical to $\mathrm{OA}\textsc{ccess}$ if $d \geq j$.
- At time $\mathsf{t} = i-1$, $F_e^{2,i,0,j}$ returns real $\mathsf{st}^{\mathrm{out}}$, but outputs the simulated accesses if $d < j$.

$\mathbf{Hyb}_{2,i,0',j}$ (the second phase):

- At time $\mathsf{t} = i-1$, $F_e^{2,i,0',j}$ outputs the simulated accesses, and returns real $\mathsf{st}^{\mathrm{out}}$ if $d \leq j$.
- At time $\mathsf{t} = i-1$, $F_e^{2,i,0',j}$ outputs the simulated accesses, but returns $\mathsf{st}^{\mathrm{out}} = \texttt{dummy}$ if $d > j$.

Note that $\mathbf{Hyb}_{2,i}$ is identical to $\mathbf{Hyb}_{2,i,0,0}$, and $\mathbf{Hyb}_{2,i,0',0}$ is also identical to $\mathbf{Hyb}_{2,i-1}$. Clearly, we need to argue the remaining hybrids are indistinguishable, $\mathbf{Hyb}_{2,i,0,0} \approx \ldots \approx \mathbf{Hyb}_{2,i,0,d_{max}} \approx \mathbf{Hyb}_{2,i,0',d_{max}} \approx \ldots \approx \mathbf{Hyb}_{2,i,0',0}$.

Lemma B.52. *Let $\mathcal{PKE}$ be an IND-CPA secure public key encryption scheme, $\mathsf{Ci}\mathcal{O}$ be a computation-trace indistinguishability obfuscation scheme in the RAM model, PRF be a secure puncturable PRF scheme; then the hybrids $\mathbf{Hyb}_{2,i}$ and $\mathbf{Hyb}_{2,i-1}$ are computationally indistinguishable for $1 \leq i \leq \mathsf{t}^*$.*

Proof. For each i, we define two second-layer hybrids $\mathbf{Hyb}_{2,i,0,j}$ and $\mathbf{Hyb}_{2,i,0',j}$ for $0 \leq j \leq d_{max}$, where d_{max} denotes the maximum depth of the ORAM tree.

$\mathbf{Hyb}_{2,i,0,j}$ In this hybrid, the program $F_e^{2,i,0,j}$ is defined in Algorithm 66. At time $\mathsf{t} = i-1$, $F_e^{2,i,0,j}$ uses $\textsc{HybOAccess}^j$ which outputs the simulated accesses if $d < j$, and it outputs those accesses identical to $\mathrm{OA}\textsc{ccess}$ if $d \geq j$ (Algorithm 67). Like previous programs, $F_e^{2,i,0,j}$ uses the $\textsc{HybOAccess}^j$ compiled program named $F_{o,hyb}^j = \mathsf{Compile}(F, \textsc{HybOAccess}^j\{K_N, K_{\mathsf{Sim}}\})$. Note that the values $(\mathsf{loc}, \mathsf{val})$ from the input, $newpos$ and pos returned by $\mathsf{PRF}(K_N, \cdot)$ and $\textsc{HybOAccess}^j(d+1, \cdot)$ are never used if $d < j$.

Hyb$_{2,i,0',j}$ In this hybrid, the program $F_e^{2,i,0',j}$ is defined in Algorithm 68. At time $\mathsf{t} = i - 1$, $F_e^{2,i,0',j}$ replaces st^{out} by `dummy` for all $d > j$.

Analysis In the remaining of this subsection, we will complete the proof of the lemma.

From Hyb$_{2,i}$ to Hyb$_{2,i,0,0}$: These two hybrids are identical.

From Hyb$_{2,i,0,j}$ to Hyb$_{2,i,0,j+1}$: We defer the discussion to Lemma B.53.

From Hyb$_{2,i,0,d_{max}}$ to Hyb$_{2,i,0',d_{max}}$: These two hybrids are identical.

From Hyb$_{2,i,0',j}$ to Hyb$_{2,i,0',j-1}$: This step can be proved via multiple hybrids. For readability, we describe the hybrids without defining them in separate algorithms. The only difference between $F_e^{2,i,0',j}$ and $F_e^{2,i,0',j-1}$ is that, in $F_e^{2,i,0',j-1}$, st_{out} is replaced by `dummy` at time $\mathsf{t} = i - 1$ and depth $d = j$. In the first hybrid, we puncture the input $(i-1, j, \texttt{Init})$ for PRF key K_E and hardwire the pseudorandomness computed from K_E. Since the computation defined by this program has identical computation trace as that in the previous hybrid, indistinguishability is guaranteed by Theorem 5.2. In the next hybrid, we replace the pseudorandomness by a truly random number. Indistinguishability is guaranteed by the security of the puncturable PRF. Then, we hardwire $\boxed{\mathsf{st}}^{\text{out}}$, which is generated by the true randomness in the previous hybrid, into the program and take away the hardwired true randomness. Since the computation defined by this program has identical computation trace as that in the previous hybrid, indistinguishability is guaranteed by Theorem 5.2. Next, we replace hardwired $\boxed{\mathsf{st}}^{\text{out}}$ by an encryption of `dummy`. Indistinguishability is guaranteed by the IND-CPA security of $\mathcal{PKE}$. Finally, we un-puncture the PRF key K_E to obtain the required hybrid. Indistinguishability is again guaranteed by the security of the puncturable PRF.

From Hyb$_{2,i,0',0}$ to Hyb$_{2,i-1}$: These two hybrids are identical.

□

Algorithm 66: $F_e^{2,i,0,j}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,   $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \le \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
9      Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
10     Compute $\mathbf{B}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;
14     if $\underline{\mathsf{t} = i-1}$ then
15         $\underline{\text{Compute } (\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F^j_{o,hyb}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})}$ ;
           // $F^j_{o,hyb} = \text{CP-ORAM.Compile}(F, \text{HYBOACCESS}^j)$
16     else
17         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
18     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
19     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
20     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
21     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
22     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
23     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
24     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
25 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

Algorithm 67: $\textsc{HybOAccess}^j\{K_N, K_{\mathsf{Sim}}\}$

```
Input  : $\mathsf{t}, d, \mathsf{loc}, \mathsf{val}$
Output : $oldval$
Data   : $K_N, K_{\mathsf{Sim}}, \alpha, MaxDepth$ (Memory size $S = \alpha^{MaxDepth}$)
1  if $d \geq MaxDepth$ then
2      return 0;
3  Pick leaf $newpos$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \mathtt{FetchR}))$ ;    // Update position map
4  $pos \leftarrow \textsc{HybOAccess}^j(\mathsf{t}, d+1, \lfloor \mathsf{loc}/\alpha \rfloor, newpos)$;
5  if $(d < j)$ then
6      Pick leaf $pos$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathsf{t}, d, \mathtt{FetchR}))$;
7      $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, pos)$;
8      $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
9      $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{dummy})$;
10     Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathsf{t}, d, \mathtt{FlushR}))$;
11     $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
12     $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
13     $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{dummy})$;
14     return;
15 else // identical to OAccess
16     $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, pos)$ ;    // Fetch
17     $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
18     $(\mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}}, oldval) \leftarrow \textsc{FetchAndUpdate}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, pos, newpos)$;
19     $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}})$;
20     Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \mathtt{FlushR}))$ ;    // Flush
21     $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
22     $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
23     $\mathbf{B}^{\mathrm{out}}_{\mathsf{flush}} \leftarrow \textsc{Flush}(\mathbf{B}_{\mathsf{flush}}, pos'')$;
24     $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{flush}})$;
25     return $oldval$;
```

Algorithm 68: $F_e^{2,i,0',j}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output $\mathtt{Reject}$;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      $\ldots$
5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
6      $\ldots$
7  else
8      Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
9      Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
10     Compute $\mathbf{B}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;
14     if $\mathsf{t} = i - 1$ then
15         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F_{o,hyb}^{d_{max}}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
16         if $(d > j$ and $a = \mathtt{Init})$ then set $\mathsf{st}^{\text{out}} = \mathtt{dummy}$;
17     else
18         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
19     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
20     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
21     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
22     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
23     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
24     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
25     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
26 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,  $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

B.5.4 Punctured ORAM: From $\mathbf{Hyb}_{2,i,0,j}$ to $\mathbf{Hyb}_{2,i,0,j+1}$

Our goal now is to show that $\mathbf{Hyb}_{2,i,0,j}$ and $\mathbf{Hyb}_{2,i,0,j+1}$ are indistinguishable, which amounts to switch time step $(\mathfrak{t}, d) = (i-1, j)$ from real computation to a simulated one. Suppose that at time $(\mathfrak{t}, d) = (i-1, j)$ the ORAM compiled program F_o wishes to access some memory location loc, which points to a cell in *block** (recall that the program is deterministic, so loc is well-defined by the computation), it reads the position map value *pos** and fetches the *block** along the path *pos** in the ORAM tree. We need to simulate the memory access pattern (induced by *pos**) and output data (including memory data and CPU state), where output data are encrypted in ciphertext and the indistinguishable simulation can be constructed by semantic security of $\mathcal{PKE}$. The main challenge here is to switch *pos** to a simulated path, since *pos** is information theoretically determined by the previous computation and *pos** must be revealed directly through memory access.

Punctured ORAM To simulate *pos**, our approach is to move to a hybrid with punctured ORAM program, where *pos** is *information-theoretically erased* so that we can switch *pos** to a simulated path. Towards this goal, let us trace closely the value *pos** in the program execution. First, we observe that *pos** is stored in two places in the ORAM data structure:

1. The block *block** itself, which is stored somewhere in the ORAM tree.
2. The position map stored in another layer of ORAM tree recursively.

Let $\mathfrak{t}_{pos}$ be that time *pos** being created by PPRF, which is also the last access time of *block** before time step $i-1$. At time $\mathfrak{t}_{pos}$, the value *pos** is stored in both *block** in the root node and the recursive position map. Note that $\mathfrak{t}_{pos}$ can be much smaller than $i-1$. To "information-theoretically erase" *pos**, we move to a hybrid program where the value *pos** is *not* generated at time $\mathfrak{t}_{pos}$. Specifically, we define an intermediate hybrid $\mathbf{Hyb}_{2,i,0,j,2}$ in which the program is replaced by a punctured ORAM program such that:

> $\mathbf{Hyb}_{2,i,0,j,2}$: At time $\mathfrak{t} = \mathfrak{t}_{pos}$, do not generate the value *pos**, and instead of putting back the encryption of the fetched *block** to the root of the ORAM tree, an encryption of empty values is put back[22]. Moreover, the position map value *pos** is not updated.

In $\mathbf{Hyb}_{2,i,0,j,2}$, the value *pos** is information-theoretically hidden before time step $i-1$. Since the *block** is not be accessed from time $\mathfrak{t}_{pos}$ to time $i-1$, the modification does not change the computation from time $\mathfrak{t}_{pos}$ to time $i-1$. Now, we can simulate the computation at time step $i-1$ as before, and switch *pos** to a simulated value by the standard PPRF argument. After the path is switched to a simulated one, we obtain $\mathbf{Hyb}_{2,i,0,j+1}$ by *un-puncturing* ORAM program F_o at the point *pos**.

We prove $\mathbf{Hyb}_{2,i,0,j}$ and $\mathbf{Hyb}_{2,i,0,j+1}$ are indistinguishable in Lemma B.53 with the help of puncturable ORAM, and we will later prove that punctured ORAM is computationally indistinguishable from its non-punctured ORAM counterpart in the next sub-section (Lemma B.54).

	$\mathfrak{t} = \mathfrak{t}_{pos}$ and $d = j$	$\mathfrak{t} = (i-1)$ and $d = j$	$\mathfrak{t} = (i-1)$ and $d < j$
$\mathbf{Hyb}_{2,i,0,j}$	Honest	Honest path and values	Sim. path, erased values
$\mathbf{Hyb}_{2,i,0,j,1}$	Honest	Honest path, erased values	Sim. path, erased values
$\mathbf{Hyb}_{2,i,0,j,2}$	Puncture *pos**	Hardwired honest path, erased values	Sim. path, erased values
$\mathbf{Hyb}_{2,i,0,j,3}$	Puncture *pos**	Sim. path, erased values	Sim. path, erased values
$\mathbf{Hyb}_{2,i,0,j+1}$	Honest	Sim. path, erased values	Sim. path, erased values

Table 6: Intuitions of the hybrid series from $\mathbf{Hyb}_{2,i,0,j}$ to $\mathbf{Hyb}_{2,i,0,j+1}$, where we focus on the simulated memory access locations as a path in the ORAM tree; "Sim." stands for simulated

[22]Recall in the ORAM construction, a bucket is a vector of K elements, where each element is either a valid block or a unique empty-slot symbol $\texttt{Empty} = (\perp, \perp, \perp)$. The value being put pack is in fact Empty (rather than *block**), which also yields a consistent ORAM tree.

Lemma B.53. *Let $\mathcal{PKE}$ be an IND-CPA secure public key encryption scheme,* $\mathsf{Ci}\mathcal{O}$ *be a computation-trace indistinguishability obfuscation scheme in the RAM model,* PRF *be a secure puncturable PRF scheme; then the hybrids* $\textbf{\textit{Hyb}}_{2,i,0,j}$ *and* $\textbf{\textit{Hyb}}_{2,i,0,j+1}$ *are computationally indistinguishable for all* $0 \leq i \leq \mathfrak{t}^*$ *and all* $0 \leq j \leq d_{max}$.

Proof. We define three third-layer hybrids $\mathbf{Hyb}_{2,i,0,j,1}$, $\mathbf{Hyb}_{2,i,0,j,2}$ and $\mathbf{Hyb}_{2,i,0,j,3}$.

$\mathbf{Hyb}_{2,i,0,j,1}$ In this hybrid, F is replaced by $F_e^{2,i,0,j,1}$ defined in Algorithm 69. $F_e^{2,i,0,j,1}$ uses $\textsc{HybOAccess}^{j,1}$ that is similar to $\textsc{HybOAccess}^{j}$, except that at $d = j$ all values written are replaced by dummy data. All other computations are carried out honestly and access locations $\mathbf{I}_{\mathsf{fetch}}$ (induced by pos^*) in $\textsc{HybOAccess}^{j,1}$ are identical to those in $\textsc{HybOAccess}^{j}$.

$\mathbf{Hyb}_{2,i,0,j,2}$ In this hybrid, F is replaced by $F_e^{2,i,0,j,2}$ defined in Algorithm 71. $F_e^{2,i,0,j,2}$ invokes different variants of F_o at different $\mathfrak{t}$: $F_{o,sim}$ for $\mathfrak{t} \geq i$, $F_{o,hyb}^{j,2}$ for $\mathfrak{t} = i - 1$, $F_o^{j,\mathsf{punct}}$ for $\mathfrak{t} = \mathfrak{t}_{pos}$, and normal F_o otherwise, where

- $F_{o,hyb}^{j,2}$ is the ORAM compiled program using $\textsc{HybOAccess}^{j,2}$ that is similar to $\textsc{HybOAccess}^{j,1}$ except that it hardwires and uses pos^* (rather than pos that returned from recursive call), where pos^* is the pre-computed value of pos at $(\mathfrak{t} = i - 1 \wedge d = j)$ in an honest evaluation of $\textsc{OAccess}$ (Algorithm 72).
- $F_o^{j,\mathsf{punct}}$ is the punctured ORAM program using $\textsc{HybOAccess}^{j,\mathsf{punct}}$ that is similar to normal $\textsc{OAccess}$, except that it is punctured at $\mathfrak{t}_{pos}$, which writes `dummy` symbol to the position map and removes $block^*$ at the j-th recursion (Algorithm 73)[23].
- $\mathfrak{t}_{pos}$ is the time $\mathfrak{t}$ so that pos^* is generated by $\mathsf{PRF}(K_N, (\mathfrak{t}, d, \mathsf{flag}))$.

$\mathbf{Hyb}_{2,i,0,j,3}$ In this hybrid, F is replaced by $F_e^{2,i,0,j,3}$ defined in Algorithm 74 $F_e^{2,i,0,j,3}$ is similar to $F_e^{2,i,0,j,2}$, except that it uses $\textsc{HybOAccess}^{j+1}$ at $\mathfrak{t} = i - 1$.

Analysis In the remaining of this subsection, we will complete the proof of the lemma.

From $\mathbf{Hyb}_{2,i,0,j}$ to $\mathbf{Hyb}_{2,i,0,j,1}$: Note that in $F_e^{2,i,0,j}$, the entire computation for $i \leq \mathfrak{t} \leq \mathfrak{t}^*$ is simulated, and when $\mathfrak{t} > \mathfrak{t}^*$, $F_e^{2,i,0,j}$ always outputs `Reject`. It will thus never be the case that, at time $\mathfrak{t} > i - 1$, the program decrypts the ciphertext written at $(i - 1, j, \mathsf{flag})$ where $\mathsf{flag} =$ `FetchW` or `FlushW`. We can therefore replace the ciphertexts output in the fetch and flush phase by the encryption of `dummy`, and replace the flush positions by a simulated version.

Formally, indistinguishability is established via the following hybrids:

1. In the first hybrid, we puncture the input $(i - 1, j,$ `FlushR`$)$ for PRF keys K_N and K_{Sim}, and the input $(i - 1, j, \mathsf{flag})$ where $\mathsf{flag} =$ `FetchW` and `FlushW` for PRF keys K_E. We hardwire the pseudorandomness computed from K_N, K_{Sim} and K_E. Since the computation defined by the program has identical computation trace as that in the previous hybrid, indistinguishability is guaranteed by Theorem 5.2.

2. In the next hybrid, we replace the pseudorandomness by truly random numbers. Indistinguishability is guaranteed by the security of the puncturable PRF.

3. Then, we hardwire $\boxed{\mathbf{B}}^{\text{out}}$ and $\mathbf{I}_{\mathsf{flush}}$, which are generated by the true randomness in the previous hybrid, into the program; and take away the hardwired true randomness. Since the computation defined by the program has identical computation trace as that in the previous hybrid, indistinguishability is guaranteed by Theorem 5.2.

[23] $\textsc{RemoveBlock}(\mathbf{B}, \mathsf{loc}, \alpha, pos)$ searches for the tuple of $block = (\lfloor \mathsf{loc}/\alpha \rfloor, pos, data)$ in each $bucket \in \mathbf{B}$. It outputs $\mathbf{B}^-$ with $block$ being removed.

4. Next, we replace the values to be written by $\texttt{dummy}$. Indistinguishability is guaranteed by the IND-CPA security of $\mathcal{PKE}$ because these values are encrypted and $F_e^{2,i,0,j,1}$ would never decrypt the ciphertext with the private key.

5. We replace $\mathbf{I}_{\mathsf{flush}}$ by a simulated version, which is generated from the hardwired true randomness that corresponds to K_{Sim}. Indistinguishability is guaranteed by the selective security of the puncturable PRF.

6. Finally, we un-puncture the PRF keys K_N, K_{Sim}, and K_E to obtain the required hybrid. Indistinguishability is again guaranteed by the security of the puncturable PRF.

From $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,2}$: We defer the discussion to Lemma B.54.

From $\mathbf{Hyb}_{2,i,0,j,2}$ to $\mathbf{Hyb}_{2,i,0,j,3}$: In $F_e^{2,i,0,j,2}$, since our ORAM program has punctured pos^* that generated at time (t_{pos}, j), and pos^* is only used at time $(i-1, j)$, we can use $\textsc{HybOAccess}^{j+1}$ instead of $\textsc{HybOAccess}^{j,2}$ by the selective security of PPRF and Theorem 5.2.

Formally, indistinguishability is established via the following hybrids:

1. In the first hybrid, we use punctured PRF keys $K_N\{(\mathsf{t}_{pos}, j, \texttt{FetchR})\}$ and $K_{\mathsf{Sim}}\{i-1, j, \texttt{FetchR}\}$. We do not use K_{Sim} at $(i-1, j, \texttt{FetchR})$ in this step, and the only value relies on the K_N at $(\mathsf{t}_{pos}, j, \texttt{FetchR})$ is pos^*, which is already hardwired. Since the program has identical computation trace as that in the previous hybrid, indistinguishability is guaranteed by Theorem 5.2.

2. In the next hybrid, we replace hardwired value pos^* (generated from $K_N\{(\mathsf{t}_{pos}, j, \texttt{FetchR})\}$) with pos^{**}, which is the random ORAM position computed from a true randomness. Indistinguishability is guaranteed by the selective security of PPRF.

3. Then, we replace hardwired value pos^{**} with pos^*_{Sim}, which is generated by randomness computed by $\mathsf{PRF}(K_{\mathsf{Sim}}, (i-1, j, \texttt{FetchR}))$. Indistinguishability is guaranteed by the selective security of PPRF.

4. Finally, we un-puncture the PRF keys K_N, K_{Sim} to obtain the required hybrid. Since the program has identical computation trace as that in the previous hybrid, indistinguishability is guaranteed by Theorem 5.2.

From $\mathbf{Hyb}_{2,i,0,j,3}$ to $\mathbf{Hyb}_{2,i,0,j+1}$: The proof is similar to that of Lemma B.54. $\square$

Algorithm 69: $F_e^{2,i,0,j,1}$

```
Input  : $\widetilde{\mathsf{st}}^{\rm in} = (\boxed{\mathsf{st}}^{\rm in}, t)$,   $\widetilde{a}^{\rm in}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\rm in}, (\boxed{\mathbf{B}}^{\rm in}, \mathbf{lw}^{\rm in}))$
Data   : $T, K_E, \mathsf{t}^*, y, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      Compute $(\mathbf{r}_1^{\rm in}, \mathbf{r}_2^{\rm in}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\rm in}, h(\mathbf{I}^{\rm in})))$;
9      Compute $(\mathbf{pk}^{\rm in}, \mathbf{sk}^{\rm in}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\rm in})$;
10     Compute $\mathbf{B}^{\rm in} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\rm in}, \boxed{\mathbf{B}}^{\rm in})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\rm in} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\rm in})$;
14     if $\underline{\mathsf{t} = i-1}$ then
15         $\underline{\text{Compute } (\mathsf{st}^{\rm out}, \mathbf{I}^{\rm out}, \mathbf{B}^{\rm out}) \leftarrow F_{o,hyb}^{j,1}(\mathsf{t}, \mathsf{st}^{\rm in}, \mathbf{I}^{\rm in}, \mathbf{B}^{\rm in})}$ ;
           // $F_{o,hyb}^{j,1} = \text{CP-ORAM.Compile}(F, \textsc{HybOAccess}^{j,1})$
16     else
17         Compute $(\mathsf{st}^{\rm out}, \mathbf{I}^{\rm out}, \mathbf{B}^{\rm out}) = F_o(\mathsf{t}, \mathsf{st}^{\rm in}, \mathbf{I}^{\rm in}, \mathbf{B}^{\rm in})$;
18     Set $\mathbf{lw}^{\rm out} = (t, \ldots, t)$;
19     Compute $(\mathbf{r}_1^{\rm out}, \mathbf{r}_2^{\rm out}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\rm out}, h(\mathbf{I}^{\rm out})))$;
20     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\rm out})$;
21     Compute $\boxed{\mathbf{B}}^{\rm out} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\rm out}; \mathbf{r}_2^{\rm out})$;
22     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
23     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
24     Compute $\boxed{\mathsf{st}}^{\rm out} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\rm out}; r_4^t)$;
25 Output $\widetilde{\mathsf{st}}^{\rm out} = (\boxed{\mathsf{st}}^{\rm out}, t+1)$,   $\widetilde{a}^{\rm out}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\rm out}, (\boxed{\mathbf{B}}^{\rm out}, \mathbf{lw}^{\rm out}))$ ;
```

Algorithm 70: $\textsc{HybOAccess}^{j,1}\{K_N, K_{\mathsf{Sim}}\}$

```
Input  : $\mathfrak{t}, d, \mathsf{loc}, \mathsf{val}$
Output : $oldval$
Data   : $K_N, K_{\mathsf{Sim}}, \alpha, MaxDepth$ (Memory size $S = \alpha^{MaxDepth}$)
1  if $d \geq MaxDepth$ then
2      return 0;
3  Pick leaf $newpos$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathfrak{t}, d, \texttt{FetchR}))$ ;   // Update position map
4  $pos \leftarrow \textsc{HybOAccess}^{j,1}(\mathfrak{t}, d+1, \lfloor \mathsf{loc}/\alpha \rfloor, newpos)$;
5  if $(d < j)$ then
6      Pick leaf $pos$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathfrak{t}, d, \texttt{FetchR}))$;
7      $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, pos)$;
8      $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
9      $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{dummy})$;
10     Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathfrak{t}, d, \texttt{FlushR}))$;
11     $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
12     $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
13     $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{dummy})$;
14     return;
15 else
16     $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, pos)$ ;   // Fetch
17     $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
18     if $(d = j)$ then
19         $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{dummy})$;
20         Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathfrak{t}, d, \texttt{FlushR}))$;
21         $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
22         $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
23         $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{dummy})$;
24         return;
25     else
26         $(\mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}}, oldval) \leftarrow \textsc{FetchAndUpdate}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, pos, newpos)$;
27         $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}})$;
28         Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathfrak{t}, d, \texttt{FlushR}))$ ;   // Flush
29         $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
30         $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
31         $\mathbf{B}^{\mathrm{out}}_{\mathsf{flush}} \leftarrow \textsc{Flush}(\mathbf{B}_{\mathsf{flush}}, pos'')$;
32         $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{flush}})$;
33         return $oldval$;
```

Algorithm 71: $F_e^{2,i,0,j,2}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,  $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
9      Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
10     Compute $\mathbf{B} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;
14     if $\underline{\mathsf{t} = i - 1}$ then
15         $\underline{\text{Compute } (\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F_{o,hyb}^{j,2}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})}$ ;
           // $F_{o,hyb}^{j,2} = \mathsf{CP\text{-}ORAM.Compile}(F, \textsc{HybOAccess}^{j,2})$
16     else if $\underline{\mathsf{t} = \mathsf{t}_{pos}}$ then
17         $\underline{\text{Compute } (\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o^{j,\mathsf{punct}}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})}$ ;
           // $F_o^{j,\mathsf{punct}} = \mathsf{CP\text{-}ORAM.Compile}(F, \textsc{HybOAccess}^{j,\mathsf{punct}})$
18     else
19         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
20     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
21     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
22     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
23     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
24     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
25     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
26     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
27 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,  $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

Algorithm 72: $\textsc{HybOAccess}^{j,2}\{K_N, K_{\mathsf{Sim}}\}$

```
Input  : $\mathsf{t}, d, \mathsf{loc}, \mathsf{val}$
Output : $oldval$
Data   : $K_N, K_{\mathsf{Sim}}, \alpha, \mathit{MaxDepth}$ (Memory size $S = \alpha^{\mathit{MaxDepth}}$), $\mathit{pos}^*$
1  if $d \geq \mathit{MaxDepth}$ then
2      return $0$;
3  Pick leaf $newpos$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \texttt{FetchR}))$ ;      // Update position map
4  $\mathit{pos} \leftarrow \textsc{HybOAccess}^{j,2}(\mathsf{t}, d+1, \lfloor \mathsf{loc}/\alpha \rfloor, newpos, K_{\mathsf{Sim}})$;
5  if $(d < j)$ then
6      Pick leaf $\mathit{pos}$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathsf{t}, d, \texttt{FetchR}))$;
7      $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, \mathit{pos})$;
8      $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
9      $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{dummy})$;
10     Pick leaf $\mathit{pos}''$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathsf{t}, d, \texttt{FlushR}))$;
11     $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, \mathit{pos}'')$;
12     $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
13     $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{dummy})$;
14     return;
15 else
16     if $(d = j)$ then
17         $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, \mathit{pos}^*)$;
18         $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
19         $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{dummy})$;
20         Pick leaf $\mathit{pos}''$ at recursion level $d$ based on $\mathsf{PRF}(K_{\mathsf{Sim}}, (\mathsf{t}, d, \texttt{FlushR}))$;
21         $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, \mathit{pos}'')$;
22         $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
23         $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{dummy})$;
24         return;
25     else
26         $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, \mathit{pos})$ ;      // Fetch
27         $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
28         $(\mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}}, oldval) \leftarrow \textsc{FetchAndUpdate}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, \mathit{pos}, newpos)$;
29         $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}})$;
30         Pick leaf $\mathit{pos}''$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \texttt{FlushR}))$ ;      // Flush
31         $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, \mathit{pos}'')$;
32         $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
33         $\mathbf{B}^{\mathrm{out}}_{\mathsf{flush}} \leftarrow \textsc{Flush}(\mathbf{B}_{\mathsf{flush}}, \mathit{pos}'')$;
34         $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{flush}})$;
35         return $oldval$;
```

Algorithm 73: $\textsc{HybOAccess}^{j,\mathsf{punct}}\{K_N\}$

```
Input  : $\mathsf{t}, d, \mathsf{loc}, \mathsf{val}$
Output : $oldval$
Data   : $K_N, \alpha, MaxDepth$ (Memory size $S = \alpha^{MaxDepth}$), $\mathsf{loc}_{i-1,j}$
1  if $d \geq MaxDepth$ then
2      return 0
3  if $(d = j)$ then
4      $pos \leftarrow \textsc{HybOAccess}^{j,\mathsf{punct}}(d+1, \lfloor \mathsf{loc}/\alpha \rfloor, \mathtt{dummy})$;
5  else
6      Pick leaf $newpos$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \mathtt{FetchR}))$ ;    // Update position map
7      $pos \leftarrow \textsc{HybOAccess}^{j,\mathsf{punct}}(d+1, \lfloor \mathsf{loc}/\alpha \rfloor, newpos)$ ;    // Fetch
8  $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, pos)$;
9  $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
10 if $(d = j)$ then
11     $(\mathbf{B}_{\mathsf{fetch},-}, oldval) \leftarrow \textsc{RemoveBlock}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \alpha, pos)$;
12     $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}_{\mathsf{fetch},-})$;
13 else
14     $(\mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}}, oldval) \leftarrow \textsc{FetchAndUpdate}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, pos, newpos)$;
15     $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}})$;
16 Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \mathtt{FlushR}))$ ;    // Flush
17 $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
18 $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
19 $\mathbf{B}^{\mathrm{out}}_{\mathsf{flush}} \leftarrow \textsc{Flush}(\mathbf{B}_{\mathsf{flush}}, pos'')$;
20 $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{flush}})$;
21 return $oldval$;
```

Algorithm 74: $F_e^{2,i,0,j,3}$

```
Input  : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, t)$,   $\tilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, (\mathbf{B}^{\mathrm{in}}, \mathbf{lw}^{\mathrm{in}}))$
Data   : $T, K_E, \mathsf{t}^*, y, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      Compute $(\mathbf{r}_1^{\mathrm{in}}, \mathbf{r}_2^{\mathrm{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{in}}, h(\mathbf{I}^{\mathrm{in}})))$;
9      Compute $(\mathbf{pk}^{\mathrm{in}}, \mathbf{sk}^{\mathrm{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{in}})$;
10     Compute $\mathbf{B}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \mathsf{st}^{\mathrm{in}})$;
14     if $\underline{\mathsf{t} = i-1}$ then
15         $\underline{\text{Compute } (\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) \leftarrow F_{o,hyb}^{j+1}(\mathsf{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})}$;
16     else if $\mathsf{t} = \mathsf{t}_{pos}$ then
17         Compute $(\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) = F_o^{j,\mathsf{punct}}(\mathsf{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;
18     else
19         Compute $(\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;
20     Set $\mathbf{lw}^{\mathrm{out}} = (t, \ldots, t)$;
21     Compute $(\mathbf{r}_1^{\mathrm{out}}, \mathbf{r}_2^{\mathrm{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{out}}, h(\mathbf{I}^{\mathrm{out}})))$;
22     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{out}})$;
23     Compute $\mathbf{B}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\mathrm{out}}; \mathbf{r}_2^{\mathrm{out}})$;
24     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
25     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
26     Compute $\mathsf{st}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\mathrm{out}}; r_4^t)$;
27 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, t+1), \tilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{out}}, (\mathbf{B}^{\mathrm{out}}, \mathbf{lw}^{\mathrm{out}}))$;
```

B.5.5 Partially Punctured ORAM: From $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,2}$

Before giving the proof details that the hybrids $\mathbf{Hyb}_{2,i,0,j,1}$ and $\mathbf{Hyb}_{2,i,0,j,2}$ are indistinguishable, we provide first the main proof ideas. Recall that in an ORAM tree, a node (as a bucket) consists of multiple blocks of bit string. In the following, let *block** be the block to be fetched at time $(\mathsf{t}, d) = (i-1, j)$, and *pos** be the corresponding position of *block**. We let t_{pos} be the time when *pos** is generated by PPRF; note that t_{pos} is also the last modification time of *block**.

We need to move from $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,2}$, or puncture ORAM compiled next-step program F_o. Our main idea is to erase *block** and *pos** value in memory and CPU state from time $\mathsf{t} = \mathsf{t}_{pos}$ to time $\mathsf{t} = i-1$. To simplify the exposition, let us focus on erasing *block**, and we note that the *pos** (which is written to ORAM recursively) can be handled analogously. In other words, consider the following simplified goal:

> $\mathbf{Hyb}_{2,i,0,j,1',\mathsf{t}_{pos}}$: (A) At time t_{pos}, instead of putting back the encryption of the fetched *block** to the root of the ORAM tree, an encryption of dummy value is put back instead.

From $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1',\mathsf{t}_{pos}}$, we wish to change the step computation at time $\mathsf{t} = \mathsf{t}_{pos}$ to $F_o^{j,\mathsf{punct}'}$. This step is non-trivial for the following reason: although *block** to be fetched at $(\mathsf{t}, d) = (i-1, j)$ is encrypted, we cannot leverage its semantic security since the PRF key K_E used to generate the ciphertext is hardwired in the program, and the security of $\mathsf{Ci}\mathcal{O}$ does not hide these hardwired values. In particular, *block** might be searched in Fetch step or be moved to another node in Flush step from time $\mathsf{t} = \mathsf{t}_{pos}$ to time $\mathsf{t} = i-1$, where both steps must first decrypt all memory input with the private key, which is generated from K_E. Specifically, following cases need to decrypt (and re-encrypt) *block**:

- Fetch step searches for another block, and *block** is on the searching path. Because *block** is not matched, it is decrypted and re-encrypted in the same bucket.
- Flush step flushes the path pos'', and *block** is not moved. Similar to Fetch, *block** is decrypted and re-encrypted in the same bucket.
- Flush step flushes the path pos'', and *block** is moved from one bucket to another bucket. *block** is decrypted and then re-encrypted in two different buckets.

Therefore, in the worst case, the adversary actually has knowledge of the position *pos** to be fetched, the access pattern at this point is actually deterministic and hence cannot be simulated.

In order to argue that we can indeed switch to $\mathbf{Hyb}_{2,i,0,j,1',\mathsf{t}_{pos}}$, the trivial approach is to hardwire input of the next time t'_{pos} that decrypts *block**, but there might exist another next time of t'_{pos} that decrypts *block**, and so on. This would mean that we need to hardwire $\Omega(T)$ information inside the program, making the construction *not* time-succinct. Instead, we will show, via the series of hybrids presented below, that we can erase the corresponding part of the ciphertexts one after another, while having only constant amount of information hardwired in every hybrid.

Partially Punctured Hybrids Our key idea to do so is to add one *partially puncturing* procedure (B) code to the ORAM program as a helper, which hardwires the *block** in plaintext and "erases" it whenever this block is decrypted in the memory accesses from time $\mathsf{t} = \mathsf{t}_{pos}$ to time $\mathsf{t} = i-1$. Consider

> $\mathbf{Hyb}_{2,i,0,j,1'',\mathsf{t}_{pos}+1}$: (A) At time t_{pos}, instead of putting back the encryption of the fetched *block** to the root of the ORAM tree, an encryption of dummy value is put back instead.
> (B) At time $\mathsf{t} \geq \mathsf{t}_{pos} + 1$, if the input state or memory contains *block**, then replace it by dummy value before performing the computation.

Since we do not put back *block** at time t_{pos}, *block** does not exist after time t_{pos}, and thus the (B) code is never executed. Therefore, the programs in $\mathbf{Hyb}_{2,i,0,j,1',\mathsf{t}_{pos}}$ and $\mathbf{Hyb}_{2,i,0,j,1'',\mathsf{t}_{pos}+1}$ have identical computation trace, and the two hybrids are indistinguishable by the security of $\mathsf{Ci}\mathcal{O}$. So, our goal reduces to move from $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1'',\mathsf{t}_{pos}+1}$.

Towards this, we will first remove the (A) code and add the (B) code only, and we will argue they are indistinguishable by IND-CPA security later. Next, we add the (B) code gradually and parametrize the (B) code by its time step condition, and consider hybrids $\mathbf{Hyb}_{2,i,0,j,1,z}$ with only the (B) code added:

$\mathbf{Hyb}_{2,i,0,j,1,z}$: (B) At time $\mathfrak{t} \geq z$, if the input state or memory contains *block**, then replace it by dummy value before performing the computation.

Note that when $z = i - 1$, the (B) code is ineffective, since computation at time step $i - 1$ is already simulated. Thus, $\mathbf{Hyb}_{2,i,0,j,1}$ and $\mathbf{Hyb}_{2,i,0,j,1,i-1}$ are indistinguishable by $\mathsf{Ci}\mathcal{O}$ security.

We next argue indistinguishability of $\mathbf{Hyb}_{2,i,0,j,1,z}$ and $\mathbf{Hyb}_{2,i,0,j,1,z-1}$, where the difference is at time step $z - 1$. If the input at time step $z - 1$ does not contain *block**, then the computation trace is identical and the hybrids are indistinguishability by $\mathsf{Ci}\mathcal{O}$ security. Now, if the input at time step $z - 1$ contains *block**, the key observation here is that since we have the (B) code added for time step $\mathfrak{t} \geq z$, when we modify the $[block^*]^{out}$ at time $z - 1$ to dummy value, it does not effect the computation after any time z. Therefore, to show that $\mathbf{Hyb}_{2,i,0,j,1,z}$ and $\mathbf{Hyb}_{2,i,0,j,1,z-1}$ are indistinguishable, we need to replace the (encrypted) $[block^*]^{out}$ in $\mathbf{Hyb}_{2,i,0,j,1,z-1}$ by (encryption of) dummy value. We hardwire the plaintext and ciphertext at time $z - 1$, and this allows us to replace the (encrypted) $[block^*]^{out}$ by (encryption of) dummy value with PPRF and $\mathcal{PKE}$ security.

Now, we have erased *block** from the *output* at time $\mathfrak{t} = z - 1$, but $\mathbf{Hyb}_{2,i,0,j,1,z-1}$ is to erase *block** from the *input*. Intuitively, any computation step from time $\mathfrak{t} = \mathfrak{t}_{pos}$ to time $\mathfrak{t} = i - 1$ does not "compute" on *block**, and *block** is transferred literally from input to output through F_o. We claim the overall output at time $\mathfrak{t} = z - 1$ are always identical by analyzing following cases in CP-ORAM:

- In Fetch step, *block** is always in the same bucket, and it implies that erasing *block** from input and erasing *block** from output are identical.
- In Flush step, if *block** is not moved, then erasing *block** from input and erasing *block** from output are identical.
- In Flush step, if *block** is on the flushing path and is moved from one bucket to another bucket, then erasing input buckets and erasing output buckets yield the identical result, where both buckets have no *block**.

Note this "commute property of erasure" is implied by CP-ORAM construction and the definitions of *block** and $\mathfrak{t}_{pos}$.

What we have done allow us to move from $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos}+1}$, which has the partially puncturing procedure injected. The difference between $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos}+1}$ and $\mathbf{Hyb}_{2,i,0,j,1'',\mathfrak{t}_{pos}+1}$ is the (A) code, which is to replace the (encrypted) $[block^*]^{out}$ at time step $\mathfrak{t}_{pos}$ by (encryption of) dummy value (with the helper (B) code added). This is the same task as above, and can be handled by the same hybrids.

Lemma B.54. *Let $\mathcal{PKE}$ be an IND-CPA secure public key encryption scheme,* $\mathsf{Ci}\mathcal{O}$ *be a computation-trace indistinguishability obfuscation scheme in the RAM model,* PRF *be a secure puncturable PRF scheme; then the hybrids* $\mathbf{Hyb}_{2,i,0,j,1}$ *and* $\mathbf{Hyb}_{2,i,0,j,2}$ *are computationally indistinguishable.*

Proof. Formally, we define fourth-layer hybrids $\mathbf{Hyb}_{2,i,0,j,1,z}$ for $\mathfrak{t}_{pos} < z \leq i - 1$, $\mathbf{Hyb}_{2,i,0,j,1',z}$ for $\mathfrak{t}_{pos} < z \leq i - 1$, $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z}$ for $\mathfrak{t}_{pos} < z \leq i - 1$ and hybrids are proceeded as follows.

- $\mathbf{Hyb}_{2,i,0,j,1} \approx \mathbf{Hyb}_{2,i,0,j,1,i-1} \approx \ldots \mathbf{Hyb}_{2,i,0,j,1,z} \cdots \approx \mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos}+1} \approx \mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}+1} \approx \mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$ $\approx \ldots \mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z} \cdots \approx \mathbf{Hyb}_{2,i,0,j,2}$
- $\mathbf{Hyb}_{2,i,0,j,1,z} \approx \mathbf{Hyb}_{2,i,0,j,1',z} \approx \mathbf{Hyb}_{2,i,0,j,1,z-1}$

Some additional notations used in $\mathbf{Hyb}_{2,i,0,j,1',z}$ are listed in Table 8.

$\mathbf{Hyb}_{2,i,0,j,1,z}$ In this hybrid, the program is replaced by $F_e^{2,i,0,j,1,z}$ defined in Algorithm 75. $F_e^{2,i,0,j,1,z}$ erases the part in the plaintext of the input bits corresponding to *block** by partially puncturing procedure that searches and erases *block** at time t, $z \leq \mathfrak{t} \leq i - 1$, and it also calls $F_{o,hyb}^{j,2}$ at time $i - 1$. Recall vector $\mathbf{B}^{\text{in}}$ is a vector of nodes on an ORAM tree path, where each node is a bucket that stores several blocks of memory, and the partially puncturing procedure is to search and erase only *block** from $\mathbf{B}^{\text{in}}$. In addition, this procedure uses the original (standard) representation in the ORAM data structure, which denotes an empty slot in a bucket by an empty symbol. Note that at time $\mathfrak{t}$ where $z < \mathfrak{t} \leq i - 1$, the memory content which corresponds to *block** has already been erased, and thus the program just executes the "normal" F_o function in the sense that it is indeed generating the "real" access pattern with respect to a particular memory which has *block** erased.

Hybrid	t_{pos}	$z-1$	z	If decrypts $z-1$	$i-1$
$\mathbf{Hyb}_{2,i,0,j,1}$	honest	honest	honest	honest	honest *pos**, simulated data
$\mathbf{Hyb}_{2,i,0,j,1,i-1}$	honest	honest	honest	honest	hardwire *pos**, simulated data
...					
$\mathbf{Hyb}_{2,i,0,j,1,z}$	honest	honest	erase $[block^*]^{in}$	erase $[block^*]^{in}$	hardwire *pos**, simulated data
$\mathbf{Hyb}_{2,i,0,j,1',z}$	honest	hardwire $[block^*]^{out}$	erase $[block^*]^{in}$	hardwire $[block^*]^{in}$ erase $[block^*]^{in}$	hardwire *pos**, simulated data
$\mathbf{Hyb}_{2,i,0,j,1'',z}$	honest	hardwire $[block^*]^{out}$ erase $[block^*]^{out}$	erase $[block^*]^{in}$	erase $[block^*]^{in}$	hardwire *pos**, simulated data
$\mathbf{Hyb}_{2,i,0,j,1,z-1}$	honest	erase $[block^*]^{in}$	erase $[block^*]^{in}$	erase $[block^*]^{in}$	hardwire *pos**, simulated data
...					
$\mathbf{Hyb}_{2,i,0,j,1'',\mathsf{t}_{pos}+1}$	hardwire $[block^*]^{out}$ erase $[block^*]^{out}$	erase $[block^*]^{in}$	erase $[block^*]^{in}$	erase $[block^*]^{in}$	hardwire *pos**, simulated data
$\mathbf{Hyb}_{2,i,0,j,1',\mathsf{t}_{pos}}$	erase $[block^*]^{out}$	honest	honest	honest	hardwire *pos**, simulated data
...					
$\mathbf{Hyb}_{2,i,0,j,1,\mathsf{t}_{pos},z}$	erase $[block^*]^{out}$	honest	erase $[pos^*]^{in}$	erase $[pos^*]^{in}$	hardwire *pos**, simulated data
...					
$\mathbf{Hyb}_{2,i,0,j,2}$	erase $[block^*]^{out}$ erase $[pos^*]^{out}$	honest	honest	honest	hardwire *pos**, simulated data

Table 7: The hybrid series from $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1',\mathsf{t}_{pos}}$, then to $\mathbf{Hyb}_{2,i,0,j,2}$, with the important instructions in the next-step program that are related to the erasure of *block** in the ORAM tree

Notation	Definition
h^*_z	The index of the node which contains *block** at time $\mathfrak{t} = z$ (If time $\mathfrak{t} = z$ has no *block** in the memory input, then let h^*_z be root index ϵ.)
b^*_z	The plaintext of the node with index h^*_z at time $\mathfrak{t} = z$
$\boxed{b}^*_z$	The ciphertext of the node b^*_z at time $\mathfrak{t} = z$
$\boxed{b}^{*-}_z$	The ciphertext the same node b^*_z except that *block** is erased

Table 8: Additional notations for the $\mathbf{Hyb}_{2,i,0,j,1',z}$

$\mathbf{Hyb}_{2,i,0,j,1',z}$ In this hybrid, the program is replaced by $F_e^{2,i,0,j,1',z}$ defined in Algorithm 76. $F_e^{2,i,0,j,1',z}$ is similar to $F_e^{2,i,0,j,1',z}$, except for the following changes.

- At $\mathfrak{t} = z - 1$, the bucket of the output ciphertext corresponding to *block**, namely $\boxed{\mathbf{B}}^{\text{out}}[h^*_{z-1}]$, is replaced by a hardwired ciphertext $\boxed{b}^*_{z-1}$.
- At $t \leq z$, an explicit check is imposed so that the private decryption key corresponding to time $\mathfrak{t} = z-1$ and $h = h^*_{z-1}$ have never be used. To keep the program working correctly, we need to hardwire the plaintext b^*_{z-1} corresponding to $\boxed{b}^*_{z-1}$ so that F_o can run with the correct input.

  We expand vector notation of PRF and $\mathcal{PKE}$ decryption to an equivalent loop form, which is easier to denote this special hardwired condition along with other honest computations.

$\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$ In this hybrid, the program is replaced by $F_e^{2,i,0,j,1',\mathfrak{t}_{pos}}$ defined in Algorithm 78, which removes the plaintext corresponding to *block** inside the ORAM access function $\text{OACCESS}^{j,\mathsf{punct}'}$ (Algorithm 79).

$\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z}$ In this hybrid, the program is replaced by $F_e^{2,i,0,j,1,\mathfrak{t}_{pos},z}$ defined in Algorithm 80. $F_e^{2,i,0,j,1,\mathfrak{t}_{pos},z}$ is similar to $F_e^{2,i,0,j,1',\mathfrak{t}_{pos}}$ except the erasure of the part in the plaintext of the input bits corresponding to *pos**.

Recall that value *pos** is generated in the j-th recursion level and also stored in the position map, which is outsourced to the $(j+1)$-th level of ORAM recursively. Let loc^* be the location of the memory cell storing *pos** in the $(j+1)$-th level ORAM, so *pos** can be found deterministically in the ORAM tree as follows: with loc^*, search the block with the format $(\lfloor \mathsf{loc}^*/\alpha \rfloor, \cdot, v)$ in the $(j+1)$-th ORAM tree, and then *pos** must be stored in the $(\mathsf{loc}^* \bmod \alpha)$-th cell in v. In this hybrid, our additional procedure hardwires loc^*, searches and erases *pos** with the unique symbol $\mathtt{dummy}$ for all time $\mathfrak{t}$, $z \leq \mathfrak{t} \leq i-1$ and recursion level $d = j+1$.

Analysis In the remaining of this subsection, we will complete the proof of the lemma. We applied Theorem 5.2 several times, and it allows arbitrary modification in the hybrid program as long as the computation trace remains identical.

From $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1,i-1}$: The difference is at time $i-1$, but the output simulated by $F^{j,2}_{o,hyb}$, which hardwires *pos**, has the identical computation trace. By Theorem 5.2, these two hybrids are computationally indistinguishable.

From $\mathbf{Hyb}_{2,i,0,j,1,z}$ to $\mathbf{Hyb}_{2,i,0,j,1',z}$: Since $F_e^{2,i,0,j,1',z}$ is obtained by hardwiring the same outputs computed from $F_e^{2,i,0,j,1,z}$, the computations defined by the two programs have identical computation trace. Therefore, by Theorem 5.2, the hybrids are computationally indistinguishable.

From $\mathbf{Hyb}_{2,i,0,j,1',z}$ to $\mathbf{Hyb}_{2,i,0,j,1,z-1}$: The indistinguishability is established via a series of intermediate hybrids, which will be described as follows:

Algorithm 75: $F_e^{2,i,0,j,1,z}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,   $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, \underline{z, block^*}, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
9      Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
10     Compute $\mathbf{B}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;
14     if $\underline{(z \leq \mathsf{t} \leq i-1) \textbf{ and } (d = j)}$ then
15         $\underline{\textbf{foreach} \text{ bucket } b \text{ in } \mathbf{B}^{\text{in}} \textbf{ do} \text{ search and erase } block^* \text{ from } b \text{ (and thus } \mathbf{B}^{\text{in}})}$;
16     if $\mathsf{t} = i-1$ then
17         $\underline{\text{Compute } (\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F_{o,hyb}^{j,2}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})}$;
18     else
19         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
20     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
21     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
22     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
23     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
24     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
25     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
26     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
27 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

Algorithm 76: $F_e^{2,i,0,j,1',z}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t), \widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, z, block^*, \underline{h^*_{z-1}, b^*_{z-1}, \boxed{b}^*_{z-1}}, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      foreach $h \in [\,|\boxed{\mathbf{B}}^{\text{in}}|\,]$ do // We expand the vector notation and only modify red-underlined condition.
9          if $(\mathsf{t} \geq z)$ and $(\mathbf{lw}^{\text{in}}[h] = z-1)$ and $(h = h^*_{z-1})$ then
10             $\underline{\text{Set } \mathbf{B}^{\text{in}}[h^*_{z-1}] = b^*_{z-1}}$ ;
11         else
12             Compute $(r_1^{\text{in}}, r_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}[h], h))$;
13             Compute $(\text{pk}^{\text{in}}, \text{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_1^{\text{in}})$;
14             Set $\mathbf{B}^{\text{in}}[h] = \mathcal{PKE}.\mathsf{Decrypt}(\text{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}}[h])$;
15     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
16     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
17     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;
18     if $(z \leq \mathsf{t} \leq i-1)$ and $(d = j)$ then
19         foreach bucket $b$ in $\mathbf{B}^{\text{in}}$ do search and erase $block^*$ from $b$ (and thus $\mathbf{B}^{\text{in}}$);
20     if $\mathsf{t} = i-1$ then
21         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F^{j,2}_{o,hyb}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
22     else
23         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
24     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
25     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
26     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
27     Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
28     $\underline{\text{if } \mathsf{t} = z-1 \text{ then set } \boxed{\mathbf{B}}^{\text{out}}[h^*_{z-1}] = \boxed{b}^*_{z-1}}$;
29     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
30     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
31     Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
32 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,  $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

Algorithm 77: $F_e^{2,i,0,j,1'',z}$

```
Input : $\widetilde{\mathsf{st}}^{\mathrm{in}} = (\boxed{\mathsf{st}}^{\mathrm{in}}, t), \widetilde{a}^{\mathrm{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\mathrm{in}}, (\boxed{\mathbf{B}}^{\mathrm{in}}, \mathbf{lw}^{\mathrm{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, z, block^*, h^*_{z-1}, \underline{\boxed{b}^{*-}_{z-1}}, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      ...
5  if $i \le \mathsf{t} < \mathsf{t}^*$ then
6      ...
7  else
8      Compute $(\mathbf{r}_1^{\mathrm{in}}, \mathbf{r}_2^{\mathrm{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{in}}, h(\mathbf{I}^{\mathrm{in}})))$;
9      Compute $(\mathbf{pk}^{\mathrm{in}}, \mathbf{sk}^{\mathrm{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{in}})$;
10     Compute $\mathbf{B}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\mathrm{in}}, \boxed{\mathbf{B}}^{\mathrm{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\mathrm{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\mathrm{in}})$;
14     if $(z \le \mathsf{t} \le i-1)$ and $(d = j)$ then
15         foreach bucket $b$ in $\mathbf{B}^{\mathrm{in}}$ do search and erase $block^*$ from $b$ (and thus $\mathbf{B}^{\mathrm{in}}$);
16     if $\mathsf{t} = i - 1$ then
17         Compute $(\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) \leftarrow F^{j,2}_{o,hyb}(\mathsf{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;
18     else
19         Compute $(\mathsf{st}^{\mathrm{out}}, \mathbf{I}^{\mathrm{out}}, \mathbf{B}^{\mathrm{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\mathrm{in}}, \mathbf{I}^{\mathrm{in}}, \mathbf{B}^{\mathrm{in}})$;
20     Set $\mathbf{lw}^{\mathrm{out}} = (t, \ldots, t)$;
21     Compute $(\mathbf{r}_1^{\mathrm{out}}, \mathbf{r}_2^{\mathrm{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\mathrm{out}}, h(\mathbf{I}^{\mathrm{out}})))$;
22     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\mathrm{out}})$;
23     Compute $\boxed{\mathbf{B}}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\mathrm{out}}; \mathbf{r}_2^{\mathrm{out}})$;
24     $\underline{\textbf{if } \mathsf{t} = z-1 \textbf{ then set } \boxed{\mathbf{B}}^{\mathrm{out}}[h^*_{z-1}] = \boxed{b}^{*-}_{z-1};}$
25     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
26     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
27     Compute $\boxed{\mathsf{st}}^{\mathrm{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\mathrm{out}}; r_4^t)$;
28 Output $\widetilde{\mathsf{st}}^{\mathrm{out}} = (\boxed{\mathsf{st}}^{\mathrm{out}}, t+1), \quad \widetilde{a}^{\mathrm{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\mathrm{out}}, (\boxed{\mathbf{B}}^{\mathrm{out}}, \mathbf{lw}^{\mathrm{out}}))$ ;
```

1. $F_e^{2,i,0,j,1',z}\{K_E\{(z-1,h^*_{z-1})\}, h^*_{z-1}, b^*_{z-1}, \boxed{b}^{*}_{z-1}\}$. In the first hybrid, the input $(z-1, h^*_{z-1})$ is punctured from the PRF key K_E. The pseudorandomness computed from K_E is hardwired in the program. Since the computation defined by this program has identical computation trace as that in $\mathbf{Hyb}_{2,i,0,j,1',z}$, the two hybrids are computationally indistinguishable by Theorem 5.2.

2. $F_e^{2,i,0,j,1',z}\{K_E\{(z-1,h^*_{z-1})\}, h^*_{z-1}, b^*_{z-1}, \boxed{b; r^*}^{*}_{z-1}\}$. In the next hybrid, we replace the hardwired ciphertext by a ciphertext encrypted with true randomness. The indistinguishability is guaranteed by the selective security of PPRF.

3. $F_e^{2,i,0,j,1',z}\{K_E\{(z-1,h^*_{z-1})\}, h^*_{z-1}, b^*_{z-1}, \boxed{b; r^*}^{*-}_{z-1}\}$. Next, we change the hardwired ciphertext $\boxed{b; r^*}^{*}_{z-1}$ to its counterpart $\boxed{b; r^*}^{*-}_{z-1}$ with *block** erased. The indistinguishability is guaranteed by the IND-CPA security of the $\mathcal{PKE}$ because $F_e^{2,i,0,j,1',z}$ would never use the private key of $(z-1, h^*_{z-1})$.

4. $F_e^{2,i,0,j,1',z}\{K_E, h^*_{z-1}, b^*_{z-1}, \boxed{b}^{*-}_{z-1}\}$. Then, we un-puncture the PRF. The indistinguishability is guaranteed by the security of the puncturable PRF.

5. $F_e^{2,i,0,j,1',z}\{K_E, h^*_{z-1}, \boxed{b}^{*-}_{z-1}\}$. We remove Line 10 and the honest plaintext b^*_{z-1} that contains *block** (Algorithm 77). The computation defined by this program has identical computation trace as the previous hybrid, and two hybrids are computationally indistinguishable by Theorem 5.2. Note that, to show two traces are identical, we observe the only difference is the input bucket b^*_{z-1} was replaced by b^{*-}_{z-1} which is decrypted from $\boxed{b}^{*-}_{z-1}$. For all $z \le \mathfrak{t} \le i-1$, however, the input to F_o always has *block** erased in advance, and it follows that double-erasing yields the identical input $\mathbf{B}^{\text{in}}$.

6. At this point, the computation trace defined by $F_e^{2,i,0,j,1',z}\{K_E, h^*_{z-1}, \boxed{b}^{*-}_{z-1}\}$ is identical to that in $\mathbf{Hyb}_{2,i,0,j,1,z-1}$, and two hybrids are computationally indistinguishable by Theorem 5.2. It is based on the "commute property of erasure", where any computation step from time $\mathfrak{t}_{pos}$ to $i-1$ does not "compute" on *block**, and *block** is transferred literally from the input to the output through F_o. Therefore, erasing *block** from output is identical to erasing *block** from the input.

From $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}+1}$ to $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$: By the same argument from $\mathbf{Hyb}_{2,i,0,j,1',z}$ to $\mathbf{Hyb}_{2,i,0,j,1'',z}$, we can see that $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}+1}$ is indistinguishable from $\mathbf{Hyb}_{2,i,0,j,1'',\mathfrak{t}_{pos}+1}$. Note that the only difference between $\mathbf{Hyb}_{2,i,0,j,1'',\mathfrak{t}_{pos}+1}$ and $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$ is the erasure of *block** in $F_e^{2,i,0,j,1',\mathfrak{t}_{pos}}$ (which is implemented in $\text{OACCESS}^{j,\mathsf{punct}'}$), and they have identical computation trace. By Theorem 5.2, the two hybrids are computationally indistinguishable.

From $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$ to $\mathbf{Hyb}_{2,i,0,j,2}$: Our task here is to remove the information corresponding to *pos** from the position map which has *pos** stored in the next recursion layer of the ORAM tree. The approach is similar to above, as we need to remove this information that *pos** is accessed for each time z in the next layer of ORAM tree. To show these two hybrids are computationally indistinguishable, we therefore define $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z}$ similar to $\mathbf{Hyb}_{2,i,0,j,1,z}$, and the argument is analogous to that from $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$. The sketched proof is briefly presented as follows:

– $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}} \approx \mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z}$ with $z = i-1$ is analogous to the series of hybrids from $\mathbf{Hyb}_{2,i,0,j,1}$ to $\mathbf{Hyb}_{2,i,0,j,1,i-1}$. The indistinguishability is guaranteed by the indistinguishability of Ci$\mathcal{O}$ (Theorem 5.2).

– $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z} \approx \mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z-1}$ for time z that $\mathfrak{t}_{pos} < z \le i-1$ is analogous to the series of hybrids from $\mathbf{Hyb}_{2,i,0,j,1,z}$ to $\mathbf{Hyb}_{2,i,0,j,1,z-1}$. The indistinguishability is guaranteed by the indistinguishability of Ci$\mathcal{O}$ (Theorem 5.2), the selective security of PPRF, the IND-CPA security of $\mathcal{PKE}$, and the "commute property of erasure" of ORAM construction.

– $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos},z} \approx \mathbf{Hyb}_{2,i,0,j,2}$ with $z = \mathfrak{t}_{pos}+1$ is analogous to the series of hybrids from $\mathbf{Hyb}_{2,i,0,j,1,\mathfrak{t}_{pos}+1}$ to $\mathbf{Hyb}_{2,i,0,j,1',\mathfrak{t}_{pos}}$. The indistinguishability is guaranteed by the indistinguishability of Ci$\mathcal{O}$ (Theorem 5.2), the selective security of PPRF, and the IND-CPA security of $\mathcal{PKE}$.

These hybrids and arguments are one-to-one mapping to their previous analogous, the only difference is that we erase the memory cell loc* storing *pos** (rather than the memory block *block** contains *pos**). The details are omitted here.

□

Algorithm 78: $F_e^{2,i,0,j,1',\mathsf{t}_{pos}}$

```
    Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\boxed{\mathsf{st}}^{\text{in}}, t)$,   $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\boxed{\mathbf{B}}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
    Data  : $T, K_E, \mathsf{t}^*, y, t_{pos}, K_N, K_{\mathsf{Sim}}, i, j$
 1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
 2  if $\mathsf{t} > \mathsf{t}^*$ then output Reject;
 3  if $\mathsf{t} = \mathsf{t}^*$ then
 4      ...
 5  if $i \leq \mathsf{t} < \mathsf{t}^*$ then
 6      ...
 7  else
 8      Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
 9      Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
10      Compute $\mathbf{B}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \boxed{\mathbf{B}}^{\text{in}})$;
11      Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12      Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13      Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \boxed{\mathsf{st}}^{\text{in}})$;
14      if $\mathsf{t} = i-1$ then
15          Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F_{o,hyb}^{j,2}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
16      else if $\mathsf{t} = \mathsf{t}_{pos}$ then
17          Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o^{j,\mathsf{punct}'}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$ ;
            // $F_o^{j,\mathsf{punct}'} = \mathsf{CP\text{-}ORAM.Compile}(F, \mathrm{OACCESS}^{j,\mathsf{punct}'})$
18      else
19          Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
20      Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
21      Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
22      Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
23      Compute $\boxed{\mathbf{B}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
24      Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
25      Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
26      Compute $\boxed{\mathsf{st}}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
27  Output $\widetilde{\mathsf{st}}^{\text{out}} = (\boxed{\mathsf{st}}^{\text{out}}, t+1)$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\boxed{\mathbf{B}}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

B.6 Proof Sketch of Theorem 8.1 (Security for $\mathcal{RE}$-PRAM)

The above $\mathcal{RE}$-PRAM is built in the same manner as that of $\mathcal{RE}$-RAM. Both of them depend on three levels of compilers, ORAM/OPRAM, $\mathcal{PKE}$, and finally Ci$\mathcal{O}$-RAM/PRAM. To argue the security of $\mathcal{RE}$-PRAM, we use similar proof techniques to go through hybrids except that we insert an additional layer to deal with each CPU agent respectively.

Algorithm 79: $\textsc{OAccess}^{j,\mathsf{punct}'}\{K_N\}$

```
Input  : $\mathsf{t}, d, \mathsf{loc}, \mathsf{val}$
Output : $oldval$
Data   : $K_N, \alpha, MaxDepth$ (Memory size $S = \alpha^{MaxDepth}$), $\mathsf{loc}_{i-1,j}$
1  if $d \geq MaxDepth$ then
2      return 0
3  Pick leaf $newpos$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \mathtt{FetchR}))$ ;      // Update position map
4  $pos \leftarrow \textsc{OAccess}(d+1, \lfloor \mathsf{loc}/\alpha \rfloor, newpos)$ ;      // Fetch
5  $\mathbf{I}_{\mathsf{fetch}} \leftarrow \textsc{Path}(d, pos)$;
6  $\mathbf{B}_{\mathsf{fetch}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{fetch}})$;
7  if $(d = j)$ then
8      $(\mathbf{B}_{\mathsf{fetch},-}, oldval) \leftarrow \textsc{RemoveBlock}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \alpha, pos)$;
9      $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}_{\mathsf{fetch},-})$;
10 else
11     $(\mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}}, oldval) \leftarrow \textsc{FetchAndUpdate}(\mathbf{B}_{\mathsf{fetch}}, \mathsf{loc}, \mathsf{val}, \alpha, pos, newpos)$;
12     $\textsc{Write}(\mathbf{I}_{\mathsf{fetch}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{fetch}})$;
13 Pick leaf $pos''$ at recursion level $d$ based on $\mathsf{PRF}(K_N, (\mathsf{t}, d, \mathtt{FlushR}))$ ;      // Flush
14 $\mathbf{I}_{\mathsf{flush}} \leftarrow \textsc{Path}(d, pos'')$;
15 $\mathbf{B}_{\mathsf{flush}} \leftarrow \textsc{Read}(\mathbf{I}_{\mathsf{flush}})$;
16 $\mathbf{B}^{\mathrm{out}}_{\mathsf{flush}} \leftarrow \textsc{Flush}(\mathbf{B}_{\mathsf{flush}}, pos'')$;
17 $\textsc{Write}(\mathbf{I}_{\mathsf{flush}}, \mathbf{B}^{\mathrm{out}}_{\mathsf{flush}})$;
18 return $oldval$;
```

Let **Real** be the real security game in which the adversary is given $\textsc{Enc}_{\mathbf{Real}}$, and **Ideal** be the security game in which the adversary is given $\textsc{Enc}_{\mathbf{Ideal}}$. The intermediate hybrids between **Real** and **Ideal** are similar to those in Appendix B.5. Roughly, we have the following hybrids **Real** $= \mathbf{Hyb}_0 \approx \mathbf{Hyb}_1 \approx \mathbf{Hyb}_{2,\mathsf{t}^*} \approx \ldots \approx \mathbf{Hyb}_{2,0}$ $=$ **Ideal**.

Let F_e^x, Π_e^x, and $\textsc{Enc}_x$ be the stateful function, computation system, and encoding in $\mathbf{Hyb}_x$ respectively.

$\mathbf{Hyb}_1$ In this hybrid, F_e^1 hardwires the output $\mathsf{st} = (\mathtt{halt}, y)$. It always returns $\perp$ if $\mathsf{t} > \mathsf{t}^*$. At time t^*, the special CPU agent returns $\mathsf{st} = (\mathtt{halt}, y)$. From $\mathbf{Hyb}_0$ to $\mathbf{Hyb}_1$, Π_e^0 and Π_e^1 have the same computation traces, and thus by applying $\mathsf{Ci}\mathcal{O}$-PRAM, $\textsc{Enc}$ and $\textsc{Enc}_1$ are computationally indistinguishable.

$\mathbf{Hyb}_{2,i}$ In this hybrid, at time t, $i \leq \mathsf{t} \leq \mathsf{t}^*$, $F_e^{2,i}$'s access pattern is a simulated access pattern provided by the OPRAM simulator, and the output state is replaced by an encryption of a special symbol $\mathtt{dummy}$. From $\mathbf{Hyb}_1$ to $\mathbf{Hyb}_{2,t*}$, we directly apply $\mathsf{Ci}\mathcal{O}$-PRAM to claim that $\textsc{Enc}_1$ and $\textsc{Enc}_{2,\mathsf{t}^*}$ are computationally indistinguishable due to $\mathsf{Trace}\langle\Pi_e^1\rangle = \mathsf{Trace}\langle\Pi_e^{2,\mathsf{t}^*}\rangle$. However, from $\mathbf{Hyb}_{2,i}$ to $\mathbf{Hyb}_{2,i-1}$, we define the next layer $\mathbf{Hyb}_{2,i,k}$ in which k is indexed by a CPU.

$\mathbf{Hyb}_{2,i,k}$ In this hybrid at time $\mathsf{t} = i-1$, $F_e^{2,i,k}$'s access pattern is a simulated access pattern provided by the OPRAM simulator if CPU $\mathtt{A} < k$, while its access pattern is a real access pattern if CPU $\mathtt{A} \geq k$. For the time i and CPU k, we consider the following cases.

- If CPU k is not a representative to access its corresponding memory location loc_k, $\mathbf{Hyb}_{2,i,k}$ and $\mathbf{Hyb}_{2,i,k+1}$ are identical.
- If CPU k is a representative to access its corresponding memory location loc_k and that memory block of loc_k is stored at the OPRAM tree path pos^*, we need to argue that $F_e^{2,i,k}$ and $F_e^{2,i,k+1}$ are computational indistinguishable. Therefore, we define the four layer hybrids $\mathbf{Hyb}_{2,i,k,0,j}$ later where j is the recursive

Algorithm 80: $F_e^{2,i,0,j,1,\mathsf{t}_{pos},z}$

```
Input : $\widetilde{\mathsf{st}}^{\text{in}} = (\mathsf{st}^{\text{in}}, t)$,   $\widetilde{a}^{\text{in}}_{\mathtt{A}\leftarrow\mathtt{M}} = (\mathbf{I}^{\text{in}}, (\mathbf{B}^{\text{in}}, \mathbf{lw}^{\text{in}}))$
Data  : $T, K_E, \mathsf{t}^*, y, \mathsf{t}_{pos}, \mathsf{loc}^*, K_N, K_{\mathsf{Sim}}, i, j$
1  Compute $\mathsf{t} = \lceil t/q_o \rceil$;
2  if $\mathsf{t} > \mathsf{t}^*$ then output $\mathtt{Reject}$;
3  if $\mathsf{t} = \mathsf{t}^*$ then
4      $\ldots$
5  if $i \le \mathsf{t} < \mathsf{t}^*$ then
6      $\ldots$
7  else
8      Compute $(\mathbf{r}_1^{\text{in}}, \mathbf{r}_2^{\text{in}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{in}}, h(\mathbf{I}^{\text{in}})))$;
9      Compute $(\mathbf{pk}^{\text{in}}, \mathbf{sk}^{\text{in}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{in}})$;
10     Compute $\mathbf{B}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(\mathbf{sk}^{\text{in}}, \mathbf{B}^{\text{in}})$;
11     Compute $(r_3^{t-1}, r_4^{t-1}) = \mathsf{PRF}(K_E, t-1)$;
12     Compute $(pk_{\mathsf{st}}, sk_{\mathsf{st}}) = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^{t-1})$;
13     Compute $\mathsf{st}^{\text{in}} = \mathcal{PKE}.\mathsf{Decrypt}(sk_{\mathsf{st}}, \mathsf{st}^{\text{in}})$;
14     if $(z \le \mathsf{t} \le i-1)$ and $(d = j+1)$ then
15         foreach bucket $b$ in $\mathbf{B}^{\text{in}}$ do
16             search for block of the form $(\lfloor \mathsf{loc}^*/\alpha \rfloor, \cdot, v)$,
17             and erase the $(\mathsf{loc}^* \bmod \alpha)$-th cell in $v$ (and thus in $\mathbf{B}^{\text{in}}$) with symbol $\mathtt{dummy}$;
18     if $\mathsf{t} = i-1$ then
19         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) \leftarrow F_{o,hyb}^{j,2}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
20     else if $\mathsf{t} = \mathsf{t}_{pos}$ then
21         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o^{j,\mathsf{punct}'}(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$ ;
           // $F_o^{j,\mathsf{punct}'} = \mathsf{CP\text{-}ORAM.Compile}(F, \mathrm{OACCESS}^{j,\mathsf{punct}'})$
22     else
23         Compute $(\mathsf{st}^{\text{out}}, \mathbf{I}^{\text{out}}, \mathbf{B}^{\text{out}}) = F_o(\mathsf{t}, \mathsf{st}^{\text{in}}, \mathbf{I}^{\text{in}}, \mathbf{B}^{\text{in}})$;
24     Set $\mathbf{lw}^{\text{out}} = (t, \ldots, t)$;
25     Compute $(\mathbf{r}_1^{\text{out}}, \mathbf{r}_2^{\text{out}}) = \mathsf{PRF}(K_E, (\mathbf{lw}^{\text{out}}, h(\mathbf{I}^{\text{out}})))$;
26     Compute $(\mathbf{pk}', \mathbf{sk}') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; \mathbf{r}_1^{\text{out}})$;
27     Compute $\mathbf{B}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(\mathbf{pk}', \mathbf{B}^{\text{out}}; \mathbf{r}_2^{\text{out}})$;
28     Compute $(r_3^t, r_4^t) = \mathsf{PRF}(K_E, t)$;
29     Compute $(pk', sk') = \mathcal{PKE}.\mathsf{Gen}(1^\lambda; r_3^t)$;
30     Compute $\mathsf{st}^{\text{out}} = \mathcal{PKE}.\mathsf{Encrypt}(pk', \mathsf{st}^{\text{out}}; r_4^t)$;
31 Output $\widetilde{\mathsf{st}}^{\text{out}} = (\mathsf{st}^{\text{out}}, t+1)$,   $\widetilde{a}^{\text{out}}_{\mathtt{M}\leftarrow\mathtt{A}} = (\mathbf{I}^{\text{out}}, (\mathbf{B}^{\text{out}}, \mathbf{lw}^{\text{out}}))$ ;
```

level.

$\mathbf{Hyb}_{2,i,k} = \mathbf{Hyb}_{2,i,k,0,0} \approx \ldots \approx \mathbf{Hyb}_{2,i,k,0,d_{\max}} \approx \mathbf{Hyb}_{2,i,k,0',d_{\max}} \approx \ldots \approx \mathbf{Hyb}_{2,i,k,0',0} = \mathbf{Hyb}_{2,i,k+1}$.

In the construction, the access pattern of the OPACCESS depends on the paths (stored in the public state st_o^t) that each CPU wants to access. Note that the access pattern is fully determined by st_o^t revealed in the execution. So, we do not erase its content in the hybrids and further guarantee the correctness of the execution. The content in the first half is simulated, while in the second half is real. In other words, we generate simulated path for each CPU, and store them in st_o^t to simulate the access pattern of OPACCESS.

As BCP-OPRAM is a generalization of CP-ORAM, it is not hard to see that the puncturing argument generalized to work for BCP-OPRAM as well. It suffices to information-theoretically hide the values of the paths p_k's to simulate the access pattern, and this can be done by injecting a puncturing code. This can be done CPU by CPU. Namely, for each p_k accessed by CPU k, we can inject a puncturing code at the corresponding time step t'_k that the value p_k is generated, to remove the generation of p_k. Moreover, we can move to this punctured hybrid by a sequence of partially punctured hybrids as before (Appendix B.5.4), by gradually puncturing the value of p_k backwards in time, per time step and per CPU. Upon reaching this punctured hybrid, we can switch p_k to a simulated one, undo the puncturing, and move to the next CPU. The argument from $\mathbf{Hyb}_{2,i,k,0,j}$ to $\mathbf{Hyb}_{2,i,k,0,j+1}$ is identical to that in Appendix B.5.4.

B.7 Proof of Theorem 9.4 (Security for $\mathcal{VE}$)

Proof. Let $\mathsf{Adv}_{\mathcal{A}}^{\beta}$ be the advantage of the adversary $\mathcal{A}$ in the hybrid $\mathbf{Hyb}_\beta$.

$\mathbf{Hyb}_0$ This is the real security experiment. The challenger chooses randomness r_1, r_2, r_3, and computes $(\mathrm{vk}, \mathrm{sk}) \leftarrow \mathsf{SIG.Gen}(1^\lambda; r_1)$, and $\textsc{enc} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(\widehat{\Pi}; r_3)$. Note that here F has (r_1, r_2, sk) hardcoded. The challenger then returns $(\textsc{enc}, \mathrm{vk})$ to the adversary $\mathcal{A}$. The adversary wins the game if it returns $(\widetilde{\pi}, \widetilde{y})$ so that $\mathcal{VE}.\mathsf{Verify}(\mathrm{vk}, \widetilde{\pi}, \widetilde{y}) = 1$ and $\widetilde{y} \neq P(x)$.

$\mathbf{Hyb}_1$ The challenger chooses randomness r_1, r_2, r_3, and computes $(\mathrm{vk}, \mathrm{sk}) \leftarrow \mathsf{SIG.Gen}(1^\lambda; r_1)$, $y = P(x)$, $\sigma = \mathsf{SIG.Sign}(\mathrm{sk}, y; r_2)$, and $\textsc{enc} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(\widehat{\Pi}'; r_3)$, where $\widehat{F}'$ (corresponds to $\widehat{\Pi}'$) has σ (rather than (r_1, r_2, sk)) hardcoded; see Algorithm 81. □

Algorithm 81: $\widehat{F}'$ `// this program is used in` $\mathbf{Hyb}_1$

```
Input : $\hat{\mathsf{st}}^{\mathrm{in}} = (\mathsf{st}^{\mathrm{in}}, t), a^{\mathrm{in}}$
Data  : $T, \underline{\sigma}$
1  Compute $(\mathsf{st}^{\mathrm{out}}, a^{\mathrm{out}}) = F(\mathsf{st}^{\mathrm{in}}, a^{\mathrm{in}})$;
2  if $\mathsf{st}^{\mathrm{out}} \neq (\mathtt{halt}, \cdot)$ then
3      Set $\hat{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, t+1)$;
4  else
5      if $y = \perp$ then
6          Set $\hat{\mathsf{st}}^{\mathrm{out}} = \mathsf{st}^{\mathrm{out}}$;
7      else
8          Set $\hat{\mathsf{st}}^{\mathrm{out}} = (\mathsf{st}^{\mathrm{out}}, \sigma)$;
9          Set $a^{\mathrm{out}} = \perp$;
10 Output $\hat{\mathsf{st}}^{\mathrm{out}}, a^{\mathrm{out}}$;
```

Analysis Our goal here is to show $\mathsf{Adv}^0_{\mathcal{A}} \leq \mathsf{negl}(\lambda)$. To achieve this goal, we prove the following lemmas.

Lemma B.55. *If* $\mathsf{Ci}\mathcal{O}$ *is a secure indistinguishability obfuscation for computation scheme in the RAM / PRAM model, then we have* $|\mathsf{Adv}^0_{\mathcal{A}} - \mathsf{Adv}^1_{\mathcal{A}}| \leq \mathsf{negl}(\lambda)$.

Proof. Assume there is an adversary $\mathcal{A}$, who can distinguish the two hybrids with non-negligible probability. By average argument, there exist r_1, r_2 such that $\mathcal{A}$ can distinguish $\mathbf{Hyb}_0^{[r_1,r_2]}$ from $\mathbf{Hyb}_1^{[r_1,r_2]}$. That means $\mathcal{A}$ can distinguish $\mathsf{Ci}\mathcal{O}(\widehat{\Pi})$ from $\mathsf{Ci}\mathcal{O}(\widehat{\Pi}')$.

□

Lemma B.56. *If* SIG *is a secure digital signature scheme, then we have* $\mathsf{Adv}^1_{\mathcal{A}} \leq \mathsf{negl}(\lambda)$.

Proof. Assume there is an adversary $\mathcal{A}$ who wins the game in $\mathbf{Hyb}_1$. Based on such adversary $\mathcal{A}$, we show how to construct a forger $\mathcal{B}$ to break the unforgeability of SIG as follows.

$\mathcal{B}$ internally simulates a copy of $\mathbf{Hyb}_1$. Upon receiving vk from $\mathcal{B}$'s SIG challenger, $\mathcal{B}$ chooses F and x, and computes $y = F(x)$. Now $\mathcal{B}$ queries its challenger with message y to obtain the corresponding signature σ. Then, based on F and σ, $\mathcal{B}$ constructs $\widehat{F}'$, and returns $\textsc{enc} \leftarrow \mathsf{Ci}\mathcal{O}.\mathsf{Obf}(\widehat{\Pi}')$ and vk to the adversary $\mathcal{A}$. Whenever $\mathcal{A}$ returns $(\widetilde{y}, \widetilde{\pi})$, $\mathcal{B}$ returns $(\widetilde{y}, \widetilde{\sigma}) = (\widetilde{y}, \widetilde{\pi})$ to SIG challenger.

□

References

[ACC+15] Prabhanjan Ananth, Yu-Chi Chen, Kai-Min Chung, Huijia Lin, and Wei-Kai Lin. Delegating RAM computations with adaptive soundness and privacy. Cryptology ePrint Archive, Report 2015/1082, 2015. http://ia.cr/2015/1082. 8

[AS16] Prabhanjan Ananth and Amit Sahai. Functional encryption for Turing machines. In *TCC 2016-A*, 2016. To appear. 5

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation: Multi-party computation for (parallel) RAM programs. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 742–762. Springer, August 2015. 4, 8

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC 2016-A*, 2016. To appear. 8, 11, 12, 15, 18, 22, 29, 35, 58

[BGI+12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Journal of the ACM*, 59(2):6, April 2012. 12

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, March 2014. 78

[BGL+15] Nir Bitansky, Sanjam Garg, Huijia Lin, Rafael Pass, and Sidharth Telang. Succinct randomized encodings and their applications. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 439–448. ACM Press, June 2015. 6, 7, 73

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, December 2013. 78

[CCHR15] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Succinct adaptive garbled RAM. Cryptology ePrint Archive, Report 2015/1074, 2015. http://ia.cr/2015/1074. 8

[CH15] Ran Canetti and Justin Holmgren. Fully succinct garbled RAM. Cryptology ePrint Archive, Report 2015/388, 2015. http://eprint.iacr.org/2015/388. 7, 8, 19

[CHJV15] Ran Canetti, Justin Holmgren, Abhishek Jain, and Vinod Vaikuntanathan. Succinct garbling and indistinguishability obfuscation for RAM programs. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 429–437. ACM Press, June 2015. 6, 7, 14, 18, 50, 73

[CLT16] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: Improved efficiency and generic constructions. In *TCC 2016-A*, 2016. To appear. 8

[CP13] Kai-Min Chung and Rafael Pass. A simple ORAM. Cryptology ePrint Archive, Report 2013/243, 2013. http://eprint.iacr.org/2013/243. 12, 14, 18, 50, 51, 53, 58

[GGH+13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *54th FOCS*, pages 40–49. IEEE Computer Society Press, October 2013. 78

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private RAM computation. In *55th FOCS*, pages 404–413. IEEE Computer Society Press, October 2014. 5, 7

[IK00] Yuval Ishai and Eyal Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *41st FOCS*, pages 294–304. IEEE Computer Society Press, November 2000. 6, 73

[KLW15] Venkata Koppula, Allison Bishop Lewko, and Brent Waters. Indistinguishability obfuscation for turing machines with unbounded memory. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 419–428. ACM Press, June 2015. 6, 8, 9, 10, 12, 13, 14, 16, 49, 50, 74, 75, 76, 129

[KPTZ13] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 669–684. ACM Press, November 2013. 78

[LO15] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. Cryptology ePrint Archive, Report 2015/1068, 2015. http://ia.cr/2015/1068. 8

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979. 12, 18

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $o((\log n)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, December 2011. 51, 58

[SW14] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In David B. Shmoys, editor, *46th ACM STOC*, pages 475–484. ACM Press, May / June 2014. 78

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986. 4