

Query-Limited RAM Programs and their Applications

Jiahui Liu*
Fujitsu Research

Justin Raizes†
NTT Research

Bhaskar Roberts‡
NTT Research

Omri Shmueli§
NTT Research

Abstract

Quantum one-time programs (Broadbent, Gutoski and Stebila, CRYPTO 2013) or OTPs for short, enable a functionality to be encoded into a quantum token that can be evaluated on a single chosen input and then becomes unusable. While powerful, this primitive is inherently stateless and tied to a setting in which a quantum token needs to be issued and distributed for every single evaluation of a circuit. This raises a natural question: can the one-time computation paradigm be extended to richer, stateful forms of controlled access, and would such an extension offer inherent advantages beyond standard OTPs?

We introduce *query-limited RAM programs* (QLPs), a RAM-generalization of QOTPs that supports structured, stateful computation under bounded or policy-driven access. QLPs allow controlled sequences of evaluations while preventing adversarial forking or rollback of computational state. This enables new applications beyond stateless one-time programs, including quantum tokens for Turing Machines whose size depends only on *code length* (and not runtime), transferable *k*-time or budget-limited programs, and low-communication mechanisms for delegating computation in settings such as Software-as-a-Service.

To construct QLPs, we introduce *one-shot programs*, unifying one-shot signatures (Amos, Georgiou, Kiayias and Zhandry, STOC 2020) with the single effective query paradigm (Gupte, Liu, Raizes, Roberts and Vaikuntanathan, STOC 2025). We prove that one-shot programs generically imply query-limited programs, demonstrating that the strengthened unclonability guarantees of one-shot signatures translate into enhanced functionality. Along the way, we clarify the relationship between signature-token primitives and quantum one-time programs via generic constructions, essentially showing that one-time signing programs imply one-time general computation.

*jiahui.liu.crypto@gmail.com

†jraizes@alumni.cmu.edu

‡bhaskarr@berkeley.edu

§omri.shmueli1@gmail.com . This work was done in part while the author was a visiting scientist at the Simons Institute for the Theory of Computing at UC Berkeley.

Contents

1	Introduction	1
1.1	Results and Future Work	2
1.2	Related Works	5
2	Technical Overview	6
2.1	Previous Work on Quantum One-Time Programs	6
2.2	One-Shot Programs	8
2.3	Query-Limited RAM Programs	11
2.4	Applications	15
3	Preliminaries	16
3.1	Compressed Oracles	16
3.2	Pseudorandom Functions	22
3.3	Blind-Unforgeable Signatures	23
3.4	Signature Tokens	23
3.5	One-Time Programs	25
3.6	qIND-qCPA Secure Secret Key Encryption	27
4	Generalized One-Time Programs	28
4.1	Definition	28
4.2	Construction	30
4.3	One-Time Programs From Any Signature Token	32
4.4	One-Shot Programs	34
4.5	Proof of Security for Generalized OTPs	34
5	Query-Limited RAM Programs	51
5.1	Definition	51
5.2	Construction	54
5.3	Proof of Security	56
5.3.1	Ordering the Database	58
5.3.2	Simulating the One-Shot Program SEQ Oracle	60
5.3.3	Simulating the Hybrid Oracle	77
6	AI Disclosure	96
A	Compressed Oracle Chaining	99
A.1	Alternative Representation of the Decompression Function	99
A.2	Proof	100
B	Applications	103
B.1	Pay-Per-Use Programs	103
B.2	Semi-Quantum Query-Limited Programs	110
B.3	One-Shot Signatures Based On Any Blind-Unforgeable Signature Scheme	116

1 Introduction

One-time programs, introduced by Goldwasser, Kalai, and Rothblum [GKR08], aim to distribute an encoded program for a functionality f that can be evaluated on a single user-chosen input and then becomes useless. In the classical world, this goal is unattainable without trusted hardware (e.g., one-time memories) to enforce single use. Quantum information suggests a radically different possibility. Broadbent, Gutoski, and Stebila [BGS13] proposed *quantum* one-time programs (which we refer to simply as OTPs), in which the program is embodied in an unclonable quantum state. Yet formalizing security against quantum adversaries proved subtle: a quantum adversary may evaluate a program coherently, partially extract information, and then uncompute, producing behaviors that do not correspond to “one query” in any physical sense. Recent work of Gupte et al. [GLR⁺25] introduced the simulation paradigm of *single effective query* (SEQ), which measures not how many times an interface is invoked, but how many times an adversary *effectively learns* an input/output pair.

In the SEQ ideal world, the oracle may be called many times coherently; nevertheless, it maintains a database of effective queries and refuses to answer once it detects knowledge of two distinct inputs. This shift enabled meaningful simulation-based security for OTPs relative to a classical oracle, together with a construction achieving such guarantees [GLR⁺25].

Stateless vs. Stateful Computation. The above developments still concern, at their core, the one-time evaluation of a *stateless* functionality. On the other hand, computation most generally may be stateful. As a basic motivating example, consider the standard scenario of Software-as-a-Service (SaaS): an entity wants to allow restricted access to some algorithm (e.g., a powerful AI model), in the form of oracle query access. The state of the computation, in that case, records not only how many queries have been made thus far but also the inner memory state of the conversation, etc. The classical solution is that the server holds the code of the AI model and for each query, the client sends its input¹ and the server computes the output. Quantum OTPs open the possibility of the server issuing a quantum token for each query².

Both the classical and quantum solutions above suffer from two fundamental limitations. First, computational complexity: the server just wants to issue access, but needs to execute proportionally to the computation of the client (and at the very least, proportionally to the number of queries). Second, communication complexity: a k -query computation will take $\Omega(k)$ communication, while in principle, there was no real need for two entities to communicate at all.

Putting the above example aside, more basically, assume we want to allow one-time access to a piece of *code*, e.g., in the form of a Turing Machine (TM), instead of a circuit. OTPs allow to issue a one-time token for running the TM, but where the quantum token’s size is proportional to the *running time* of that TM³. This leaves open a basic question that arises throughout cryptography: what happens when the hidden functionality is not naturally a circuit that is meant to be evaluated once, but rather a stateful program whose future behavior depends on how it has already been used? In many cryptographic settings, such as indistinguishability obfuscation and fully homomorphic encryption, lifting results from circuits to RAM-style computation is notoriously nontrivial and often incurs significant overhead [GGH⁺16, WCS15, HHWW19, LMW23, CHJV15]. These considerations naturally raise a question:

Can the quantum one-time paradigm be extended beyond a single stateless evaluation to richer, stateful forms of controlled access?

¹If privacy is wanted, at least theoretically, we can use tools like fully-homomorphic encryption.

²Note that even using OTPs, it isn’t immediately clear how to prevent a malicious user from abusing this service by forking the inner state, which is created between different queries. In the classical solution from before this is not possible, since the server is keeping the state of itself, and gives the user only the answer.

³More precisely, the worst-case running time, taken over all inputs of some fixed size n .

Ideally, we would like to issue a quantum token where its size (and optimistically, even the computational work for generating it) is proportional to the actual hidden information, namely, the size of the program’s code, rather than depending on the amount of possible future computation. This is relevant for both; the above example of restricted query-access to an AI model, or generally issuing tokens for Turing Machines. To this end, we introduce the notion of *query-limited programs* (QLPs). A local query-limited program offers a qualitatively different guarantee: the user can evaluate the program *locally*, while the access policy is enforced by the program itself rather than by an always-online provider.

Leveraging One-Shot Signatures for One-Time Stateful Computation. A different line of work in quantum cryptography studies a particularly strong form of unclonability, embodied by *one-shot signatures* (OSS) [AGKZ20, SZ25a]. On one hand, OSS can be viewed as OTPs for a restricted class of functionalities – namely, signing algorithms, closely related to tokenized (one-time) signatures [BDS23]. On the other hand, OSS strengthen the no-cloning requirement in a striking way: given only a (classical) common reference string, the signing capability is a quantum state that is unclonable *even to the quantum algorithm that generates it*. In particular, any quantum polynomial-time generator cannot produce two usable signing states for the same public serial number. While the canonical OSS application is one-shot *signing*, the generator-resistant flavor of unclonability they capture is exactly the kind of “decentralized” control that seems necessary to push beyond stateless OTPs.

Along different axes, OTPs and OSS provide complementary guarantees: OTPs emphasize one-time obfuscated functionality, while OSS emphasize generator-resistant unclonability. In this work we explore whether the stronger unclonability inherent in OSS can be leveraged to fundamentally strengthen the *functionality* of quantum one-time programs. To this end we develop a unified framework that bridges these notions and show new functionalities enabled by their interaction. Along the way, we investigate their formal relationship and ask whether unclonable *signing* can serve as a generic mechanism for unclonable *general computation*. In particular, we investigate whether decentralized unclonable signing capability suggests a quantum capability can gate not just a single output, but a *sequence* of state transitions, without relying on trusted hardware or an online authority.

1.1 Results and Future Work

The results of this work consist of a chain of new notions, their construction and applications, which answer the above question positively. Our work has three main parts. The first two parts introduce two new notions; (1) one-shot programs and (2) query-limited programs, where the latter is built on the former. The third part of our work discusses examples of applications. We further elaborate below on our contributions.

First Part: Defining and Constructing One-Shot Programs. We begin by putting forth and defining the notion of *one-shot programs* (OSP), which upgrades quantum one-time programs by extending their no-cloning guarantee to that of one-shot signatures. At a high level, a one-shot program OSP scheme consists of two efficient algorithms; a classical algorithm PGen for compiling programs and a quantum algorithm TokenGen for generating tokens. Given a classical functionality f we can classically compile it $P_f \leftarrow \text{PGen}(f)$. The program P_f can be thought of as an ideal obfuscation of a function which depends on f (and executes it, conditioned on some initial verification procedure on the input). Then, TokenGen allows to sample one-time tokens $(|T\rangle, s) \leftarrow \text{TokenGen}$, which in turn consist of an unclonable state $|T\rangle$ and a classical serial number s . Correctness says that one can use P_f and the token to execute f on any chosen input x and sampled serial number s .

Informally, security guarantees that once the instance P_f is sampled honestly, it is computationally hard to execute f twice using the same serial number s , on any two inputs x, x' .

Some intuition behind one-shot programs. At a first glance, the move from OTPs to OSPs seems rather strange: the whole point of OTPs is to give one time access to a functionality. However, if an adversary can generate tokens $(|T\rangle, s) \leftarrow \text{TokenGen}(\text{pk})$ locally and on demand, it can effectively execute f many times on different values of x . The resolution to this, is in the fact that the function f acts on *both* parts; x and the serial number s . While x is to the choosing of the evaluator, s is not. To elaborate, the previous interface of functions used for OTPs was $f(x; r_x)$, where f is a randomized functionality, the evaluator chooses x , and r_x is an x -dependent randomness. Here, the interface is $f(x, s; r_x)$ where x, r_x are as before, but s is independently derived from the token generation. Intuitively, this exact combination seem to create our divergence from OTPs. On the one hand, we let the (possibly malicious) evaluator sample its own tokens. On the other hand, each token is necessarily tied to a serial number s , which in turn plays two roles: (1) it acts as a second input to f and thus influences its execution, and furthermore (2) enforces a "collision-resistance" property against the first part of its input, x , which the evaluator *does* have control over.

We formally define OSPs in Definition 4.2 (which is based on Definition 4.1), and show how to construct these generically from any one-shot signature [AGKZ20, SZ25a]. Furthermore, our proofs are modular in a way that creates a correspondence between one-time signing capabilities and one-time general computation. By essentially the same construction and security proof, we show:

- **One-Shot Programs from One-Shot Signatures:** In a classical oracle model, a one-shot program compiler can be constructed from any one-shot signature scheme (Theorem 4.3).
- **One-Time Programs from Signature Tokens:** In a classical oracle model, a one-time program compiler can be constructed from any quantum signature token scheme (Theorem 4.1).

Purely in the context of OTPs, the above improves our understanding: the previous work of [GLR⁺25] construct OTPs directly and by using techniques based on coset states. Coset states have a specific algebraic structure which the previous proofs use, and additionally, one can construct quantum signature tokens from them [BDS23, CLLZ21]. Here, we show that *any* quantum signature token scheme can be used to construct a OTP compiler, in particular, not relying on the algebraic structure of coset states. Finally, as an additional sanity check for our definition, we show that one-shot programs imply one-shot signatures in a classical oracle model (Section B.3). The oracles can all be heuristically instantiated efficiently using indistinguishability obfuscation and one-way functions.

Second Part: Defining and Constructing Query-Limited RAM Programs. We next define a new notion called Query-Limited RAM Programs (QLPs), and its security. As briefly explained in the introduction, a QLP compiler is a generalization of a one-time program compiler, in the sense that it allows the generation of a quantum one-time token for a stateful functionality. A QLP compiler consists of two efficient quantum algorithms; a program compiler $\tilde{P}_0 \leftarrow \text{PGen}(P)$ for a classical program P , and an evaluation algorithm $(y, \widetilde{P_{st'}}) \leftarrow \text{Eval}(x, \widetilde{P_{st}})$. The evaluation algorithm is given a classical input x and a quantum token $\widetilde{P_{st}}$ of a stateful program (here, the program is in "state st"). It outputs a classical y and advances the program's state. Similarly to OSPs, also here we give the (possibly malicious) evaluator delicate control: while it can choose x arbitrarily, the security guarantee of QLPs prevent it from changing the inner state in any illegal way. We formally define QLPs in Definition 5.1 and construct them in the classical oracle model. Specifically, we do this in the following steps.

1. **Introducing the Limited-Effective Query (LEQ) Oracle.** To define the security of QLP, we introduce the *limited-effective* query (LEQ) oracle, as a RAM-generalization of the previous *single-effective* query (SEQ) oracle. The SEQ oracle was introduced in [GLR⁺25] as part

of their resolution addressing barriers for constructing OTPs [BGS13]. Our new LEQ oracle plays a central role for realizing QLP compilers. In the technical overview we elaborate on its definition and the main ideas behind it.

2. **QLPs from OSPs.** We construct a Query-Limited Program compiler from a One-Shot Program compiler. We provide our construction and a formal analysis and security proof with respect to a classical oracle in Section 5 (Theorem 5.1), and explain the main ideas in our technical overview.

Third Part: Applications. Finally, we give preliminary results demonstrating the power of QLPs as an abstract primitive as well as our construction’s framework, through several new applications. First, as mentioned in the introduction, QLP compilers directly yield one-time programs for code/RAM programs and, more generally, k -time programs, which allow a possibly stateful program to be executed for up to k steps. Importantly, both the token size and, in our construction, its generation time, depend only polylogarithmically on k .

Pay-per-use (PPU) programs. An additional application we show is the ability to *retroactively* determine the bound k , which is captured by our pay-per-use (PPU) programs. We consider a setting where a user repeatedly evaluates a program and later proves (to a verifier) an upper bound on the *total cost* incurred. Each step of a PPU evaluation advances the internal state, which is kept hidden, and produces an output y . Separately, the user can run an explicit proof interface that yields a (classically verifiable) proof that the program’s accumulated cost equals (or is bounded by) c .

New forms of issuing code access. Both our k -time programs and pay-per-use programs enable new capabilities for delegating restricted access to computation while using minimal communication and computation. For example, both suggest, for the first time, different solutions for enabling Software-as-a-Service while keeping resources small, as was explained in our Introduction (Section 1).

Semi-quantum query-limited programs. Finally, following a sequence of works on semi-quantum public-key cryptographic primitives [Gav12, RS19, AGKZ20, Shm22a, Shm22b], we show how to construct a *semi-quantum* QLP compiler (Definition B.2). A semi-quantum QLP compiler can have a purely classical issuer that transfers the QLP restricted capability over a classical channel. Our construction does *not* transform any QLP generically into a semi-quantum version. Instead, we focus on an intermediate structure which is generically implied by one-shot programs in order to construct it.

Definition (specifics). A *semi-quantum* QLP compiler (cf. Definition B.2) augments (PGen, Eval) with a classical-message interactive protocol Transfer between a sender holding $\tilde{P}$ and a receiver, after which the receiver obtains a new program $\tilde{P}'$ that continues from the same logical RAM state. Correctness is required to hold for arbitrary polynomial-length *interleavings* of Eval operations and Transfer operations (where Transfer is modeled as a no-op on the ideal RAM state).

Theorem 1.1 (Informal: Semi-quantum QLP). From any QLP compiler based on OSP tokens, there exists a semi-quantum QLP compiler supporting a classical-message Transfer protocol (cf. Figure 16).

The protocol is conceptually simple: the receiver generates a fresh next-token serial number s' , the sender consumes its current token to execute a special “transfer” step that authenticates s' , and the receiver thereby obtains the updated classical state plus a fresh token enabling the next evaluation.

Future work. For future work, we can consider two directions.

Work	Security Notion	Assumptions and Tools	Applications	Classical Vendor?
[GLR ⁺ 25]	OTP, SEQ model, sim-based security	Classical oracle (or iO for weaker security), coset states	One-time PRFs, NIZKs, signatures	No, fully quantum
[GM24]	OTP, game-based (implied by [GLR ⁺ 25]’s security)	Classical oracle, signature tokens	Same [GLR ⁺ 25] (with corresponding security)	No, fully quantum
[Sta25a]	OTP, single-physical query ^a , sim-based	<i>Classically-accessible</i> classical oracle ^b , semi-quantum signature tokens	[Sta25a] RAM programs and applications (below)	Yes, classical vendor and communication
[Sta25a]	RAM programs, single-physical query, sim-based	<i>Classically-accessible</i> classical oracle, [Sta25a] OTP	Long-lived OTP and copy protection	Yes, classical vendor and communication
This work	OSP (OTP resp.), SEQ model, sim-based	Classical oracle, OSS (signature tokens resp.)	Same as [GLR ⁺ 25]; QLP and its applications (see below)	Yes, classical vendor and communication
This work	QLP, limited-effective query (LEQ) model, sim-based security	OSP, classical oracle, signatures	OTP for RAM/code, k -time programs, budget programs, pay-per-use programs, semi-quantum QLP, OSS	Yes, classical vendor and communication

^a As discussed in the introduction, a single physical query oracle allows literally one query. Un-computing and re-querying on the same input are not allowed. A single-effective query oracle allows multiple queries as long as its database records no more than one distinct query.

^b A classically-accessible oracle is an oracular interface that allows no superposition queries and measures any incoming query before answering. A classical oracle is a a black-box classical function that allows superposition queries.

Table 1: Comparison of our results with most relevant previous works. See Section 1.2 for detailed discussion.

1. Formulating the security definition for query-limited access for stateful/RAM programs where constructions exist in the plain model (without oracles), without running into impossibilities.⁴
2. Building on item 1, constructions in the plain model from indistinguishability obfuscation and more desirably, weaker standard assumptions, for generic or specialized functionalities .

1.2 Related Works

As mentioned in the introduction, [BGS13] put forward the definition for quantum one time programs for quantum functionalities and show a strong impossibility result. [GLR⁺25] proposes the SEQ oracle model and gets around [BGS13]’s impossibility result, giving a construction for all classical functions (meaningful only for high-entropy functions) in the classical oracle/ideal obfuscation model. [GLR⁺25] additionally proposes some weaker operational security definitions where they obtain one-time PRFs, NIZKs, signatures in the plain model from iO with respect to this operational security. [GM24] proposes a game-based security where a somewhat natural class of attackers cannot produce a second output as long as the functionality samples a high min-entropy output on every input. They construct one-time programs from any signature tokens under this definition. However, it does not rule out a more malicious attacker that can still produce multiple outputs; one

⁴Since OTP is a special case of our QLP, the impossibilities of realizing the SEQ simulation security in the plain model, as discussed in [GLR⁺25], apply to our QLP as well.

example is the impossibility of [GLR⁺25]. The follow-up works [ABG26, AB26] improve the assumptions in the plain model for OTP of PRFs and explores going from OTP to non-interactive MPC with preprocessing.

The work of [Sta25a] also propose RAM oracles similar to our QLP notion and semi-quantum OTPs from semi-quantum signature tokens [Shm22b], satisfying strong simulation-based security which allows only single-physical queries in the ideal world. The program vendor and communication can also be made classical. However, their constructions use classical measured oracles. A measured oracle is an oracle where adversaries' query registers have to be measured (in the computational basis) by the program interface upon querying, which gives a different model and relies on additional hardware assumptions. Their applications include OTP and semi-quantum copy protection with no need of long-term quantum memories, also in the classically-accessible measured oracle model. Besides, compared to the semi-quantum token based scheme in [Sta25a] where the program vendor has to communicate with every incoming user/receiver, our OSS based OSP and QLP has an additional "decentralized" feature that allows anyone to generate their own tokens and transfer programs to others via classical communication.

Apart from the above, [Sta25b] studies information-theoretic security for a specialized adversarial model called geometric local QNC₀ adversaries. [LSZ20] studies the security of classical one-time memory against quantum superposition attacks. [Liu23] constructs quantum one-time memory in the quantum random-oracle model with depth-bounded adversaries.

2 Technical Overview

In this section we explain the main ideas behind our new notions, their definitions and constructions. Before going into these details, we first recall the idea of one-time program (OTP) compilers, their security definition and construction.

2.1 Previous Work on Quantum One-Time Programs

We begin by recalling quantum one-time programs for classical randomized functionalities. Let $f : \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y}$. An OTP compiler consists of two efficient quantum algorithms,

$$\tilde{f} \leftarrow \text{PGen}(1^\lambda, f) , \quad y \leftarrow \text{Eval}(\tilde{f}, x) .$$

$\text{PGen}(1^\lambda, f)$ samples a quantum state $\tilde{f}$, and then $\text{Eval}(\tilde{f}, x)$ takes the quantum token and a later-chosen input x and computes a classical output y . Correctness requires that honest evaluation on any chosen input x produce an output statistically close to $f(x; r)$ over a uniformly random r . Importantly, the evaluator chooses x , but not r .

On the side of security, we would ideally like to enforce that "the quantum token $\tilde{f}$ gives exactly one-query access to f ". The work of Broadbent, Gutoski, and Stebila [BGS13] defined this as

$$\text{OTP}(f) \approx_c \text{Sim}^{f_1} ,$$

where Sim is a simulator and f_1 denotes a single-query access to f , which may be a superposition (coherent) query. In other words, the quantum token can be indistinguishably simulated by a simulator that has access to one physical query. This definition was formalized and subsequently proved impossible by [BGS13]. An issue observed by [BGS13] as part of their impossibility is that, for example, if f is a deterministic function, such definition cannot hold. This is due to the evaluator's ability to measure only the output, which, due to the determinism of f , will incur a non-disruptive measurement and keep the quantum token as good as new, allowing a second query. On the other hand, a one-time physical access to f by the simulator will not be able to answer a second query.

The Single-Effective Query Approach. A solution to this tension proposed in [GLR⁺25] is to give a definition referring to how much "effective information" overall was extracted from one token. Formally, the authors define the Single-Effective Query (SEQ) oracle (which gives more information than f_1), and then prove that access to the quantum token is no better than access to the oracle. Concretely, the SEQ oracle O_f^{SEQ} (defined in Figure 1) implements $f(x; H(x))$, where $H : \mathcal{X} \rightarrow \mathcal{R}$ is an internal random oracle represented using Zhandry's compressed-oracle technique [Zha18]. The use of compressed oracles allows us to not only simulate H , but to also monitor the queries made thus far, which we keep in a database register D . The register D is initially empty and remains supported on databases containing at most one entry (x, r) . On a query x , the SEQ oracle does nothing in branches where the database already records a different input $x' \neq x$. Otherwise, it coherently computes $r = H(x)$, XORs $f(x; r)$ into the output register, and uncomputes its temporary workspace.

To observe one clear point of difference between the SEQ setting and the single physical query setting of f_1 , consider what happens when the simulator queries on the same input x as the one recorded in the database. More generally, for an arbitrary f , consider a simulator which makes a superposition query in the form

$$\sum_x \alpha_x \cdot |x\rangle_{\text{input}} |u\rangle_{\text{output}} \rightarrow \sum_x \alpha_x \cdot |x\rangle_{\text{input}} |u + f(x)\rangle_{\text{output}} ,$$

without measuring the output. In the setting of f_1 , oracle access to the simulator now shuts down, while in the SEQ setting the simulator can query the SEQ oracle again (using the same two registers) to un-compute the output register and goes back to the state $\sum_x \alpha_x |x\rangle_{\text{input}} |u\rangle_{\text{output}}$.

SEQ security requires an efficient simulator that reproduces the encoded program using only this restricted oracle access. Informally, modulo security parameters and public auxiliary information, we ask for the computational indistinguishability

$$(f, \text{OTP}(f)) \approx_c \left(f, \text{Sim}_{O_f^{\text{SEQ}}} \right) .$$

Connection to the zero-knowledge simulation paradigm. Intuitively, the above somewhat follows the zero-knowledge simulation paradigm, where one separates the information which the interaction thinks of a private, from the actual information sent during the interaction. Given some minimal information (which we think of as public) only the latter needs to be simulated and be indistinguishable even given arbitrary side information. For example, in zero knowledge, the simulator has access to the verifier's code (which we think of as minimal information needed for simulation) and should simulate the real interaction with the prover. The indistinguishability should hold even given arbitrary independent side information, for example, even given a witness for the instance. Analogously, in the setting of OTPs, the simulator gets access to some minimal information (here, it is access to the SEQ oracle) and simulates a quantum state that's indistinguishable from the real quantum OTP token. Indistinguishability needs to stay intact even given the side information of the circuit of f .

Previous construction of One-Time Program Compilers. Previous work by [GLR⁺25] realizes SEQ security in the classical-oracle model using hidden subspace states. For simplicity, first consider a one-bit input $x \in \{0, 1\}$. The compiler samples a uniformly random subspace $A \subseteq \mathbb{F}_2^n$ of dimension $n/2$ and provides one copy of

$$|A\rangle = \frac{1}{\sqrt{|A|}} \sum_{u \in A} |u\rangle .$$

Writing $A^0 = A$ and $A^1 = A^\perp$, the program additionally provides access to a quantumly accessible classical oracle that acts as

$$|x, v, u\rangle \mapsto \begin{cases} |x, v, u \oplus f(x; G(v))\rangle , & v \in A^x \setminus \{0\}, \\ |x, v, u\rangle , & \text{otherwise,} \end{cases}$$

where G is an internal random oracle. To evaluate on $x = 0$, the evaluator uses $|A\rangle$ directly; for $x = 1$, it first applies a Hadamard transform, obtaining $|A^\perp\rangle$. Querying the oracle and measuring the output produces the desired sample, except with negligible correctness error. For an m -bit input, the construction uses one independent subspace state per bit, checks the resulting tuple $\mathbf{v} = (v_1, \dots, v_m)$ coordinate-wise, and derives the randomness as $G(\mathbf{v})$. So, the information of $\mathbf{v}$ contains the information of the input x , and this is used to derive the randomness r .

In a nutshell, the previous security proof shows that recording queries inside the SEQ oracle implies disturbance to these subspace states. In the real construction randomness is indexed by the vectors $\mathbf{v}$, while in the ideal functionality it is indexed by x . A coherent caching procedure bridges these representations. Finally, direct product hardness of subspace states implies the impossibility of finding any two vectors $\mathbf{v} := (v_1, \dots, v_m)$, $\mathbf{v}' := (v'_1, \dots, v'_m) \in \mathbb{F}_2^{n \cdot m}$ where there is some $i \in [m]$ and $b \in \{0, 1\}$ such that $v_i \in A^b$ and $v'_i \in A^{-b}$. This eventually rules out the recording of any two incompatible outputs.

2.2 One-Shot Programs

We next explain the definition and construction of our new primitive of a One-Shot Program (OSP) compiler, formally given in Definition 4.2. For the functions which are encoded by OSP compilers we consider f having a slightly different input interface. Before, f had a main input x and an auxiliary randomness input r . Here, the main input itself is divided into two parts, x and s :

$$f : \mathcal{S}_\lambda \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y} .$$

An OSP compiler consists of four algorithms:

$$\begin{aligned} \text{KeyGen}(1^\lambda) &\longrightarrow \text{crs} , \\ \text{PGen}(\text{crs}, f) &\longrightarrow P_f , \\ \text{TokenGen}(\text{crs}) &\longrightarrow (|T\rangle, s) , \\ \text{Eval}(P_f, s, x, |T\rangle) &\longrightarrow y . \end{aligned}$$

Here, P_f is classical and may include oracle access. The issuer compiles f , whereas the evaluator can generate tokens locally using only the public crs. These procedures are separate: program compilation does not require a token or its serial number. Correctness requires that evaluating with an honestly generated token produce a distribution statistically close to $f(s, x; r)$ for uniform r .

Security: single-effective query per serial number. We generalize the preceding SEQ oracle by giving it inputs (s, x) and using an internal random oracle $H(s, x)$ to generate the evaluation randomness. Its compressed database now records tuples (s, x, r) . On a query (s, x) , it acts as the identity in branches containing a record (s, x', r') with $x' \neq x$. Otherwise, it coherently computes $f(s, x; H(s, x))$ and uncomputes its workspace, exactly as before. The database may therefore contain many entries, but *at most one entry for each serial number*.

Security (Definition 4.1) requires that P_f , including its oracle interface, be simulatable using only this generalized SEQ access:

$$P_f \approx_c \text{Sim}^{O_f^{\text{SEQ}}}(\text{crs}, \text{aux}_f) .$$

The above indistinguishability should hold given public parameters, the description of f , and a common token and serial. The fact that indistinguishability holds even with this extra information, makes the security definition be at least as strong as arbitrarily simulating the quantum token (because the simulator can hold the key that allows to generate the quantum token).

As we briefly explained in Section 1.1, allowing public token generation may initially seem contrary to the purpose of one-time programs: an evaluator can generate many tokens and perform

many evaluations. The distinction is that *the serial number is an input to f* and influences its execution. A new token enables evaluation at its associated serial, while the security restriction applies separately to each serial. Thus, we seek one effective choice of x per s , rather than one effective evaluation overall. As a side note, this combination, of publicly generated tokens and serial-dependent functionality, is what we will later exploit for one-time stateful computation.

Construction from one-shot signatures. To get the right intuition for constructing OSP compilers, we recall the previous construction of OTP compilers given above, which uses subspace states. The subspace states in the preceding construction really play the role of one-time signature tokens: the input x specifies the message to sign, and the vectors supplied to the classical oracle constitute its signature. Here we will use signature tokens directly. Specifically, since we are dealing with one-shot program compilers rather than one-time program compilers, we will use one-shot signatures (Definition 3.6).

The construction uses the same "sign, then evaluate" structure of the previous OTP construction. Setup and token generation are inherited from a strongly unforgeable one-shot signature scheme OSS. To compile f , we sample an internal random oracle

$$G : \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \longrightarrow \mathcal{R} ,$$

and provide quantum access to the classical oracle

$$|s, x, \sigma, u\rangle \longmapsto \begin{cases} |s, x, \sigma, u \oplus f(s, x; G(s, x, \sigma))\rangle & \text{OSS.Ver(crs, } s, x, \sigma) = 1 \\ |s, x, \sigma, u\rangle & \text{otherwise} \end{cases}$$

An honest evaluator uses $|T\rangle$ to sign x , obtaining σ , and queries this oracle. Signature correctness and the independent randomness of G give the required output distribution. Importantly, the randomness is indexed by the signature as well as the serial and input.

One-shot programs security proof overview. The intuition for security is that the oracle evaluates f only when the query contains a valid signature σ on x under serial number s . However, ruling out the production of two signatures is not enough: simulation must reproduce arbitrary coherent queries, partial measurements, and un-computation. The main challenge is that the real-world adversary has access to the stateless oracle from our construction, whereas the simulator has access only to the stateful SEQ oracle (defined in Figure 2).

First, let us better understand the mismatch between the real and ideal worlds. The real oracle derives its randomness as $r = G(s, x, \sigma)$, whereas the ideal oracle uses $r = H(s, x)$. Recall that the SEQ oracle maintains a coherent database of records (s, x, r) , with at most one record per serial. On a query (s, x) , it does nothing in branches containing (s, x', r') with $x' \neq x$; otherwise, it coherently evaluates $f(s, x; H(s, x))$ and uncomputes its workspace. Records may be created or erased by these operations. We therefore want to reach a hybrid where the randomness indexing and the SEQ-style check can *ignore* σ , all at the same time without losing the correlations that the real oracle creates with the signature register.

Our proof builds on the coherent caching approach of [GLR⁺25], but must extend it from a single record to many records indexed by serial numbers. In their proof, the cache synchronizes a single tuple of subspace vectors with the one effective query recorded by the SEQ oracle. In our setting, many effective queries with distinct serial numbers may remain recorded simultaneously, and the adversary may query and un-compute across these serials in superposition. We therefore need a more general reversible cache that maintains the correspondence between each record (s, x, r) and its

signature information (s, x, σ) as records are created or erased. Together with a reversible database-splitting operation, this will allow us to remove σ from the ideal oracle's database without losing the information needed for un-computing. We do this in three main steps.

Step 1: Introducing an SEQ-style check. We first implement G as a compressed oracle (see Section 3.1), whose database records tuples (s, x, σ, r) . Our next step is to add a check that refuses a valid query (s, x, σ) whenever the database already records (s, x', σ', r') with $(x', \sigma') \neq (x, \sigma)$. If this changed the adversary's view noticeably, we could extract the current query and a conflicting database entry to obtain two distinct valid message-signature pairs under the same serial, contradicting strong unforgeability of the OSS. Thus, we may restrict the database to at most one recorded pair (x, σ) per serial.

Step 2: Separating the signature information reversibly. We now want the compressed database to record only (s, x, r) , as in the ideal oracle. Simply deleting σ would not preserve the joint quantum state. Instead, we introduce a quantum cache register C , initially empty, and a reversible change of representation Split. On databases whose serial numbers are all distinct, this operation separates each record into two matching parts:

$$\{(s_i, x_i, \sigma_i, r_i)\}_i \longleftrightarrow \left(\{(s_i, x_i, r_i)\}_i, \{(s_i, x_i, \sigma_i)\}_i \right).$$

The first component is the smaller database and the second is the cache. The transformation is reversible because the distinct serial numbers identify which entries belong together, allowing the original database to be reconstructed. Thus, signature information is *moved, not discarded*. We show that the compressed-oracle operations on the original database can be implemented using the smaller database together with reversible cache updates before and after evaluation. The cache records the signature associated with an outstanding evaluation, but allows that record to be erased when the evaluation is un-computed.

We can then make the SEQ-style check ignore σ as well. The old check rejects a different pair (x', σ') under the same serial; the new check rejects only a different input $x' \neq x$. The checks differ only when $x' = x$ but $\sigma' \neq \sigma$, with both signatures valid. A noticeable difference would again violate *strong* unforgeability, this time by producing two signatures on the same message. After this change, the smaller database and its check have exactly the form required by the ideal SEQ oracle.

Step 3: Implementing the cache using only SEQ access. It remains to implement the cache updates without reading the ideal oracle's private database. For each valid query (s, x, σ) , the simulator makes a coherent probe on $|s, \bar{x}, 0, 0\rangle$, where $\bar{x} = x \oplus 1 \neq x$ denotes x with a fixed bit flipped. The last register is the probe's answer bit b . If $b = 0$, the simulator toggles membership of (s, x, σ) in C , inserting it if absent and removing it if present; it then uncomputes the probe. This implements a coherent test for a record (s, x', r') with $x' \neq \bar{x}$. On branches where the main query is allowed, such a record can only be at x , so the probe supplies precisely the control needed for the cache. Neither the answer bit nor the database is measured.

The simulator performs this cache routine, makes the main SEQ query on $|s, x, u\rangle \otimes |+\rangle_B$, and performs the cache routine again. Here u is the adversary's output accumulator; using $|+\rangle_B$ for the unused answer target prevents it from retaining acceptance information. The cache routine is its own inverse, but the two applications need not cancel when the main query changes the database: their purpose is to track that change reversibly. On rejected branches, the database is unchanged and the two cache operations do cancel. The simulator therefore needs only public signature verification, its own cache, and SEQ access to f .

One-time Programs from Any Signature Tokens We also construct one-time programs from any signature tokens via a similar manner. The construction satisfies the simulation based SEQ security in [GLR⁺25] and meanwhile uses any signature token instead of the subspace coset states used in [GLR⁺25]. Note that [GM24] also constructs one-time programs from any signature tokens

but they prove it under a weaker security notion where the challenger will enforce measurements on the adversary's registers. We therefore achieve the best of both worlds proposed by these past works. The security proof resembles the one-shot programs security proof and we omit the discussions here.

2.3 Query-Limited RAM Programs

We are now ready to define and construct our query-limited programs for code/RAM functionalities, using one-shot programs.

RAM Programs. A randomized RAM program consists of a function $P : \mathcal{M} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{M} \times \mathcal{Y}$ and an initial memory state $\text{st}_1 = \perp$. Each evaluation takes the current state, a user-chosen input, and fresh randomness, and computes

$$P(\text{st}_i, x_i; r_i) = (\text{st}_{i+1}, y_i).$$

A query-limited program (QLP) compiler (Definition 5.1) consists of two efficient quantum algorithms, and encodes P into an object that supports these successive evaluations:

$$\tilde{P}_1 \leftarrow \text{PGen}(1^\lambda, P), \quad (y_i, \tilde{P}_{i+1}) \leftarrow \text{Eval}(\tilde{P}_i, x_i), \quad ,$$

where the encodings $\tilde{P}_i$ are quantum states. For every polynomial-length input sequence, correctness requires that the joint distribution of outputs from honest evaluation be statistically close to that of the underlying RAM program. The evaluator chooses the inputs, but does not directly choose the internal memory states.

The first thing to notice is that this interface captures simple access policies. For a k -time program, the state includes a counter recording how many evaluations have been completed, and the program rejects once the counter reaches k . For a more general budget-limited program, it instead tracks the accumulated cost and rejects a query that would exceed the budget. Even more generally, the state can contain the program's working memory, not merely an access counter.

Operational Security Definition. Before introducing our general security definition, consider the intuitive goal of preventing an evaluator from obtaining two incompatible executions. An execution has the form

$$\text{exe} = (((\text{st}_1, x_1, r_1), y_1), \dots, ((\text{st}_n, x_n, r_n), y_n)) \quad ,$$

where $P(\text{st}_i, x_i; r_i) = (\text{st}_{i+1}, y_i)$ at every step. We would like an adversary given $\text{QLP}(P)$ to be unable to produce two valid executions from the same initial state that differ at a common step. Of course, outputting one execution and a shorter prefix of it should not constitute an attack.

However, such an operational guarantee cannot hold for every family of programs. In particular, one can sometimes successfully obtain deterministic outputs while the corresponding computation is uncomputed. We therefore seek a simulation-based definition that captures the access provided by the encoded program, while leaving functionality-specific operational guarantees to a separate analysis, as in the SEQ framework of [GLR⁺25].

The Limited-Effective-Query (LEQ) Model for RAM Programs. Our ideal functionality is the LEQ oracle O_P^{LEQ} . Informally, a QLP compiler is secure if its entire encoded program can be simulated using only LEQ access:

$$\text{QLP}(P) \approx_c \text{Sim}^{O_P^{\text{LEQ}}}.$$

The comparison includes the program's oracle interfaces and any auxiliary information allowed by Definition 5.1. Sim can interact with the LEQ oracle by submitting queries (i, x_i) and receiving an

output y_i , which O_p^{LEQ} computes using the i 'th memory state st_i . We emphasize that the memory state st_i is *NOT* exposed to Sim at any point⁵.

The LEQ oracle meaningfully limits access to P while still faithfully modeling the fact that a real quantum adversary can run the program forward and then uncompute it (as long as it does not measure). In contrast to the SEQ oracle, which limits the evaluator to one “effective query”, the LEQ oracle allows the evaluator to repeatedly evaluate the program as the state updates. Instead, it limits the evaluator to one “effective execution path” by maintaining a superposition over totally ordered databases of queries and program states

$$D = \left\{ \begin{array}{l} 1 : (\text{st}_1, x_1, r_1, y_1), \\ 2 : (\text{st}_2, x_2, r_2, y_2), \\ \dots \\ i_{\max} : (\text{st}_{i_{\max}}, x_{i_{\max}}, r_{i_{\max}}, y_{i_{\max}}) \end{array} \right\}$$

which are internally consistent:

$$P(\text{st}_i, x_i; r_i) = (\text{st}_{i+1}, y_i) \quad \text{for all } i$$

Because the database is totally ordered, the adversary cannot simultaneously explore different branches of the program. If D contains $(i, \text{st}_i, x_i, r_i)$, then the adversary cannot also evaluate on a different (i, x'_i) , or evaluate the same (i, x_i) using a different program state st'_i .

Relation to the Operational Definition. We briefly pause to describe the relation of the general and operational definitions before continuing with the mechanics of maintaining the database. Suppose we could show for a particular class of programs that any execution the adversary finds must be recorded in the LEQ database. Then if the adversary found two executions, the LEQ database would contain two executions – against its definition.

Although we have not yet described *how* the LEQ database records queries, there are two intuitive criteria for when knowledge of an execution implies its presence in the database. First, the adversary should not be able to “forge” query responses by computing them without actually querying the LEQ oracle. Second, each query’s response should be sufficiently randomized, or else the program state will not change significantly by measuring the query’s response and uncomputing the query. The precise conditions are quite similar to those for the SEQ oracle and we refer the reader to [GLR⁺25] for an in-depth discussion.

Definition: Recording Queries in O_p^{LEQ} . . The main challenge of defining O_p^{LEQ} is the mechanism for recording an “effective execution path” corresponding to the evaluations that the evaluator knows. O_p^{LEQ} will maintain a register $\mathcal{S}$ containing the current state of the program and a sequence of ancilla registers $\mathcal{A}_1, \dots, \mathcal{A}_k$ corresponding to the individual steps of the computation. When it receives a basis state query $|i, x, u\rangle$, it will use $\mathcal{A}_i$ as a workspace to answer the query. If $|i, x, u\rangle$ is an “effective query” then (i, x_i) should end up recorded in $\mathcal{A}_i$. As an overly simplified view, we can imagine the database as a sequence of memory states

$$\text{st}_1 \rightarrow \text{st}_2 \rightarrow \text{st}_3 \rightarrow \dots$$

corresponding to the evaluations that the evaluator knows.

[GLR⁺25] already gives a solution to recording a single effective query by using Zhandry’s compressed oracle [Zha19]. Indeed, a natural approach to the LEQ oracle is to imagine internally running a sequence of SEQ oracles for each recorded i ; when the i 'th SEQ oracle records a new query, create

⁵This notion does not conflict with some applications where the users may want to read part of the memory state information (e.g. remaining queries, budget) where we can put this information in the output.

a new SEQ oracle for $i + 1$. However, maintaining an ordered database of queries requires more care. There are two issues we need to address: 1) maintaining an unbroken chain of memory states and 2) preventing the oracle from “getting stuck”.

Definition: Maintaining order. First, we need to ensure that the database contains a complete, ordered history of the execution with no gaps. The subtlety is that a quantum adversary can always erase its own memory of an evaluation by making another evaluation, as observed by [BGS13, GM24, GLR⁺25]. If we permit an evaluator to make a second query to st_1 , for example, it might *erase its own memory of evaluating on st_1* , which would cause the 1st SEQ oracle to in turn *erase the record of st_2 in D* . Now there is a gap in the database:

$$st_1 \rightarrow ??? \rightarrow st_3 \rightarrow \dots$$

Preventing gaps is as simple as restricting the adversary from accessing middle memory states. In other words, O_p^{LEQ} provides access to D as a stack. The evaluator has two options for its queries: either repeat the most recent query $(i, x_{i_{\max}})$, or make a new query $(i_{\max} + 1, x_{i_{\max}+1})$ to be evaluated using the next memory state $st_{i_{\max}+1}$ (obtained from $P(st_{i_{\max}}, x_{i_{\max}}; r_{i_{\max}})$). They are not allowed to access the middle of the stack, ensuring that the database maintains a complete history of the computation so far.

Of course, an evaluator who has access to the real $QLP(P)$ can always re-evaluate middle queries. Ensuring indistinguishability from the LEQ oracle which forbids middle queries becomes an important step in proving any QLP construction secure.

Definition: Preventing Sticking. Second, there is a subtle issue with implementing D naively using a compressed oracle: it can get stuck after a deterministic query. Suppose that an evaluator honestly makes subsequent queries to x_1 , then x_2 . They expect the second query to use memory state st_2 , which may depend on x_1 . However, a key feature of [GLR⁺25]’s SEQ oracle is that it does *not* record “gentle” queries. If P were deterministic, then x_1 is never recorded and O_p^{LEQ} would answer the second query x_2 using the *first* memory state st_1 .⁶ Since Sim can never learn the correct response to the second query, a naive implementation of D results in an unrealizable security notion.

To ensure that LEQ security is possible to achieve, we implement the database D differently.⁷ Instead, we implement D using ideas from [GLQ⁺26]’s recent generalization of one-time programs to quantum functionalities. To answer a query to $|i, x_i, u\rangle$, the LEQ oracle swaps $\mathcal{A}_i$ between a fully un-used state $|0^n, 0\rangle$ and a fresh run of P_{OSP} :

$$|i, x_i, u\rangle_Q \otimes (|st_i\rangle_S \otimes |0^{n+1}\rangle_{\mathcal{A}_i}) \leftrightarrow \sum_{r_i} |i, x_i, u \oplus y_{i,r_i}\rangle_Q \otimes (|st_{i+1}\rangle_S \otimes |(st_i, x_i, r_i), 1\rangle_{\mathcal{A}_i})$$

where $(st_{i+1}, r_i, y_{i,r_i}) = P(st_i, x_i; r)$ and n is a placeholder length. Essentially, this approach strips out the compressed oracle’s final attempt to re-compress the database after each query, allowing x_i to be recorded and the computation to continue even if y_i ignored r_i .

Construction. Our construction is conceptually simple: give the evaluator a one-shot program for the functionality P_{OSP} that

⁶For example, P might be a deterministic counting program where the program outputs $y_i = i$, maintaining state $st_i = i$ and ignoring x_i . Here, the evaluator would expect outputs 1, then 2. If we implement D naively for O_p^{LEQ} , then any simulator would only see 1 forever and not know how to properly answer the second query.

⁷A particularly simple solution is to append a truly random value $r_2 \in \{0, 1\}^\lambda$ to P ’s output. Now if an honest evaluator (or the simulator) wants to avoid getting stuck, it just has to remember r_2 . However, appending r_2 is unsatisfactory because then the ideal functionality depends on the security parameter λ . Despite this, this idea provides a useful conceptual stepping-stone for our proofs.

1. allows them to use a token $|T_i\rangle$ to evaluate $(st_{i+1}, y_i) = P(st_i, x_i; r_i)$ if they have a signature σ on the memory state st_i together with the matching serial number s_i ,
2. then signs the new state st_{i+1} together with the next serial number s_{i+1} .⁸

Explicitly, the evaluator chooses the next serial number s_{i+1} , then evaluates the OSP on input $(x_i, st_i, \sigma_i, s_{i+1})$ using serial number s_i . Additionally providing them with a signature on a starting serial number and initial memory state $s_1 \| st_1$ allows the evaluator to kick-start its computation.

At any time the evaluator only holds a *single* valid chain element (st_i, σ_i) tied to the current serial number s_i ; advancing to the next step requires consuming the current one-shot token $|T_i\rangle$ and obtaining from P_{OSP} a new signature σ_{i+1} on a fresh serial s_{i+1} . Thus, any attempt to fork the computation (e.g., reuse the same state under two different next inputs) would force either reusing the same one-shot serial s on two distinct effective inputs, or forging a fresh signature on some new (s_{i+1}, st_{i+1}) -message.

To prevent forging a signature for a forked computation, we use [GLR⁺25]’s one-time signature construction: to sign m using randomness r , sign $m \| r \| G_{\text{Sign}}(m \| r)$ where G_{Sign} is a random oracle. We show that their construction also gives useful guarantees for stateful programs where the OSP signs different messages depending on st_i .

Security: Overview. To prove security, we need to lift the one-time guarantees provided by the OSP to a stateful program. The SEQ oracle for the OSP from the construction maintains a database containing many queries $(i, x_i, st_i, \sigma_i, s_{i+1})$, along with the associated serial number s_i , and the answer y_i and a signature σ_{i+1} associated the next serial number s_{i+1} with the next memory state st_{i+1} . Ultimately, we need to enforce that the database is *ordered* and *consistent*:⁹

$$D = \left\{ \begin{array}{l} [\text{serial} = s_1, \text{ input} = (1, x_1, st_1, \sigma'_1, s_2), \text{ output} = (y_1, \sigma_2), \text{ state} = st_2], \\ [\text{serial} = s_2, \text{ input} = (2, x_2, st_2, \sigma'_2, s_3), \text{ output} = (y_2, \sigma_3), \text{ state} = st_3], \\ \dots \end{array} \right\}$$

Note that the above ordering asks that the s_2 and st_2 recorded in the first row exactly match those in the second row, establishing continuity of the computation.

To prove security, we need to 1) establish this consistent ordering and 2) translate between the evaluator’s queries (which are recorded directly in the SEQ oracle) and the LEQ oracle’s interface. Although prior works provide a useful starting point, the greater structure of the LEQ oracle requires new ideas.

Ordering the Database: Dynamically Constrained Signatures. The evaluator can compute on a program state st if and only if it has a signature σ on st together with a serial number s . Intuitively, the evaluator’s behavior can be controlled by constraining the signatures to a certain set. If the program were stateless, then this is sufficient. However, for RAM programs, the kinds of messages the program is willing to sign *change* as the program is queried.

The key to dynamically constraining signatures lies in the structure of entry 2’s input signature σ'_2 : it contains some randomness $r'_{2,\text{Sign}} = r'_2 \| G_{\text{Sign}}(r'_2 \| m)$, along with a baseline signature on $m \| r'_{2,\text{Sign}}$. We use this structure to show that entry 1’s output signature σ_2 also contains the *same* randomness $r'_{2,\text{Sign}}$, which is only possible if the same s_2 and st_2 were also recorded in the first entry.

⁸In the construction, we hide the memory from the adversary states st_{i+1} by encrypting them, so that we achieve a stronger security. We omit this step here for simplicity of exposition.

⁹This database has been simplified for the purposes of this overview.

We show this in two steps. First, we argue that knowledge of σ_{i+1} implies that G_{Sign} records s_{i+1} , st_{i+1} , and the randomness used by the OSP for the first query. Second, we strengthen the compressed oracle chaining lemma from [GLR⁺25] to establish a strong functional connection from the signed message to the first entry. As compared to their original chaining lemma, the strengthened version has much greater flexibility in the relation between the SEQ oracle's database entry and the G_{Sign} entry. We refer the reader to Section 5.3.1 for more details.

Translating Between SEQ and LEQ. The final step is to translate between the SEQ oracle, which handles all the cryptographic tools used to enforce consistent computation, and the LEQ oracle, which purely deals with the program functionality. The translation is handled via a carefully designed cache system. For SEQ security, it was sufficient to just cache the evaluator's inputs, but more is necessary for proving LEQ security.

The primary difference between the SEQ oracle and the LEQ oracle is the way that the LEQ enforces a consistent history by providing access only to the most recent query i or a new query $i + 1$, like a stack. It doesn't answer old queries $i - 1$, even though a real evaluator would certainly be able to re-evaluate old queries. To match the evaluator's view, the simulator needs to answer old queries, which necessitates *also* caching the output of each query. However, the input cache *depends* on whether the output is known outside of the LEQ oracle. Ensuring that the output cache does not interfere with the input cache requires some additional care. We refer the reader to Section 5.3 for more details.

2.4 Applications

As we have discussed, the notion of query-limited RAM programs already captures applications such as k -time program and programs with a prefixed budget. Here, we demonstrate more applications.

Pay-Per-Use Programs We propose the following paradigm. A pay-per-use program can be queried any number of times, and afterward, the user pays for the cost of the queries that they made. At the time of the payment, the user has to provide a proof that its previous *effective* queries (queries whose information is successfully obtained; uncomputed queries should not count) actually cost the value c as they claim. Soundness is defined via effective queries, formalized through the REQ oracle: after using the program, the user gives a challenger its claimed cost c and a proof; the challenger first interacts with the user to verify the validity of the the proof and if the proof is valid, the challenger then reads from the database of the LEQ oracle to check if it is equal to c .

In our construction, the RAM program has two modes: the regular Eval mode that evaluates functionality P ; and a Prove mode that outputs the cost c' , a random string r_1 , and a signature on (c', r_1) . During the cost verification protocol, the verifier will check if the alleged cost matches the signed message and the validity of the signature. After one uses the RAM program in Prove mode, it will enter a Halt mode that rejects to answer anything. This prevents the user from a "prove-then-query-more" attack.

Semi-Quantum Query-Limited Programs A semi-quantum query-limited programs allows one user to transfer the capability of evaluating a program to another user via only *classical communication*. In addition to the original RAM program syntax, the semi-quantum query-limited programs have a Transfer algorithm that lets a sender send a program to a receiver via classical interactions. Our construction also consists of all components in the RAM program construction, with an additional Transfer protocol.

The Transfer protocol roughly works as follows: the receiver generates a token $|T'\rangle$ and corresponding serial number s' after receiving the crs from the sender; the sender, after getting the receiver's serial number s' , evaluates the program in Transfer mode with a dummy input (e.g. a

0-string) using its own token $|T\rangle$ and the new serial s' . The sender then sends the output of the Transfer evaluation, the classical part of the program (i.e. the classical oracle) to the receiver. After the transfer, the sender's token will be consumed and the program the receiver obtains will work correctly under its new token $|T'\rangle$. The receiver will acquire a program that inherits the sender's program's RAM state. In particular, the receiver will be able to evaluate the "remaining queries" in the program. Suppose the original program is a k -time program and the sender has used k' queries, then the receiver will obtain a $(k - k')$ -time program. Similar scenario applies to programs with budget/cost policies.

One-Shot Signatures from Blind-Unforgeable Signature Scheme Finally, we show that OSP can be used not only as an input primitive (via OSS) but also as an *amplifier* that upgrades standard signature assumptions into one-shot signatures with more flexible signature formats.

Given a blind-unforgeable signature scheme BUS, which one can obtain from post-quantum subexponentially secure signature schemes and a one-shot program compiler OSP, there exists a one-shot signature scheme OSS whose verification algorithm is essentially BUS's verification. Here the one-shot program implements a signing functionality for BUS keyed by a serial number; OSP security enforces one-shotness per serial number, while blind-unforgeability supplies the underlying post-quantum unforgeability needed for the resulting signature format.

3 Preliminaries

A distribution over oracles (a.k.a. an oracle distribution) samples parameters pp from a given distribution and then implements the oracle O_{pp} defined by those parameters. Next, let us define the indistinguishability of two oracle distributions. Two oracle distributions are **perfectly indistinguishable** if any *computationally unbounded* distinguisher, given quantum query access to an oracle drawn from one of the distributions, has 0 advantage at distinguishing the two distributions. Next, two oracle distributions are **computationally indistinguishable** if any *QPT* distinguisher, given quantum query access to an oracle drawn from one of the distributions, has $\text{negl}(\lambda)$ advantage at distinguishing the two distributions.

We use $*$ to denote wildcards in strings. For example, $x_1||*$ denotes any state from the set $\{x_1||x_2\}_{x_2}$.

3.1 Compressed Oracles

Compressed oracles are a technique for implementing a random oracle that allow the oracle to record queries [Zha18]. Essentially, the oracle purifies its randomness, maintaining a superposition over functions in the distribution and responding to queries by operating coherently on this superposition.

Database Register D : Let H be a random oracle mapping $\mathcal{X} \rightarrow \mathcal{Y}$. The compressed oracle has a database register D that stores a partial truth table d for the function. d is a set of tuples of the form $(x, y) \in \mathcal{X} \times \mathcal{Y}$. Every tuple in d has a unique x -value. If $(x, y) \in d$, we say that $x \in d$ (slightly abusing notation) and $d(x) = y$. Alternatively, for a given x , if there exists no y such that $(x, y) \in d$, we say that $x \notin d$ and $d(x) = \perp$.

Next, let us define a useful eigenbasis for register D . For any $x \notin d$, any $\tilde{y} \in \mathcal{Y}$,

$$\text{let } |\psi(d, x, \tilde{y})\rangle_D = \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |d \cup \{(x, y)\}\rangle_D \quad (1)$$

For any fixed $x \in \mathcal{X}$, we will define an eigenbasis as follows. Our eigenbasis will include $|d\rangle$ and $|\psi(d, x, \tilde{y})\rangle$ for every database d such that $x \notin d$ and every $\tilde{y} \in \mathcal{Y}$.

Subroutines: The compressed oracle uses the following subroutines: $\{\text{Decomp}_x\}_{x \in \mathcal{X}}$, Decomp , CO' . They are unitaries that operate on the database register D and sometimes also on a query register Q , which stores values of the form $(x, y) \in \mathcal{X} \times \mathcal{Y}$.

- $\{\text{Decomp}_x\}_{x \in \mathcal{X}}$: For a given $x \in \mathcal{X}$, let d be a database such that $x \notin d$, and let $\tilde{y} \in \mathcal{Y} \setminus \{0\}$.

$$\begin{aligned} |d\rangle &\xrightarrow{\text{Decomp}_x} |\psi(d, x, 0)\rangle \\ |\psi(d, x, 0)\rangle &\xrightarrow{\text{Decomp}_x} |d\rangle \\ |\psi(d, x, \tilde{y})\rangle &\xrightarrow{\text{Decomp}_x} |\psi(d, x, \tilde{y})\rangle \end{aligned}$$

- Decomp : Let $(x, y) \in \mathcal{X} \times \mathcal{Y}$, and let d be a database.

$$|x, y\rangle_Q \otimes |d\rangle_D \xrightarrow{\text{Decomp}} |x, y\rangle_Q \otimes (\text{Decomp}_x |d\rangle_D)$$

- CO' : Let $(x, y) \in \mathcal{X} \times \mathcal{Y}$, and let d be a database such that $x \in d$. Then CO' acts as follows.

$$|x, y\rangle_Q \otimes |d\rangle_D \xrightarrow{\text{CO}'} |x, y \oplus d(x)\rangle_Q \otimes |d\rangle_D$$

CO' acts as the identity on all states orthogonal to the ones above.

Compressed Oracle: Now we can describe how the compressed oracle operates. The D register is initialized to $|\emptyset\rangle$. On each query, the user submits a query register $Q = Q_x \times Q_y$. Then the oracle applies the following sequence of operations to $Q \times D$:

$$\text{CO} := \text{Decomp} \circ \text{CO}' \circ \text{Decomp}$$

Finally, the Q register is returned to the user.

A Useful Eigenbasis: Let us define a useful basis for $Q_x \times Q_y \times D$. The basis is easy to work with because queries to the compressed oracle simply permute these basis vectors. First, Q_x (which stores the x -value of Q) will be in the computational basis. Second, Q_y (which stores the y -value of Q) will be in the Fourier basis. The Fourier basis comprises the following basis vectors. For any $\tilde{y} \in \mathcal{Y}$,

$$\text{let } |\phi(\tilde{y})\rangle_{Q_y} = \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |y\rangle_{Q_y} \quad (2)$$

Third, the basis for D will be the one defined above (eq. (1)).

In total, the basis for $Q_x \times Q_y \times D$ comprises all vectors of the form

$$\begin{aligned} &|x\rangle_{Q_x} |\phi(\tilde{y})\rangle_{Q_y} |d\rangle_D, \text{ or} \\ &|x\rangle_{Q_x} |\phi(\tilde{y})\rangle_{Q_y} |\psi(d, x, \tilde{y}')\rangle_D \end{aligned} \quad (3)$$

where $x \in \mathcal{X}$, $\tilde{y}, \tilde{y}' \in \mathcal{Y}$, d is a database s.t. $x \notin d$

Lemma 3.1. The basis vectors in eq. (3) are an orthonormal basis for $Q_x \times Q_y \times D$.

Proof. First, the vectors $\{|\phi(\tilde{y})\rangle\}_{\tilde{y} \in \mathcal{Y}}$ are orthonormal. For any $\tilde{y}, \tilde{y}' \in \mathcal{Y}$,

$$\begin{aligned} \langle \phi(\tilde{y}) | \phi(\tilde{y}') \rangle &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y, y' \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} \cdot (-1)^{\langle y', \tilde{y}' \rangle} \langle y | y' \rangle \\ &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle + \langle y, \tilde{y}' \rangle} = \frac{1}{|\mathcal{Y}|} \cdot \sum_{y \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} + \tilde{y}' \rangle} \\ &= \frac{1}{|\mathcal{Y}|} \cdot |\mathcal{Y}| \cdot \mathbb{1}_{\tilde{y} + \tilde{y}' = 0} \\ &= \mathbb{1}_{\tilde{y} = \tilde{y}'} \end{aligned}$$

We used the fact that when $\tilde{y} + \tilde{y}' \neq 0$, then $\langle y, \tilde{y} + \tilde{y}' \rangle = 0$ for exactly half of the y -values in $\mathcal{Y}$, and $\langle y, \tilde{y} + \tilde{y}' \rangle = 1$ for the other half. We also used the fact that $\tilde{y} + \tilde{y}' \neq 0$ if and only if $\tilde{y} = \tilde{y}'$. This is because we are working over the additive group of binary strings, so every vector is its own inverse.

Second, $|d\rangle$ is orthogonal to $|\psi(d, x, \tilde{y}')\rangle$ for any $x \in \mathcal{X}$, $\tilde{y}' \in \mathcal{Y}$, and any database d for which $x \notin d$. This is because d does not contain x , but $|\psi(d, x, \tilde{y}')\rangle$ is a superposition only over databases that contain x . $|\psi(d, x, \tilde{y}')\rangle$ is a superposition over databases of the form $|d \cup \{(x, y)\}\rangle$ for some $y \in \mathcal{Y}$. Such databases do contain x .

Third, the vectors $\{|\psi(d, x, \tilde{y})\rangle\}_{d, x, \tilde{y}}$ (where $x \notin d$) are orthonormal. For any $\tilde{y}, \tilde{y}' \in \mathcal{Y}$,

$$\begin{aligned} \langle \psi(d, x, \tilde{y}) | \psi(d, x, \tilde{y}') \rangle &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y, y' \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} \cdot (-1)^{\langle y', \tilde{y}' \rangle} \langle d \cup \{(x, y)\} | d \cup \{(x, y')\} \rangle \\ &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle + \langle y, \tilde{y}' \rangle} \\ &= \mathbb{1}_{\tilde{y}=\tilde{y}'} \end{aligned}$$

Fourth, for any $y \in \mathcal{Y}$, $|y\rangle$ can be written as a linear combination of $|\phi(\tilde{y})\rangle$ vectors as follows:

$$|y\rangle = \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |\phi(\tilde{y})\rangle$$

This is because

$$\begin{aligned} \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |\phi(\tilde{y})\rangle &= \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} \cdot \left(\frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y' \in \mathcal{Y}} (-1)^{\langle y', \tilde{y} \rangle} |y'\rangle \right) \\ &= \sum_{y' \in \mathcal{Y}} |y'\rangle \cdot \left(\frac{1}{|\mathcal{Y}|} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y+y', \tilde{y} \rangle} \right) \\ &= \sum_{y' \in \mathcal{Y}} |y'\rangle \cdot \left(\frac{1}{|\mathcal{Y}|} \cdot |\mathcal{Y}| \cdot \mathbb{1}_{y+y'=0} \right) = \sum_{y' \in \mathcal{Y}} |y'\rangle \cdot \mathbb{1}_{y=y'} \\ &= |y\rangle \end{aligned}$$

Fifth, for any $(x, y) \in \mathcal{X} \times \mathcal{Y}$ and any database d that does not contain x , the combined database $|d \cup \{(x, y)\}\rangle$ can be written as a linear combination of $|\phi(d, x, \tilde{y})\rangle$ vectors as follows:

$$|d \cup \{(x, y)\}\rangle = \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |\phi(d, x, \tilde{y})\rangle$$

This is because

$$\begin{aligned} \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |\phi(d, x, \tilde{y})\rangle &= \frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} \cdot \left(\frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y' \in \mathcal{Y}} (-1)^{\langle y', \tilde{y} \rangle} |d \cup \{(x, y')\}\rangle \right) \\ &= \sum_{y' \in \mathcal{Y}} |d \cup \{(x, y')\}\rangle \cdot \left(\frac{1}{|\mathcal{Y}|} \cdot \sum_{\tilde{y} \in \mathcal{Y}} (-1)^{\langle y+y', \tilde{y} \rangle} \right) \\ &= \sum_{y' \in \mathcal{Y}} |d \cup \{(x, y')\}\rangle \cdot \mathbb{1}_{y=y'} = |d \cup \{(x, y)\}\rangle \end{aligned}$$

Sixth, the previous parts imply that the vectors in eq. (3) each have unit norm, and the vectors are orthogonal to each other. Furthermore, any computational basis eigenstate $|x\rangle_{Q_x} |y\rangle_{Q_y} |d\rangle_D$ can be written as a linear combination of the basis vectors. $\square$

The next lemma says that a query to the compressed oracle just permutes the basis vectors of eq. (3), as long as $\tilde{y}' \neq 0$, and furthermore, the query only modifies the value of the D register, when viewed in this basis.

Lemma 3.2. For any $x \in \mathcal{X}$, any $\tilde{y}, \tilde{y}' \in \mathcal{Y}$ such that $\tilde{y}' \neq 0$, and any database d such that $x \notin d$,

$$\begin{aligned} |x\rangle_{Q_x} |\phi(0)\rangle_{Q_y} |d\rangle_D &\xrightarrow{\text{CO}} |x\rangle_{Q_x} |\phi(0)\rangle_{Q_y} |d\rangle_D \quad (\tilde{y} = 0) \\ |x\rangle_{Q_x} |\phi(\tilde{y})\rangle_{Q_y} |d\rangle_D &\xrightarrow{\text{CO}} |x\rangle_{Q_x} |\phi(\tilde{y})\rangle_{Q_y} |\psi(d, x, \tilde{y})\rangle_D \quad (\text{if } \tilde{y} \neq 0) \\ |x\rangle_{Q_x} |\phi(\tilde{y}')\rangle_{Q_y} |\psi(d, x, \tilde{y}')\rangle_D &\xrightarrow{\text{CO}} |x\rangle_{Q_x} |\phi(\tilde{y}')\rangle_{Q_y} |d\rangle_D \quad (\tilde{y} = \tilde{y}') \\ |x\rangle_{Q_x} |\phi(\tilde{y})\rangle_{Q_y} |\psi(d, x, \tilde{y}')\rangle_D &\xrightarrow{\text{CO}} |x\rangle_{Q_x} |\phi(\tilde{y})\rangle_{Q_y} |\psi(d, x, \tilde{y} + \tilde{y}')\rangle_D \quad (\text{if } \tilde{y} \neq \tilde{y}') \end{aligned}$$

Proof. First, let us examine how CO' acts. For any $x \in \mathcal{X}$, any $\tilde{y}, \tilde{y}' \in \mathcal{Y}$, and any database d that does not contain x , CO' maps

$$|x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}')\rangle \xrightarrow{\text{CO}'} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}' + \tilde{y})\rangle$$

This is because:

$$\begin{aligned} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}')\rangle &= |x\rangle \left(\frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle} |y\rangle \right) \left(\frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y' \in \mathcal{Y}} (-1)^{\langle y', \tilde{y}' \rangle} |d \cup \{(x, y')\}\rangle \right) \\ &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y, y' \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle + \langle y', \tilde{y}' \rangle} \cdot |x\rangle |y\rangle |d \cup \{(x, y')\}\rangle \\ &\xrightarrow{\text{CO}'} \frac{1}{|\mathcal{Y}|} \cdot \sum_{y, y' \in \mathcal{Y}} (-1)^{\langle y, \tilde{y} \rangle + \langle y', \tilde{y}' \rangle} \cdot |x\rangle |y + y'\rangle |d \cup \{(x, y')\}\rangle \\ &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y', y'' \in \mathcal{Y}} (-1)^{\langle y'' - y', \tilde{y} \rangle + \langle y', \tilde{y}' \rangle} \cdot |x\rangle |y''\rangle |d \cup \{(x, y')\}\rangle \quad \text{where } y'' = y + y' \\ &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y', y'' \in \mathcal{Y}} (-1)^{\langle y'', \tilde{y} \rangle + \langle y', \tilde{y}' - \tilde{y} \rangle} \cdot |x\rangle |y''\rangle |d \cup \{(x, y')\}\rangle \\ &= \frac{1}{|\mathcal{Y}|} \cdot \sum_{y', y'' \in \mathcal{Y}} (-1)^{\langle y'', \tilde{y} \rangle + \langle y', \tilde{y}' + \tilde{y} \rangle} \cdot |x\rangle |y''\rangle |d \cup \{(x, y')\}\rangle \\ &= |x\rangle \left(\frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y'' \in \mathcal{Y}} (-1)^{\langle y'', \tilde{y} \rangle} |y''\rangle \right) \left(\frac{1}{\sqrt{|\mathcal{Y}|}} \cdot \sum_{y' \in \mathcal{Y}} (-1)^{\langle y', \tilde{y}' + \tilde{y} \rangle} |d \cup \{(x, y')\}\rangle \right) \\ &= |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}' + \tilde{y})\rangle \end{aligned}$$

We used the fact that $\tilde{y} = -\tilde{y}$ because we are working over the additive group of binary strings.

Second, let us examine how CO acts on $|x\rangle |\phi(0)\rangle |d\rangle$ (where $\tilde{y} = 0$).

$$\begin{aligned} |x\rangle |\phi(0)\rangle |d\rangle &\xrightarrow{\text{Decomp}} |x\rangle |\phi(0)\rangle |\psi(d, x, 0)\rangle \\ &\xrightarrow{\text{CO}'} |x\rangle |\phi(0)\rangle |\psi(d, x, 0)\rangle \\ &\xrightarrow{\text{Decomp}} |x\rangle |\phi(0)\rangle |d\rangle \end{aligned}$$

Third, let us examine how CO acts on $|x\rangle |\phi(\tilde{y})\rangle |d\rangle$ when $\tilde{y} \neq 0$.

$$\begin{aligned} |x\rangle |\phi(\tilde{y})\rangle |d\rangle &\xrightarrow{\text{Decomp}} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, 0)\rangle \\ &\xrightarrow{\text{CO}'} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y})\rangle \\ &\xrightarrow{\text{Decomp}} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y})\rangle \end{aligned}$$

Fourth, let us examine how CO acts on $|x\rangle |\phi(\tilde{y}')\rangle |\psi(d, x, \tilde{y}')\rangle$ (where $\tilde{y} = \tilde{y}'$ and $\tilde{y}' \neq 0$).

$$\begin{aligned} |x\rangle |\phi(\tilde{y}')\rangle |\psi(d, x, \tilde{y}')\rangle &\xrightarrow{\text{Decomp}} |x\rangle |\phi(\tilde{y}')\rangle |\psi(d, x, \tilde{y}')\rangle \\ &\xrightarrow{\text{CO}'} |x\rangle |\phi(\tilde{y}')\rangle |\psi(d, x, 0)\rangle \\ &\xrightarrow{\text{Decomp}} |x\rangle |\phi(\tilde{y}')\rangle |d\rangle \end{aligned}$$

Fifth, let us examine how CO acts on $|x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}')\rangle$ when $\tilde{y} \neq \tilde{y}'$ (and $\tilde{y}' \neq 0$).

$$\begin{aligned} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}')\rangle &\xrightarrow{\text{Decomp}} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y}')\rangle \\ &\xrightarrow{\text{CO}'} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y} + \tilde{y}')\rangle \\ &\xrightarrow{\text{Decomp}} |x\rangle |\phi(\tilde{y})\rangle |\psi(d, x, \tilde{y} + \tilde{y}')\rangle \end{aligned}$$

□

Next, we show a particular situation in which the compressed oracle acts almost as if it were just the CO' operation, without Decomp. Intuitively, this occurs whenever an input-output pair $(x, H(x))$ is known external to the oracle. Furthermore, it also holds when only part of $H(x)$ is known, proportional to how much is known.

Lemma 3.3. Let $H : \mathcal{X} \rightarrow \mathcal{R}_1 \times \mathcal{R}_2$ be a random oracle implemented as a compressed oracle, where $\mathcal{R}_1, \mathcal{R}_2$ are finite and nonempty. Let $|\psi\rangle$ be a normalized state of the form

$$|\psi\rangle = \sum_{x, r_2, D'_x, u} \alpha_{x, r_2, D'_x, u} |\phi_{x, r_2}\rangle |x, u\rangle \otimes \sum_{r_1 \in \mathcal{R}_1} \beta_{u, D'_x, x, r_1, r_2} |D'_x \cup \{(x, r_1 \| r_2)\}\rangle,$$

where the sum ranges over $x \in \mathcal{X}$, $r_2 \in \mathcal{R}_2$, databases D'_x with $D'_x(x) = \perp$, and computational-basis labels u . Suppose that $\langle \phi_{x, r_2} | \phi_{x', r'_2} \rangle = 0$ whenever $(x, r_2) \neq (x', r'_2)$, and that Decomp is controlled by the displayed query-input register containing x . Then, with trace distance defined using the factor $1/2$,

$$\text{TD}(\text{Decomp } |\psi\rangle, |\psi\rangle) \leq \sqrt{\frac{2}{|\mathcal{R}_2|} - \frac{1}{|\mathcal{R}_2|^2}} \leq \sqrt{\frac{2}{|\mathcal{R}_2|}}.$$

Moreover,

$$\|(\text{Decomp} - I) |\psi\rangle\| \leq \sqrt{\frac{2}{|\mathcal{R}_2|}}.$$

Proof. Write $N_1 = |\mathcal{R}_1|$, $N_2 = |\mathcal{R}_2|$, and $N = N_1 N_2$. For $d(x) = \perp$, define

$$|+_{x, d}\rangle := \frac{1}{\sqrt{N}} \sum_{a \in \mathcal{R}_1, b \in \mathcal{R}_2} |d \cup \{(x, a \| b)\}\rangle.$$

For fixed x , these states are orthonormal as d varies. Each is orthogonal to every database basis state whose cell at x is $\perp$. Let

$$P_{+, x} := \sum_{d: d(x) = \perp} |+_{x, d}\rangle \langle +_{x, d}|, \quad P_+ := \sum_x |x\rangle \langle x|_{Q_x} \otimes P_{+, x},$$

with identity operators on all other registers understood. Thus P_+ projects the queried database cell onto its uniform non- $\perp$ state.

For brevity, let $j = (x, r_2, D'_x, u)$, $\alpha_j = \alpha_{x, r_2, D'_x, u}$, $\beta_{j, a} = \beta_{u, D'_x, x, a, r_2}$, and

$$w_j := |\alpha_j|^2 \|\phi_{x, r_2}\|^2.$$

The orthogonality assumption, the computational-basis registers x, u , and the database basis give the normalization identity

$$1 = \|\psi\rangle\|^2 = \sum_j w_j \sum_{a \in \mathcal{R}_1} |\beta_{j,a}|^2. \quad (4)$$

In particular, no separate normalization of α or β is needed. The factors $\|\phi_{x,r_2}\|^2$ can be omitted when these external vectors are normalized.

For each $a \in \mathcal{R}_1$,

$$P_{+,x} |D'_x \cup \{(x, a|r_2)\}\rangle = \frac{1}{\sqrt{N}} |_{+,D'_x}\rangle.$$

Consequently,

$$P_+ |\psi\rangle = \frac{1}{\sqrt{N}} \sum_j \alpha_j \left(\sum_{a \in \mathcal{R}_1} \beta_{j,a} \right) |\phi_{x,r_2}\rangle |x, u\rangle |_{+,D'_x}\rangle.$$

The vectors indexed by distinct j in this expression are mutually orthogonal: different x or u are distinguished by the query registers; different r_2 for fixed x are distinguished by the external vectors; and different D'_x for fixed x give orthogonal uniform database states. Therefore, by Cauchy–Schwarz and (4),

$$\begin{aligned} p := \|P_+ |\psi\rangle\|^2 &= \frac{1}{N} \sum_j w_j \left| \sum_{a \in \mathcal{R}_1} \beta_{j,a} \right|^2 \\ &\leq \frac{N_1}{N} \sum_j w_j \sum_{a \in \mathcal{R}_1} |\beta_{j,a}|^2 = \frac{1}{N_2}. \end{aligned}$$

Let $P_\perp$ project onto databases whose queried cell equals $\perp$, controlled on Q_x .

$$P_{\perp,x} := \sum_{d: d(x)=\perp} |d\rangle \langle d|, \quad P_\perp := \sum_{x \in \mathcal{X}} |x\rangle \langle x|_{Q_x} \otimes P_{\perp,x},$$

with identity operators on all other registers understood. Every term in the prescribed state $|\psi\rangle$ has a non- $\perp$ entry at its queried input, so $P_\perp |\psi\rangle = 0$.

The prescribed form of $|\psi\rangle$ implies $P_\perp |\psi\rangle = 0$. By definition, Decomp_x swaps $|_{+,d}\rangle$ with $|d\rangle$ and fixes their orthogonal complement, for every $d(x) = \perp$. Set

$$|v\rangle = P_+ |\psi\rangle, \quad |z\rangle = (I - P_+) |\psi\rangle, \quad |b\rangle = \text{Decomp} |v\rangle.$$

Since P_+ is the orthogonal projector onto the uniform non- $\perp$ subspace of the queried cell, the definition of $|z\rangle$ removes precisely this component:

$$P_+ |z\rangle = P_+ (I - P_+) |\psi\rangle = (P_+ - P_+^2) |\psi\rangle = 0.$$

Moreover, $P_\perp P_+ = 0$ because the uniform non- $\perp$ subspace is orthogonal to the $\perp$ subspace; together with $P_\perp |\psi\rangle = 0$, this gives

$$P_\perp |z\rangle = P_\perp (I - P_+) |\psi\rangle = P_\perp |\psi\rangle - P_\perp P_+ |\psi\rangle = 0.$$

Thus $|z\rangle$ is orthogonal to both subspaces exchanged by Decomp , so $\text{Decomp} |z\rangle = |z\rangle$.

Moreover, $|b\rangle$ lies in the range of $P_\perp$, whereas $|v\rangle$ and $|z\rangle$ lie in its kernel. Thus $|v\rangle, |z\rangle, |b\rangle$ are pairwise orthogonal and

$$|\psi\rangle = |z\rangle + |v\rangle, \quad \text{Decomp} |\psi\rangle = |z\rangle + |b\rangle, \quad \|v\rangle\|^2 = \|b\rangle\|^2 = p, \quad \|z\rangle\|^2 = 1 - p.$$

It follows exactly that

$$\langle \psi | \text{Decomp} | \psi \rangle = 1 - p, \quad \|(\text{Decomp} - I) | \psi \rangle\|^2 = \| |b\rangle - |v\rangle \|^2 = 2p.$$

Using the trace-distance identity for normalized pure states,

$$\text{TD}(\text{Decomp} | \psi \rangle, | \psi \rangle) = \sqrt{1 - |\langle \psi | \text{Decomp} | \psi \rangle|^2} = \sqrt{2p - p^2}.$$

Since $p \leq 1/N_2 \leq 1$ and $2p - p^2$ is increasing on $[0, 1]$,

$$\text{TD}(\text{Decomp} | \psi \rangle, | \psi \rangle) \leq \sqrt{\frac{2}{N_2} - \frac{1}{N_2^2}} = \sqrt{\frac{2}{|\mathcal{R}_2|} - \frac{1}{|\mathcal{R}_2|^2}} \leq \sqrt{\frac{2}{|\mathcal{R}_2|}}, \quad \|(\text{Decomp} - I) | \psi \rangle\| \leq \sqrt{\frac{2}{N_2}}.$$

Therefore we also have

$$|\langle \psi | \text{Decomp} | \psi \rangle|^2 \geq 1 - \frac{1}{|\mathcal{R}_2|}$$

□

We mention a few other useful lemmas which were introduced by prior works. Zhandry showed that if a quantum algorithm would find $(x, H(x))$ pairs after a polynomial number of queries, then H records a corresponding (x, r) pair with overwhelming probability when it is implemented as a compressed oracle.

Lemma 3.4 ([Zha19], Lemma 5). Consider a quantum algorithm $\mathcal{A}$ making queries to a random oracle H and outputting tuples $(x_1, \dots, x_k, y_1, \dots, y_k)$. Let R be a collection of such tuples. Suppose with probability p , $\mathcal{A}$ outputs a tuple such that (1) the tuple is in R and (2) $H(x_i) = y_i$ for all i . Now consider running $\mathcal{A}$ with the compressed oracle, and suppose the database D is measured after $\mathcal{A}$ produces its output. Let p' be the probability that (1) the tuple is in R , and (2) $D(x_i) = y_i$ for all i (and in particular $D(x_i) \neq \perp$). Then:

$$\sqrt{p} \leq \sqrt{p'} + \sqrt{k/|\mathcal{Y}|}$$

[GLR⁺25] observed that the compressed oracle treats all inputs identically, and so it equivalent up to renaming.

Claim 3.1 ([GLR⁺25]). Let SWITCH_{x_1, x_2} be the unitary which maps any database D to the unique D' defined by $D'(x_1) = D(x_2)$, $D'(x_2) = D(x_1)$, and $D'(x_3) = D(x_3)$ for all $x_3 \notin \{x_1, x_2\}$ and acts as the identity on all orthogonal states. Let U_{x_1, x_2} be the unitary which maps $|x_1\rangle \mapsto |x_2\rangle$, $|x_2\rangle \mapsto |x_1\rangle$ and acts as the identity on all orthogonal states. Then

$$\text{CO} = (U_{x_1, x_2} \otimes I \otimes \text{SWITCH}_{x_1, x_2}) \text{CO} (U_{x_1, x_2} \otimes I \otimes \text{SWITCH}_{x_1, x_2})$$

Proof. This follows from the observation that $(U_{x_1, x_2} \otimes I \otimes \text{SWITCH}_{x_1, x_2})$ commutes with both Decomp and CO' and it is self-inverse. □

3.2 Pseudorandom Functions

Here we define pseudorandom functions (PRFs), following definition 3.24 of [KL20].

A **keyed function** $F : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ takes as input a key k and a main input x and outputs y . The function is **length preserving** if for any key length $\lambda \in \mathbb{N}$, and any $k \in \{0, 1\}^\lambda$, $F(k, \cdot)$ is only defined for x -values in $\{0, 1\}^\lambda$, and for any such x , $F(k, x) \in \{0, 1\}^\lambda$.

Next, let Func_λ be the set of all functions mapping $\{0, 1\}^\lambda \rightarrow \{0, 1\}^\lambda$.

Definition 3.1 (Pseudorandom Function). An efficient, length preserving, keyed function $\text{PRF} : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^*$ is a **pseudorandom function** if for all QPT distinguishers D , there is a negligible function negl such that for any $\lambda \in \mathbb{N}$:

$$\left| \Pr \left[b = 1 : \begin{array}{l} k \xleftarrow{\$} \{0, 1\}^\lambda \\ b \leftarrow D^{\text{PRF}(k, \cdot)}(1^\lambda) \end{array} \right] - \Pr \left[b = 1 : \begin{array}{l} f \xleftarrow{\$} \text{Func}_\lambda \\ b \leftarrow D^{f(\cdot)}(1^\lambda) \end{array} \right] \right| \leq \text{negl}(\lambda)$$

3.3 Blind-Unforgeable Signatures

We recall here the definition of a (classical) signature scheme and the definition of blind unforgeability, a notion of post-quantum security.

The following definition of a signature scheme comes from [KL20].

Definition 3.2 (Signature Scheme). A (classical) signature scheme for message space $\mathcal{M}$ comprises three PPT algorithms (KeyGen, Sign, Ver) with the following **syntax**:

- $\text{KeyGen}(1^\lambda) \rightarrow (\text{sk}, \text{pk})$: Takes a security parameter $\lambda \in \mathbb{N}$ and generates a secret signing key sk and a public verification key pk . We assume that sk, pk are each at least λ bits long and that λ can be determined from sk or pk .
- $\text{Sign}(\text{sk}, m) \rightarrow \sigma$: Takes a message $m \in \mathcal{M}$ and outputs a signature σ .
- $\text{Ver}(\text{pk}, m, \sigma) \rightarrow \{0, 1\}$: Verifies the signature σ on message m . Ver outputs 1 for valid signatures and 0 for invalid signatures.

All variables $(\lambda, \text{sk}, \text{pk}, m, \sigma)$ are classical strings.

Additionally, the signature scheme satisfies **perfect correctness**, which says that for any $\lambda \in \mathbb{N}$ and any message $m \in \mathcal{M}$,

$$\Pr \left[b = 1 : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda) \\ \sigma \leftarrow \text{Sign}(\text{sk}, m) \\ b \leftarrow \text{Ver}(\text{pk}, m, \sigma) \end{array} \right] = 1$$

Next, here is [GLR⁺25]'s variant on the notion of blind unforgeability for signatures [AMRS20], verbatim.

Definition 3.3 (Quantum Blind Unforgeability). A signature scheme (KeyGen, Sign, Ver) for message space $\mathcal{M}$ (definition 3.2) is *blind-unforgeable* if for every QPT adversary $\mathcal{A}$ and blinding set $B \subset \mathcal{M}$,

$$\Pr \left[m \in B \wedge \text{Ver}(\text{pk}, m, \sigma) = \text{Accept} : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda) \\ (m, \sigma) \leftarrow \mathcal{A}^{\text{Sign}_B(\text{sk}, \cdot)}(\text{pk}) \end{array} \right] = \text{negl}(\lambda),$$

where $\text{Sign}_B(\text{sk}, \cdot)$ denotes a (quantumly-accessible) signature oracle that signs messages m using sk if $m \in B$, and otherwise outputs $\perp$.

This definition differs from the original in that the adversary may choose its blinding set B . [AMRS20] show that the hardness of this task is polynomially related to their original definition, which samples B uniformly at random.

We note that any sub-exponentially secure signature scheme is blind-unforgeable¹⁰, since the adversary could simply query for all signatures in $\mathcal{M} \setminus B$, then simulate the blind-unforgeability experiment. [AMRS20] also gives several other signature schemes which are blind-unforgeable (under the original definition).

3.4 Signature Tokens

A signature token is a quantum state that can be used to sign one and only one message [BDS23]. Here, we expand the definition of signature token schemes to encompass one-shot signatures as well.

In our definition (definition 3.4), a signature token also comes with a unique serial number s , and the security guarantee is that an adversary cannot produce two valid message-signature pairs *for the same serial number*. This guarantee resembles the guarantee of one-shot signatures.

¹⁰Here we need the scheme to be secure against subexponential time and queries, and we also need that the message space is smaller than this subexponential bound.

Indeed the one-shot signature primitive is a special case of definition 3.4 where $sk = pk$ so that signature tokens can be generated publicly. Additionally, our definition is general enough to capture regular signature tokens as well. A regular signature token scheme simply enforces that there is only one valid serial number. Along with the strong unforgeability property, this ensures that an adversary cannot generate two valid message-signature pairs.

The following definition of a generalized signature token scheme is adapted from [BDS23] and [SZ25a] definition 9.

Definition 3.4 (Generalized Signature Token Scheme).

- **Syntax:** A generalized signature token scheme comprises the following QPT algorithms:
 - $\text{KeyGen}(1^\lambda) \rightarrow (sk, pk)$: Takes a security parameter $\lambda \in \mathbb{N}$ and generates a secret signing key sk and a public verification key pk .
 - $\text{TokenGen}(sk) \rightarrow (|T\rangle, s)$: Generates a quantum signing token $|T\rangle$ and a classical serial number $s \in \mathcal{S}_\lambda \cup \{\perp\}$ associated with $|T\rangle$. TokenGen may output $(|T\rangle, s) = (\perp, \perp)$ with negligible probability.
 - $\text{Sign}(|T\rangle, m) \rightarrow \sigma$: Takes a signing token $|T\rangle$ and a message m from the message space $\mathcal{M}_\lambda$ and outputs a signature $\sigma \in \Sigma_\lambda$.
 - $\text{Ver}(pk, s, m, \sigma) \rightarrow \{0, 1\}$: Verifies the signature σ on message m with serial number s . Ver outputs 1 for valid signatures and 0 for invalid signatures.

The variables $(\lambda, sk, pk, s, m, \sigma)$ are classical, and $|T\rangle$ is a quantum state.

- **Correctness:** The scheme is *correct* if there is a negligible function $\text{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,

$$\Pr \left[(|T\rangle, s) = (\perp, \perp) : \begin{array}{l} (sk, pk) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(sk) \end{array} \right] \leq \text{negl}(\lambda)$$

and there exists a negligible function $\text{negl}'(\lambda)$ such that for all $\lambda \in \mathbb{N}$, all $m \in \mathcal{M}_\lambda$, all (sk, pk) in the support of $\text{KeyGen}(1^\lambda)$, and all $(|T\rangle, s)$ in the support of $\text{TokenGen}(sk)$ for which $(|T\rangle, s) \neq (\perp, \perp)$,

$$\Pr \left[b = 1 : \begin{array}{l} \sigma \leftarrow \text{Sign}(|T\rangle, m) \\ b \leftarrow \text{Ver}(pk, s, m, \sigma) \end{array} \right] \geq 1 - \text{negl}'(\lambda)$$

- **Strong Unforgeability:** The scheme is *strongly unforgeable* if for any QPT adversary $\mathcal{A}$, there exists a negligible function $\text{negl}(\lambda)$ such that for any $\lambda \in \mathbb{N}$,

$$\Pr \left[\begin{array}{l} (m_0, \sigma_0) \neq (m_1, \sigma_1) \wedge \\ 1 \leftarrow \text{Ver}(pk, s', m_0, \sigma_0) \wedge \\ 1 \leftarrow \text{Ver}(pk, s', m_1, \sigma_1) \end{array} : \begin{array}{l} (sk, pk) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(sk) \\ (m_0, m_1, \sigma_0, \sigma_1, s') \leftarrow \mathcal{A}(1^\lambda, pk, |T\rangle, s) \end{array} \right] \leq \text{negl}(\lambda)$$

The signature tokens defined in [BDS23] (definitions 4-6, 25) are a special case of the tokens in definition 3.4 where s has the same value s^* for every token and Ver requires that $s = s^*$. This ensures that no adversary can generate two different valid message-signature pairs. We call such a scheme a *one-time* signature token scheme. In particular, we would use a one-time signature token scheme with strong unforgeability.

Definition 3.5 (One-Time Signature Token Scheme). A **one-time signature token** scheme $(\text{KeyGen}, \text{TokenGen}, \text{Sign}, \text{Ver})$ is a type of generalized signature token scheme (definition 3.4) where for any $\lambda \in \mathbb{N}$, the set of valid serial numbers $\mathcal{S}_\lambda$ contains just one element s^* . Then for any $\lambda \in \mathbb{N}$ and any (sk, pk) in the support of $\text{KeyGen}(1^\lambda)$,

- $\text{TokenGen}(\text{sk})$ always outputs $s \in \{s^*, \perp\}$.
- For any (s, m, σ) such that $s \neq s^*$, $\text{Ver}(\text{pk}, s, m, \sigma)$ always outputs 0.

One-shot signatures are a special case of definition 3.4 in which TokenGen is a public procedure ($\text{sk} = \text{pk}$).

Definition 3.6 (One-Shot Signature Scheme). A **one-shot signature** scheme $(\text{KeyGen}, \text{TokenGen}, \text{Sign}, \text{Ver})$ is a type of generalized signature token scheme (definition 3.4) where for any $\lambda \in \mathbb{N}$ and any (sk, pk) in the support of $\text{KeyGen}(1^\lambda)$,

$$\text{sk} = \text{pk}.$$

[SZ25a] construct one-shot signatures, which satisfy all the properties of definition 3.4 and satisfy $\text{sk} = \text{pk}$.¹¹

3.5 One-Time Programs

The following definitions come from [GLR⁺25] section 4.

Syntax: Let $\mathcal{F}$ be a family of classical randomized functions $f : \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y}$. $\mathcal{X}, \mathcal{R}, \mathcal{Y}$ are some sets whose descriptions are included in the description of f . f may be associated with some public, classical auxiliary information aux_f . Next, each $f \in \mathcal{F}$ takes a user-defined input $x \in \mathcal{X}$, samples $r \xleftarrow{\$} \mathcal{R}$, and outputs $y = f(x; r)$.

A one-time program compiler for a function family $\mathcal{F}$ takes a description of an $f \in \mathcal{F}$ and outputs a quantum program P_f , which allows the user to evaluate f .

Definition 3.7 (One-Time Program Compiler for $\mathcal{F}$). A one-time program compiler for $\mathcal{F}$ is a pair of QPT algorithms $(\text{PGen}, \text{Eval})$, which have the following syntax:

- $P_f \leftarrow \text{PGen}(1^\lambda, f)$: Takes as input the security parameter 1^λ and a description of the function $f \in \mathcal{F}$, and outputs a quantum program P_f , which may comprise a quantum state and an oracle.
- $y \leftarrow \text{Eval}(P_f, x)$: Takes a quantum program P_f and a classical input $x \in \mathcal{X}$ and outputs a classical value $y \in \mathcal{Y}$.

Correctness: The one-time program is correct if querying the program on any input x results in a distribution over y -values that is statistically close to the output of the classical randomized functionality.

Definition 3.8 (Correctness, [GLR⁺25] definition 4.2). A one-time program compiler for $\mathcal{F}$ is **correct** if for every $f \in \mathcal{F}$ and $x \in \mathcal{X}$, the following distributions over y -values have negligible statistical distance:

$$\{\text{Eval}[\text{PGen}(1^\lambda, f), x]\} \quad \text{and} \quad \{f(x; r)\}$$

The randomness in the second distribution is over the sampling of $r \xleftarrow{\$} \mathcal{R}$.

¹¹We can map the notation of [SZ25a], definition 9, to the notation of our definition 3.4 as follows: $\text{Setup} \rightarrow \text{KeyGen}$, $\text{Gen} \rightarrow \text{TokenGen}$, $\text{CRS} \rightarrow \text{pk}$, $\text{CRS} \rightarrow \text{sk}$, $\text{pk} \rightarrow s$, $(\text{pk}, |\text{sk}\rangle, \text{CRS}) \rightarrow |T\rangle$.

We also note that [SZ25a] defines a weaker notion of security (regular unforgeability), but their construction satisfies strong unforgeability as well.

Security: We will use the notion of single-effective query (SEQ) security, which is a simulation-based notion of security introduced in [GLR⁺25] that is designed to model uncomputation attacks. A real-world adversary can always uncompute their query and return to the original program state if they did not make any destructive measurements, so our ideal-world functionality, called the SEQ oracle, allows the adversary to uncompute their queries as well. The SEQ oracle uses the compressed oracle technique to record the adversary's queries and stops answering new queries once one query has been recorded. Security says that the real one-time program for f can be simulated given only the SEQ oracle for f .

The SEQ oracle O_f^{SEQ} implements the function mapping $x \rightarrow f(x; H(x))$, where H is a random oracle mapping $\mathcal{X} \rightarrow \mathcal{R}$. The SEQ oracle implements H as a compressed oracle, which allows it to record queries, and it refuses to answer any query that would increase the number of recorded queries to 2 or more. The following definition of the SEQ oracle comes from [GLR⁺25], section 4.2.

O_f^{SEQ} has an internal database register D , which can store classical values of the form $d = (x, r) \in \mathcal{X} \times \mathcal{R}$ or $d = \emptyset$. D is initialized to $|\emptyset\rangle$. O_f^{SEQ} also acts on a query register $Q = Q_x \times Q_u \times Q_B$, which stores classical values of the form $(x, u, b) \in \mathcal{X} \times \mathcal{Y} \times \{0, 1\}$. On each query, O_f^{SEQ} acts as a unitary on $Q \times D$ and an ancilla register Q' according to the procedure given in fig. 1.

To respond to a query, O_f^{SEQ} acts as a unitary on $Q \times D$ and an ancilla register Q' . Given some basis state $|x, u, b\rangle_Q \otimes |d\rangle_D$, O_f^{SEQ} does the following:

1. If $d = (x', r)$ for some $x' \neq x$, then skip the following steps. Otherwise, continue.
2. Prepare the state $|x, 0\rangle$ in an ancillary register Q' . Query H by applying the compressed oracle unitary CO to $Q' \times D$.
3. Apply the following isometry to registers $Q \times Q'$:

$$|x, u, b\rangle_Q \otimes |x, r\rangle_{Q'} \rightarrow |x, u \oplus f(x; r), b \oplus 1\rangle_Q \otimes |x, r\rangle_{Q'}$$

4. Uncompute item 2.

Figure 1: Querying the SEQ Oracle O_f^{SEQ}

Now we can define our notion of security, which says that the one-time program for f can be simulated given only the SEQ oracle O_f^{SEQ} .

Definition 3.9 (SEQ Security, [GLR⁺25] definition 4.8). Let λ be the security parameter. Let f be a function in a function family $\mathcal{F}$ that possibly comes with a public, classical auxiliary input aux_f . Let D be a QPT distinguisher.

A one-time program compiler for $\mathcal{F}$ satisfies **SEQ security** if there exists a QPT simulator Sim^{12} such that for every $f \in \mathcal{F}$ and all distinguishers D , there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[1 \leftarrow D \left(\text{PGen}(1^\lambda, f), f, \text{aux}_f \right) \right] - \Pr \left[1 \leftarrow D \left(\text{Sim}^{O_f^{\text{SEQ}}}(1^\lambda, \text{aux}_f), f, \text{aux}_f \right) \right] \right| \leq \text{negl}(\lambda)$$

¹²We allow Sim to take 1^λ as input. We believe [GLR⁺25] should have included this feature in their definition of SEQ security because the simulator that they construct in figure 4 does take 1^λ as input.

3.6 qIND-qCPA Secure Secret Key Encryption

In this section, we introduce a security notion for SKE where the quantum adversary can make superposition queries in both the query phase and challenge phase. The definition is formalized in [GHS16].

Definition 3.10 (Quantum encryption oracle). Let $\mathcal{E} = (\text{KeyGen}, \text{Enc}, \text{Dec})$ be a secret-key encryption scheme, and let $k \leftarrow \text{KeyGen}(1^n)$. The quantum encryption oracle associated with Enc_k is the unitary

$$U_{\text{Enc}_k} : \sum_{x,y} \alpha_{x,y} |x\rangle |y\rangle \mapsto \sum_{x,y} \alpha_{x,y} |x\rangle |y \oplus \text{Enc}_k(x)\rangle.$$

Within a single oracle query, the same encryption randomness is used coherently across all basis states in superposition; for each new query, fresh independent randomness is used.

Definition 3.11 (qCPA learning phase). In the qCPA learning phase, the adversary $\mathcal{A}$ is given oracle access to U_{Enc_k} .

Definition 3.12 (Type-(2) encryption operator). For length-preserving encryption, define the type-(2) operator

$$U_{\text{Enc}_k}^{(2)} : \sum_x \alpha_x |x\rangle \mapsto \sum_x \alpha_x |\text{Enc}_k(x)\rangle.$$

More generally, if encryption expands the message length, one may view $U_{\text{Enc}_k}^{(2)}$ as a unitary acting on an input register together with an ancilla register initialized to $|0\rangle$, such that applying it to $\rho \otimes |0\rangle\langle 0|$ yields the honest encryption of ρ .

Definition 3.13 (qIND challenge phase). The qIND challenge phase proceeds as follows.

1. The adversary $\mathcal{A}$ chooses two m -qubit states ρ_0, ρ_1 having efficient classical descriptions, and sends these classical descriptions to the challenger $\mathcal{C}$.
2. The challenger samples a uniformly random bit

$$b \xleftarrow{\$} \{0, 1\}.$$

3. The challenger replies with the encryption of ρ_b obtained by applying the type-(2) operator $U_{\text{Enc}_k}^{(2)}$ to ρ_b .
4. The adversary outputs a bit b' and wins iff $b' = b$.

Security Game $\text{qIND-qCPA}_{\mathcal{E}}^{\mathcal{A}}(n)$

1. Generate a key:

$$k \xleftarrow{\$} \text{KeyGen}(1^n).$$

2. Give $\mathcal{A}$ oracle access to U_{Enc_k} (the qCPA learning phase).
3. $\mathcal{A}$ outputs two m -qubit states ρ_0, ρ_1 .

4. Sample

$$b \xleftarrow{\$} \{0, 1\}.$$

5. Return to $\mathcal{A}$ the challenge ciphertext state

$$\sigma := U_{\text{Enc}_k}^{(2)} \rho_b (U_{\text{Enc}_k}^{(2)})^\dagger.$$

6. $\mathcal{A}$ outputs a bit b' .
7. Output 1 iff $b' = b$, and output 0 otherwise.

Definition 3.14 (qIND-qCPA security). A secret-key encryption scheme $\mathcal{E}$ is *qIND-qCPA secure* if, for every quantum probabilistic polynomial-time adversary $\mathcal{A}$,

$$\Pr[\text{qIND-qCPA}_{\mathcal{E}}^{\mathcal{A}}(n) = 1] \leq \frac{1}{2} + \text{negl}(n).$$

Equivalently, the advantage

$$\text{Adv}_{\mathcal{E}}^{\text{qIND-qCPA}}(\mathcal{A}, n) := \left| \Pr[\text{qIND-qCPA}_{\mathcal{E}}^{\mathcal{A}}(n) = 1] - \frac{1}{2} \right|$$

is negligible in n .

Theorem 3.1 ([GHS16]). Assuming post-quantum secure pseudorandom permutations (PRP), there exists qIND-qCPA secure SKE.

4 Generalized One-Time Programs

The first construction of an OTP compiler that satisfied SEQ security [GLR⁺25] used the specific tokenized signature scheme of [BDS23]. The intuition for the construction is that the user is required to sign their input in order to obtain a response, and the signature token scheme limits them to signing just one input.

Here, we generalize the construction and security definition in two ways. First, we show that an SEQ-secure OTP compiler can be constructed from *any* one-time signature token scheme (definition 3.5), not just [BDS23]’s scheme. Second, we show that if the signature token is replaced with a one-shot signature scheme (definition 3.6), the result is a new primitive, called a one-shot program (definition 4.2), that allows the user to query the program on many inputs as long as they have distinct serial numbers. One-shot programs will be used to construct query-limited RAM programs in section 5.

Both of the results above have similar constructions and security proofs, so we construct a generalized one-time program primitive (definition 4.1) to capture both results.

4.1 Definition

Here we define a generalized one-time program compiler. Informally, the compiler generates a program P_f that implements a randomized functionality f . The program also limits the user to making one effective query for each possible serial number.

The Function Family. Let $\mathcal{F}$ be a family of randomized functions mapping $\mathcal{S}_{\lambda} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y}$, for some given sets $\mathcal{S}_{\lambda}, \mathcal{X}, \mathcal{R}, \mathcal{Y}$. Each $f \in \mathcal{F}$ takes as input a serial number $s \in \mathcal{S}_{\lambda}$ and a main input $x \in \mathcal{X}$. f also takes a uniformly random string $r \xleftarrow{\$} \mathcal{R}$ and then computes $y = f(s, x; r) \in \mathcal{Y}$. Additionally, each f may be associated with some public, classical auxiliary information aux_f .

For example, f may generate a signature on (s, x, r) . In this case, f is parametrized by a signing key sk for a blind-unforgeable signature scheme, and $f(s, x)$ computes $y = [r, \text{Sign}(\text{sk}, (s, x, r))]$. Additionally, aux_f is the pk of the signature scheme. Alternatively, f may require s to be some fixed value s^* if we are constructing regular one-time programs.

The guarantee of the generalized OTP is that the adversary cannot evaluate any given f on the same s with two different x -values.

To respond to a query, the oracle acts as a unitary on $Q \times \mathcal{D}$ and an ancilla register Q' . Given some basis state $|s, x, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}}$, the oracle does the following:

1. If the database D contains an entry of the form (s', x', r') such that $s' = s$ but $x' \neq x$, then skip the following steps. Otherwise, continue.
2. Prepare the state $|s, x, 0\rangle$ in an ancillary register Q' . Query $H(s, x)$ by applying the compressed oracle unitary CO to $Q' \times \mathcal{D}$.
3. Apply the following isometry to registers $Q \times Q'$:

$$|s, x, u, b\rangle_Q \otimes |s, x, r\rangle_{Q'} \rightarrow |s, x, u \oplus f(s, x; r), b \oplus 1\rangle_Q \otimes |s, x, r\rangle_{Q'}$$

4. Uncompute item 2.

Figure 2: The SEQ Oracle O_f^{SEQ}

The SEQ Oracle. Figure 2 describes an oracle O_f^{SEQ} that is a variant of the SEQ oracle for $f \in \mathcal{F}$. O_f^{SEQ} takes (s, x) as input and computes $f(s, x; H(s, x))$ as long as its database does not already record (s, x', r) for an $x' \neq x$. This means the database may record many queries, but they will have distinct s -values.

Generalized One-Time Program Compiler. This primitive generates quantum program tokens using

$$(|T\rangle, s) \leftarrow \text{TokenGen}(\text{sk})$$

and each token allows one evaluation of the program. The primitive also generates a *classical* program

$$P_f \leftarrow \text{PGen}(\text{pk}, f)$$

which enables the evaluation of f by any user who possesses a program token. We have chosen to separate the program's classical component P_f from its quantum component $|T\rangle$, as it will simplify the construction of semi-quantum query-limited programs, which can be sent over classical channels (appendix B.2).

Definition 4.1 (Generalized One-Time Program Compiler for $\mathcal{F}$).

- **Syntax:** A generalized one-time program compiler for $\mathcal{F}$ comprises the following QPT algorithms:
 - $\text{KeyGen}(1^\lambda) \rightarrow (\text{sk}, \text{pk})$: Takes a security parameter $\lambda \in \mathbb{N}$ and outputs a classical secret key sk and classical public key pk .
 - $\text{TokenGen}(\text{sk}) \rightarrow (|T\rangle, s)$: Generates a program token $|T\rangle$ and the corresponding serial number $s \in \mathcal{S}_\lambda \cup \{\perp\}$. TokenGen may output $(|T\rangle, s) = (\perp, \perp)$ with negligible probability.
 - $\text{PGen}(\text{pk}, f) \rightarrow P_f$: Takes the key pk and a function $f \in \mathcal{F}$ and outputs a *classical* program P_f , which may include an oracle.
 - $\text{Eval}(P_f, s, x, |T\rangle) \rightarrow y$: Takes a program P_f , a serial number $s \in \mathcal{S}_\lambda$, a main input $x \in \mathcal{X}$, and a signature token $|T\rangle$ that is associated with s . Then Eval outputs $y \in \mathcal{Y}$. For a correct evaluation, $y = f(s, x; r)$ for a uniformly random $r \xleftarrow{\$} \mathcal{R}$.

- **Correctness:** A generalized one-time program compiler for $\mathcal{F}$ is **correct** if there is a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\Pr \left[(|T\rangle, s) = (\perp, \perp) : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(\text{sk}) \end{array} \right] \leq \text{negl}(\lambda)$$

and there exists a negligible function $\text{negl}'(\cdot)$ such that for all $(\lambda, f, x) \in \mathbb{N} \times \mathcal{F} \times \mathcal{X}$, all (sk, pk) in the support of $\text{KeyGen}(1^\lambda)$, and all $(|T\rangle, s)$ in the support of $\text{TokenGen}(\text{sk})$ for which $(|T\rangle, s) \neq (\perp, \perp)$, the following distributions over y -values are $\text{negl}'(\lambda)$ -close in statistical distance:

$$\left\{ \begin{array}{l} P_f \leftarrow \text{PGen}(\text{pk}, f) \\ y \leftarrow \text{Eval}(P_f, s, x, |T\rangle) \end{array} \right\} \quad \text{and} \quad \left\{ \begin{array}{l} r \xleftarrow{\$} \mathcal{R} \\ y = f(s, x; r) \end{array} \right\}$$

- **Security:** A generalized one-time program compiler for $\mathcal{F}$ is **SEQ-secure** if there exists a QPT simulator Sim such that for all $f \in \mathcal{F}$ and all QPT distinguishers D , there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[b = 1 : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(\text{sk}) \\ P_f \leftarrow \text{PGen}(\text{pk}, f) \\ b \leftarrow D(P_f, |T\rangle, s, \text{pk}, f, \text{aux}_f) \end{array} \right] - \Pr \left[b = 1 : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(\text{sk}) \\ P_f^{\text{Sim}} \leftarrow \text{Sim}_{O_f^{\text{SEQ}}}^f(1^\lambda, \text{pk}, \text{aux}_f) \\ b \leftarrow D(P_f^{\text{Sim}}, |T\rangle, s, \text{pk}, f, \text{aux}_f) \end{array} \right] \right| \leq \text{negl}(\lambda)$$

where O_f^{SEQ} is the oracle in fig. 2.

4.2 Construction

Here we construct a generalized one-time program compiler for any randomized function family $\mathcal{F}$ (definition 4.1) using a generalized signature token scheme (definition 3.4). At a high level, the construction works as follows. The program includes an oracle that computes $f(s, x; r)$ on any inputs (s, x) as long as the user can provide a valid signature σ on message x with serial number s . The user cannot evaluate the program on two inputs with the same s value, by the security of the generalized signature token scheme. Furthermore, if f is highly random, any evaluation of the oracle will consume the signature token because the random string r is chosen by hashing $G(s, x, \sigma)$.

Figure 3 describes the construction in detail. Let $\text{SigToken}(\text{KeyGen}, \text{TokenGen}, \text{Sign}, \text{Ver})$ be a generalized signature token scheme (definition 3.4). For any $\lambda \in \mathbb{N}$, let Σ_λ be the domain of signatures, and let $\mathcal{S}_\lambda$ be the domain of the serial numbers.

Now we prove that the construction in fig. 3 is a generalized one-time program compiler for $\mathcal{F}$.

Theorem 4.1. The construction in fig. 3 is a generalized one-time program compiler for $\mathcal{F}$ (definition 4.1).

Proof. First, the construction in fig. 3 clearly satisfies the syntax of a generalized one-time program compiler for $\mathcal{F}$.

Second, to begin the proof of correctness, we must show that

$$\Pr \left[(|T\rangle, s) = (\perp, \perp) : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(\text{sk}) \end{array} \right] \leq \text{negl}(\lambda)$$

This follows directly from the correctness property of SigToken (definition 3.4).

Third, to finish the proof of correctness, we must show that for any $(\lambda, f, x, \text{sk}, \text{pk}, |T\rangle, s)$ for which $(|T\rangle, s) \neq (\perp, \perp)$, the following distributions over y -values are statistically close.

$$\left\{ \begin{array}{l} P_f \leftarrow \text{PGen}(\text{pk}, f) \\ y \leftarrow \text{Eval}(P_f, s, x, |T\rangle) \end{array} \right\} \quad \text{and} \quad \left\{ \begin{array}{l} r \xleftarrow{\$} \mathcal{R} \\ y = f(s, x; r) \end{array} \right\} \quad (5)$$

- $\text{KeyGen}(1^\lambda)$: Compute and output

$$(\text{sk}, \text{pk}) \leftarrow \text{SigToken.KeyGen}(1^\lambda)$$

- $\text{TokenGen}(\text{sk})$: Compute and output

$$(|T\rangle, s) \leftarrow \text{SigToken.TokenGen}(\text{sk})$$

- $\text{PGen}(\text{pk}, f)$:

1. Sample a random oracle $G : \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \rightarrow \mathcal{R}$.
2. Generate an oracle $O_{f,G,\text{pk}}$ that acts as follows:

Hardcoded: f, G, pk

Inputs: Takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$. Here, s is a serial number, x is the evaluator's input, σ is a signature on x , and u is in the register that stores the output of the function.

- (a) Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
- (b) Query G to compute $r = G(s, x, \sigma)$:

$$|s, x, \sigma\rangle_{Q_s \times Q_x \times Q_\sigma} \otimes |0\rangle_R \rightarrow |s, x, \sigma\rangle \otimes |r\rangle$$

- (c) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$

- (d) Uncompute item 2b.
- (e) Output register Q .

3. Output $P_f = O_{f,G,\text{pk}}$.

- $\text{Eval}(P_f, s, x, |T\rangle)$:

1. Compute $\sigma = \text{SigToken.Sign}(|T\rangle, x)$.
2. Query $O_{f,G,\text{pk}}$ on input $(s, x, \sigma, 0)$ to obtain output (s, x, σ, y) . Then output y .

Figure 3: Generalized One-Time Program Compiler Construction

- $\text{PGen}(1^\lambda, f)$:
 1. Compute $(\text{sk}, \text{pk}) \leftarrow \text{GeneralOTP.KeyGen}(1^\lambda)$
 2. Compute $(|T\rangle, s) \leftarrow \text{GeneralOTP.TokenGen}(\text{sk})$
 3. Let $f' : \mathcal{S}_\lambda \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y}$ be the following function that ignores its first input:

$$f'(s, x; r) = f(x; r)$$
 4. Compute $P_{f'} \leftarrow \text{GeneralOTP.PGen}(\text{pk}, f')$
 5. Output $P_f = (|T\rangle, s, \text{pk}, P_{f'})$.
- $\text{Eval}(P_f, x)$:
 1. Parse $P_f = (|T\rangle, s, \text{pk}, P_{f'})$.
 2. Compute and output $y \leftarrow \text{GeneralOTP.Eval}(P_{f'}, s, x, |T\rangle)$.

Figure 4: Construction of a One-Time Program Compiler for $\mathcal{F}$

The left distribution of eq. (5) is equivalent to the following distribution $\mathcal{D}$:

$$\begin{aligned}
 \sigma &\leftarrow \text{SigToken.Sign}(|T\rangle, x) \\
 b &\leftarrow \text{SigToken.Ver}(\text{pk}, s, x, \sigma) \\
 r &\xleftarrow{\$} \mathcal{R} \\
 y &= \begin{cases} f(s, x; r), & b = 1 \\ 0, & b = 0 \end{cases}
 \end{aligned}$$

This follows by unpacking PGen and Eval.

Next, $\mathcal{D}$ is statistically close to the right distribution of eq. (5). This is because $\text{SigToken.Ver}(\text{pk}, s, x, \sigma)$ equals 1 with overwhelming probability, by the correctness of SigToken (definition 3.4). This shows that the left and right distributions of eq. (5) are statistically close, and it completes the proof of correctness.

Fourth, the proof of security (theorem 4.4) is given in section 4.5. $\square$

4.3 One-Time Programs From Any Signature Token

We will use the generalized one-time program compiler constructed in fig. 3 to construct a (regular) one-time program compiler (definition 3.7).

Let us instantiate SigToken in fig. 3 with a one-time signature token scheme (definition 3.5). Additionally, the KeyGen and TokenGen functions of the generalized OTP compiler are merged into PGen.

Figure 4 constructs the OTP compiler for $\mathcal{F}$ formally. Let $\text{GeneralOTP}(\text{KeyGen}, \text{TokenGen}, \text{PGen}, \text{Eval})$ be the generalized OTP compiler constructed in fig. 3, where SigToken is instantiated with a one-time signature token scheme (definition 3.5). Additionally, for every $f \in \mathcal{F}$, let $f : \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{Y}$.

Theorem 4.2. The construction in fig. 4 is a one-time program compiler for $\mathcal{F}$, which satisfies the syntax (definition 3.7), correctness (definition 3.8), and SEQ security (definition 3.9) properties.

Proof. First, it is clear by inspection that the construction satisfies the syntax of a one-time program compiler.

Second, correctness follows from the correctness of GeneralOTP. The correctness of GeneralOTP implies that for any f, x , the following two distributions over y -values are negligibly close in statistical distance:

$$\left\{ \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{GeneralOTP.KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{GeneralOTP.TokenGen}(\text{sk}) \\ P_{f'} \leftarrow \text{GeneralOTP.PGen}(\text{pk}, f') \\ y \leftarrow \text{GeneralOTP.Eval}(P_{f'}, s, x, |T\rangle) \end{array} \right\} \quad \text{and} \quad \left\{ \begin{array}{l} r \xleftarrow{\$} \mathcal{R} \\ y = f(x; r) \end{array} \right\}$$

The left distribution is the same as

$$\left\{ \begin{array}{l} P_f \leftarrow \text{PGen}(1^\lambda, f) \\ y \leftarrow \text{Eval}(P_f, x) \end{array} \right\}$$

This proves the correctness of the OTP compiler.

Third, to prove SEQ security, let us start with the SEQ security property of GeneralOTP, which says the following: there exists a simulator Sim' such that for all $f \in \mathcal{F}$ and all QPT distinguishers D , there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[b = 1 : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{GeneralOTP.KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{GeneralOTP.TokenGen}(\text{sk}) \\ P_{f'} \leftarrow \text{GeneralOTP.PGen}(\text{pk}, f') \\ b \leftarrow D(P_{f'}, |T\rangle, s, \text{pk}, f', \text{aux}_{f'}) \end{array} \right] - \Pr \left[b = 1 : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{GeneralOTP.KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{GeneralOTP.TokenGen}(\text{sk}) \\ P_{f'}^{\text{Sim}} \leftarrow \text{Sim}'^{O_f^{\text{SEQ}}}(1^\lambda, \text{pk}, \text{aux}_{f'}) \\ b \leftarrow D(P_{f'}^{\text{Sim}}, |T\rangle, s, \text{pk}, f', \text{aux}_{f'}) \end{array} \right] \right| \leq \text{negl}(\lambda) \quad (6)$$

where O_f^{SEQ} is the oracle in fig. 2.

We can simplify eq. (6) to be:

$$\left| \Pr \left[b = 1 : \begin{array}{l} P_f \leftarrow \text{PGen}(1^\lambda, f) \\ b \leftarrow D(P_f, f', \text{aux}_{f'}) \end{array} \right] - \Pr \left[b = 1 : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{GeneralOTP.KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{GeneralOTP.TokenGen}(\text{sk}) \\ P_{f'}^{\text{Sim}} \leftarrow \text{Sim}'^{O_f^{\text{SEQ}}}(1^\lambda, \text{pk}, \text{aux}_{f'}) \\ b \leftarrow D(P_{f'}^{\text{Sim}}, |T\rangle, s, \text{pk}, f', \text{aux}_{f'}) \end{array} \right] \right| \leq \text{negl}(\lambda)$$

Fourth, let us reconcile the difference in SEQ oracles. The OTP compiler uses O_f^{SEQ} from fig. 1 to define security, whereas the generalized OTP compiler uses $O_{f'}^{\text{SEQ}}$ from fig. 2. O_f^{SEQ} is similar to $O_{f'}^{\text{SEQ}}$, except that O_f^{SEQ} does not take s as input, does not record s in the database, and does not consider s when deciding to answer the query. However, note that because SigToken is a one-time signature token, $\mathcal{S}_\lambda$ contains a single element s^* . Therefore, every query to $O_{f'}^{\text{SEQ}}$ has $s = s^*$ and every entry of the oracle database records $s = s^*$. Furthermore, $O_{f'}^{\text{SEQ}}$ answers a query x if and only if the database does not record (s^*, x', r') for some $x' \neq x$. Then querying O_f^{SEQ} on input (x, u, b) is equivalent to querying $O_{f'}^{\text{SEQ}}$ on input (s^*, x, u, b) and ignoring the s^* value after the query.

Fifth, let us define the simulator Sim for the OTP compiler.

1. Given query access to O_f^{SEQ} , Sim simulates queries to $O_{f'}^{\text{SEQ}}$ as follows. Take as input a query (s, x, u, b) . If $s \notin \mathcal{S}_\lambda$, then do not answer the query. Otherwise, query O_f^{SEQ} on input (x, u, b) to obtain (x, u', b') , and output (s^*, x, u', b') .
2. Sim computes

$$\begin{aligned} (\text{sk}, \text{pk}) &\leftarrow \text{GeneralOTP.KeyGen}(1^\lambda) \\ (|T\rangle, s) &\leftarrow \text{GeneralOTP.TokenGen}(\text{sk}) \\ P_{f'}^{\text{Sim}} &\leftarrow \text{Sim}'^{O_f^{\text{SEQ}}}(1^\lambda, \text{pk}, \text{aux}_{f'}) \end{aligned}$$

where $O_{f'}^{\text{SEQ}}$ is simulated using the procedure above.

3. Sim outputs $P_f^{\text{Sim}} = (|T\rangle, s, \text{pk}, P_{f'}^{\text{Sim}})$.

With this choice of Sim, eq. (6) simplifies further to:

$$\left| \Pr \left[b = 1 : \begin{array}{l} P_f \leftarrow \text{PGen}(1^\lambda, f) \\ b \leftarrow D(P_f, f', \text{aux}_{f'}) \end{array} \right] - \Pr \left[b = 1 : \begin{array}{l} P_f^{\text{Sim}} \leftarrow \text{Sim}_{f'}^{O^{\text{SEQ}}} (1^\lambda, \text{aux}_{f'}) \\ b \leftarrow D(P_f^{\text{Sim}}, f', \text{aux}_{f'}) \end{array} \right] \right| \leq \text{negl}(\lambda)$$

Furthermore, we can set $\text{aux}_{f'} = \text{aux}_f$, and we can give D access to f instead of f' because they provide equivalent information. Then we have the following: there exists a simulator Sim such that for all $f \in \mathcal{F}$ and all QPT distinguishers D , there exists a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[b = 1 : \begin{array}{l} P_f \leftarrow \text{PGen}(1^\lambda, f) \\ b \leftarrow D(P_f, f, \text{aux}_f) \end{array} \right] - \Pr \left[b = 1 : \begin{array}{l} P_f^{\text{Sim}} \leftarrow \text{Sim}_f^{O^{\text{SEQ}}} (1^\lambda, \text{aux}_f) \\ b \leftarrow D(P_f^{\text{Sim}}, f, \text{aux}_f) \end{array} \right] \right| \leq \text{negl}(\lambda)$$

This proves the SEQ security of the OTP compiler. $\square$

4.4 One-Shot Programs

A one-shot program compiler is a type of generalized OTP compiler where $\text{sk} = \text{pk}$. This property gives the evaluator significant power. Now, if they possess the public key pk , then they can generate many program tokens $(|T\rangle, s)$, albeit with unique serial numbers, and evaluate the program once per token.

Definition 4.2 (One-Shot Program Compiler for $\mathcal{F}$). A **one-shot program compiler** for $\mathcal{F}$ (KeyGen , TokenGen , PGen , Eval) is a type of generalized one-time program compiler for $\mathcal{F}$ (definition 4.1) where for any $\lambda \in \mathbb{N}$ and any (sk, pk) in the support of $\text{KeyGen}(1^\lambda)$,

$$\text{sk} = \text{pk}.$$

Remark 1. For one-shot programs, we sometimes introduce the name $\text{crs} = \text{sk} = \text{pk}$, and we may notate the output of KeyGen as

$$\text{crs} \leftarrow \text{KeyGen}(1^\lambda)$$

Our construction of one-shot programs is the same as the construction in fig. 3, except we instantiate SigToken with a one-shot signature scheme (definition 3.6).

Theorem 4.3. Given the construction in fig. 3, let SigToken be instantiated with a one-shot signature scheme (definition 3.6). Then the construction is a one-shot program compiler for $\mathcal{F}$ (definition 4.2).

Proof. The one-shot signature token scheme used in the construction is a type of generalized signature token scheme. Then by theorem 4.1, the construction is a generalized one-time program compiler.

Next, note that when SigToken is instantiated with a one-shot signature scheme, the $\text{KeyGen}(1^\lambda)$ function in the construction always outputs (sk, pk) such that $\text{sk} = \text{pk}$. Therefore, the construction is a one-shot program compiler. $\square$

4.5 Proof of Security for Generalized OTPs

Theorem 4.4 (Security). The construction in fig. 3 satisfies the security property of a generalized one-time program compiler for $\mathcal{F}$ (definition 4.1).

The rest of section 4.5 proves theorem 4.4.

Simulator: We will construct the SEQ simulator Sim in fig. 5, which simulates P_f .

1. **Inputs:** Sim gets quantum query access to the SEQ oracle O_f^{SEQ} and also receives $(1^\lambda, \text{pk}, \text{aux}_f)$.
2. Prepare a *cache register* C , which holds a set of any number of tuples of the form $(s, x, \sigma) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda$ and is initialized to $|\emptyset\rangle$ (empty).
3. Prepare an oracle $O_{\text{Sim}}^{O_f^{\text{SEQ}}}$, which acts as a unitary on classical eigenstates as follows:
 - (a) $O_{\text{Sim}}^{O_f^{\text{SEQ}}}$ takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
 - (b) Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
 - (c) **Cache 1:**
 - i. Prepare $|s, x \oplus 1, 0, 0\rangle$ on an internal register Q' , and query O_f^{SEQ} on Q' . Let $|s, x \oplus 1, y, b\rangle$ be the final state of Q' .
 - ii. If $b = 0$, then do the following:
 - A. If C contains entry (s, x, σ) , then remove this entry from C .
 - B. Else, add (s, x, σ) to C .
 - iii. Uncompute item 3(c)i.
 - (d) Initialize $|+\rangle_B$. Query O_f^{SEQ} on $|s, x, u\rangle_{Q_s \times Q_x \times Q_u} \otimes |+\rangle_B$.
 - (e) **Cache 2:** Repeat item 3c.
 - (f) Output register Q .
4. Output the program $P_f^{\text{Sim}} = O_{\text{Sim}}^{O_f^{\text{SEQ}}}$.

Figure 5: SEQ Simulator $\text{Sim}^{O_f^{\text{SEQ}}}$

Next, the following sequence of hybrids transforms the oracle $O_{f,G,\text{pk}}$ from the real-world program P_f into the oracle $O_{\text{Sim}}^{O_f^{\text{SEQ}}}$ from the ideal-world simulator $\text{Sim}^{O_f^{\text{SEQ}}}$. The hybrids are computationally indistinguishable.

Let us be more precise about the distinguisher's inputs. At the start of each hybrid, a challenger takes λ, f and samples:

$$\begin{aligned} (\text{sk}, \text{pk}) &\leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) &\leftarrow \text{TokenGen}(\text{sk}) \end{aligned}$$

Then the challenger samples any additional parameters of the hybrid's oracle, such as G , and implements the oracle $O_{f,G,\text{pk}}$ that is defined by the parameters above. Next, the distinguisher's inputs are

$$O_{f,G,\text{pk}}, |T\rangle, s, \text{pk}, f, \text{aux}_f$$

Note that these are the inputs given to the distinguisher in the SEQ security definition.

We say that two hybrids are *computationally indistinguishable* if any QPT distinguisher, given access to the input distribution above, has negligible advantage at distinguishing the hybrids. We say that two hybrids are *perfectly indistinguishable* if any computationally unbounded distinguisher, given access to the input distribution above, has 0 advantage at distinguishing the hybrids.

Finally, for each hybrid description below, any differences from the previous hybrid will be written in **red**.

Hybrid 1. The real-world oracle $O_{f,G,\text{pk}}$ from program P_f (fig. 3).

- **Additional Parameters:** The oracle is parametrized by a random oracle G sampled uniformly at random.
- 1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
- 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
- 3. Query G to compute $r = G(s, x, \sigma)$:

$$|s, x, \sigma\rangle_{Q_s \times Q_x \times Q_\sigma} \otimes |0\rangle_R \rightarrow |s, x, \sigma\rangle \otimes |r\rangle$$

- 4. Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$

- 5. Uncompute item 3.
- 6. Output register Q .

Hybrid 2. Implement the random oracle G as a compressed oracle.

- **The oracle has a database register D that stores a (possibly empty) set of tuples of the form $(s', x', \sigma', r') \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{R}$. D is initialized to the empty list $|\emptyset\rangle$.**
- 1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
- 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
- 3. Query G on $|s, x, \sigma\rangle_{Q_s \times Q_x \times Q_\sigma} \otimes |0\rangle_R \otimes |D\rangle$ by applying the following sequence of operations (from right to left):

$$\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}$$

- 4. Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$

- 5. Uncompute item 3.
- 6. Output register Q .

Lemma 4.1. Hybrids 1 and 2 are perfectly indistinguishable.

Proof. The only difference between the hybrids is that the random oracle G is implemented as a compressed oracle in hybrid 2. The two hybrids are perfectly indistinguishable by [Zha18], lemma 4. $\square$

Hybrid 3. Add an SEQ check to ensure that the compressed oracle never records two distinct entries with the same s -value.

- The oracle has a database register D that stores a (possibly empty) set of tuples of the form $(s', x', \sigma', r') \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{R}$. D is initialized to the empty list $|\emptyset\rangle$. **The oracle initializes a control qubit B to $|0\rangle$. At the start of every query, $B = |0\rangle$.**
- 1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
- 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
- 3. **SEQ Check:** Check that D contains no entry (s', x', σ', r') such that $s' = s$ but $(x', \sigma') \neq (x, \sigma)$. If the check passes, then flip B to $|1\rangle$.
- 4. **If $B = |1\rangle$:**
 - (a) Query G on $|s, x, \sigma\rangle_{Q_s \times Q_x \times Q_\sigma} \otimes |0\rangle_R \otimes |D\rangle$ by applying the following sequence of operations (from right to left):
$$\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}$$
 - (b) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :
$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$
 - (c) Uncompute item 4a by applying the following sequence of operations (from right to left):
$$\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}$$
- 5. **Uncompute step 3.**
- 6. Output register Q .

Lemma 4.2. Hybrids 2 and 3 are computationally indistinguishable.

Proof. The difference between the hybrids is that hybrid 3 adds an SEQ check (step 3), which aborts if D already records a value (s, x', σ') such that $(x', \sigma') \neq (x, \sigma)$. If hybrids 2 and 3 were not computationally indistinguishable, then we could break the strong security of the signature scheme (definition 3.4). If the hybrids are distinguishable, then with non-negligible probability, the SEQ check in hybrid 3, step 3 fails to flip B to $|1\rangle$. If this occurs, we can measure the Q and D registers to obtain values $(x, \sigma) \neq (x', \sigma')$.

Furthermore, both values are valid message-signature pairs for serial number s . We know that $\text{Ver}(\text{pk}, s, x, \sigma) = 1$ because otherwise, the oracle would have aborted in step 2 before reaching step 3. We also know that $\text{Ver}(\text{pk}, s, x', \sigma') = 1$ because (s, x', σ') are recorded on the database D . The only values of (s, x', σ') recorded on D are values that were previously queried to the compressed oracle in steps 4a or 4c. The only values that were queried in those steps must satisfy $\text{Ver}(\text{pk}, s, x', \sigma') = 1$ because otherwise, the query would have aborted in step 2.

Let us explain the attack above in more detail. Assume toward contradiction that there is a QPT adversary that queries the hybrid 3 oracle and with non-negligible probability, at least one of these queries distinguishes hybrids 2 and 3 with non-negligible advantage. Next, in the strong unforgeability game (definition 3.4), the challenger samples

$$\begin{aligned} (\text{sk}, \text{pk}) &\leftarrow \text{SigToken.KeyGen}(1^\lambda) \\ (|T\rangle, s) &\leftarrow \text{SigToken.TokenGen}(\text{sk}) \end{aligned}$$

and gives the signature adversary $(1^\lambda, \text{pk}, |T\rangle, s)$. Then our signature adversary does the following:

1. Implement the oracle described in hybrid 3 using (pk, f) .
2. Run the distinguisher, giving it query access to the hybrid 3 oracle.
3. If step 3 of the oracle ever ends with $B = |0\rangle$, then measure the Q and D registers to obtain (s, x, σ) and (s, x', σ') .
4. Output $(x, x', \sigma, \sigma', s)$.

First, pk has the same distribution as it does in hybrid 3. Second, our signature adversary runs in polynomial time because so does the distinguisher. Third, with non-negligible probability, step 3 ends with $B = |0\rangle$ because we've correctly simulated hybrid 3, and the distinguisher distinguishes the two hybrids with non-negligible advantage. Fourth, when step 3 ends with $B = |0\rangle$, the signature adversary outputs a correct answer, which satisfies:

$$\begin{aligned}
 (x, \sigma) &\neq (x', \sigma') \\
 1 &= \text{Ver}(pk, s, x, \sigma) \\
 1 &= \text{Ver}(pk, s, x', \sigma')
 \end{aligned}$$

Therefore, the signature adversary breaks the strong security of the OSS scheme.

In summary, we've shown that hybrids 2 and 3 are computationally indistinguishable, assuming that the OSS scheme is strongly secure. $\square$

Hybrid 4. We will split our database D into two registers. A smaller database register D stores tuples of the form (s', x', r') , and a cache register C stores tuples of the form (s', x', σ') . Previously, the database stored all these values together: (s', x', σ', r') .

First, the combined database d' should have unique s' -values in order to be splittable. Let d be a set of tuples of the form (s', x', σ', r') . d is **splittable** if no two entries of d have the same s' value.

Second, here is an algorithm Split that splits d into a smaller database d' and cache c' .

1. Split takes as input an (empty) cache c and a combined database d .
2. If $c \neq \emptyset$ or d is not splittable, then output (c, d) and halt. Otherwise continue.
3. Initialize $c' = d' = \emptyset$.
4. For every entry $(s', x', \sigma', r') \in d$, add (s', x', σ') to c' , and add (s', x', r') to d' .
5. Output (c', d') .

Lemma 4.3. Split is invertible.

Proof. Let us consider how $\text{Split}(c, d)$ acts on the following exhaustive cases:

1. Case 1: $(c, d) = (\emptyset, \emptyset)$. Then $\text{Split}(c, d)$ outputs $(c', d') = (\emptyset, \emptyset) = (c, d)$.
2. Case 2: $c \neq \emptyset$. Then $\text{Split}(c, d)$ outputs $(c', d') = (c, d)$.
3. Case 3: $c = \emptyset$ and $d \neq \emptyset$ and d is not splittable. Then $\text{Split}(c, d)$ outputs $(c', d') = (c, d)$.
4. Case 4: $c = \emptyset$ and $d \neq \emptyset$ and d is splittable. Then $\text{Split}(c, d)$ follows through and actually splits (c, d) . d' is a non-empty database containing tuples of the form (s', x', r') , where every s' is unique. c' is a non-empty database containing tuples of the form (s', x', σ') . Furthermore, there is a perfect matching between entries of c' and entries of d' , where the matching pairs up entries with the same s' value.

Next, here is how to invert $\text{Split}(c, d)$ given its output (c', d') . First, check whether d' is a non-empty database containing tuples with three entries each: (s', x', r') . If so, then we must be in case 4. For cases 1 - 3, $d' = d$, so d' is either empty or it contains tuples with four entries each: (s', x', σ', r') .

Second, if we are in case 4, then we set $c = \emptyset$ and construct d as follows. For each pair of entries $(s', x', \sigma') \in c'$ and $(s', x', r') \in d'$ with the same s' -value, add (s', x', σ', r') to d . This constructs d correctly because every entry of d has a unique s' -value.

Third, if we are not in case 4, then set $(c, d) = (c', d')$. This correctly inverts Split on cases 1-3. $\square$

Next, let U_{Split} be a unitary that computes $\text{Split}(c, d)$ in-place:

$$|c, d\rangle_{C \times D} \rightarrow |\text{Split}(c, d)\rangle_{C \times D}$$

U_{Split} is a unitary because Split is invertible.

Finally, let us define the oracle for hybrid 4.

- The oracle has a database register D , a cache register C , and control qubit B . $B \times C \times D$ are initialized to $|0, \emptyset, \emptyset\rangle$.

1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
3. **SEQ Check:**
 - (a) Apply $U_{\text{Split}}^\dagger$ to $C \times D$.
 - (b) Check that D contains no entry (s', x', σ', r') such that $s' = s$ but $(x', \sigma') \neq (x, \sigma)$. If the check passes, then flip B to $|1\rangle$.
 - (c) Apply U_{Split} to $C \times D$.
4. If $B = |1\rangle$:
 - (a) Query G on $|s, x, \sigma\rangle_{Q_s \times Q_x \times Q_\sigma} \otimes |0\rangle_R \otimes |c, d\rangle_{C \times D}$ by applying the following sequence of operations (from right to left):

$$U_{\text{Split}} \circ \text{Decomp}_{s, x, \sigma} \circ \text{CO}' \circ \text{Decomp}_{s, x, \sigma} \circ U_{\text{Split}}^\dagger$$

- (b) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$

- (c) Uncompute item 4a by applying the following sequence of operations (from right to left):

$$U_{\text{Split}} \circ \text{Decomp}_{s, x, \sigma} \circ \text{CO}' \circ \text{Decomp}_{s, x, \sigma} \circ U_{\text{Split}}^\dagger$$

5. Uncompute step 3.
6. Output register Q .

Lemma 4.4. Hybrids 3 and 4 are perfectly indistinguishable.

Proof. Hybrid 4 differs from hybrid 3 in the following ways. Hybrid 4 includes a cache register C , and the oracle applies $U_{\text{Split}}^\dagger$ and U_{Split} many times. We will show that most of these applications of $U_{\text{Split}}^\dagger$ and U_{Split} cancel out with each other.

First, the first time that the oracle applies $U_{\text{Split}}^\dagger$ or U_{Split} in hybrid 4 is during step 3 on the first query that reaches that step without aborting. Initially, $C \times D$ are in the state $|\emptyset, \emptyset\rangle$, and the application of $U_{\text{Split}}^\dagger$ maps this state to $|\emptyset, \emptyset\rangle$. Note that the state of D after this application ($D = |\emptyset\rangle$) is the same as the initial state of D in hybrid 3.

Second, the final time that the oracle applies $U_{\text{Split}}^\dagger$ or U_{Split} is during step 5. This application of U_{Split} does not affect the distinguisher's view because the distinguisher makes no further queries, and U_{Split} acts on internal registers of the oracle.

Third, we claim that all other applications of $U_{\text{Split}}^\dagger$ or U_{Split} (between the first and last applications) cancel out with each other. It suffices to consider branches of the superposition on which B is flipped to $|1\rangle$ because otherwise, steps 3 and 5 cancel out. The application of U_{Split} in step 3 cancels with the application of $U_{\text{Split}}^\dagger$ in step 4a. Next, the application of U_{Split} in 4a cancels with the application of $U_{\text{Split}}^\dagger$ in 4c. $U_{\text{Split}}^\dagger$ commutes with step 4b because $U_{\text{Split}}^\dagger$ acts on registers $C \times D$, and step 4b acts on a disjoint set of registers $Q \times R$. The application of U_{Split} in 4c cancels out with the application of $U_{\text{Split}}^\dagger$ in step 5. Finally, the application of U_{Split} in step 5 cancels out with the application of $U_{\text{Split}}^\dagger$ in step 3 of the next query. $U_{\text{Split}}^\dagger$ (which acts on $C \times D$) commutes with step 2 (which acts on Q) because they act on disjoint registers. $\square$

Hybrid 5. In this hybrid, the compressed oracle operations Decomp and CO' act on a smaller database that only records tuples of the form (s', x', r') (and which do not include σ').

Since the compressed oracle no longer records σ' , we must change our Decomp and CO' procedures. Instead of using $\text{Decomp}_{s,x,\sigma}$, we use $\text{Decomp}_{s,x}$, which is defined analogously (section 3.1). $\text{Decomp}_{s,x}$ acts on databases d comprising tuples of the form (s', x', r') . Given any database d that does not contain a tuple of the form $(s, x, *)$, $\text{Decomp}_{s,x}$ swaps the states

$$|d\rangle \text{ and } |\psi(d, (s, x), 0)\rangle$$

and acts as the identity on all other states.

Additionally, let CO' be replaced with CO'' that ignores σ . CO'' takes a query (s, x, r) and a database d that contains a tuple of the form (s, x, r') , and does the following:

$$|s, x, r\rangle_{Q \times R} \otimes |d\rangle_D \rightarrow |s, x, r \oplus r'\rangle_{Q \times R} \otimes |d\rangle_D$$

Next, let us define a unitary $\text{Cache}_{s,x,\sigma}$ that updates register C to record (s, x, σ) whenever D records (s, x, r') . $\text{Cache}_{s,x,\sigma}$ resembles the chacheing step of Sim . $\text{Cache}_{s,x,\sigma}$ acts on $C \times D$ as follows:

1. Check if D contains an entry (s', x', r') such that $s' = s$ and $x' \neq x \oplus 1$. If so, then do the following:
 - (a) If C contains the entry (s, x, σ) , then remove this entry from C .
 - (b) Else, add (s, x, σ) to C .

Finally, let us define the oracle distribution for hybrid 5.

- The oracle has a database register D , a cache register C , and control qubit B . $B \times C \times D$ are initialized to $|0, \emptyset, \emptyset\rangle$.

1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
3. **SEQ Check:**
 - (a) Apply $U_{\text{Split}}^\dagger$ to $C \times D$.
 - (b) Check that D contains no entry (s', x', σ', r') such that $s' = s$ but $(x', \sigma') \neq (x, \sigma)$. If the check passes, then flip B to $|1\rangle$.
 - (c) Apply U_{Split} to $C \times D$.
4. If $B = |1\rangle$:
 - (a) Query G on $|s, x, \sigma\rangle_{Q_s \times Q_x \times Q_\sigma} \otimes |0\rangle_R \otimes |c, d\rangle_{C \times D}$ by applying the following sequence of operations (from right to left):

$$\text{Cache}_{s,x,\sigma}^\dagger \circ \text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x} \circ \text{Cache}_{s,x,\sigma}$$
 - (b) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$
 - (c) Uncompute item 4a by applying the following sequence of operations (from right to left):

$$\text{Cache}_{s,x,\sigma}^\dagger \circ \text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x} \circ \text{Cache}_{s,x,\sigma}$$
5. Uncompute step 3.
6. Output register Q .

Lemma 4.5. $\text{Cache}_{s,x,\sigma}$ is its own inverse: $\text{Cache}_{s,x,\sigma}^\dagger = \text{Cache}_{s,x,\sigma}$.

Proof. $\text{Cache}_{s,x,\sigma}$ changes the state on register C (by adding or removing (s, σ)) controlled on the state of register D . Applying this operation a second time will undo the action of the first. If the first $\text{Cache}_{s,x,\sigma}$ added (s, σ) to C , then the second $\text{Cache}_{s,x,\sigma}$ removes (s, σ) , and vice versa. $\square$

Lemma 4.6. Hybrids 4 and 5 are perfectly indistinguishable.

Proof. The only difference between the hybrids is how they implement queries to G . See steps 4a and 4c of hybrid 4 and steps 4a and 4c of hybrid 5. We will show that each of these steps computes the same function as their analog in hybrid 4.

First, let us define $\Pi_{s,x,\sigma}$ to project the $C \times D$ registers onto *valid* states, where $C \times D$ are the result of splitting some combined database d , and (s, x, σ) is an allowed query for this database. Formally, $\Pi_{s,x,\sigma}$ projects onto all eigenstates of the following form.

$$|c'\rangle_C \otimes |d'\rangle_D, \quad \text{or} \\ |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D$$

where $\tilde{r}' \in \mathcal{R} \setminus \{0\}$, $(c', d') = \text{Split}(\emptyset, d)$,

and d is a splittable database that does not contain

any tuple (s', x', σ', r') for which $s' = s$

(7)

Recall from eq. (1) that

$$|\psi(d', (s, x), \tilde{r}')\rangle = \frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r}' \rangle} |d' \cup \{(s, x, r')\}\rangle$$

Second, let us assume that we are given a state in the span of $\Pi_{s,x,\sigma}$ and let us apply to the state either the operation in hybrid 4, step 4a

$$U_{\text{Split}} \circ \text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma} \circ U_{\text{Split}}^\dagger \quad (8)$$

or the operation in hybrid 5, step 4a

$$\text{Cache}_{s,x,\sigma}^\dagger \circ \text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x} \circ \text{Cache}_{s,x,\sigma} \quad (9)$$

We will show the two operations act equivalently on the state and that the final state is also in the span of $\Pi_{s,x,\sigma}$.

Note that $\text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x}$ is simply the query operation CO to a compressed oracle that stores tuples of the form (s, x, r) . Likewise, $\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}$ is simply the query operation CO to a compressed oracle that stores tuples of the form (s, x, σ, r) . We can use lemma 3.2 to describe how these operations act on eigenstates of $\Pi_{s,x,\sigma}$.

Furthermore, let us express the R register in the $\{|\phi(\tilde{r})\rangle\}_{\tilde{r} \in \mathcal{R}}$ basis. Recall from eq. (2) that

$$|\phi(\tilde{r})\rangle = \frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r \in \mathcal{R}} (-1)^{\langle r, \tilde{r} \rangle} |r\rangle$$

Additionally, lemma 3.1 shows that $\{|\phi(\tilde{r})\rangle\}_{\tilde{r} \in \mathcal{R}}$ is an orthonormal basis for register R .

Now, let us consider the following cases:

1. Case 1: $R \times C \times D$ contains $|\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D$ where $\tilde{r} = 0$ and (c', d') satisfy the conditions of eq. (7).

Let us apply the operation from hybrid 5 (eq. (9)):

$$\begin{aligned} |\phi(0)\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D &\xrightarrow{\text{Cache}_{s,x,\sigma}^\dagger} |\phi(0)\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D \\ &\xrightarrow{\text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x}} |\phi(0)\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D \\ &\xrightarrow{\text{Cache}_{s,x,\sigma}^\dagger} |\phi(0)\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D \end{aligned}$$

Likewise, let us apply the operation from hybrid 4 (eq. (8)):

$$\begin{aligned} |\phi(0)\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D &\xrightarrow{U_{\text{Split}}^\dagger} |\phi(0)\rangle_R \otimes |\mathbf{0}\rangle_C \otimes |d\rangle_D \\ &\xrightarrow{\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}} |\phi(0)\rangle_R \otimes |\mathbf{0}\rangle_C \otimes |d\rangle_D \\ &\xrightarrow{U_{\text{Split}}} |\phi(0)\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D \end{aligned}$$

This shows that the operations in eqs. (8) and (9) act equivalently on this state. Furthermore, the final state $|c'\rangle_C \otimes |d'\rangle_D$ is still in the span of $\Pi_{s,x,\sigma}$.

2. Case 2: $R \times C \times D$ contains $|\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D$ where $\tilde{r} \neq 0$ and (c', d') satisfy the conditions of eq. (7).

Let us apply the operation from hybrid 5 (eq. (9)):

$$\begin{aligned}
|\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D &\xrightarrow{\text{Cache}_{s,x,\sigma}} |\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D \\
&\xrightarrow{\text{Decomp}_{s,x} \circ \text{CO}' \circ \text{Decomp}_{s,x}} |\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |\psi(d', (s, x), \tilde{r})\rangle_D \\
&\xrightarrow{\text{Cache}_{s,x,\sigma}^\dagger} |\phi(\tilde{r})\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r})\rangle_D
\end{aligned}$$

Likewise, let us apply the operation from hybrid 4 (eq. (8)):

$$\begin{aligned}
|\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D &\xrightarrow{U_{\text{Split}}^\dagger} |\phi(\tilde{r})\rangle_R \otimes |\emptyset\rangle_C \otimes |d\rangle_D \\
&\xrightarrow{\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}} |\phi(\tilde{r})\rangle_R \otimes |\emptyset\rangle_C \otimes |\psi(d, (s, x, \sigma), \tilde{r})\rangle_D \\
&\xrightarrow{U_{\text{Split}}} |\phi(\tilde{r})\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r})\rangle_D
\end{aligned}$$

This shows that the operations in eqs. (8) and (9) act equivalently on this state. Furthermore, the final state $|c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r})\rangle_D$ is still in the span of $\Pi_{s,x,\sigma}$.

3. Case 3: $R \times C \times D$ contains $|\phi(\tilde{r})\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D$, where $\tilde{r} = \tilde{r}'$, $\tilde{r}' \neq 0$, and (c', d') satisfy the conditions of eq. (7).

Let us apply the operation from hybrid 5 (eq. (9)):

$$\begin{aligned}
|\phi(\tilde{r}')\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D &\xrightarrow{\text{Cache}_{s,x,\sigma}} |\phi(\tilde{r}')\rangle_R \otimes |c'\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D \\
&\xrightarrow{\text{Decomp}_{s,x} \circ \text{CO}' \circ \text{Decomp}_{s,x}} |\phi(\tilde{r}')\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D \\
&\xrightarrow{\text{Cache}_{s,x,\sigma}^\dagger} |\phi(\tilde{r}')\rangle_R \otimes |c'\rangle_C \otimes |d'\rangle_D
\end{aligned}$$

Likewise, let us apply the operation from hybrid 4 (eq. (8)):

$$\begin{aligned}
|\phi(\tilde{r}')\rangle |c' \cup \{(s, x, \sigma)\}\rangle |\psi(d', (s, x), \tilde{r}')\rangle &= |\phi(\tilde{r}')\rangle \left(\frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r}' \rangle} \cdot |c' \cup \{(s, x, \sigma)\}\rangle |d' \cup \{(s, x, r')\}\rangle \right) \\
&\xrightarrow{U_{\text{Split}}^\dagger} |\phi(\tilde{r}')\rangle \left(\frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r}' \rangle} \cdot |\emptyset\rangle |d \cup \{(s, x, \sigma, r')\}\rangle \right) \\
&= |\phi(\tilde{r}')\rangle |\emptyset\rangle |\psi(d, (s, x, \sigma), \tilde{r}')\rangle \\
&\xrightarrow{\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}} |\phi(\tilde{r}')\rangle |\emptyset\rangle |d\rangle \\
&\xrightarrow{U_{\text{Split}}} |\phi(\tilde{r}')\rangle |c'\rangle |d'\rangle
\end{aligned}$$

This shows that the operations in eqs. (8) and (9) act equivalently on this state. Furthermore, the final state $|c'\rangle |d'\rangle$ is still in the span of $\Pi_{s,x,\sigma}$.

4. Case 4: $R \times C \times D$ contains $|\phi(\tilde{r})\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D$, where $\tilde{r} \neq \tilde{r}'$, $\tilde{r}' \neq 0$, and (c', d') satisfy the conditions of eq. (7).

Let us apply the operation from hybrid 5 (eq. (9)):

$$\begin{aligned}
|\phi(\tilde{r})\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D &\xrightarrow{\text{Cache}_{s,x,\sigma}} |\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |\psi(d', (s, x), \tilde{r}')\rangle_D \\
&\xrightarrow{\text{Decomp}_{s,x} \circ \text{CO}' \circ \text{Decomp}_{s,x}} |\phi(\tilde{r})\rangle_R \otimes |c'\rangle_C \otimes |\psi(d', (s, x), \tilde{r} + \tilde{r}')\rangle_D \\
&\xrightarrow{\text{Cache}_{s,x,\sigma}^\dagger} |\phi(\tilde{r})\rangle_R \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |\psi(d', (s, x), \tilde{r} + \tilde{r}')\rangle_D
\end{aligned}$$

Likewise, let us apply the operation from hybrid 4 (eq. (8)):

$$\begin{aligned}
|\phi(\tilde{r})\rangle |c' \cup \{(s, x, \sigma)\}\rangle |\psi(d', (s, x), \tilde{r}')\rangle &= |\phi(\tilde{r})\rangle \left(\frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r}' \rangle} \cdot |c' \cup \{(s, x, \sigma)\}\rangle |d' \cup \{(s, x, r')\}\rangle \right) \\
&\xrightarrow{U_{\text{Split}}^\dagger} |\phi(\tilde{r})\rangle \left(\frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r}' \rangle} \cdot |\emptyset\rangle |d \cup \{(s, x, \sigma, r')\}\rangle \right) \\
&= |\phi(\tilde{r})\rangle |\emptyset\rangle |\psi(d, (s, x, \sigma), \tilde{r}')\rangle \\
&\xrightarrow{\text{Decomp}_{s,x,\sigma} \circ \text{CO}' \circ \text{Decomp}_{s,x,\sigma}} |\phi(\tilde{r})\rangle |\emptyset\rangle |\psi(d, (s, x, \sigma), \tilde{r} + \tilde{r}')\rangle \\
&= |\phi(\tilde{r})\rangle \left(\frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r} + \tilde{r}' \rangle} \cdot |\emptyset\rangle |d \cup \{(s, x, \sigma, r')\}\rangle \right) \\
&\xrightarrow{U_{\text{Split}}} |\phi(\tilde{r})\rangle \left(\frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r' \in \mathcal{R}} (-1)^{\langle r', \tilde{r} + \tilde{r}' \rangle} \cdot |c' \cup \{(s, x, \sigma)\}\rangle |d' \cup \{(s, x, r')\}\rangle \right) \\
&= |\phi(\tilde{r})\rangle |c' \cup \{(s, x, \sigma)\}\rangle |\psi(d', (s, x), \tilde{r} + \tilde{r}')\rangle
\end{aligned}$$

This shows that the operations in eqs. (8) and (9) act equivalently on this state. Furthermore, the final state $|c' \cup \{(s, x, \sigma)\}\rangle |\psi(d', (s, x), \tilde{r} + \tilde{r}')\rangle$ is still in the span of $\Pi_{s,x,\sigma}$.

Third, the SEQ check (step 3 in hybrid 4 or step 3 in hybrid 5) is the same in the two hybrids. Likewise, step 5 in hybrid 4 or step 5 in hybrid 5 are the same. Furthermore, if the state of $C \times D$ is in the span of $\Pi_{s,x,\sigma}$ at the beginning of the SEQ check, then it will still be in the span of $\Pi_{s,x,\sigma}$ at the end of the SEQ check. Intuitively, this is because the SEQ check always flips B in step 3b if the initial state is in the span of $\Pi_{s,x,\sigma}$, and the applications of U_{Split} and $U_{\text{Split}}^\dagger$ cancel out.

If a state of $C \times D$ is in the span of $\Pi_{s,x,\sigma}$, then it is in the span of states of the form:

$$\begin{aligned}
|c'\rangle_C \otimes |d'\rangle_D \\
|c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |d' \cup \{(s, x, r')\}\rangle_D
\end{aligned}$$

where $(c', d') = \text{Split}(\emptyset, d)$ and d is a splittable database that does not contain $(s, *, *, *)$.

Let us consider how the SEQ check acts on each of the states listed above.

1. Case 1: $B \times C \times D$ contains $|0\rangle_B \otimes |c'\rangle_C \otimes |d'\rangle_D$. Then the SEQ check acts on this state as follows:

$$\begin{aligned}
|0\rangle_B \otimes |c'\rangle_C \otimes |d'\rangle_D &\xrightarrow{U_{\text{Split}}^\dagger} |0\rangle_B \otimes |\emptyset\rangle_C \otimes |d\rangle_D \\
&\xrightarrow{\text{Step 3b}} |1\rangle_B \otimes |\emptyset\rangle_C \otimes |d\rangle_D \\
&\xrightarrow{U_{\text{Split}}} |1\rangle_B \otimes |c'\rangle_C \otimes |d'\rangle_D
\end{aligned}$$

The reason that step 3b flips B is that d does not contain any entry of the form $(s, *, *, *)$.

2. Case 2: $B \times C \times D$ contains $|0\rangle_B \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |d' \cup \{(s, x, r')\}\rangle_D$. Then the SEQ check acts on this state as follows:

$$\begin{aligned}
|0\rangle_B \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |d' \cup \{(s, x, r')\}\rangle_D &\xrightarrow{U_{\text{Split}}^\dagger} |0\rangle_B \otimes |\emptyset\rangle_C \otimes |d \cup \{(s, x, \sigma, r')\}\rangle_D \\
&\xrightarrow{\text{Step 3b}} |1\rangle_B \otimes |\emptyset\rangle_C \otimes |d \cup \{(s, x, \sigma, r')\}\rangle_D \\
&\xrightarrow{U_{\text{Split}}} |1\rangle_B \otimes |c' \cup \{(s, x, \sigma)\}\rangle_C \otimes |d' \cup \{(s, x, r')\}\rangle_D
\end{aligned}$$

The reason that step 3b flips B is that (s, x, σ, r') is the only entry of $d \cup \{(s, x, \sigma, r')\}$ with that particular s -value. Then there is no entry (s', x', σ', r') of $d \cup \{(s, x, \sigma, r')\}$ such that $s = s'$ but $(x', \sigma') \neq (x, \sigma)$.

This shows that if the state of $C \times D$ is in the span of $\Pi_{s,x,\sigma}$ at the start of the SEQ check, then the check will flip B to $|1\rangle$ with certainty, and the final state of $C \times D$ will be the same as its initial state.

Fourth, at the start of any application of eq. (8) in hybrid 4 or eq. (9) in hybrid 5, the state of the $C \times D$ registers is in the span of $\Pi_{s,x,\sigma}$. We will prove this inductively. For the base case: initially, $C \times D$ is in the state $|\emptyset\rangle |\emptyset\rangle$.

For the inductive case: the only steps that act on the $C \times D$ registers are the SEQ check (steps 3 and 5) or the queries to the compressed oracle (steps 4a and 4c). We've already shown that these steps map states in the span of $\Pi_{s,x,\sigma}$ to states in the span of $\Pi_{s,x,\sigma}$. Therefore, at the start of any query to the compressed oracle, the state of $C \times D$ will be in the span of $\Pi_{s,x,\sigma}$.

Fifth, we know that the compressed oracle queries in hybrid 4 (eq. (8)) and hybrid 5 (eq. (9)) act equivalently on any state in the span of $\Pi_{s,x,\sigma}$. The hybrids are identical except for their compressed oracle queries. Therefore, hybrids 4 and 5 perform equivalent operations. $\square$

Hybrid 6. We cancel out adjacent applications of $\text{Cache}_{s,x,\sigma}$ and $\text{Cache}_{s,x,\sigma}^\dagger$, and we explicitly switch to a compressed oracle H that takes queries of the form (s, x) instead of (s, x, σ) .

Previously, we implemented a compressed oracle $G : \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \rightarrow \mathcal{R}$. Now we implement a compressed oracle $H : \mathcal{S}_\lambda \times \mathcal{X} \rightarrow \mathcal{R}$.

- **The oracle has a database register D implementing a compressed random oracle $H : \mathcal{S}_\lambda \times \mathcal{X} \rightarrow \mathcal{R}$.** The oracle also has a cache register C , and control qubit B . $B \times C \times D$ are initialized to $|0, \emptyset, \emptyset\rangle$.
- 1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
- 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
- 3. **SEQ Check:**
 - (a) Apply $U_{\text{Split}}^\dagger$ to $C \times D$.
 - (b) Check that D contains no entry (s', x', σ', r') such that $s' = s$ but $(x', \sigma') \neq (x, \sigma)$. If the check passes, then flip B to $|1\rangle$.
 - (c) Apply U_{Split} to $C \times D$.
- 4. If $B = |1\rangle$:
 - (a) **Cache 1:** Check if D contains an entry (s', x', r') such that $s' = s$ and $x' \neq x \oplus 1$. If so, then do the following:
 - i. If C contains the entry (s, x, σ) , then remove this entry from C .
 - ii. Else, add (s, x, σ) to C .
 - (b) Query H on $|s, x\rangle_{Q_s \times Q_x} \otimes |0\rangle_R \otimes |d\rangle_D$.
 - (c) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$

- (d) Uncompute item 4b.
 - (e) **Cache 2: Repeat step 4a.**
5. Uncompute step 3.
 6. Output register Q .

Lemma 4.7. Hybrids 5 and 6 are perfectly indistinguishable.

Proof. We can change hybrid 5 into hybrid 6 without changing the function that the hybrid computes.

First, the $\text{Cache}_{s,x,\sigma}$ operation at the start of hybrid 5 step 4a is now written as its own step in hybrid 6 step 4a. Additionally, the $\text{Cache}_{s,x,\sigma}^\dagger$ operation at the end of hybrid 5 step 4c is written as its own step in hybrid 6 step 4e. Note that $\text{Cache}_{s,x,\sigma} = \text{Cache}_{s,x,\sigma}^\dagger$ (lemma 4.5), so the two cache steps of hybrid 6 (steps 4a and 4e) compute the same function.

Second, in hybrid 5, the $\text{Cache}_{s,x,\sigma}^\dagger$ operation at the end of step 4a cancels out with the $\text{Cache}_{s,x,\sigma}$ operation at the start of step 4c. Note that $\text{Cache}_{s,x,\sigma}$ commutes with the intervening step 4b because they act on disjoint registers. $\text{Cache}_{s,x,\sigma}$ acts on registers $C \times D$ whereas step 4b acts on registers $Q \times R$.

Third, the operation

$$\text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x}$$

in hybrid 5 steps 4a and 4c is replaced by a query to the compressed oracle H in hybrid 6, steps 4b and 4d. This is just a notational change because $\text{Decomp}_{s,x} \circ \text{CO}'' \circ \text{Decomp}_{s,x}$ are anyways the operation that the compressed oracle query performs.

These changes transform the hybrid 5 oracle into the hybrid 6 oracle without changing the function that the oracle computes. $\square$

Hybrid 7. Change the SEQ check to only act on D and not C .

- The oracle has a database register D implementing a compressed random oracle $H : \mathcal{S}_\lambda \times \mathcal{X} \rightarrow \mathcal{R}$. The oracle also has a cache register C , and control qubit B . $B \times C \times D$ are initialized to $|0, \emptyset, \emptyset\rangle$.
1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
 3. **SEQ Check:** Check that D contains no entry (s', x', r') such that $s' = s$ but $x' \neq x$. If the check passes, then flip B to $|1\rangle$.
 4. If $B = |1\rangle$:
 - (a) **Cache 1:** Check if D contains an entry (s', x', r') such that $s' = s$ and $x' \neq x \oplus 1$. If so, then do the following:
 - i. If C contains the entry (s, x, σ) , then remove this entry from C .
 - ii. Else, add (s, x, σ) to C .
 - (b) Query H on $|s, x\rangle_{Q_s \times Q_x} \otimes |0\rangle_R \otimes |d\rangle_D$.
 - (c) Compute $y = f(s, x, r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x, r)\rangle_Q \otimes |r\rangle_R$$

- (d) Uncompute item 4b.
- (e) **Cache 2:** Repeat step 4a.
- 5. Uncompute step 3.
- 6. Output register Q .

Lemma 4.8. Hybrids 6 and 7 are computationally indistinguishable.

Proof. The main difference between the hybrids is that the SEQ check in hybrid 6 depends on (s', x', σ') , whereas the SEQ check in hybrid 7 only depends on (s', x') . We can transform the SEQ check in hybrid 6 into the SEQ check in hybrid 7 in a way that gives a distinguisher negligible distinguishing advantage.

First, change the check in hybrid 6 step 3b to the following:

$$\begin{aligned} &\text{Check that } D \text{ contains no entry } (s', x', \sigma', r') \text{ such that} \\ &s' = s \text{ but } x' \neq x. \text{ If the check passes, then flip } B \text{ to } |1\rangle. \end{aligned} \tag{10}$$

Now we check that $x' \neq x$ instead of checking that $(x', \sigma') \neq (x, \sigma)$.

Second, any PPT distinguisher has negligible advantage at distinguishing an oracle with the old check from an oracle with a new check. The only time that the two checks give different results is when Q contains (s, x, σ, u) and D contains (s', x', σ', r') such that $s' = s$, $x' = x$, but $\sigma' \neq \sigma$. In this case, we could measure the Q and D registers to obtain an answer $(x, x', \sigma, \sigma', s)$ that breaks the strong unforgeability of the OSS scheme. The formal proof of this follows the same argument as the proof of lemma 4.2.

Third, the SEQ check with the change in eq. (10) is the following:

• **SEQ Check:**

1. Apply $U_{\text{Split}}^\dagger$ to $C \times D$.
2. Check that D contains no entry (s', x', σ', r') such that $s' = s$ but $x' \neq x$. If the check passes, then flip B to $|1\rangle$.
3. Apply U_{Split} to $C \times D$.

We claim that this SEQ check is equivalent to the SEQ check from hybrid 7:

- **SEQ Check:** Check that D contains no entry (s', x', r') such that $s' = s$ but $x' \neq x$. If the check passes, then flip B to $|1\rangle$.

This is because the check in eq. (10) only depends on the (s', x') values in D , and Split does not change the (s', x') values in D . If the database is d is splittable, then $\text{Split}(\emptyset, d)$ removes the σ' value from each entry (s', x', σ', r') of d . If d is not splittable, then $\text{Split}(\emptyset, d)$ leaves d unchanged. In either case $\text{Split}(\emptyset, d)$ leaves the (s', x') values of each entry of d unchanged. Therefore, we can remove the $U_{\text{Split}}^\dagger$ and U_{Split} operations from the SEQ check. □

Hybrid 8. Move the cache 1 and cache 2 steps to occur regardless of the value of B .

- The oracle has a database register D implementing a compressed random oracle $H : \mathcal{S}_\lambda \times \mathcal{X} \rightarrow \mathcal{R}$. The oracle also has a cache register C , and control qubit B . $B \times C \times D$ are initialized to $|0, \emptyset, \emptyset\rangle$.
- 1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.

2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
3. **SEQ Check:** Check that D contains no entry (s', x', r') such that $s' = s$ but $x' \neq x$. If the check passes, then flip B to $|1\rangle$.
4. **Cache 1:** Check if D contains an entry (s', x', r') such that $s' = s$ and $x' \neq x \oplus 1$. If so, then do the following:
 - (a) If C contains the entry (s, x, σ) , then remove this entry from C .
 - (b) Else, add (s, x, σ) to C .
5. If $B = |1\rangle$:
 - (a) Query H on $|s, x\rangle_{Q_s \times Q_x} \otimes |0\rangle_R \otimes |d\rangle_D$.
 - (b) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$
 - (c) Uncompute item 5a.
6. **Cache 2:** Repeat step 4.
7. Uncompute step 3.
8. Output register Q .

Lemma 4.9. Hybrids 7 and 8 are perfectly indistinguishable.

Proof. The only difference between the hybrids is where the cache 1 and cache 2 steps occur. In hybrid 7, they only occur if $B = |1\rangle$. In hybrid 8, they occur regardless of B 's value.

First, consider the case where $B = |1\rangle$ at the end of the SEQ check. Then the two hybrids act the same because they both apply cache 1 and cache 2.

Second, consider the case where $B = |0\rangle$ at the end of the SEQ check. Then the two hybrids still compute equivalent functions. Hybrid 7 does not apply either cache 1 or cache 2, whereas hybrid 8 applies cache 1 followed immediately by cache 2. Note that Cache 1 and cache 2 are inverses because the cache operation is its own inverse (lemma 4.5). Therefore, applying the cache operation twice is equivalent to not applying it at all.

We've shown that regardless of the value of B , hybrids 7 and 8 compute equivalent functions. $\square$

Hybrid 9. Change the order of the cache 1 step and the SEQ check.

- The oracle has a database register D implementing a compressed random oracle $H : \mathcal{S}_\lambda \times \mathcal{X} \rightarrow \mathcal{R}$. The oracle also has a cache register C , and control qubit B . $B \times C \times D$ are initialized to $|0, \emptyset, \emptyset\rangle$.
1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
 3. **Cache 1:** Check if D contains an entry (s', x', r') such that $s' = s$ and $x' \neq x \oplus 1$. If so, then do the following:

- (a) If C contains the entry (s, x, σ) , then remove this entry from C .
- (b) Else, add (s, x, σ) to C .
- 4. **SEQ Check:** Check that D contains no entry (s', x', r') such that $s' = s$ but $x' \neq x$. If the check passes, then flip B to $|1\rangle$.
- 5. If $B = |1\rangle$:
 - (a) Query H on $|s, x\rangle_{Q_s \times Q_x} \otimes |0\rangle_R \otimes |d\rangle_D$.
 - (b) Compute $y = f(s, x; r)$, and CNOT y onto Q_u :

$$|s, x, \sigma, u\rangle_Q \otimes |r\rangle_R \rightarrow |s, x, \sigma, u \oplus f(s, x; r)\rangle_Q \otimes |r\rangle_R$$
 - (c) Uncompute item 5a.
- 6. **SEQ Check:** Uncompute step 4.
- 7. **Cache 2:** Repeat step 3.
- 8. Output register Q .

Lemma 4.10. Hybrids 8 and 9 are perfectly indistinguishable.

Proof. The difference between the hybrids is the order in which they perform the cache and SEQ check operations.

However, the cache operation commutes with the SEQ check. Cache applies a unitary to C controlled on the computational basis value of $Q \times D$. The SEQ check applies a unitary to B controlled on the computational basis value of $Q \times D$. Such operations commute.

Therefore, hybrids 8 and 9 compute equivalent functions. $\square$

Hybrid 10. Replace steps 4 - 6 from hybrid 9 with a call to the SEQ oracle.

- The oracle stores an instantiation of the SEQ oracle O_f^{SEQ} (fig. 2), which stores a database register D . The oracle also has a cache register C . $C \times D$ are initialized to $|\emptyset, \emptyset\rangle$.

1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
3. **Cache 1:** Check if D contains an entry (s', x', r') such that $s' = s$ and $x' \neq x \oplus 1$. If so, then do the following:
 - (a) If C contains the entry (s, x, σ) , then remove this entry from C .
 - (b) Else, add (s, x, σ) to C .
4. **Initialize $|+\rangle_B$.** Query O_f^{SEQ} on $|s, x, u\rangle_{Q_s \times Q_x \times Q_u} \otimes |+\rangle_B$.
5. **Cache 2:** Repeat step 3.
6. Output register Q .

Lemma 4.11. Hybrids 9 and 10 are perfectly indistinguishable.

Proof. The differences between the hybrids are as follows. Hybrid 9 stores a database D and includes an SEQ check (step 4), a procedure for computing the answer (step 5), and an uncomputation of the SEQ check (step 6). Hybrid 10 has query access to the SEQ oracle, which stores the database D internally. Furthermore, steps 4 - 6 of hybrid 9 are replaced with a query to the SEQ oracle in step 4 of hybrid 10. Note that hybrid 10 queries the SEQ oracle on $B = |+\rangle$. That way, the state of B is unchanged by the bitflip operation X .

The two hybrids are equivalent because the SEQ oracle stores the database D internally, and the query to the SEQ oracle computes the same function as steps 4 - 6 of hybrid 9. $\square$

Hybrid 11. Modify the cache 1 and cache 2 steps to make queries to O_f^{SEQ} rather than operating on D directly.

- The oracle stores an instantiation of the SEQ oracle O_f^{SEQ} (fig. 2), which stores a database register D . The oracle also has a cache register C . $C \times D$ are initialized to $|\emptyset, \emptyset\rangle$.
- 1. **Inputs:** The oracle takes a query register $Q = Q_s \times Q_x \times Q_\sigma \times Q_u$ with eigenstates of the form $|s, x, \sigma, u\rangle$ where $(s, x, \sigma, u) \in \mathcal{S}_\lambda \times \mathcal{X} \times \Sigma_\lambda \times \mathcal{Y}$.
- 2. Check that $\text{SigToken.Ver}(\text{pk}, s, x, \sigma) = 1$. If so, then continue. If not, then abort and output register Q .
- 3. **Cache 1:**
 - (a) Prepare $|s, x \oplus 1, 0, 0\rangle$ on an internal register Q' , and query O_f^{SEQ} on Q' . Let $|s, x \oplus 1, y, b\rangle$ be the final state of Q' .
 - (b) If $b = 0$, then do the following:
 - i. If C contains entry (s, x, σ) , then remove this entry from C .
 - ii. Else, add (s, x, σ) to C .
 - (c) Uncompute item 3a.
- 4. Initialize $|+\rangle_B$. Query O_f^{SEQ} on $|s, x, u\rangle_{Q_s \times Q_x \times Q_u} \otimes |+\rangle_B$.
- 5. **Cache 2:** Repeat step 3.
- 6. Output register Q .

Lemma 4.12. Hybrids 10 and 11 are perfectly indistinguishable.

Proof. The difference between the hybrids is how they implement cache 1 (and cache 2). Let us trace the operation of cache 1 in both hybrids and show that the implementations compute the same function.

First, the $C \times D$ registers have computational-basis eigenstates of the form $|c\rangle_C \otimes |d\rangle_D$. It suffices to consider the operation of cache 1 on each eigenstate $|c\rangle \otimes |d\rangle$.

Second, the cache 1 step of hybrid 10 checks the following condition:

$$\begin{aligned} & d \text{ contains an entry } (s', x', r') \\ & \text{such that } s' = s \text{ and } x' \neq x \oplus 1 \end{aligned} \tag{11}$$

If and only if condition 11 is satisfied, then hybrid 10's cache 1 operation will execute steps 3a and 3b.

Third, let us consider hybrid 11's cache 1 operation. Step 3a queries O_f^{SEQ} on $|s, x \oplus 1, 0, 0\rangle$ to obtain $|s, x \oplus 1, y, b\rangle$. We claim that $b = 0$ if and only if condition 11 is satisfied. This is because O_f^{SEQ}

checks in step 1 whether condition 11 is satisfied. If the condition is satisfied, then O_f^{SEQ} outputs the query register unchanged, so $b = 0$. If the condition is not satisfied, then O_f^{SEQ} answers the query and applies a bitflip to b , resulting in $b = 1$.

Fourth, step 3b of hybrid 11 will execute the same operation as hybrid 10's steps 3a and 3b, if and only if $b = 0$. This step does not change the state of the D register.

Fifth, step 3c of hybrid 11 uncomputes step 3a. This returns the Q' register to its initial state.

In total, the cache 1 operation of hybrid 11 checks whether condition 11 is satisfied, and if so, it modifies C according to steps 3a and 3b of hybrid 10. This shows that hybrids 10 and 11 implement equivalent cache 1 operations, so the two hybrids are equivalent. $\square$

Now we can finish the proof of security. Note that the oracle distribution in hybrid 1 is the real-world distribution of $O_{f,G,\text{pk}}$ from program P_f (fig. 3). Additionally, the oracle distribution in hybrid 11 is the ideal-world distribution of $O_{\text{Sim}}^{\text{SEQ}}$ implemented by Sim (fig. 5).

Next, we have shown that hybrid 1 and hybrid 11 are computationally indistinguishable. This means that any QPT distinguisher given $(|T\rangle, s, \text{pk}, f, \text{aux}_f)$ and oracle access to an oracle sampled from either hybrid 1 or hybrid 11 will have $\text{negl}(\lambda)$ advantage at distinguishing these two cases. This is exactly what the distinguisher in the definition of security (definition 4.1) is asked to distinguish.

In summary, we have constructed a QPT simulator $\text{Sim}^{O_f^{\text{SEQ}}}(1^\lambda, \text{pk}, \text{aux}_f)$ (fig. 5) such that for all $f \in \mathcal{F}$ and all QPT distinguishers D that are given $(|T\rangle, s, \text{pk}, f, \text{aux}_f)$ and either $O_{f,G,\text{pk}}$ or $O_{\text{Sim}}^{\text{SEQ}}(1^\lambda, \text{pk}, \text{aux}_f)$, D has $\text{negl}(\lambda)$ distinguishing advantage. This completes the proof of security.

5 Query-Limited RAM Programs

A useful ability for programs is the ability to maintain and update a state. In a RAM program, the program may modify the memory contents as it is evaluated, resulting in a sequence of memory states $\text{st}_0, \text{st}_1, \dots$. In this section, we show how to limit queries to a RAM program so that any adversarial evaluator can only access the program on a program state that is consistent with the set of queries so far.

As a motivating example, suppose that, instead of purchasing a single query, a customer purchased some budget $\$B$ worth of queries. A given query might cost more or less, depending on how complicated it is. The merchant can give out a program that limits the customer's queries to a total cost of $\$B$.

More formally, the merchant can give out a RAM program whose internal state is the total cost $\$C$ of all queries made so far (this can be much smaller than the list of all queries). When a query is made that costs $\$c_q$, the program updates the state to $\$(C + c_q)$. However, once the internal state goes above $\$B$, the program only ever outputs a helpful message "Budget reached. Please contact us to buy more." Since the evaluator is limited to program states resulting from a consistent evaluation, they cannot evaluate a sequence of queries with budget exceeding $\$B$.

5.1 Definition

RAM Programs. A randomized RAM program consists of a function $P : \mathcal{M} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{M} \times \mathcal{Y}$ and an initial state st_1 . P takes as input a memory state st_i , an input x , and randomness r , then outputs an updated memory state st_{i+1} and the output y :

$$P : (\text{st}_i, x; r) \mapsto (\text{st}_{i+1}, y)$$

Correctness and Intended Usage. A RAM program is intended to be used as follows. Starting from $st_1 = \perp$ and some initial input x_1 , sample randomness r_1 and compute $(st_2, y_1) = P(st_1, x_1; r_1)$. Then, choose a new input x_2 and repeat the process: sample r_2 , then compute $(st_3, y_2) = P(st_2, x_2; r_2)$. This continues until the user is done evaluating the RAM program. As a simplification, the user can sample their randomness $r_i = H(i||x_i)$ using a random oracle H . Including i in the input to H , as opposed to st_i , ensures that fresh randomness is used for every query, even if the state loops.

RAM Programs in the Effective Query Model. Next, we give the formal definition of a RAM program in the effective query model in Figure 6, which we call the LEQ oracle. At a high level, the LEQ oracle maintains a coherent list of evaluations made so far and the program states associated with them. To enforce a consistent history of the computation so far, it only allows query in the form of a stack - the last q 'th query must be uncomputed to access the $q - 1$ 'th query.

Registers. The oracle acts on the following registers:

- Q : The query register taken as input. Q has standard basis states of the form $|t, x, u, b\rangle_Q$. t denotes the tag, x denotes the input, u is the register where the output will be written, and b serves as an indicator for whether the query was answered.
- S : A register containing the current state st of the program. S is initialized to $|st_1\rangle$.
- $\mathcal{A} = \mathcal{A}_1 \otimes \dots \otimes \mathcal{A}_k$: A variable-length register containing ancillas for the evaluations. Initially, $\mathcal{A}$ has length 0.
- $\mathcal{G} = \mathcal{G}_1 \otimes \dots \otimes \mathcal{G}_k$: A variable-length register where each $\mathcal{G}_i$ contains a bit g_i indicating whether the oracle is “open” (the last operation on tag i was U_P) or “closed” (the last operation on tag i was $U_P^\dagger$) for tag i . Initially, $\mathcal{G}$ has length 0.
- $\mathcal{B}$: A work register to validate the query. $\mathcal{B}$ is initialized to $|0\rangle$.

Sub-Routine. The oracle makes use of a unitary U_P that starts from fresh ancillas $|0\rangle_{\mathcal{A}_i}$, samples randomness r , and computes P on its input. Concretely, for any $x \in \mathcal{X}$ and $u \in \mathcal{Y}$, the unitary U_P maps

$$|x, u\rangle_Q \otimes |st\rangle_S \otimes |0\rangle_{\mathcal{A}_i} \leftrightarrow \frac{1}{\sqrt{|\mathcal{R}|}} \sum_r |x, u \oplus y_r\rangle_Q \otimes |st'_{x,r}\rangle_S \otimes |st, x, r\rangle_{\mathcal{A}_i}$$

where $P(st, x; r) = (st'_{x,r}, y_r)$. Furthermore, U_P acts as the identity on orthogonal states.

Queries. To respond to a query, the oracle acts on $Q \times S \times \mathcal{A} \times \mathcal{G} \times \mathcal{B}$ as the following unitary. Given some basis state $|t, x, u, b\rangle_Q \otimes |st\rangle_S \otimes |a\rangle_{\mathcal{A}} \otimes |g\rangle_{\mathcal{G}} \otimes |0\rangle_{\mathcal{B}}$, the oracle does the following:

1. **Expand $\mathcal{G}$ and $\mathcal{A}$ by One Register.** Append $|0\rangle_{\mathcal{G}_{k+1}}$ to $\mathcal{G}$, and append $|0\rangle_{\mathcal{A}_{k+1}}$ to $\mathcal{A}$.
2. **Validate Query.** Let $k_{\max}$ be the maximum index $i \in [k]$ such that $g_i = 1$, or if no such i exists, then let $k_{\max} = 0$.
If $t \in \{k_{\max}, k_{\max} + 1\}$, then apply X to register $\mathcal{B}$ to obtain $|1\rangle_{\mathcal{B}}$.
3. **Answer Query.** If $\mathcal{B}$ contains 1, then answer the query as follows.

- (a) **Close U_P if g_t is Currently “Open”.** If $g_t = 1$, then apply $U_P^\dagger$ to registers $Q \otimes S \otimes \mathcal{A}_t$.

- (b) **Switch Mode if Returned to Original State.** If register $\mathcal{A}_t$ contains $|0\rangle$, then flip g_t and b by applying $X \otimes X$ to $\mathcal{G}_t \times \mathcal{Q}_b$. In other words, apply

$$|0\rangle\langle 0|_{\mathcal{A}_t} \otimes X_{\mathcal{G}_t} \otimes X_{\mathcal{Q}_b} + (I - |0\rangle\langle 0|_{\mathcal{A}_t}) \otimes I_{\mathcal{G}_t} \otimes I_{\mathcal{Q}_b}$$

to registers $\mathcal{A}_t \times \mathcal{G}_t \times \mathcal{Q}_b$.

- (c) **Open U_P if g_t is Set to “Open”.** If $g_t = 1$, apply U_P to registers $\mathcal{Q} \otimes \mathcal{S} \otimes \mathcal{A}_t$.

4. Uncompute step 2.

Figure 6: The Limited Effective Query (LEQ) Oracle O_P^{LEQ} for RAM Programs

A useful fact about the LEQ oracle is that it enforces a consistent history of the computation so far. In particular, the ancilla registers contain a superposition over ordered sequences of queries, randomnesses, and resulting states (entangled with the evaluator’s state). This follows from the fact that the oracle restricts access to only the top layer of queries.

Lemma 5.1. Let A be an oracle algorithm interacting with the REQ oracle O_P^{LEQ} from Figure 6. After every query in this interaction, the mode register $\mathcal{G}$ is supported on strings $g = 1^{k_{\max}} \| 0^*$ with an all-ones prefix for some $k_{\max}$. Furthermore, the joint ancilla register $\mathcal{A}_1 \otimes \dots \otimes \mathcal{A}_{k_{\max}}$ and current state register $\mathcal{S}$ are supported only on computational basis states

$$|(st_1, x_1, r_1), \dots, (st_{k_{\max}}, x_{k_{\max}}, r_{k_{\max}})\rangle_{\mathcal{A}_{[k_{\max}]}} \otimes |st_{k_{\max}+1}\rangle_{\mathcal{S}}$$

for which there exist outputs $y_1, \dots, y_{k_{\max}+1}$ satisfying

$$P(st_i, x_i; r_i) = (st_{i+1}, y_i) \quad \forall i \in [k_{\max}].$$

Proof. We proceed by induction over the sequence of oracle queries. This clearly holds when the oracle is initialized. Now fix an interaction step and consider any computational basis state in the support of $\mathcal{Q} \otimes \mathcal{S} \otimes \mathcal{A} \otimes \mathcal{G} \otimes \mathcal{B}$ immediately before the oracle call. Let $k_{\max}$ be the maximum index with $g_{k_{\max}} = 1$ (or $k_{\max} = 0$ if no such index exists). By the validation step, any answered query has tag $t \in \{k_{\max}, k_{\max} + 1\}$.

We first isolate the net effect of Step 3 on the registers associated with tag t . Consider any state where the ancillas are $|0\rangle$ in step 3b. Observe that step 3 acts as the identity on any states where the ancillas are orthogonal to $|0\rangle$ in step 3b, since it either applies I or $U_P^\dagger U_P = I$, depending on g_t . Otherwise, we can explicitly compute the behavior by pre-and-post-applying U_P controlled on g_t . Thus, step 3 has the overall effect

$$|x, u\rangle_{\mathcal{Q}} \otimes |st\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \leftrightarrow \frac{1}{\sqrt{|\mathcal{R}|}} \sum_r |x, u \oplus y_r\rangle_{\mathcal{Q}} \otimes |st'_r\rangle_{\mathcal{S}} \otimes |st, x, r\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t}$$

for every $(x, u) \in \mathcal{X} \times \mathcal{Y}$ and acts as the identity otherwise, where $P(st, x; r) = (st'_r, y_r)$.

Therefore $\mathcal{A}_{t-1,t} \otimes \mathcal{G}_{t-1,t} \otimes \mathcal{S}$ satisfy the requirements of the lemma. Finally, the fact that $t \in \{k_{\max}, k_{\max} + 1\}$ implies that 1) all indices before or after t satisfy the requirements by the inductive hypothesis 2) all indices $t' > t$ are $g_{t'} = 1$, and 3) all indices after $t' < t$ are $g_{t'} = 0$. $\square$

Definition 5.1 (Query-Limited RAM Program (QLP) Compiler). Let $\mathcal{P} = \{P : \mathcal{M} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{M} \times \mathcal{Y}\}_P$ be a family of RAM programs. A **query-limited RAM program compiler** for $\mathcal{P}$ comprises the efficient algorithms (PGen, Eval) and has the following syntax, correctness, and security guarantees.

- **Syntax:**

- $\text{PGen}(1^\lambda, P) \rightarrow \tilde{P}$: Take a security parameter λ and a functionality $P \in \mathcal{P}$ then outputs a program $\tilde{P}$ that implements P .
- $\text{Eval}(\tilde{P}, x) \rightarrow (y, \tilde{P}')$: Takes a program $\tilde{P}$ and an input x , and evaluates the program on x to produce output y and a new program state $\tilde{P}'$.
- **Correctness:** For every $P \in \mathcal{P}$, every sufficiently large λ , and every polynomial-length (i.e. $n = \text{poly}(\lambda)$) query sequence $(x_1, \dots, x_n) \in X^*$, the following distributions are statistically close:

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} (r_1, \dots, r_n) \leftarrow \mathcal{R}^n \\ (\text{st}_1, y_1) \leftarrow P(\text{st}_0, x_1; r_1) \\ \dots \\ (\text{st}_n, y_n) \leftarrow P(\text{st}_{n-1}, x_n; r_n) \end{array} \right\} \approx_s \left\{ (y_1, \dots, y_n) : \begin{array}{l} \tilde{P}_0 \leftarrow \text{PGen}(1^\lambda, P) \\ (y_1, \tilde{P}_1) \leftarrow \text{Eval}(\tilde{P}_0, x_1) \\ \dots \\ (y_n, \tilde{P}_n) \leftarrow \text{Eval}(\tilde{P}_{n-1}, x_n) \end{array} \right\}$$

where $\text{st}_0 = \perp$.

- **Security:** The scheme satisfies LEQ security if there exists a QPT simulator $\mathcal{S}$ such that for any randomized RAM program $P \in \mathcal{P}$ and any distinguisher D with (quantum) auxiliary input $\text{aux}_{P, \mathcal{P}}$ (which may depend non-uniformly on P and $\mathcal{P}$), there exists a negligible function $\text{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[D(1^\lambda, \text{PGen}(1^\lambda, P), \text{aux}_{P, \mathcal{P}}) \rightarrow 1 \right] - \Pr \left[D \left(1^\lambda, \mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda), \text{aux}_{P, \mathcal{P}} \right) \rightarrow 1 \right] \right| \leq \text{negl}(\lambda)$$

5.2 Construction

Here we construct a RAM program compiler (definition 5.1). We use the following tools:

- A one-shot program compiler (definition 4.2) $\text{OSP}.$ ($\text{KeyGen}, \text{PGen}, \text{TokenGen}, \text{Eval}$) with serial number space $\mathcal{S}_{\text{OSP}}$.
- A blind-unforgeable signature scheme $\text{Sig}.$ ($\text{KeyGen}, \text{Sign}, \text{Ver}$) with signature space Σ_{Sig} .
- A secret-key encryption scheme $\text{SKE}.$ ($\text{KeyGen}, \text{Enc}, \text{Dec}$) with ciphertext space $\mathcal{C}_{\text{SKE}}$ satisfying qIND-qCPA security (Definition 3.14).
- A random oracle $G_{\text{Sign}} : \mathcal{S}_{\text{OSP}} \times \mathcal{C}_{\text{SKE}} \times \{0, 1\}^{2\lambda} \rightarrow \{0, 1\}^\lambda$.
- A random oracle $G' : (X \times \mathcal{C}_{\text{SKE}} \times \{0, 1\}^\lambda \times \Sigma_{\text{Sig}} \times \mathcal{S}_{\text{OSP}}) \times \{0, 1\}^\lambda \rightarrow \{0, 1\}^{2\lambda}$.

The random oracles G_{Sign} and G' can be replaced by pseudorandom functions. We present the construction as if they were oracles to slightly simplify the proof of security by removing an indistinguishable transition from PRF to random oracle.

Theorem 5.1. The construction in fig. 8 is query-limited RAM program compiler (Definition 5.1) for any family of randomized RAM programs $\mathcal{F}$.

Proof. We need to show correctness and simulation security.

Correctness. Consider the set of serial numbers and tokens generated for any polynomial number of evaluations. By correctness and union bound, none of the generated pairs are $(\perp, \perp)$, except with negligible probability. Conditioned on this, the output of every query $(s; (x, \tilde{\text{st}}, r_s), \sigma, s')$ made to the one-shot program $\tilde{P}_{\text{OSP}}$ is distributed statistically close to $P_{\text{OSP}}((x, \tilde{\text{st}}, r_s), \sigma, s'; s; r)$ as defined in Figure 8, for uniform r and a polynomial number of queries. Now consider directly querying P_{OSP} . Observe that if σ is a valid signature on $s \parallel \tilde{\text{st}} \parallel r_s$, then P_{OSP} computes $(\text{st}', y) \leftarrow P(\text{st}, x; r_x)$,

Hardcoded: $(\text{sk}, \text{pk}, \text{sk}_{\text{Enc}}, P)$.

Input: $(x, \tilde{\text{st}}, r_s, \sigma, s'; s; r)$. x is the chosen input, $\tilde{\text{st}}$ is an encryption of the current program state, r_s is an arbitrary string, σ is a signature, s is the evaluation serial number for the one-shot program, s' is an arbitrary serial number, and r is the sampled randomness. For ease of notation, we denote $\text{inp} = (x, \tilde{\text{st}}, r_s, \sigma, s')$.

1. Compute $\text{Sig.Ver}(\text{pk}, s \parallel \text{st} \parallel r_s, \sigma)$. If the signature is valid, continue. Otherwise output $\perp$.
2. Decrypt $\text{st} \leftarrow \text{SKE.Dec}(\text{sk}_{\text{Enc}}, \tilde{\text{st}}; r_{\text{Enc}})$.
3. Parse $r = r_x \parallel r'_{s'}$.
4. Evaluate $(\text{st}', y) \leftarrow P(\text{st}, x; r_x)$.
5. Compute randomness for signing and encrypting $r_{\text{Enc}} \parallel r_{\text{Sign}} = G'(s \parallel \text{inp})$.
6. Encrypt $\tilde{\text{st}}' \leftarrow \text{SKE.Enc}(\text{sk}_{\text{Enc}}, \text{st}'; r_{\text{Enc}})$.
7. Compute the following two values:

$$\begin{aligned} r_{s'} &= r'_{s'} \parallel G_{\text{Sign}}(s' \parallel \tilde{\text{st}}' \parallel r'_{s'}) \\ \sigma' &= \text{Sig.Sign}(\text{sk}, s' \parallel \tilde{\text{st}}' \parallel r_{s'}; r_{\text{Sign}}). \end{aligned}$$

8. Output $(y, (\tilde{\text{st}}', r_{s'}, \sigma'))$.

Figure 7: Classical Sub-Program P_{OSP} for the Query-Limited RAM Compiler

for uniform r_x , and outputs y , an encryption $\tilde{\text{st}}'$ of st' , and a signature σ' on $s' \parallel \tilde{\text{st}}' \parallel r_{s'}$ along with a string $r_{s'}$. Starting from the initial state $\text{st}_0 = \perp$ and a valid σ , the sequence of $y_1, \dots, y_n$ resulting from directly querying P_{OSP} a total of $n = \text{poly}(\lambda)$ is distributed as

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} (r_1, \dots, r_n) \leftarrow \mathcal{R}^n \\ (\text{st}_1, y_1) \leftarrow P(\text{st}_0, x_1; r_1) \\ \dots \\ (\text{st}_n, y_n) \leftarrow P(\text{st}_{n-1}, x_n; r_n) \end{array} \right\}$$

Security. By the security of the one-shot program compiler, the OSP $\tilde{P}_{\text{OSP}}$ can be indistinguishably replaced by the OSP simulator with oracle access to the OSP SEQ functionality $O_{\text{SEQ}}^{P_{\text{OSP}}}$ (Figure 2) for the program P_{OSP} given in Figure 8. Thus, it suffices to show that $O_{\text{SEQ}}^{P_{\text{OSP}}}$ can be indistinguishably simulated using oracle access to O_{LEQ}^P .

We first introduce a hybrid oracle O_{Hyb}^P in Figure 9 and show that it can be used to simulate the one-shot-program SEQ oracle $O_{\text{SEQ}}^{P_{\text{OSP}}}$ (Figure 2) in Proposition 1 (Section 5.3.2). Then, we show how to use O_{LEQ}^P to simulate oracle access to O_{Hyb}^P in Proposition 2 (Section 5.3.3). $\square$

Informally, transitioning to O_{Hyb}^P achieves two important steps in preparation for the final transition to O_{LEQ}^P . First, it moves the core functionality – evaluating $P(\text{st}_{t-1}, x; r_1)$ and tracking the new st_t – into the SEQ oracle while stripping out the cryptographic tools P_{OSP} uses to enforce good behavior. Second, it enforces the contiguous history of the computation by restricting the adversary's

$\text{PGen}(1^\lambda, P)$:

1. Sample a signature key pair $(\text{sk}, \text{pk}) \leftarrow \text{Sig.KeyGen}(1^\lambda)$.
2. Generate a one-shot program common reference string $\text{crs} \leftarrow \text{OSP.KeyGen}(1^\lambda)$.^a
3. Generate a one-shot program $\tilde{P}_{\text{OSP}} \leftarrow \text{OSP.PGen}(\text{crs}, P_{\text{OSP}})$ where P_{OSP} is the classical randomized functionality defined in Figure 7.
4. Generate a one-shot program token $(s_1, |T_1\rangle) \leftarrow \text{OSP.TokenGen}(\text{crs})$.^b
5. Encrypt $\tilde{\text{st}}_1 \leftarrow \text{SKE.Enc}(\text{sk}_{\text{Enc}}, \perp)$.
6. Sign $\sigma_1 \leftarrow \text{Sig.Sign}(\text{sk}, s_0 \parallel \tilde{\text{st}}_1 \parallel 0^{2\lambda})$.
7. Output $\tilde{P} = (\text{crs}, \tilde{P}_{\text{OSP}}, (\tilde{\text{st}}_1, 0^{2\lambda}, \sigma_1), s_1, |T_1\rangle)$.

$\text{Eval}(P_f, x)$:

1. Parse $P_f = (\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}, r_s, \sigma), s, |T\rangle)$.
2. Generate $(s', |T'\rangle) \leftarrow \text{OSP.TokenGen}(\text{crs})$.
3. Evaluate $(y, (\text{st}', r_{s'}, \sigma')) \leftarrow \text{OSP.Eval}(\tilde{P}_{\text{OSP}}, s, (x, \text{st}, r_s), \sigma, s'), |T\rangle)$.
4. Update $\tilde{P}$ to $(\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}', r_{s'}, \sigma'), |T'\rangle)$.
5. Output y .

^aAlternatively, the program may use a pre-existing OSP common reference string, since the common reference string is independent of the OSP functionality.

^bIf interaction is permissible, s may be taken externally to make all communication classical.

Figure 8: Query-Limited RAM Program Compiler Construction

queries to either a new query or repeating the most recent query. Thus, the adversary cannot attempt to erase the oracle's memory of old queries without first erasing the newer ones.

5.3 Proof of Security

In this section, we complete the proof of security by defining the hybrid oracle O_{Hyb}^P (fig. 9), showing how to simulate oracle access to $O_{\text{SEQ}}^{P_{\text{OSP}}}$ using O_{Hyb}^P (Proposition 1) and in turn how to simulate oracle access to O_{Hyb}^P using O_P^{LEQ} (Proposition 2).

Registers. The oracle acts on the following registers:

- Q : The query register taken as input. Q has standard basis states of the form $|t, x, u, b\rangle_Q$. t denotes the tag, x denotes the input, u is the register where the output will be written, and b serves as an indicator for whether the query was answered.
- $\mathcal{D}$: An internal register that represents the database of a compressed oracle $H : [2^\lambda] \times \mathcal{X} \rightarrow \mathcal{R} \times \{0, 1\}^\lambda$. $\mathcal{D}$ is initialized to $|\emptyset\rangle$.
- $\mathcal{W}$: An internal register that is used to query the compressed oracle. $\mathcal{W}$ is initialized to $|0, 0\rangle$.
- $\mathcal{B}$: An internal register to validate the query. $\mathcal{B}$ is initialized to $|0\rangle$.

Sub-Routine. The oracle uses a subroutine $\text{State}_t(D)$ that computes the state of the RAM program at the start of the t -th query. $\text{State}_t(D)$ is parameterized by an index $t \in \mathbb{N}$ and takes as input a database $D = \{((i, x_i), r_{i,x} \| r_{i,\text{Sign}})\}_{i \in \{1, \dots, |D|\}}$. $\text{State}_t(D)$ computes the following values

$$\begin{aligned} \text{st}_1 &= \perp \\ (\text{st}_{i+1}, y_i) &= P(\text{st}_i, x_i; r_{i,x}), \quad \forall i \in [t-1] \end{aligned}$$

and returns st_t .

Queries. To respond to a query, the oracle acts on $Q \times \mathcal{D} \times \mathcal{W} \times \mathcal{B}$ as the following unitary. Given some basis state $|t, x, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}} \otimes |0, 0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}}$, the oracle does the following:

1. **Validate Query.** If either $t = |D| \wedge x = x_{|D|}$ or if $t = |D| + 1$, then apply X to register $\mathcal{B}$ to obtain $|1\rangle_{\mathcal{B}}$.
2. **Answer Query.** If $\mathcal{B}$ contains 1, then answer the query as follows.
 - (a) CNOT (t, x) from Q to $\mathcal{W}$. Then query $|(t, x), 0\rangle_{\mathcal{W}}$ to the compressed oracle to obtain a superposition over $|(t, x), r\rangle$.
 - (b) Let $\text{st}_t = \text{State}_t(D)$ and parse $r = (r_1, r_2) \in \mathcal{R} \times \{0, 1\}^\lambda$. Then compute $(\text{st}_{t+1}, y) = P(\text{st}_t, x; r_1)$.^a
 - (c) Using CNOTs, write the value (y, r_2) to the sub-register of Q containing $|u\rangle$ and flip $|b\rangle$:

$$\begin{aligned} |t, x, u, b\rangle_Q \otimes |(t, x), r\rangle_{\mathcal{W}} \\ \mapsto \\ |t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q \otimes |(t, x), r\rangle_{\mathcal{W}} \end{aligned}$$

- (d) Uncompute step 2a.

3. Uncompute step 1.

^aThis is a syntactic step for the sake of exposition. The computation can be absorbed into the next step.

Figure 9: A Hybrid REQ Oracle O_{Hyb}^P

5.3.1 Ordering the Database

Before showing how to simulate $O_{\text{SEQ}}^{\text{POSP}}$ using O_{Hyb}^P , we first establish a property of the SEQ oracle's database which will be useful for establishing a total ordering of the queries recorded in its database. Intuitively, the claim states that after interacting with the SEQ oracle for P_{OSp} , it cannot sign any messages $s\|\tilde{\text{st}}\|*$ which are inconsistent with either the starting state $(s_0, \perp)$ or some state (s_j, st_j) which is recorded by the SEQ oracle.

In particular, this holds even if the adversary gets to see a measured copy of the SEQ oracle's database after interacting with it. Since every entry in the SEQ database includes a valid signature on some $s\|\tilde{\text{st}}\|*$, it must either use the first serial number s_0 or have a precursor entry also recorded in the SEQ database. This allows us to iteratively order the SEQ database starting from s_0 as

$$\begin{aligned} & (s_0, (x_0, \perp, *, 0^\lambda\|0^\lambda, s_1), r_{0,x}\|r'_1) \\ & (s_1, (x_1, \tilde{\text{st}}_1, *, r'_1\|*, s_2), r_{1,x}\|r'_2) \\ & \dots \\ & (s_i, (x_i, \tilde{\text{st}}_i, *, r'_i\|*, s_{i+1}), r_{i,x}\|r'_{i+1}) \end{aligned}$$

Claim 5.1 (Dynamically Constrained Signatures). Let $A^{(\cdot)}$ be a QPT oracle algorithm, let A_2 be a QPT algorithm, and let P be a RAM program. Consider the following experiment $\text{DC-Unforg}(A^{(\cdot)}, A_2, P, q, \lambda)$, parameterized by a query number $q \in \mathbb{N}$ and a security parameter $\lambda \in \mathbb{N}$.

1. Sample $(\text{sk}, \text{pk}) \leftarrow \text{Sig.KeyGen}(1^\lambda)$ and $\text{sk}_{\text{Enc}} \leftarrow \text{SKE.KeyGen}(1^\lambda)$.
2. Sample $(s_0, |T_0\rangle) \leftarrow \text{OSP.TokenGen}(\text{crs})$, encrypt $\tilde{\text{st}}_0 = \text{SKE.Enc}(\text{sk}_{\text{Enc}}, \perp)$ and sign $\sigma_0 \leftarrow \text{Sig.Sign}(\text{sk}, s_0\|\tilde{\text{st}}_0\|0)$.
3. Run $A^{(\cdot)}(s_0, |T_0\rangle, \tilde{\text{st}}_0, \sigma_0)$ with oracle access to $O_{\text{SEQ}}^{\text{POSP}[\text{sk}, \text{pk}, \text{sk}_{\text{Enc}}]}$ until q queries are made, where $(\text{sk}, \text{pk}, \text{sk}_{\text{Enc}})$ are hard-coded into P_{OSp} . Let register $\mathcal{A}$ contain A 's state at the end of this step.
4. Measure the SEQ oracle's compressed database in the computational basis to get D_q .
5. Run $A_2(\mathcal{A}, D_q)$ to obtain σ' and m .
6. Output “lose” if $\text{Sig.Ver}(\text{pk}, m, \sigma')$ rejects or if m can be parsed as $m = s\|\tilde{\text{st}}\|r_s$ satisfying either of the following two conditions.

- (a) $s = s_0$ and $\text{SKE.Dec}(\text{sk}_{\text{Enc}}, \tilde{\text{st}}) = \perp$
- (b) or there exists an entry $(*, (x_j, \tilde{\text{st}}_j, *, *, s), r_{j,x}\|r'_s) \in D_q$ such that the program states $\text{st}_j = \text{SKE.Dec}(\text{sk}_{\text{Enc}}, \tilde{\text{st}}_j)$ and $\text{st}_{j+1} = \text{SKE.Dec}(\text{sk}_{\text{Enc}}, \tilde{\text{st}})$ satisfy

$$\begin{aligned} & P(\text{st}_j, x_j; r_{j,x}) = (\text{st}_{j+1}, *) \\ & \text{and } r_s = r'_s\|* \end{aligned}$$

Otherwise, output “win”.

Assuming Sig is blind-unforgeable, then for all sufficiently large $\lambda \in \mathbb{N}$, every $q = \text{poly}(\lambda)$, every QPT oracle algorithm $A^{(\cdot)}$, every QPT A_2 , and every RAM program P ,

$$\Pr[\text{win} \leftarrow \text{DC-Unforg}(A^{(\cdot)}, A_2, P, q, \lambda)] = \text{negl}(\lambda)$$

To aid in proving this claim, we introduce the following strengthening of the compressed oracle chaining lemma from [GLR⁺25]. Intuitively, [GLR⁺25]'s original version showed that if an adversary has access to a composition of random oracles $H \circ G$ and H 's compressed database records a value

$H(y) = z$, then G 's compressed database records a matching value $G(x) = y$, for some x . The lemma below strengthens this in two ways. First, instead of directly evaluating H on $G(x)$, we consider evaluating H on some arbitrary function f of x and $G(x)$, along with half of $G(x)$.¹³ Second, it enforces that the *input* x_g recorded by G is a preimage of the input x_h recorded by H .

Lemma 5.2 (Compressed Oracle Chaining). Let $G : \mathcal{X}_G \rightarrow \mathcal{Y}_1 \times \mathcal{Y}_2$ and $H : \mathcal{X}_H \times \mathcal{Y}_2 \rightarrow \mathcal{Z}$ be random oracles implemented by the compressed oracle technique.

Let $f : \mathcal{X}_G \times \mathcal{Y}_1 \times \mathcal{Y}_2 \rightarrow \mathcal{X}_H$ be an arbitrary function. Define the function $F_f : \mathcal{X} \rightarrow \mathcal{Z}$ by the following computation for $F(x_g)$:

1. Compute $y_1 \| y_2 \coloneqq G(x_g)$.
2. Compute $x_h \coloneqq f(x_g, y_1 \| y_2)$.
3. Output $H(x_h, y_2)$.

Consider running an interaction of an oracle algorithm with F_f until query t , then measuring the internal state of G and H to obtain D_G and D_H .

Let E_t be the event that after the measurement at time t , for all $(x_h \| y_2, z) \in D_H$, there exists an entry $(x_g, y_1 \| y_2) \in D_G$ such that $f(x_g, y_1 \| y_2) = x_h$. Then

$$\Pr[E_t] \geq 1 - \frac{8t^2}{|\mathcal{Y}_2|}$$

The proof of the strengthened compressed oracle chaining lemma is similar to that of the original, but requires some careful modifications. We defer it to Appendix A in favor of focusing on the direct analysis of RAM programs.

Proof of Claim 5.1. Observe that if A submits a query $((s, (*, \tilde{s}t, r_s, \sigma, *)), r)$ to the SEQ oracle where σ were *not* a valid signature on $s \| \tilde{s}t \| r_s$, then P_{OSP} deterministically outputs $\perp$. Since the SEQ oracle never records deterministic queries, D_q never records queries failing this condition.

Next, we show that either $s = s_0$ or $r_s = r'_s \| G_{\text{Sign}}(s \| \tilde{s}t \| r'_s)$. Observe that P_{OSP} only ever signs messages $s \| \tilde{s}t \| r_s$ where $r_s = r'_s \| G_{\text{Sign}}(s \| \tilde{s}t \| r'_s)$. Furthermore, the experiment can be internally simulated given query access to a signing oracle that only signs such messages. Therefore by the blind-unforgeability of Sig , A_2 can only find signatures on either

- $s_0 \| \tilde{s}t_0 \| 0^{2\lambda}$, which was signed as σ_0 ,
- or on $s \| \tilde{s}t \| r_s$ where $r_s = r'_s \| G_{\text{Sign}}(s \| \tilde{s}t \| r'_s)$ for some r'_s .

Since the adversary loses in the former case, we restrict our attention to the latter. Let E_q be the event that A_2 outputs a valid signature satisfying the latter after q queries. Also consider the event E_q^* where condition 6b holds for this output and the measured database D_q . We want to reason about $\Pr[E_q \wedge \neg E_q^*]$.

Consider the experiment where G_{Sign} is implemented as a compressed oracle and let $E_{q, G_{\text{Sign}}}$ be the event that A_2 successfully signs a message $s \| \tilde{s}t \| r_s$ where G_{Sign} 's compressed database records an entry $(s \| \tilde{s}t \| r'_s, r''_s)$ where $r_s = r'_s \| r''_s$. By Lemma 3.4,

$$\Pr[E_{q, G_{\text{Sign}}}] \geq \left(\sqrt{\Pr[E_q]} - \sqrt{1/2^\lambda} \right)^2 = \Pr[E_q] - \text{negl}(\lambda)$$

¹³Including half of $G(x)$ ensures that H 's input has a minimal dependence on $G(x)$. Otherwise, f could ignore its inputs when choosing the input to H .

Furthermore, by the compressed oracle chaining lemma (Lemma 5.2), D_q must have a corresponding entry $((s_{i-1}, (x_{i-1}, \tilde{st}_{i-1}, *, *, s)), r'_s \| *)$ satisfying condition 6b after any polynomial number of queries, except with negligible probability.

In particular, the chaining lemma requires that the entry $(x_g, y_1 \| y_2) \in G$ matching $(x_h \| y_2, *) \in H$ satisfies $x_h = f(x_g, G(x_g))$ and y_2 appears in both databases. Set $H = G_{\text{Sign}}$, G to be the SEQ oracle's internal compressed oracle, $x_g = ((s_{i-1}, (x_{i-1}, \tilde{st}_{i-1}, *, *, s)),$ and $y_2 = r'_s$ where $H(x_g) = * \| r'_s$. Then, set $f(x_g, r'_s \| r'_s)$ to be the function which 1) decrypts $\tilde{st}_{i-1}$, then 2) evaluates P , then 3) encrypts the new state st to produce $\tilde{st}$, and finally 4) outputs $s \| \tilde{st} \| r'_s$. Together with the correctness of SKE, it must be the case that $P(st_{i-1}, x; r_{x,i-1}) = (st, *)$ where $st_{i-1} = \text{Dec}(\text{sk}_{\text{Enc}}, \tilde{st}_{i-1})$ and $st = \text{Dec}(\text{sk}_{\text{Enc}}, \tilde{st})$. Furthermore, $y_2 = r'_s$ is recorded in both the input of H (also known as G_{Sign}) and the output of G (also known as the SEQ oracle's internal compressed oracle). Therefore

$$\Pr[E_q^*] \geq \Pr[E_{q, G_{\text{Sign}}}] - \text{negl}(\lambda) \geq \Pr[E_q] - \text{negl}(\lambda).$$

Finally, $\Pr[E_t \wedge \neg E_{k,t}^*] = \text{negl}(\lambda)$ because E_q^* is a sub-event of E_q . $\square$

5.3.2 Simulating the One-Shot Program SEQ Oracle

Proposition 1. *There exists a simulator Sim_1 such that for all RAM programs P , all QPT oracle algorithms $A^{(\cdot)}$ and all sufficiently large $\lambda \in \mathbb{N}$,*

$$\left\{ A_{\text{SEQ}}^{O_{\text{OSP}}^P[\text{sk}, \text{pk}, \text{sk}_{\text{Enc}}]} : \begin{array}{l} (\text{sk}, \text{pk}) \leftarrow \text{Sig.KeyGen}(1^\lambda) \\ \text{sk}_{\text{Enc}} \leftarrow \text{SKE.KeyGen}(1^\lambda) \end{array} \right\} \approx_c \left\{ A^{\text{Sim}_1^{O_{\text{Hyb}}^P}} \right\}$$

where $P_{\text{OSP}}[\text{sk}, \text{pk}, \text{sk}_{\text{Enc}}]$ denotes the functionality P_{OSP} with sk , pk , and sk_{Enc} hard-coded.

Proof. The simulator Sim_1 is given in Figures 10 and 11.

Sim_1 maintains a cache register $C = (C_{\text{Sl}}, C_{\text{InpC}}, C_{\text{OutC}})$ containing three key-value stores:

- $\text{Sl} = \{(s : i_s)\}_s$ is a mapping from serial numbers s to indices i_s . It is initialized to $\{(0 : s_0)\}$.
- $\text{InpC} = \{(i : \text{inp}_i)\}_i$ is a mapping from indices i to inputs $\text{inp}_i = (x_i, \tilde{st}_i, r_{s_i}, \sigma_i, s')$ for P_{OSP} . It is initialized to $\emptyset$.
- $\text{OutC} = \{(i : \text{outp}_i)\}_i$ is a mapping from indices i to outputs $\text{outp}_i = (y, (\tilde{st}', r_{s'}, \sigma'))$ resulting from evaluating P_{OSP} .

Additionally, it uses a $\text{Refl}_i^{O_{\text{Hyb}}^P}$ subroutine parameterized by an index i and an input $x \in \mathcal{X}$ which acts on a state $|b\rangle$ as follows.^a Fix any $x \neq x' \in \mathcal{X}$.

1. Prepare and query $\propto |(i, x)\rangle \otimes \sum_u |u\rangle \otimes |0\rangle$ to O_{Hyb}^P to obtain $\propto |(i, x)\rangle \otimes \sum_u |u\rangle \otimes |b_1\rangle$.
2. Prepare and query $\propto |(i, x')\rangle \otimes \sum_u |u\rangle \otimes |b_2\rangle$ to O_{Hyb}^P to obtain $\propto |(i, x')\rangle \otimes \sum_u |u\rangle \otimes |b_2\rangle$.
3. Compute $|b \oplus (b_1 \wedge b_2)\rangle$.
4. Uncompute steps 2 and 1.

^aIntuitively, this routine reflects around whether O_{Hyb}^P has recorded some query $(i, *)$.

Figure 10: Work Registers and Subroutines for $\text{Sim}_1^{O_{\text{Hyb}}^P}$

Claim 5.2. $\text{Ref}_i^{O_{\text{Hyb}}^P}$ is equivalent to the following procedure: flip b if $\nexists((i, *), *) \in D$.

Proof. First, we show that step 1 is equivalent to flipping $b_1 = 0$ controlled on $\nexists((i, x''), *) \in D$ for all $x'' \neq x$. If there is such an entry $((i, x''), *) \in D$, then the query to O_{Hyb}^P acts as the identity. Note that in the case of $i < \text{MaxEntry}(D)$, the former holds and O_{Hyb}^P also acts as the identity. Furthermore, writing the output of the P evaluation to $\sum_u |u\rangle$ does not change the state and so the evaluations of the compressed oracle cancel with each other. Similarly, step 2 is equivalent to flipping $b_2 = 0$ controlled on $\nexists((i, x''), *) \in D$ for all $x'' \neq x'$.

Now observe that since $x' \neq x$, both $b_1 = 1$ and $b_2 = 1$ if and only if $\nexists((i, *), *) \in D$. Finally, uncomputing b_1, b_2 , and the work registers removes any effect other than the flip. $\square$

To answer a query $|s, \text{inp}, u, b\rangle_Q$ where $\text{inp} = (x, \tilde{\text{st}}, r_s, \sigma, s')$ do the following.

1. **SEQ Check:** Check that the cache does not contain a conflicting $\text{InpC}[\text{SI}[s]] = \text{inp}'$ where $\text{inp}' \notin \{\perp, \text{inp}\}$.^a If it does, then skip the following steps.
2. **Signature Validation:** Compute $\text{Sig.Ver}(\text{pk}, s \parallel \tilde{\text{st}} \parallel r_s, \sigma)$. If the signature is valid, then flip b , write $\perp$ to u , and skip the following steps.
3. **Early Answer:** Let $i_{\max}$ be the highest index recorded in SI. If $\text{SI}[s] < i_{\max} - 1$, compute

$$|s, \text{inp}, u \oplus \text{OutC}[\text{SI}[s]], b \oplus 1\rangle_Q$$

from the output cache and skip the following steps.

4. **Cache Clear:**

- (a) Initialize a temporary ancilla register $|g\rangle = |1\rangle$. Controlled on the input, evaluate $\text{Ref}|_{\text{SI}[s], x}^{O_{\text{Hyb}}^P} |g\rangle$.
- (b) **Serial and Input Cache:** If $g = 1$, controlled on the input register $|s, \text{inp}, *, *\rangle$, set $i_s = \text{SI}[s]$ and update the serial and input caches to

$$\begin{aligned} \text{SI} &\mapsto \text{SI} \Delta \{(s' : i_s + 1)\} \\ \text{InpC} &\mapsto \text{InpC} \Delta \{(i_s : (s, \text{inp}))\} \end{aligned}$$

in the cache registers C_{SI} and C_{InpC} .

- (c) **Output Cache:** If $g = 0$, controlled on the input register $|s, \text{inp}, *, *\rangle$, set $i_s = \text{SI}[s]$ and look up $(s_{i_s-1}, \text{inp}_{i_s-1}) = \text{InpC}[i_s - 1]$. Then, compute $\text{outp} = P_{\text{OSP}}(\text{inp}_{i_s-1}; s; r)$ where $r = H(\text{SI}[s], x)$ as in the **Evaluation** step, and update

$$\text{OutC} \mapsto \text{OutC} \Delta \{(i_s - 1 : \text{outp})\}$$

in the cache register C . Note that this requires two queries to O_{Hyb}^P .

- (d) Repeat step 4a, then discard the temporary register.

5. **Evaluation:**

- (a) **Query** $O_{\text{SEQ}}^{\text{POSP}}$: Prepare and query $|(\text{SI}[s], x), 0\rangle$ to $O_{\text{SEQ}}^{\text{POSP}}$ to obtain $|(\text{SI}[s], x), y, r_2\rangle$.
- (b) Set $r_{\text{Enc}} \parallel r_{\text{Sign}} = G'(s \parallel \text{inp})$.
- (c) **Compute Next Encrypted State.** Compute $\tilde{\text{st}}' = \text{Enc}(\text{sk}_{\text{Enc}}, 0; r_{\text{Enc}})$.
- (d) **Compute Next Signature.** Compute the following two values (as in step 7 of Figure 8):

$$\begin{aligned} r_{s'} &= r_{s'}' \parallel G_{\text{Sign}}(s' \parallel \tilde{\text{st}}' \parallel r_{s'}') \\ \sigma' &= \text{Sig.Sign}(\text{sk}, s' \parallel \tilde{\text{st}}' \parallel r_{s'}'; r_{\text{Sign}}). \end{aligned}$$

- (e) **Answer Query.** Write the response to the query register containing u as $|u\rangle \mapsto |u \oplus (y, (\tilde{\text{st}}', r_{s'}, \sigma'))\rangle$ and flip the success indicator bit b .
- (f) **Unquery** $O_{\text{SEQ}}^{\text{POSP}}$: Repeat step 5a

6. **Cache Fill:** Repeat step 4.

^aWe define $\text{InpC}[\perp] = \perp$ to handle the case where $\text{SI}[s] = \perp$.

Consider the following sequence of hybrids.

Hyb₀ is $A_{\text{SEQ}}^{O_{\text{OSP}}[\text{sk}, \text{pk}, \text{sk}_{\text{Enc}}]}$'s view with freshly sampled (sk, pk) and sk_{Enc} . It can generally be divided into two steps.

1. **SEQ Check:** Check that the database D does not contain a conflicting entry $(s, \text{inp}', *)$ where $\text{inp}' \neq (x, \tilde{\text{st}}, r_s, \sigma, s')$; if it does, skip the following steps.
2. **Evaluation:** Compute $P_{\text{OSP}}(x, \tilde{\text{st}}, r_s, \sigma, s'; s; r)$ where $r = H(s, x, \tilde{\text{st}}, r_s, \sigma, s')$. Write the result to u and flip b . This involves two queries to the compressed oracle H : one to obtain r and another to erase it.

Hyb₁ (Move Signature Validation) moves the signature check step in P_{OSP} to before the compressed oracle is queried. Explicitly, respond to oracle queries $|s, (x, \tilde{\text{st}}, r_s, \sigma, s'), u, b\rangle$ as follows.

1. **SEQ Check:** as before.
2. **Signature Validation (Moved):** Compute $\text{Sig.Ver}(\text{pk}, s || \tilde{\text{st}} || r_s, \sigma)$. If the signature is valid, then flip b , write $\perp$ to u , and go straight to outputting, skipping the following steps. Otherwise, continue
3. **Evaluation: (Modified):** Compute $P_{\text{OSP}}(x, \tilde{\text{st}}, r_s, \sigma, s'; s; r)$ where $r = H(s, x, \tilde{\text{st}}, r_s, \sigma, s')$ starting from step 2 in the construction (Figure 8). Write the result to u and flip b . This includes 2 queries to H .

Hyb_0 is equivalent to Hyb_1 since the signature validation is independent of r .

Hyb₂ (Add History Check) adds an additional “history consistency” check after signature validation. Specifically, respond to oracle queries $|s, (x, \tilde{\text{st}}, r_s, \sigma, s'), u, b\rangle$ as follows.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **History Consistency Check (New):** Check that either $s = s_0$ or that there exist entries

$$\begin{array}{lll} (s_0, & (x_0, \perp, *, 0^\lambda || 0^\lambda, s_1), & r_{0,x} || r'_1) \\ (s_1, & (x_1, \tilde{\text{st}}_1, *, r'_1 || *, s_2), & r_{1,x} || r'_2) \\ & \dots & \\ (s_{i-1}, & (x_{i-1}, \tilde{\text{st}}_{i-1}, *, r'_{i-1} || *, s_i), & r_{i-1,x} || r'_i) \end{array}$$

in D where $s_i = s$ and $r_s = r'_i || *$ such that for every $j < i$, the program states $\text{st}_j = \text{SKE.Dec}(\text{sk}_{\text{Enc}}, \tilde{\text{st}}_j)$ and $\text{st}_{j+1} = \text{SKE.Dec}(\text{sk}_{\text{Enc}}, \tilde{\text{st}}_{j+1})$ satisfy

$$P(\text{st}_j, x_j; r_{j,x}) = (\text{st}_{j+1}, *)$$

where $r_j = r_{j,x} || r'_{j,s_{j+1}}$ and $\text{st}_0 = \perp$.

Measure whether this occurs and if not, abort the experiment completely.¹⁴

4. **Evaluation:** as before.

We prove computational indistinguishability to Hyb_1 in Claim 5.3.

¹⁴This is the only physical measurement that occurs while answering a query – all other computations are done coherently.

Hyb₃ (st from Database) changes how st is computed in the **Evaluation** step. Instead of decrypting it from $\tilde{st}$, st is computed using the compressed database D . To answer a query $|s, (x, \tilde{st}, r_s, \sigma, s'), u, b\rangle_Q$, do the following.

1. **SEQ Check**: as before.
2. **Signature Validation**: as before.
3. **History Consistency Check**: as before.
4. **Cache Clear**: as before.
5. **Evaluation (Modified)**: Compute $P_{\text{OSP}}(x, \tilde{st}, r_s, \sigma, s'; s; r)$ where $r = H(s, x, \tilde{st}, r_s, \sigma, s')$, starting from step 2 in the construction (Figure 8).

(a) **Compute st (Modified)**: Order the database, starting from s_0 , as containing

$$\begin{aligned} & (s_0, (x_0, \perp, *, *, s_1), r_{0,x} \| *) \\ & (s_1, (x_1, \tilde{st}_1, *, *, s_2), r_{1,x} \| *) \\ & \dots \\ & (s_i, (x_i, \tilde{st}_i, *, *, s_{i+1}), r_{i,x} \| *) \end{aligned}$$

Find i_s such that $s_{i_s} = s$, then iteratively compute

$$P(st_j, x_j; r_{j,x}) = (st_{j+1}, *)$$

starting from $j = 0$ and $st_0 = \perp$ to $j = i_s - 1$. Set $st = st_{i_s}$.

- (b) Continue from step 3 in Figure 8.
- (c) Write the output to u and flip b .

6. **Cache Fill**: as before.

We prove equivalence to Hyb₂ in Claim 5.4.

Hyb₄ (Simplified History Check) simplifies the history check by removing the condition on $\tilde{st}_j$. Explicitly, answer queries by doing the following.

1. **SEQ Check**: as before.
2. **Signature Validation**: as before.
3. **History Consistency Check (Modified)**: Check that either $s = s_0$ or that there exist entries

$$\begin{aligned} & (\textcolor{red}{s}_0, (x_0, \tilde{st}, *, \textcolor{red}{0}^\lambda \| 0^\lambda, \textcolor{blue}{s}_1), r_{0,x} \| \textcolor{blue}{r}'_1) \\ & (\textcolor{blue}{s}_1, (x_1, \tilde{st}, *, \textcolor{blue}{r}'_1 \| *, \textcolor{violet}{s}_2), r_{1,x} \| \textcolor{violet}{r}'_2) \\ & \dots \\ & (s_{i-1}, (x_{i-1}, \tilde{st}, *, r'_i \| *, s_i), r_{i-1,x} \| r'_i) \end{aligned}$$

in D where $s_i = s$ and $r_s = r'_i \| *$.

4. **Evaluation**: as before.

We prove computational indistinguishability to Hyb₃ in Claim 5.5.

Hyb₅ (Hide st) changes how $\tilde{st}'$ is computed in the **Evaluation** of $P_{\text{OSP}}(\text{inp}; s; r)$ (step 5). Instead of computing it as an encryption of the next state st' (step 6 of Figure 8), generate $\tilde{st}' = \text{Enc}(\text{sk}_{\text{Enc}}, 0; r_{\text{Enc}})$ as an encryption of 0. We prove computational indistinguishability to Hyb₄ in Claim 5.6.

Hyb₆ (Input Cache) adds the serial-to-index cache SI and the input cache InpC, contained in a variable-length register C . $\text{SI} = \{(s : i_s)\}_s$ is a mapping from serial numbers s to indices i_s . $\text{InpC} = \{(i : \text{inp}_i)\}_i$ is a mapping from indices i to inputs inp_i . SI is initialized to $\{(0 : s_0)\}$, while InpC is initialized to $\emptyset$.

To answer a query $|s, \text{inp}, u, b\rangle_Q$, do the following.

1. **SEQ Check**: as before.
2. **Signature Validation**: as before.
3. **History Check**: as before.
4. **Cache Clear (New)**: If there exists an entry $(s, (*, *, *, *, *), *) \in D$, where D is the compressed database, do the following.
 - Set $i_s = \text{SI}[s]$ and update the caches to¹⁵

$$\begin{aligned}\text{SI} &\mapsto \text{SI} \Delta \{(s' : i_s + 1)\} \\ \text{InpC} &\mapsto \text{InpC} \Delta \{i_s : (s, \text{inp})\}\end{aligned}$$

5. **Evaluation**: as before.
6. **Cache Fill (New)**: Repeat step 4.

We prove equivalence to Hyb₅ in Claim 5.8.

Hyb₇ (SEQ Check using Cache) changes the **SEQ Check** to use the cache instead of looking in the database. To answer a query $|s, \text{inp}, u, b\rangle_Q$, do the following.

1. **SEQ Check (Modified)**: Check that the cache does not contain a conflicting $\text{InpC}[\text{SI}[s]] = (s', \text{inp}')$ where $(s', \text{inp}') \notin \{\perp, (s, \text{inp})\}$.¹⁶ If it does, then skip the following steps.
2. **Signature Validation**: as before.
3. **History Check**: as before.
4. **Cache Clear**: as before.
5. **Evaluation**: as before.
6. **Cache Fill**: as before.

We prove equivalence to Hyb₆ in Claim 5.9.

¹⁵A query using s registers the *next* serial number s' into SI and the *current* input inp into InpC under s . **Cache Clear** prepares for answering each query by (if necessary) de-registering the next serial number s' from SI and clearing the current input inp from InpC for s . The symmetric difference Δ is used to ensure the operation is unitary. A new element is never added in **Cache Clear** because it is (indirectly) conditioned on an element already existing.

¹⁶We define $\text{InpC}[\perp] = \perp$ to handle the case where $\text{SI}[s] = \perp$.

Hyb₈ (H Input Renaming) renames the inputs to the compressed oracle H . s is renamed to its index $\text{SI}[s]$ and $\text{inp} = (x, \tilde{\text{st}}, r_s, \sigma, s')$ is renamed to just the chosen input x . For example, instead of querying H on (s, inp) , the simulator would query it on $(\text{SI}[s], x)$.

To answer a query $|s, (x, \tilde{\text{st}}, r_s, \sigma, s'), u, b\rangle_Q$, do the following.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **History Check (Modified):** Check that either $s = s_0$ or there exist entries

$$\begin{aligned} &((0, x_0), r_0) \\ &((1, x_1), r_1) \\ &\dots \\ &((\text{SI}[s] - 1, x_{\text{SI}[s]-1}), r_{\text{SI}[s]-1}) \end{aligned}$$

in D .

4. **Cache Clear (Modified):** If there exists an entry $((\text{SI}[s], *), *) \in D$, where D is the compressed database, clear the cache as before.
5. **Evaluation (Modified):** Compute $P_{\text{OSP}}(\text{inp}; s; r)$ as before, except $r = H(\text{SI}[s], x)$
6. **Cache Fill (Modified):** Repeat step 4.

We prove equivalence to Hyb₇ in Claim 5.10.

Hyb₉ (Output Cache) adds an output cache OutC to the cache register C . $\text{OutC} = \{(i : \text{outp}_i)\}$ is a key-value store mapping indices i to outputs $\text{outp}_i = (y, (\tilde{\text{st}}', r_{s'}, \sigma'))$ resulting from querying P_{OSP} on $(s_i, \text{InpC}[i])$.

To answer a query $|s, (x, \tilde{\text{st}}, r_s, \sigma, s'), u, b\rangle_Q$, do the following.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **History Check:** as before.
4. **Cache Clear (Modified):**

- (a) If there exists an entry $(\text{SI}[s], x, *) \in D$, update the serial-to-index and input caches to

$$\begin{aligned} \text{SI} &\mapsto \text{SI} \Delta \{(s' : i_s + 1)\} \\ \text{InpC} &\mapsto \text{InpC} \Delta \{i_s : \text{inp}\} \end{aligned}$$

- (b) If there *does not* exist an entry $(\text{SI}[s], x, *) \in D$, look up $(s_{i-1}, \text{inp}_{i-1}) = \text{InpC}[\text{SI}[s] - 1]$ and parse $\text{inp}_{i-1} = (x_{i-1}, \dots)$. Then, compute $r_{i-1} = D(\text{SI}[s] - 1, x_{i-1})$ *directly from the database register* and compute $\text{outp}_{i-1} = P_{\text{OSP}}(s_{i-1}, \text{inp}_{i-1}; r_{i-1})$ as in the **evaluation** step.¹⁷ Finally, update the output cache to

$$\text{OutC} \mapsto \text{OutC} \Delta \{(\text{SI}[s] - 1 : \text{outp})\}$$

5. **Evaluation:** as before.
6. **Cache Fill (Modified):** Repeat step 4.

We prove equivalence to Hyb₈ in Claim 5.12.

¹⁷We emphasize that this does not query the compressed oracle to obtain r_{i-1} .

Hyb₁₀ (Early Answer) uses the output cache to answer queries which use a serial number in the middle of the recorded serial numbers. To answer a query $|s, \text{inp}, u, b\rangle_Q$ where $\text{inp} = (x, \tilde{\text{st}}, r_s, \sigma, s')$, do the following.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **History Check:** as before.
4. **Early Answer (New):** Let $i_{\max}$ be the highest index recorded in SI. If $\text{SI}[s] < i_{\max}$, compute

$$|s, \text{inp}, u \oplus \text{OutC}[\text{SI}[s]], b \oplus 1\rangle_Q$$

from the output cache and skip the following steps.

5. **Cache Clear:** as before (if no early answer).
6. **Evaluation:** as before (if no early answer).
7. **Cache Fill:** as before (if no early answer).

We prove equivalence to Hyb₉ in Claim 5.13.

Hyb₁₁ (Output Cache Queries H) modifies how the output cache works. Instead of directly using $r = D(i, x)$ from the database, it queries the random oracle H on (i, x) to obtain r .

To answer a query $|s, (x, \tilde{\text{st}}, r_s, \sigma, s'), u, b\rangle_Q$, do the following.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **Early Answer:** as before.
4. **Cache Clear (Modified):** If there exists an entry $(\text{SI}[s], x, *) \in D$, clear the cache as follows.
 - (a) **Serial-to-Index and Input Caches:** Same as before.
 - (b) **Output Cache (Modified):** If there *does not* exist an entry $(\text{SI}[s], x, *) \in D$, look up $(s_{i-1}, \text{inp}_{i-1}) = \text{InpC}[\text{SI}[s] - 1]$ and parse $\text{inp}_{i-1} = (x_{i-1}, \dots)$. Then, compute $r_{i-1} = H(\text{SI}[s] - 1, x_{i-1})$ by querying the compressed oracle H and compute $\text{outp}_{i-1} = P_{\text{OSP}}(s_{i-1}, \text{inp}_{i-1}; r_{i-1})$ as in the **evaluation** step.¹⁸ Finally, update the output cache to

$$\text{OutC} \mapsto \text{OutC} \Delta \{(\text{SI}[s] - 1 : \text{outp})\}$$

- (c) Flip g if there is an entry $(s, x, *) \in D$. Measure whether $g = 0$ and abort the experiment if not.
5. **Evaluation:** as before.
6. **Cache Fill (Modified):** Repeat step 4.

We prove statistical closeness to Hyb₁₀ in Claim 5.14.

¹⁸The change is to compute r by querying the compressed oracle.

Hyb₁₂ (Remove History Check) removes the **history consistency check**.

We prove statistical closeness to Hyb₁₁ in Claim 5.15.

Hyb₁₃ (Cache Uses Oracle) replaces the conditions for the cache to activate. Instead of the cache activating if $\exists((SI[s], *), *) \in D$ matching the input $(s, *)$, it uses $\text{Ref}_{SI[s]}^{O_{\text{Hyb}}^P}$ to decide whether to activate the cache.

To answer a query $|s, \text{inp}, u, b\rangle_Q$, do the following.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **Early Answer:** as before.
4. **Cache Clear (Modified):** (if no early answer)
 - (a) **(Modified)** Run $\text{Ref}_{SI[s]}^{O_{\text{Hyb}}^P}$ on $|1\rangle$ to obtain $|g\rangle$.¹⁹
 - (b) **Input Cache:** If $g = 1$, then update SI and InpC as before.
 - (c) **Output Cache:** If $g = 0$, then update OutC as before.
 - (d) **(Modified)** Repeat step 4a to erase $|g\rangle$.
5. **Evaluation:** as before (if no early answer).
6. **Cache Fill (Modified):** repeat step 4 (if no early answer).

Hyb₁₃ is equivalent to Hyb₁₂ by Claim 5.2, which states that $\text{Ref}_{SI[s]}^{O_{\text{Hyb}}^P}$ is equivalent to checking whether $\nexists((SI[s], *), *) \in D$, and the fact that g starts as 1.

Hyb₁₄ = Sim^{O_{Hyb}^P} (Evaluation Uses Oracle) separates the computation of $\tilde{st}'$, the encryption of the next state, and σ , the signature on the next serial number and encrypted next state, from the computation of $P(st, x; r_x)$ inside the **Evaluation** step. This allows the computation of $P(st, x; r_x)$ to be re-interpreted as a query to O_{Hyb}^P .

To answer a query $|s, (x, \tilde{st}, r_s, \sigma, s'), u, b\rangle_Q$, do the following.

1. **SEQ Check:** as before.
2. **Signature Validation:** as before.
3. **Cache Clear:** as before.
4. **Evaluation (Modified):**
 - (a) **Query $O_{\text{SEQ}}^{P_{\text{OSP}}}$ (Separated):** Prepare and query $|(SI[s], x), 0\rangle$ to O_{Hyb}^P to obtain $|(SI[s], x), y, r_2\rangle$.
 - (b) Set $r_{\text{Enc}} \| r_{\text{Sign}} = G'(s \| \text{inp})$.
 - (c) **Compute Next State.** Compute $\tilde{st}' = \text{Enc}(\text{sk}_{\text{Enc}}, 0; r_{\text{Enc}})$.

¹⁹The computation of $|g\rangle$ is implicit in prior hybrids. The modification is in *how* it is computed, whether by directly checking the database D or by using $\text{Ref}_{SI[s]}^{O_{\text{Hyb}}^P}$.

(d) **Compute Next Signature.** Compute the following two values (as in step 7 of Figure 8):

$$\begin{aligned} r_{s'} &= r'_{s'} \parallel G_{\text{Sign}} \left(s' \parallel \tilde{\text{st}}' \parallel r'_{s'} \right) \\ \sigma' &= \text{Sig.Sign} \left(\text{sk}, s' \parallel \tilde{\text{st}}' \parallel r_{s'}; r_{\text{Sign}} \right). \end{aligned}$$

(e) **Answer Query.** Write the response to the query register containing u as $|u\rangle \mapsto |u \oplus (y, (\tilde{\text{st}}', r_{s'}, \sigma'))\rangle$

(f) **Unquery** $O_{\text{SEQ}}^{\text{OSP}}$ (**Separated**): Repeat step 4a

5. **Cache Fill:** as before.

We prove equivalence to Hyb_{13} in Claim 5.17.

We show statistical or computational indistinguishability between each pair of subsequent hybrids in claims 5.3 to 5.6, 5.8 to 5.10 and 5.12 to 5.17. Therefore the real world Hyb_0 is indistinguishable from $\text{Sim}^{O_{\text{Hyb}}^P} = \text{Hyb}_{14}$.

Claim 5.3 (Hybrid: History Check). For all QPT adversaries,

$$\text{Hyb}_1 \approx_c \text{Hyb}_2$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_2 .

Proof. It suffices to show that the **history consistency check** fails with negligible probability in Hyb_2 and is therefore gentle. Suppose it failed with noticeable probability. Since the SEQ check restricts D to record a single entry for each serial number s , either the new query $|s, (x, \tilde{\text{st}}, \sigma, r_s, s'), u, b\rangle$ or an existing entry $((s, (x, \tilde{\text{st}}, \sigma, r_s, s'), r) \in D$ does not have a precursor entry $(*, (*, *, *, *, s), *) \in D$ satisfying the state requirements. In either case, σ is a valid signature on $s \parallel \tilde{\text{st}} \parallel r_s$ due to the **signature validation** step.

Therefore if this occurs with noticeable probability after any $q = \text{poly}(\lambda)$ queries, we can construct a QPT adversary that wins the DC-Unforg security game from Claim 5.1 with noticeable probability. Specifically, run Hyb_2 and stop after random query $q' < q$, before it is answered. Find and output the signature σ and message $s \parallel \tilde{\text{st}} \parallel r_s$ corresponding to the query or entry which does not have a precursor in D . $\square$

Claim 5.4 (Hybrid: st from Database). For all adversaries,

$$\text{Hyb}_2 = \text{Hyb}_3$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_3 .

Proof. The only difference between Hyb_2 and Hyb_3 is how st is computed. In Hyb_2 it is computed by decrypting $\tilde{\text{st}}$, whereas in Hyb_3 it is iteratively computed by decrypting st_j in the database and computing $P(\text{st}_j, x_j; r_{j,x}) = (\text{st}_{j+1}, *)$. The **history consistency check** requires that these are equivalent, and the experiment aborts otherwise. Finally, since the two experiments are perfectly indistinguishable and the history check failing can be detected, it must happen with the same probability in Hyb_3 as in Hyb_2 , i.e. negligible. $\square$

Claim 5.5 (Hybrid: Simplified History Check). For all QPT adversaries,

$$\text{Hyb}_3 \approx_c \text{Hyb}_4$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_4 .

Proof. The hybrids differ only in the **History Consistency Check** (Step 3). Hyb_4 performs a subset of the checks that Hyb_3 does. Thus, it is equivalent in Hyb_3 to first perform the full check, then additionally perform the simplified check from Hyb_4 . Furthermore, 5.4 states that the first check fails with negligible probability. Therefore it is gentle and the state in Hyb_3 is negligibly far in trace distance from the state if the full check was not performed; in other words, the state in Hyb_4 .

Finally, since the two experiments are computationally indistinguishable and the history check failing can be detected, it must happen in Hyb_3 with probability negligibly different from in Hyb_2 , i.e. negligible. $\square$

Claim 5.6 (Hybrid: Hide st). For all QPT adversaries,

$$\text{Hyb}_4 \approx_c \text{Hyb}_5$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_5 .

Proof. The difference between Hyb_4 and Hyb_5 is that each $\tilde{st}$ is an encryption of either a program state st in Hyb_4 or a fixed string 0 in Hyb_5 . Observe that Hyb_4 and Hyb_5 never decrypt any $\tilde{st}$ and only use the key to encrypt. Therefore indistinguishability of Hyb_4 and Hyb_5 can be reduced to security of the encryption scheme under quantum queries in both the learning and challenge portions of the security game, i.e. qInd-qCPA security (Definition 3.14).

Finally, since the two experiments are computationally indistinguishable and the history check failing can be detected, it must happen in Hyb_5 with probability negligibly different from in Hyb_4 , i.e. negligible. $\square$

Before showing that Hyb_5 and Hyb_6 are perfectly indistinguishable in Claim 5.8, we first show a useful property of the cache in Hyb_6 .

Claim 5.7 (Input Cache Invariant). In Hyb_5 , after every query, there are entries

$$(s_{i+1} : i + 1) \in \text{SI} \\ \text{and } (i : (s_i, (\text{inp}', s_{i+1}))) \in \text{InpC}$$

for some index $i > 0$, some serial numbers s_i and s_{i+1} , and some partial input $\text{inp}' = (x, \tilde{st}, r_s)$ if and only if there is an entry

$$((s_i, (\text{inp}', s_{i+1})), r_i) \in D$$

Proof. We induct over the number of queries. This property clearly holds at 0 queries. Consider the state of the simulator upon receiving the $(q + 1)$ 'th query. Before answering it, the invariant holds because it holds after answering the q 'th query by the inductive hypothesis and the adversary's operations between queries do not affect the simulator's state.

Consider how the simulator answers a query $|s, (\text{inp}', s')\rangle$. If either the **SEQ check** or **history consistency check** fail, then the invariant holds after answering the query.

Now consider the effect of **cache clear**. We claim that at the end of this step, $\text{SI}[s'] = \perp$ and $\text{InpC}[\text{SI}[s]] = \perp$, i.e. the cache does not contain entries corresponding to the new query. There are two cases at this stage, which may be in superposition or entangled with an external state: either there exists an entry $(s, (\text{inp}', s'), *) \in D$ or there does not. In the former case, by the invariant and the SEQ check, $\text{SI}[s'] = \text{SI}[s] + 1$ and $\text{InpC}[\text{SI}[s]] = (\text{inp}', s')$. Furthermore in this case, **cache clear** computes

$$\begin{aligned} \text{SI} &\mapsto \text{SI} \Delta \{(s' : \text{SI}[s] + 1)\} \\ \text{InpC} &\mapsto \text{InpC} \Delta \{(\text{SI}[s] : (\text{inp}', s'))\} \end{aligned}$$

which clears both entries. In the latter, $SI[s'] = \perp$ and $\text{InpC}[SI[s]] = \perp$ at the beginning of the step and **cache clear** acts as the identity.

Evaluation only queries H on , so the only possible difference between the database D at the start of **Evaluation** and the updated D' at the end it is at $D(s, (\text{inp}', s'))$ and $D'(s, (\text{inp}', s'))$. These D and D' are the same as at the start and end of answering the query, because no other operation modifies the database in the computational basis.

Finally, consider the effect of **cache fill**. At the start of this step, $SI[s'] = \perp$ and $\text{InpC}[SI[s]] = \perp$. Again there are two cases (potentially in superposition or entangled externally): either $(s, (\text{inp}', s'), *) \in D'$ or not. In the former case, applying the symmetric difference causes $SI[s'] = SI[s] + 1$ and $\text{InpC}[SI[s]] = (\text{inp}', s')$ as required. In the latter case, the cache is unmodified and $SI[s'] = \perp$ and $\text{InpC}[SI[s]] = \perp$ as required. $\square$

Claim 5.8 (Hybrid: Input Cache). For all adversaries,

$$\text{Hyb}_5 = \text{Hyb}_6$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_5 .

Proof. Consider the unitary U which, controlled on the compressed database D , maps the cache to

$$\begin{aligned} SI &\mapsto SI \Delta \{(s' : SI[s] + 1) \mid \forall (s, (\text{inp}', s'), r) \in D\} \\ \text{InpC} &\mapsto \text{InpC} \Delta \{(SI[s] : (\text{inp}', s')) \mid \forall (s, (\text{inp}', s'), r) \in D\} \end{aligned}$$

Using the invariant shown in Claim 5.7, U maps between the state of the simulator after each query in Hyb_5 and its state in Hyb_4 (with some additional $|0\rangle$ ancillas). Specifically, if applied at the start of a query in Hyb_5 , it completely empties the cache, and if applied at the end of a query in Hyb_4 , it completely fills the cache according to D . Therefore Hyb_5 can be computed by applying U at the start of a query, answering it as in Hyb_4 , then applying U again after the query.

Finally, since the two experiments are perfectly indistinguishable and the history check failing can be detected, it must happen with in Hyb_3 as in Hyb_2 , i.e. negligible. $\square$

Claim 5.9 (Hybrid: SEQ Using Cache). For all adversaries,

$$\text{Hyb}_6 = \text{Hyb}_7$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_7 .

Proof. It suffices to show that the **SEQ check** from Hyb_6 rejects if and only if the version from Hyb_7 would reject. The Hyb_5 version rejects if there is an entry $((s, \text{inp}'), *) \in D$ where $\text{inp} \neq \text{inp}'$. The Hyb_7 version rejects if there is a cache entry $\text{InpC}[SI[s]] = \text{inp}'$ where $\text{inp}' \notin \text{inp}$. By the cache invariant (Claim 5.7), $((s, \text{inp}'), *) \in D$ if and only if $\text{InpC}[SI[s]] = \text{inp}'$ (since inp' is recorded in D , $\text{inp}' \neq \perp$).

Finally, since the two experiments are perfectly indistinguishable and the history check failing can be detected, it must happen with the same probability in Hyb_7 as in Hyb_6 , i.e. negligible. $\square$

Claim 5.10 (Hybrid: H Input Renaming). For all adversaries,

$$\text{Hyb}_7 = \text{Hyb}_8$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_8 .

Proof. Recall that the cache invariant (Claim 5.7) requires that there is an entry $\text{InpC}[i] = \text{inp} \neq \perp$ if and only if $((s_i, \text{inp}), *) \in D$ for some s_i such that $\text{Sl}[s_i] = i$. Since D can only record one inp_i for each s_i , the caches Sl and InpC define a bijection

$$(s_i, \text{inp}) \rightarrow (\text{Sl}[s_i], \text{InpC}[\text{Sl}[s_i]])$$

covering every entry in D . Furthermore, compressed oracles are insensitive to bijective renaming (Claim 3.1).

Finally, since the two experiments are perfectly indistinguishable and the history check failing can be detected, it must happen with the same probability in Hyb_6 as in Hyb_5 , i.e. negligible. $\square$

Before showing that Hyb_8 and Hyb_9 , which introduces the output cache, are perfectly indistinguishable in Claim 5.12, we first show a useful property of the output cache.

Claim 5.11 (Output Cache Invariant). In Hyb_9 , after every query, there are entries

$$\begin{aligned} & (s_{i+1} : i + 1) \in \text{Sl} \\ & \text{and } (i : (s_i, ((x, \text{inp}_i, s_i)) \in \text{InpC} \\ & \text{and } (i - 1 : P_{\text{OSP}}(x_{i-1}, \text{inp}_{i-1}, s_{i-1}; s_i; r_{i-1})) \in \text{OutC}, \end{aligned}$$

where $\text{inp}_{i-1} = (\sigma_{i-1}, r_{\sigma, i-1})$ for some σ_{i-1} and $r_{\sigma, i-1}$, if and only if there is an entry

$$((i, x), r) \in D$$

Proof. The conditions on Sl and InpC follow from the same argument as Claim 5.7. It remains to analyze the output cache. We show this by induction. We claim that at the end of the first cache step, $\text{OutC}[i - 1] = P_{\text{OSP}}(x, \text{inp}', s_{i+1}; s; r)$ where $(i - 1, x_{i-1}, r_{i-1}) \in D$. This is true because the inductive hypothesis requires that $\text{OutC}[i - 1] = \perp$ if there are no entries $(i, *)$ in D and the symmetric difference in the update swaps between $\perp$ and $P_{\text{OSP}}(x, \text{inp}', s_{i+1}; s; r)$.²⁰ Then, after the second cache step, there are two cases: either there is an entry $(i, *) \in D$ or there is not. In the first case, $\text{OutC}[i - 1]$ is not modified and remains filled, but $\text{Sl}[i + 1]$ and $\text{InpC}[i]$ are filled. In the second case, $\text{OutC}[i - 1]$ is emptied, but $\text{Sl}[i + 1]$ and $\text{InpC}[i]$ are not modified and remain empty. In all cases, these are the only indices which are modified, so the inductive hypothesis holds. $\square$

Claim 5.12 (Hybrid: Output Cache). For all adversaries,

$$\text{Hyb}_8 = \text{Hyb}_9$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_9 .

Proof. Consider the unitary U which, controlled on the compressed database D , maps the output cache to

$$\text{OutC} \mapsto \text{OutC} \Delta \left\{ (\text{Sl}[s] : P_{\text{OSP}}(\text{State}_i(D), x; r)) \mid \forall ((i, x_i), r_i) \in D \right\}$$

Using the invariant shown in Claim 5.7, U maps between the state of the simulator after each query in Hyb_9 and its state in Hyb_8 (with some additional $|0\rangle$ ancillas). Specifically, if applied at the start of a query in Hyb_9 , it completely empties the output cache OutC , and if applied at the end of a query in Hyb_8 , it completely fills the output cache according to D . Therefore Hyb_9 can be computed by applying U at the start of a query, answering it as in Hyb_8 , then applying U again after the query.

Finally, since the two experiments are perfectly indistinguishable and the history check failing can be detected, it must happen with the same probability in Hyb_9 as in Hyb_8 , i.e. negligible. $\square$

²⁰If we had instead obtained r by querying H , the database register *can* change, which is more complicated to keep track of. We avoid this issue for now by directly reading r off of the database.

Claim 5.13 (Hybrid: Early Answer). For all adversaries,

$$\text{Hyb}_9 \approx_s \text{Hyb}_{10}$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_{10} .

Proof. Consider the projector Π_{early} onto joint states of the query, database, and cache registers which trigger an early answer (including passing the SEQ check, signature validation, and history check). Use the early-answer condition $\text{SI}[s] < i_{\max} - 1$. Let $|\psi\rangle$ be any normalized state resulting from the computation so far, including the database registers, the query registers, the cache registers, and A 's internal registers. We abuse notation slightly by writing $\text{Hyb}_9 |\psi\rangle$ (respectively $\text{Hyb}_{10} |\psi\rangle$) to be one query in Hyb_9 (respectively Hyb_{10}) applied to $|\psi\rangle$.

For this comparison, use the same purification of the common history-check measurement in both hybrids, and include its outcome and all work registers in the states. respectively. They agree whenever no early answer is triggered, so

$$(\text{Hyb}_9 - \text{Hyb}_{10})(I - \Pi_{\text{early}}) |\psi\rangle = 0.$$

Since we have

$$\begin{aligned} 0 &\leq \left| 1 - \langle \psi | \text{Hyb}_9^\dagger \text{Hyb}_{10} | \psi \rangle \right|^2 \\ &= 1 - 2 \operatorname{Re} \left(\langle \psi | \text{Hyb}_9^\dagger \text{Hyb}_{10} | \psi \rangle \right) + \left| \langle \psi | \text{Hyb}_9^\dagger \text{Hyb}_{10} | \psi \rangle \right|^2. \end{aligned}$$

Rearranging this inequality and using the trace-distance formula for normalized pure states gives

$$\begin{aligned} \text{TD}[\text{Hyb}_9 |\psi\rangle, \text{Hyb}_{10} |\psi\rangle]^2 &= 1 - \left| \langle \psi | \text{Hyb}_9^\dagger \text{Hyb}_{10} | \psi \rangle \right|^2 \\ &\leq 2 - 2 \operatorname{Re} \left(\langle \psi | \text{Hyb}_9^\dagger \text{Hyb}_{10} | \psi \rangle \right) \\ &= \|\text{Hyb}_9 |\psi\rangle\|^2 + \|\text{Hyb}_{10} |\psi\rangle\|^2 - 2 \operatorname{Re} \left(\langle \psi | \text{Hyb}_9^\dagger \text{Hyb}_{10} | \psi \rangle \right) \\ &= \|\text{Hyb}_9 |\psi\rangle - \text{Hyb}_{10} |\psi\rangle\|^2 \\ &= \|(\text{Hyb}_9 - \text{Hyb}_{10}) |\psi\rangle\|^2. \end{aligned}$$

Therefore,

$$\begin{aligned} \text{TD}[\text{Hyb}_9 |\psi\rangle, \text{Hyb}_{10} |\psi\rangle] &\leq \|(\text{Hyb}_9 - \text{Hyb}_{10}) |\psi\rangle\| \\ &\leq \|(\text{Hyb}_9 - \text{Hyb}_{10}) \Pi_{\text{early}} |\psi\rangle + (\text{Hyb}_9 - \text{Hyb}_{10})(I - \Pi_{\text{early}}) |\psi\rangle\| \\ &= \|(\text{Hyb}_9 - \text{Hyb}_{10}) \Pi_{\text{early}} |\psi\rangle\|. \end{aligned}$$

If $\Pi_{\text{early}} |\psi\rangle = 0$, this distance is zero. Otherwise, set

$$|\eta\rangle = \frac{\Pi_{\text{early}} |\psi\rangle}{\|\Pi_{\text{early}} |\psi\rangle\|}.$$

It suffices to bound $\|(\text{Hyb}_9 - \text{Hyb}_{10}) |\eta\rangle\|$ uniformly for such normalized early-answer states satisfying the cache invariants. The bound for the original state is then multiplied by $\|\Pi_{\text{early}} |\psi\rangle\|$.

Recall how a query $H(i, x)$ works. It uses an operation $\text{Decomp}_{i,x}$, which swaps

$$|D'\rangle \leftrightarrow \frac{1}{\sqrt{|\mathcal{R}|2^\lambda}} \sum_{r \in \mathcal{R} \times \{0,1\}^\lambda} |D' \cup \{(i, x), r)\}\rangle,$$

where $D'(i, x) = \perp$ and fixes the orthogonal complement of these two vectors, for each fixed residual database D' . Then, it classical-copies the result r to the output register and re-applies $\text{Decomp}_{i,x}$.

On query $(s, (x, \text{inp}', s_{i_s+1}))$, set $i_s = \text{SI}[s]$. Hybrid Hyb_9 first performs **Cache Clear**. It then computes $P_{\text{OSP}}(x, \text{inp}', s_{i_s+1}; s; r)$ using two queries to $H(i_s, x)$ to compute and uncompute r , and finally performs **Cache Fill**. The middle **evaluation** step can be simplified by expanding how queries work. The two middle Decomp cancel: the intermediate computation uses r in the work register and uses only predecessor database cells to compute the program state. This results in the following procedure for Hyb_9 , conditioned on Π_{early} .

1. Perform **Cache Clear** as defined in Hyb_9 . On an early-answer state, this clears $\text{SI}[s_{i_s+1}]$ and $\text{InpC}[i_s]$; it leaves $\text{SI}[s]$, $\text{InpC}[i_s + 1]$, and $\text{OutC}[i_s]$ unchanged.
2. $\text{Decomp}_{i_s,x}$.
3. Compute and classical-copy $P_{\text{OSP}}(x, \text{inp}', s_{i_s+1}; s; r)$ to the output register, controlled on $r = D(i_s, x)$, and flip b . Uncompute the temporary registers.
4. $\text{Decomp}_{i_s,x}$.
5. Perform **Cache Fill** as defined in Hyb_9 .

Now consider the following sub-hybrid experiments where we gradually modify Hyb_9 into Hyb_{10} on the early-answer component.

- $\text{Hyb}_{9,a}$ removes the first Decomp operation.
- $\text{Hyb}_{9,b}$ removes the second Decomp operation.
- $\text{Hyb}_{9,c}$ swaps the order of clearing the cache and the computation of $P_{\text{OSP}}(x, \text{inp}', s_{i_s+1}; s; r)$. At this point, **Cache Clear** and **Cache Fill** take place immediately one after another and therefore cancel.
- Hyb_{10} differs from $\text{Hyb}_{9,c}$ only by answering the query using the pre-computed value of P_{OSP} stored in the output cache, instead of computing it from scratch using randomness $r = D(i_s, x)$. Using the cache invariant on Hyb_{10} , this is equivalent.

On the early-answer component, the fact that

$$\text{Hyb}_{9,b} = \text{Hyb}_{9,c}$$

follows from the observation that clearing the cache leaves the database and query inputs unchanged, while evaluating P_{OSP} does not use the cache entries being cleared: the current input is in the query register, and computing the program state uses only predecessor entries. Thus these two operations commute. With both Decomp removed, the queried database cell remains present, so the two cache operations restore precisely the entries they clear.

Next, we bound the vector distance between $\text{Hyb}_9 |\eta\rangle$ and $\text{Hyb}_{9,a} |\eta\rangle$, and between $\text{Hyb}_{9,a} |\eta\rangle$ and $\text{Hyb}_{9,b} |\eta\rangle$. It suffices to show that Decomp is close to the identity on the state immediately before each removed operation. For the second comparison, this is the state in $\text{Hyb}_{9,a}$, where the first decompression has already been removed.

An early answer is triggered for query (s, inp_x) if $\exists (s_{\text{max}} : i_{\text{max}}) \in \text{SI}$ such that $\text{SI}[s] < i_{\text{max}} - 1$. Here we denote $i_s := \text{SI}[s]$ and $\text{inp}_x = (x, \text{inp}', s_{i_s+1})$. In particular, there must be an s_{i_s+2} where $\text{SI}[s_{i_s+2}] = i_s + 2$. Combining this with the output cache invariant (Claim 5.11) and history check, there must be entries

$$\begin{aligned} ((i_s, x), r) &\in D, \quad \text{where } r = * \| r'_{i_s+1}, \\ (i_s + 1 : (*, *, r'_{i_s+1} \| *, *, s_{i_s+2})) &\in \text{InpC}. \end{aligned}$$

Furthermore, these two properties also imply the existence of input and output cache entries for s and inp_x , which are constrained by the SEQ check to be

$$(i_s : \text{inp}_x) \in \text{InpC},$$

$$(i_s : P_{\text{OSP}}(\text{inp}_x; s; r)) \in \text{OutC}.$$

Here the input-cache entries display their input component, suppressing the accompanying serial number when it is stored as part of the entry. In particular, the input cache entry $\text{InpC}[i_s + 1]$ records r'_{i_s+1} and is not modified by answering a query to s , and the database entry $((i_s, x), *||r'_{i_s+1}) \in D$ can only be modified by Decom operations.

Let $|\chi\rangle$ be the normalized state immediately before either of the two decompressions being removed. Before the first, **Cache Clear** has not changed the queried database cell or the successor input-cache entry. Before the second in $\text{Hyb}_{9,a}$, the first decomposition is absent, and the intermediate evaluation also leaves both of these records unchanged. Thus the queried cell is non- $\perp$, and its last λ bits agree with r'_{i_s+1} recorded in $\text{InpC}[i_s + 1]$ in both cases. The queried address is also recorded outside the database by the query input and the unchanged entry $\text{SI}[s]$.

We apply the vector estimate from Lemma 3.3, with $\mathcal{R}_1 = \mathcal{R}$ and $\mathcal{R}_2 = \{0, 1\}^\lambda$. For completeness, the same estimate holds with arbitrary entanglement among the remaining registers. Group the state according to the queried address (i_s, x) , the recorded value r'_{i_s+1} , and the residual database. Write such a group as $j = (i_j, x_j, z_j, D'_j)$, where $D'_j(i_j, x_j) = \perp$. Then

$$|\chi\rangle = \sum_j \sum_{r \in \mathcal{R}} |\xi_{j,r}\rangle \otimes |D'_j \cup \{(i_j, x_j), r||z_j)\}\rangle, \quad \sum_j \sum_{r \in \mathcal{R}} \| |\xi_{j,r}\rangle \|^2 = 1.$$

The vectors $|\xi_{j,r}\rangle$ contain all registers other than the database. Distinct values of (i_j, x_j, z_j) have orthogonal external records; for fixed (i_j, x_j, z_j) , distinct residual databases remain orthogonal.

Let P_+ project the queried cell onto its uniform non- $\perp$ state, controlled on the queried address. By Cauchy-Schwarz,

$$\begin{aligned} \|P_+ |\chi\rangle\|^2 &= \frac{1}{|\mathcal{R}|2^\lambda} \sum_j \left\| \sum_{r \in \mathcal{R}} |\xi_{j,r}\rangle \right\|^2 \\ &\leq \frac{1}{2^\lambda} \sum_j \sum_{r \in \mathcal{R}} \| |\xi_{j,r}\rangle \|^2 = 2^{-\lambda}. \end{aligned}$$

There is no $\perp$ component in the queried cell. Since Decom swaps its uniform non- $\perp$ state with $|\perp\rangle$ and fixes their orthogonal complement, the removed uniform component and the newly created $\perp$ component are orthogonal. Let $|v\rangle := P_+ |\chi\rangle$. Hence

$$\begin{aligned} \|(\text{Decomp} - I) |\chi\rangle\|^2 &= \|\text{Decomp} |v\rangle - |v\rangle\|^2 \\ &= \|\text{Decomp} |v\rangle\|^2 + \| |v\rangle \|^2 - 2 \text{Re}(\langle v | \text{Decomp} |v\rangle) \\ &= \| |v\rangle \|^2 + \| |v\rangle \|^2 \\ &= 2 \|P_+ |\chi\rangle\|^2 \leq \frac{1}{2^{\lambda-1}}. \end{aligned}$$

The operations following each replacement are identical isometries in the two purified circuits, so they preserve vector norm. Therefore

$$\begin{aligned} \|(\text{Hyb}_9 - \text{Hyb}_{9,a}) |\eta\rangle\| &\leq \sqrt{1/2^{\lambda-1}}, \\ \|(\text{Hyb}_{9,a} - \text{Hyb}_{9,b}) |\eta\rangle\| &\leq \sqrt{1/2^{\lambda-1}}. \end{aligned}$$

Combining these bounds with the exact remaining changes gives

$$\|(\text{Hyb}_9 - \text{Hyb}_{10}) |\eta\rangle\| \leq 2\sqrt{1/2^{\lambda-1}}.$$

By the initial reduction and homogeneity,

$$\text{TD}[\text{Hyb}_9 |\psi\rangle, \text{Hyb}_{10} |\psi\rangle] \leq \|(\text{Hyb}_9 - \text{Hyb}_{10}) |\psi\rangle\| \leq 2\sqrt{1/2^{\lambda-1}} \|\Pi_{\text{early}} |\psi\rangle\| \leq 2\sqrt{1/2^{\lambda-1}}.$$

Let q be the number of oracle calls made by the adversary. adversary's operations. Writing $|\psi_t\rangle$ for the state before query t in Hyb_{10} and $|\Psi_9\rangle, |\Psi_{10}\rangle$ for the final purified states, we have

$$\| |\Psi_9\rangle - |\Psi_{10}\rangle \| \leq \sum_{t=1}^q \|(\text{Hyb}_9 - \text{Hyb}_{10}) |\psi_t\rangle\| \leq 2q\sqrt{1/2^{\lambda-1}} = \text{negl}(\lambda)$$

for polynomial q . Their trace distance, including the recorded outcomes of the history checks, is at most the same bound. This proves $\text{Hyb}_9 \approx_s \text{Hyb}_{10}$, i.e. statistically indistinguishable.

Finally, since the two experiments are statistically indistinguishable and the history check failing can be detected, it must happen with almost the same probability in Hyb_{10} as in Hyb_9 , i.e. negligible. $\square$

Claim 5.14 (Hybrid: Output Cache Queries H). For all adversaries,

$$\text{Hyb}_{10} \approx_s \text{Hyb}_{11}$$

Furthermore, the **history consistency check** fails with negligible probability in Hyb_9 .

Proof. The only difference between Hyb_{10} and Hyb_{11} is in the caching routine, specifically when updating the output cache $\text{OutC}[\text{SI}[s]]$ in step 4b.²¹ In particular, in Hyb_{11} the randomness r is read directly from D 's database, while in Hyb_{10} the random oracle H is queried to obtain r . Simplifying the computation as before, the output cache update in the two hybrids can be written as

1. (Hyb_{11} only) Compute $\text{Decomp}_{\text{SI}[s],x}$ on the database register.
2. Compute and classical-copy $P_{\text{OSP}}(x, \text{inp}', s_{i+1}; s; r)$ to the output cache register using a symmetric difference, controlled on $r = D(i_s, x)$.
3. (Hyb_{11} only) Compute $\text{Decomp}_{\text{SI}[s],x}$ on the database register.

Thus, we need to show that whenever the output cache would be either cleared or filled, the two Decomp operations act statistically close to the identity. The output cache is only cleared or filled at index $i_s - 1$ when a query is made to a serial number s where $\text{SI}[s] = i_s$ and an early answer is *not* triggered.²² By the history check, there exists an entry $(i_s - 1, x_{i_s-1}, r_{i_s-1}) \in D$ such the inp_{i_s} part of the query contains the last that the last λ bits of r_{i_s-1} . By the SEQ check and cache invariant, there exists an entry $\text{InpC}[i_s - 1] \neq \perp$ containing x_{i_s-1} . Therefore by Lemma 3.3, Decomp acts close to the identity on the query $(i_s - 1, x_{i_s-1}$ which is made to modify the output cache.

Finally, since the two experiments are statistically indistinguishable and the history check failing can be detected, it must happen with almost the same probability in Hyb_{10} as in Hyb_9 , i.e. negligible. $\square$

Claim 5.15 (Hybrid: Remove History Check). For all QPT adversaries,

$$\text{Hyb}_{11} \approx_c \text{Hyb}_{12}$$

²¹The usage of the bit g is implicit in Hyb_{10} .

²²Unlike in the proof of Claim 5.13, we cannot rely on the existence of a "later" entry in the cache because the early answer does *not* trigger, meaning there is no later entry recorded.

Proof. The only difference between Hyb_{11} and Hyb_{12} is Hyb_{11} measures the history check and aborts if it fails. By Claim 5.13, this happens with negligible probability and so is gentle. $\square$

Claim 5.16 (Hybrid: Cache Uses Oracle). For all adversaries,

$$\text{Hyb}_{12} = \text{Hyb}_{13}$$

Proof. By Claim 5.2, $\text{Refl}_i^{O_{\text{Hyb}}^P}$ is equivalent to flipping g controlled on D not containing any entries $((i, *), *)$. Since g starts at 1, the cache steps activate in Hyb_{12} if and only if there is an entry $((i, *), *) \in D$, the same as in Hyb_{11} . $\square$

Claim 5.17 (Hybrid: Evaluation uses Oracle). For all adversaries,

$$\text{Hyb}_{13} = \text{Hyb}_{14}$$

Proof. Expanding the definition of O_{Hyb}^P , there are three differences between Hyb_{13} and Hyb_{14} . First, the computation of $P(\text{st}, x; r_x)$ is separated from the computation of the signatures. This clearly is only a syntactic difference.

Second, the compressed oracle H is evaluated a second time to uncompute r_x (note that r_2 is provided as part of the output of O_{Hyb}^P , and so is not uncomputed) at the end of querying O_{Hyb}^P in step 5a and start of unquerying O_{Hyb}^P in the last step of **Evaluation**. Observe that the steps in between querying O_{Hyb}^P do not operate on the database register $\mathcal{D}$ or the work register used for querying H (the output (y, r_2) is moved to a separate register). Therefore the extra two queries to H commute with the operations in between querying O_{Hyb}^P and cancel with each other.

Third, O_{Hyb}^P only answers queries $|i, x, u, b\rangle$ if $\exists((i, x), *) \in D$ and no other entry has a higher index i' , or if i is one higher than the highest index recorded in D . By the cache invariant (Claim 5.7) and the **early answer** step, Hyb_{12} does not do **Evaluation** if there is an entry $((i', *), *) \in D$ where $i' > i$. Furthermore, if i matches the maximum entry $((i, x'), *) \in D$, then the **SEQ check** guarantees that $x' = x$. $\square$

Combining these claims, $A_{\text{Sim}_1}^{O_{\text{Hyb}}^P, O_{\text{SEQ}}^{P_{\text{OSP}}[\text{sk}, \text{pk}, \text{sk}_{\text{SE}}]}} = \text{Hyb}_0$ is computationally indistinguishable from $\text{Hyb}_{12} = A_{\text{Sim}_1}^{O_{\text{Hyb}}^P}$. $\square$

5.3.3 Simulating the Hybrid Oracle

Here we show that the hybrid oracle O_{Hyb}^P (fig. 9) can be simulated with the REQ oracle O_P^{LEQ} (fig. 6). The simulation is indistinguishable from the real hybrid oracle to any distinguisher that makes polynomially-many queries. Note that the distinguisher implicitly knows the program P because the distinguisher is quantified after P , so the distinguisher may have P hardwired.

Proposition 2. *There exists a simulator Sim such that for all RAM programs P and all distinguishers $D(1^\lambda)$ that make $\text{poly}(\lambda)$ -many queries to a given oracle, there is a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,*

$$\left| \Pr \left[1 \leftarrow D^{O_{\text{Hyb}}^P(1^\lambda)}(1^\lambda) \right] - \Pr \left[1 \leftarrow D^{\text{Sim}^{O_P^{\text{LEQ}}}(1^\lambda)}(1^\lambda) \right] \right| \leq \text{negl}(\lambda)$$

where O_P^{LEQ} is the oracle given in fig. 6, and O_{Hyb}^P is the oracle given in fig. 9.

Proof.

The simulator.

First, we describe the simulator Sim in fig. 13.

Registers: The simulator acts on the following registers:

- Q : The query register taken as input. Q has standard basis states of the form $|t, x, u, b\rangle_Q$. t denotes the tag, x denotes the input, u is the register where the output will be written, and b serves as an indicator for whether the query was answered.
- $C = C_1 \otimes \cdots \otimes C_{k'}$: An internal register of variable length that stores a cache $C = [(t_i, x'_i, y_i, g'_i)]_{i \in [k']}$. t_i is the tag, x'_i is the input, y_i is the output, and g'_i is a bit indicating whether this cache location is in use ($g'_i = 0$) or not ($g'_i = 1$). Initially, C has length 0.
- $\mathcal{D}$: An internal register that represents the database of a compressed oracle $G : [2^\lambda] \times \mathcal{X} \rightarrow \{0, 1\}^\lambda$. $\mathcal{D}$ is initialized to $|\emptyset\rangle$.
- $\mathcal{B}'$: An internal register that decides whether to answer the query from the user. $\mathcal{B}'$ is initialized to $|0\rangle$, and additionally $\mathcal{B}' = |0\rangle$ at the start of every query.

Figure 12: Registers for The Simulator $\text{Sim}^{O_P^{\text{LEQ}}}$ for O_{Hyb}^P

Queries. To respond to a query, the simulator acts on $\mathcal{Q} \times \mathcal{C} \times \mathcal{D} \times \mathcal{W} \times \mathcal{B}'$ as the following unitary. Given some basis state $|t, x, u, b\rangle_{\mathcal{Q}} \otimes |C\rangle_{\mathcal{C}} \otimes |D\rangle_{\mathcal{D}} \otimes |0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}'}$, the simulator does the following:

1. **Expand C by Two Registers.** Append two new entries $|0, 0, 0, 0\rangle_{C_{k'+1}} \otimes |0, 0, 0, 0\rangle_{C_{k'+2}}$ to C .
2. **Validate Query.** Let $k_{\max}$ be the maximum index $i \in [k']$ such that $g'_i = 1$, or if no such i exists, then let $k_{\max} = 0$.
If $\mathcal{Q}$ contains $t = k_{\max}$ and $x = x'_{k_{\max}}$, or if $\mathcal{Q}$ contains $t = k_{\max} + 1$, then apply X to $\mathcal{B}'$.

3. **Manage Cache.**

- (a) **ReflectAndCopy:** If $g'_t = 0$, then CNOT (t, x) from $\mathcal{Q}$ to C_t :

$$|t, x'_t, y_t, 0\rangle_{C_t} \mapsto |t \oplus t, x'_t \oplus x, y_t, 0\rangle_{C_t}$$

- (b) If $g_i = 1$, $\text{Decomp}_{t,x}$ to $\mathcal{D}$.

- (c) **Query** O_P^{LEQ} : If $\mathcal{D}$ is in the span of states of the form

$$|D\rangle \quad \text{or} \quad \frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

for all databases D that do not contain $(t, x, *)$, then query O_P^{LEQ} on C_t .^a Otherwise, query O_P^{LEQ} on a dummy query with $t = -1$.

- (d) Repeat step 3b.

- (e) **ReflectAndCopy:** Repeat step 3a.

4. **Answer Query:** If $\mathcal{B}'$ contains 1, then answer the query by copying (y, r_2) from $C_t \times \mathcal{D}$ onto $\mathcal{Q}$ and flipping b :

$$|t, x, u, b\rangle_{\mathcal{Q}} \mapsto |t, x, u \oplus (y, r_2), b \oplus 1\rangle_{\mathcal{Q}}$$

5. **Manage Cache.** Repeat step 3.

6. (Validate Query) Repeat step 2.

^aThe last bit g_t in C_t serves as the response indicator bit for this query. It flips if the query is answered.

Figure 13: The Simulator $\text{Sim}^{O_P^{\text{LEQ}}}$ for O_{Hyb}^P

State Transition Function of O_P^{LEQ} .

We show that step 3 of O_P^{LEQ} can be replaced with an operation that simply swaps two states (lemma 5.3).

Definition 5.2. For any query $(t, x, u, b) \in [2^\lambda] \times \mathcal{X} \times \mathcal{Y} \times \{0, 1\}$ to O_P^{LEQ} and any program state st , let

$$\left| \phi_{t,x,u,b,\text{st}}^{\text{LEQ}} \right\rangle = \frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r_1 \in \mathcal{R}} |t, x, u \oplus y_{r_1}, b \oplus 1\rangle_{\mathcal{Q}} \otimes |\text{st}'_{x,r_1}\rangle_{\mathcal{S}} \otimes |\text{st}, x, r_1\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t}$$

where $(\text{st}'_{x,r_1}, y_{r_1}) = P(\text{st}, x; r_1)$.

Definition 5.3 (Reachable States of O_P^{LEQ}). Let the state of S^{LEQ} be the space of states on O_P^{LEQ} 's internal registers for which $\mathcal{G}_i = |1\rangle$ or $\mathcal{G}_i \times \mathcal{A}_i = |0\rangle \otimes |0\rangle$ for all $i \in [k]$.

Then lemma 5.3 describes the state transition function of step 3 on any state in S^{LEQ} .

Lemma 5.3. For any query $(t, x, u, b) \in [2^\lambda] \times \mathcal{X} \times \mathcal{Y} \times \{0, 1\}$ to O_P^{LEQ} and any program state st , step 3 of O_P^{LEQ} swaps the following states if $\mathcal{B} = |1\rangle$:

$$|t, x, u, b\rangle_Q \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \leftrightarrow \left| \phi_{t,x,u,b,\text{st}}^{\text{LEQ}} \right\rangle$$

and acts as the identity on all states in S^{LEQ} that are orthogonal to the ones above.

Proof. We will consider an orthonormal basis for S^{LEQ} and show that for every basis state, step 3 acts the way that lemma 5.3 claims.

Case 1: The input to step 3 has $\mathcal{B} = |0\rangle$.

Then step 3 aborts and acts as the identity on this state.

Case 2: The input to step 3 is has $\mathcal{A}_t \times \mathcal{G}_t \times \mathcal{B} = |0\rangle \otimes |0\rangle \otimes |1\rangle$.

Then the input is in the span of states of the form: $|t, x, u, b\rangle_Q \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}}$.

Let us step through the action of step 3 on this state. Since $\mathcal{B} = |1\rangle$, step 3 does not abort. Next, step 3a does nothing because $\mathcal{G}_t = |0\rangle$. Step 3b verifies that $\mathcal{A}_t = |0\rangle$ and then maps:

$$|0\rangle_{\mathcal{G}_t} \otimes |b\rangle_{Q_b} \mapsto |1\rangle_{\mathcal{G}_t} \otimes |b \oplus 1\rangle_{Q_b}$$

Finally, step 3c verifies that $\mathcal{G}_t = |1\rangle$ and then applies U_P , which acts as follows:

$$\begin{aligned} & |t, x, u, b \oplus 1\rangle_Q \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \\ & \mapsto \frac{1}{\sqrt{|\mathcal{R}|}} \sum_{r_1 \in \mathcal{R}} |t, x, u \oplus y_{r_1}, b \oplus 1\rangle_Q \otimes |\text{st}'_{x,r_1}\rangle_S \otimes |\text{st}, x, r_1\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \\ & = \left| \phi_{t,x,u,b,\text{st}}^{\text{LEQ}} \right\rangle \end{aligned}$$

This shows that step 3 maps

$$|t, x, u, b\rangle_Q \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \mapsto \left| \phi_{t,x,u,b,\text{st}}^{\text{LEQ}} \right\rangle$$

Case 3: The input to step 3 has $\mathcal{A}_t \times \mathcal{G}_t \times \mathcal{B} = |a\rangle \otimes |0\rangle \otimes |1\rangle$, where $a \neq 0$.

This state is not in S^{LEQ} , so we don't need to consider it.

Case 4: The input $|\psi\rangle$ to step 3 has $\mathcal{G}_t \times \mathcal{B} = |1\rangle \otimes |1\rangle$ and satisfies $|\psi\rangle = U_P \cdot |0\rangle_{\mathcal{A}_t} \cdot U_P^\dagger \cdot |\psi\rangle$.

We claim that in this case, $|\psi\rangle$ is in the span of states of the form $\left| \phi_{t,x,u,b,\text{st}}^{\text{LEQ}} \right\rangle$. First, $U_P^\dagger \cdot |\psi\rangle$ is in the span of $|0\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}}$. Then

$$U_P^\dagger \cdot |\psi\rangle = \sum_{t,x,u,b',\text{st}} \alpha(t, x, u, b', \text{st}) \cdot |t, x, u, b'\rangle_Q \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}}$$

for some amplitude function α . Next, we set $b = b' \oplus 1$:

$$U_P^\dagger \cdot |\psi\rangle = \sum_{t,x,u,b,\text{st}} \alpha(t, x, u, b \oplus 1, \text{st}) \cdot |t, x, u, b \oplus 1\rangle_Q \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}}$$

Finally, we apply U_P :

$$|\psi\rangle = U_P \cdot U_P^\dagger \cdot |\psi\rangle = \sum_{t,x,u,b,st} \alpha(t,x,u,b \oplus 1, st) \cdot \left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle$$

This shows that $|\psi\rangle$ is in the span of states of the form $\left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle$.

Next, let us step through the action of step 3 on each basis state $\left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle$. Since $\mathcal{G}_t \times \mathcal{B} = |1\rangle \otimes |1\rangle$, step 3 does not abort, and then step 3a applies $U_P^\dagger$, which acts as follows:

$$\begin{aligned} \left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle &= \frac{1}{\sqrt{|\mathcal{R}|}} \cdot \sum_{r_1 \in \mathcal{R}} |t, x, u \oplus y_{r_1}, b \oplus 1\rangle_Q \otimes |st'_{x,r_1}\rangle_S \otimes |st, x, r_1\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}} \\ &\mapsto |t, x, u, b \oplus 1\rangle_Q \otimes |st\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |1\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}} \end{aligned}$$

Next, step 3b verifies that $\mathcal{A}_t = |0\rangle$ and then does the following:

$$|1\rangle_{\mathcal{G}_t} \otimes |b \oplus 1\rangle_{Q_b} \mapsto |0\rangle_{\mathcal{G}_t} \otimes |b\rangle_{Q_b}$$

Finally, step 3c does nothing because $\mathcal{G}_t = |0\rangle$. In total, this shows that step 3 maps:

$$\left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle \mapsto |t, x, u, b\rangle_Q \otimes |st\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}}$$

Case 5: The input $|\psi\rangle$ to step 3 has $\mathcal{G}_t \times \mathcal{B} = |1\rangle \otimes |1\rangle$ and satisfies $|\psi\rangle = U_P \cdot (I - |0\rangle\langle 0|)_{\mathcal{A}_t} \cdot U_P^\dagger \cdot |\psi\rangle$.

Let us step through the action of step 3 on $|\psi\rangle$. Since $\mathcal{G}_t \times \mathcal{B} = |1\rangle \otimes |1\rangle$, step 3 does not abort, and then step 3a applies $U_P^\dagger$ to $|\psi\rangle$. Note that $U_P^\dagger \cdot |\psi\rangle$ is in the span of $(I - |0\rangle\langle 0|)_{\mathcal{A}_t}$. Next, step 3b acts as the identity because the state is in the span of $(I - |0\rangle\langle 0|)_{\mathcal{A}_t}$. Finally, step 3c applies U_P because $\mathcal{G}_t = |1\rangle$, which restores the state to

$$U_P \cdot U_P^\dagger \cdot |\psi\rangle = |\psi\rangle$$

In total, this shows that step 3 acts as the identity in this case. $\square$

The following corollary says that S^{LEQ} contains all reachable states of O_P^{LEQ} .

Corollary 5.1. After any number of quantum queries to O_P^{LEQ} , the state of O_P^{LEQ} 's internal registers is in S^{LEQ} (definition 5.3).

Proof. We will prove this inductively. The initial state of O_P^{LEQ} is in S^{LEQ} because $\mathcal{A}$ and $\mathcal{G}$ have length $k = 0$. Next, let us assume that at the start of a given query, the state of O_P^{LEQ} is in S^{LEQ} . Then we will show that at the end of the query, the state is still in S^{LEQ} .

During a query to O_P^{LEQ} , step 1 appends registers $|0\rangle_{\mathcal{G}_{k+1}} \otimes |0\rangle_{\mathcal{A}_{k+1}}$, which satisfy the condition to be in S^{LEQ} . Step 2 does not change the computational-basis values of $\mathcal{G} \times \mathcal{A}$, so the state after step 2 is still in S^{LEQ} .

According to lemma 5.3, step 3 swaps

$$|t, x, u, b\rangle_Q \otimes |st\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}} \leftrightarrow \left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle$$

and acts as the identity on all states in S^{LEQ} that are orthogonal to the ones above. Note that both states above are in S^{LEQ} : $|t, x, u, b\rangle_Q \otimes |st\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \otimes |1\rangle_{\mathcal{B}}$ satisfies $\mathcal{G}_t \times \mathcal{A}_t = |0\rangle \otimes |0\rangle$ and $\left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle$ satisfies $\mathcal{G}_t = |1\rangle$. Therefore, step 3 maps any state in S^{LEQ} to a state in S^{LEQ} .

Finally, step 4 does not change the computational-basis values of $\mathcal{G} \times \mathcal{A}$, so the state at the end of the query is still in S^{LEQ} . $\square$

Hybrids

Hybrid 1: The simulator $\text{Sim}^{O_P^{\text{LEQ}}}(1^\lambda)$ in fig. 13.

Hybrid 2: This hybrid opens up the queries to O_P^{LEQ} and replaces it with its state transition function.

Given some basis state $|t, x, u, b\rangle_Q \otimes |C\rangle_C \otimes |D\rangle_{\mathcal{D}} \otimes |0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}'}$, the simulator does the following:

1. **Expand $C \times \mathcal{G} \times \mathcal{A}$ by Two Registers.** Append the following registers to $C \times \mathcal{G} \times \mathcal{A}$:

$$|0\rangle_{C_{k+1}} \otimes |0\rangle_{C_{k+2}} \otimes |0\rangle_{\mathcal{G}_{k+1}} \otimes |0\rangle_{\mathcal{G}_{k+2}} \otimes |0\rangle_{\mathcal{A}_{k+1}} \otimes |0\rangle_{\mathcal{A}_{k+2}}$$

2. **Validate Query.** Let $k_{\max}$ be the maximum index $i \in [k]$ such that $g'_i = 1$, or if no such i exists, then let $k_{\max} = 0$.

If Q contains $t = k_{\max}$ and $x = x'_{k_{\max}}$, or if Q contains $t = k_{\max} + 1$, then apply X to $\mathcal{B}'$.

3. **Manage Cache.**

- (a) **ReflectAndCopy:** If $g'_t = 0$, then CNOT (t, x) from Q to C_t :

$$|t_t, x'_t, u_t, 0\rangle_{C_t} \mapsto |t_t \oplus t, x'_t \oplus x, u_t, 0\rangle_{C_t}$$

- (b) If $g'_t = 1$, apply $\text{Decomp}_{t,x}$ to $\mathcal{D}$.

- (c) (Query O_P^{LEQ}) If $\mathcal{D}$ is in the span of states of the form

$$|D\rangle \quad \text{or} \quad \frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

for all databases D that do not contain $(t, x, *)$, then swap states of the following form:

$$|t_t, x'_t, u_t, g'_t\rangle_{C_t} \otimes |st\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \leftrightarrow |\phi_{t_t, x'_t, u_t, g'_t, st}^{\text{LEQ}}\rangle$$

- (d) If $g'_t = 1$, apply $\text{Decomp}_{t,x}$ to $\mathcal{D}$.

- (e) **ReflectAndCopy:** Repeat step 3a.

4. **Answer Query:** If $\mathcal{B}'$ contains 1, then answer the query by copying (y, r_2) from $C_t \times \mathcal{D}$ onto Q and flipping b :

$$|t, x, u, b\rangle_Q \mapsto |t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q$$

5. **Manage Cache.** Repeat step 3.

6. (Validate Query) Repeat step 2.

Lemma 5.4. Hybrids 1 and 2 are perfectly indistinguishable after any number of quantum queries.

Proof. The first difference between hybrids 1 and 2 is that step 1 of hybrid 1 appends two registers to C whereas step 1 of hybrid 2 appends two registers to each of C , $\mathcal{G}$, and $\mathcal{A}$. Note that hybrid 1 always makes two queries to O_P^{LEQ} , and each query appends one register to $\mathcal{G}$ and $\mathcal{A}$ each. So Hybrid 1 also appends two registers to each of C , $\mathcal{G}$, and $\mathcal{A}$, but some of these append operations occur inside O_P^{LEQ} . Hybrid 2 simply moves all these append operations to step 1.

The second difference between hybrids 1 and 2 is that in hybrid 1, step 3 of O_P^{LEQ} has been replaced by the state transition function given in lemma 5.3. These operations are equivalent according to lemma 5.3.

The third and final difference between hybrids 1 and 2 is that hybrid 2 omits the query validation steps of O_p^{LEQ} (steps 2 and 4 of O_p^{LEQ}). This change is undetectable because hybrid 1 implements its own query validation step in step 2. If step 2 accepts the query, then the subsequent calls to O_p^{LEQ} will also accept the query and flip $\mathcal{B}$. Furthermore, if step 2 rejects the query, then the hybrid acts as the identity (other than appending new registers). If step 2 rejects the query, then step 4 acts as the identity, and the remaining steps cancel out: steps 2 - 3e cancel out with steps 5 - 6. $\square$

Hybrid 3: Modify when the $\text{Decomp}_{t,x}$ operations trigger. Recall that in Hybrid 2, the first ReflectAndCopy operation is followed by the three steps: apply $\text{Decomp}_{t,x}$ if $g'_t = 1$; perform the RAM query only on components for which the database has no $(t, x, *)$ entry; and again apply $\text{Decomp}_{t,x}$ if $g'_t = 1$. The reverse sequence is used after **Answer Query**, between the third and fourth ReflectAndCopy operations.

In step 3b, instead of checking g'_t , trigger $\text{Decomp}_{t,x}$ if $C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t$ is orthogonal to $|\phi_{t,x,u,b,st}^{\text{LEQ}}\rangle$. Then, in step 3d, trigger $\text{Decomp}_{t,x}$ unconditionally. Their reverse sequence after answering the query is also replaced accordingly.

Given some basis state $|t, x, u, b\rangle_Q \otimes |C\rangle_C \otimes |D\rangle_{\mathcal{D}} \otimes |0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}'}$, the simulator does the following:

1. **Expand $C \times \mathcal{G} \times \mathcal{A}$ by Two Registers.** Append the following registers to $C \times \mathcal{G} \times \mathcal{A}$:

$$|0\rangle_{C_{k+1}} \otimes |0\rangle_{C_{k+2}} \otimes |0\rangle_{\mathcal{G}_{k+1}} \otimes |0\rangle_{\mathcal{G}_{k+2}} \otimes |0\rangle_{\mathcal{A}_{k+1}} \otimes |0\rangle_{\mathcal{A}_{k+2}}$$

2. **Validate Query.** Let $k_{\max}$ be the maximum index $i \in [k]$ such that $g'_i = 1$, or if no such i exists, then let $k_{\max} = 0$.

If Q contains $t = k_{\max}$ and $x = x'_{k_{\max}}$, or if Q contains $t = k_{\max} + 1$, then apply X to $\mathcal{B}'$.

3. **Manage Cache.**

- (a) ReflectAndCopy : If $g'_t = 0$, then CNOT (t, x) from Q to C_t :

$$|t_t, x'_t, u_t, 0\rangle_{C_t} \mapsto |t_t \oplus t, x'_t \oplus x, u_t, 0\rangle_{C_t}$$

- (b) Apply $\text{Decomp}_{t,x}$ to $\mathcal{D}$, **controlled on the component of $C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t$ orthogonal to all the following states:**

$$\begin{aligned} &|t, x, u, b\rangle_{C_t} \otimes |st\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{G}_t} \otimes |0\rangle_{\mathcal{A}_t}, \quad \forall (u, b, st), \\ &|\phi_{t,x,u,b,st}^{\text{LEQ}}\rangle, \quad \forall (u, b, st). \end{aligned}$$

- (c) (Query O_p^{LEQ}) If $\mathcal{D}$ is in the span of states of the form

$$|D\rangle \quad \text{or} \quad \frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

for all databases D that do not contain $(t, x, *)$, then swap states of the following form:

$$|t_t, x'_t, u_t, g'_t\rangle_{C_t} \otimes |st\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t} \leftrightarrow |\phi_{t_t, x'_t, u_t, g'_t, st}^{\text{LEQ}}\rangle$$

- (d) Apply $\text{Decomp}_{t,x}$ to $\mathcal{D}$ **unconditionally**.
- (e) ReflectAndCopy : Repeat step 3a.

4. **Answer Query:** If $\mathcal{B}'$ contains 1, then answer the query by copying (y, r_2) from $C_t \times \mathcal{D}$ onto Q and flipping b :

$$|t, x, u, b\rangle_Q \mapsto |t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q$$

5. **Manage Cache.**

- (a) (ReflectAndCopy) Repeat step 3e.
- (b) (**Unconditional** $\text{Decomp}_{t,x}$) Repeat step 3d.
- (c) (Query O_p^{LEQ}) Repeat step 3c.
- (d) **Repeat step 3b.**
- (e) (ReflectAndCopy) Repeat step 3a.
- (f) (Validate Query) Repeat step 2.

Lemma 5.5. Hybrids 2 and 3 are perfectly indistinguishable after any number of quantum queries.

Proof. The only changes are during the **manage cache** step, in particular steps 3b to 3d (and the reverse in the repetition of **manage cache**). We claim that outside of these steps, the state of the simulator is identical between Hybrids 2 and 3. By linearity, it suffices to show this for computational basis queries $|t, x', u, 0\rangle$.

Let $V_{t,x}$ be the span of states of the form

$$|D\rangle \quad \text{or} \quad \frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

for all databases D that do not contain $(t, x, *)$. Observe that $\text{Decomp}_{t,x}$ acts trivially on the orthogonal space $V_{t,x}^\perp$ and denote

$$\left| \xi_{t,t,x',u_t,g'_t,\text{st}}^{\text{LEQ}} \right\rangle := |t, x', u_t, g'_t\rangle_{C_t} \otimes |\text{st}\rangle_S \otimes |0\rangle_{\mathcal{A}_t} \otimes |0\rangle_{\mathcal{G}_t}$$

so that the query operation swaps

$$\left| \xi_{t,t,x',u_t,g'_t,\text{st}}^{\text{LEQ}} \right\rangle \leftrightarrow \left| \phi_{t,t,x',u_t,g'_t,\text{st}}^{\text{LEQ}} \right\rangle$$

Let W_{LEQ} be the span of these two states.

Consider the following sequence of hybrid experiments which modify only the three steps 3b to 3d. First, Hybrid 2.1 inserts two $\text{Decomp}_{t,x}$ operations after the conditional query:

1. If $g'_t = 1$, apply $\text{Decomp}_{t,x}$.
2. Conditional Query: If $\mathcal{D}$ is supported on $V_{t,x}$, swap $\left| \xi_{t,t,x',u_t,g'_t,\text{st}}^{\text{LEQ}} \right\rangle \leftrightarrow \left| \phi_{t,t,x',u_t,g'_t,\text{st}}^{\text{LEQ}} \right\rangle$.
3. **New:** If $g'_t = 0$, apply $\text{Decomp}_{t,x}$.
4. **New:** If $g'_t = 0$, apply $\text{Decomp}_{t,x}$.
5. If $g'_t = 1$, apply $\text{Decomp}_{t,x}$.

Since $\text{Decomp}_{t,x}$ is an involution, the new steps are together the identity. Observe that the last two steps can be combined together as *always* do $\text{Decomp}_{t,x}$.

Next, Hybrid 2.2 moves the new step 3 to *before* the conditional LEQ evaluation:

1. If $g'_t = 1$, apply $\text{Decomp}_{t,x}$.

2. Moved and Modified:

- If $(C_t, \mathcal{S}, \mathcal{A}_t, \mathcal{G}_t)$ are supported on W_{LEQ} and $g'_t = 0$, apply $\text{Decomp}_{t,x}$.
 - If $(C_t, \mathcal{S}, \mathcal{A}_t, \mathcal{G}_t)$ are *not* supported on W_{LEQ} and $g'_t = 1$, apply $\text{Decomp}_{t,x}$.
3. Conditional Query: If $\mathcal{D}$ is supported on $V_{t,x}$, swap $\left| \xi_{t_t, x'_t, u_t, g'_t, \text{st}}^{\text{LEQ}} \right\rangle \leftrightarrow \left| \phi_{t_t, x'_t, u_t, g'_t, \text{st}}^{\text{LEQ}} \right\rangle$.
4. Apply $\text{Decomp}_{t,x}$.

To see equivalence, first observe that since $V_{t,x}$ is invariant under $\text{Decomp}_{t,x}$, an unconditional application of $\text{Decomp}_{t,x}$ commutes with the conditional query in step 3. When $(C_t, \mathcal{S}, \mathcal{A}_t, \mathcal{G}_t)$ are supported on W_{LEQ} , step 3 flips the control bit g'_t . Therefore switching the ordering is equivalent with the control flipped when $(C_t, \mathcal{S}, \mathcal{A}_t, \mathcal{G}_t)$ are supported on W_{LEQ} .

Finally, we do a case analysis to combine the three conditional applications of $\text{Decomp}_{t,x}$ before step 3. The first and third step cancel whenever $(C_t, \mathcal{S}, \mathcal{A}_t, \mathcal{G}_t)$ are *not* supported on W_{LEQ} . Otherwise, $\text{Decomp}_{t,x}$ is applied when $g'_t = 0$ and when $g'_t = 1$. In other words, the aggregate effect of these three steps is to apply $\text{Decomp}_{t,x}$ when $(C_t, \mathcal{S}, \mathcal{A}_t, \mathcal{G}_t)$ are supported on W_{LEQ} . Thus Hybrid 2.2 is equivalent to Hybrid 3. $\square$

State Transition Function of Sim.

Definition 5.4 (States of Sim in Hybrid 4). Let $s \in (\mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda)^*$ be a (potentially empty) tuple, and for each $i \in [|s|]$, let

$$s_i = (x_i, r_{1,i}, r_{2,i}) \in \mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda.$$

Let $k \in \{0\} \cup \mathbb{N}$ satisfy $k \geq |s|$. Let $\text{st}_1 = \perp$, and for each $i \in [|s|]$, let

$$\begin{aligned} (\text{st}_{i+1}, y_i) &= P(\text{st}_i, x_i; r_{1,i}) \\ a_i &= (\text{st}_i, x_i, r_{1,i}) \\ c_i &= (i, x_i, y_i, 1) \\ d_i &= (i, x_i, r_{2,i}) \\ g_i &= 1 \end{aligned}$$

Then, for each $i \in (|s|, k]$, let

$$\begin{aligned} a_i &= (0, 0, 0) \\ c_i &= (0, 0, 0, 0) \\ g_i &= 0 \end{aligned}$$

For each such k, s , let us define the following computational-basis eigenstate:

$$\begin{aligned} \left| \psi_{k,s}^{\text{Sim}} \right\rangle &= |0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}'} \\ &\otimes |c_1\rangle_{C_1} \otimes \cdots \otimes |c_k\rangle_{C_k} \otimes \left| \{d_i\}_{i \in [|s|]} \right\rangle_{\mathcal{D}} \\ &\otimes |a_1\rangle_{\mathcal{A}_1} \otimes \cdots \otimes |a_k\rangle_{\mathcal{A}_k} \otimes |g_1\rangle_{\mathcal{G}_1} \otimes \cdots \otimes |g_k\rangle_{\mathcal{G}_k} \\ &\otimes |\text{st}_{|s|+1}\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{B}} \end{aligned}$$

Let us define a Fourier basis state as follows. For any such k, s and any $x \in \mathcal{X}$, $\tilde{r}_1 \in \mathcal{R}$ and $\tilde{r}_2 \in \{0, 1\}^\lambda$, let:

$$\left| \phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}} \right\rangle = \frac{1}{\sqrt{|R|} \cdot 2^\lambda} \cdot \sum_{(r_1, r_2) \in \mathcal{R} \times \{0, 1\}^\lambda} (-1)^{\langle r_1, \tilde{r}_1 \rangle + \langle r_2, \tilde{r}_2 \rangle} \cdot \left| \psi_{k,s}^{\text{Sim}} \right\rangle_{(x, r_1, r_2)}$$

Note that the initial state of Sim's internal registers is $|\psi_{0,\emptyset}^{\text{Sim}}\rangle$:

$$|\psi_{0,\emptyset}^{\text{Sim}}\rangle = |0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}'} \otimes |\emptyset\rangle_{\mathcal{D}} \otimes |\perp\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{B}}$$

Claim 5.18. For any given $k \in \{0\} \cup \mathbb{N}$, the states $\left\{ |\psi_{k,s}^{\text{Sim}}\rangle \right\}_s$ of definition 5.4 are orthonormal.

Proof. First, each $|\psi_{k,s}^{\text{Sim}}\rangle$ is a computational-basis eigenstate, so it has unit norm.

Next, let us consider two different s -values $s \neq s'$ that have the same length $k_{\max}$. We will show that $\langle \psi_{k,s}^{\text{Sim}} | \psi_{k,s'}^{\text{Sim}} \rangle = 0$. First, $k_{\max} \geq 1$ because there is only one s -value ($s = \emptyset$) of length 0. Second, every component of s or s' is written on some register of $|\psi_{k,s}^{\text{Sim}}\rangle$ or $|\psi_{k,s'}^{\text{Sim}}\rangle$ respectively. For each $i \in [k_{\max}]$, $(x_i, r_{1,i})$ are written on register $\mathcal{A}_i$, and $r_{2,i}$ is written in the i -th entry d_i of register $\mathcal{D}$. If $s \neq s'$, then $|\psi_{k,s}^{\text{Sim}}\rangle$ and $|\psi_{k,s'}^{\text{Sim}}\rangle$ will disagree on one of the registers listed above. Therefore, $\langle \psi_{k,s}^{\text{Sim}} | \psi_{k,s'}^{\text{Sim}} \rangle = 0$.

Finally, let us consider two s -values (s, s') of different lengths. We will show that $\langle \psi_{k,s}^{\text{Sim}} | \psi_{k,s'}^{\text{Sim}} \rangle = 0$. Let $k_{\max}$ be the length of s , and let $k'_{\max}$ be the length of s' . Without loss of generality, let us assume that $k_{\max} > k'_{\max}$. Then the register $\mathcal{G}_{k_{\max}}$ will store $|1\rangle$ in $|\psi_{k,s}^{\text{Sim}}\rangle$ and will store $|0\rangle$ in $|\psi_{k,s'}^{\text{Sim}}\rangle$. Therefore, $\langle \psi_{k,s}^{\text{Sim}} | \psi_{k,s'}^{\text{Sim}} \rangle = 0$.

In summary, we've shown that if $s \neq s'$, then $\langle \psi_{k,s}^{\text{Sim}} | \psi_{k,s'}^{\text{Sim}} \rangle = 0$. $\square$

Hybrid 4: The initial state of the simulator is $|\psi_{0,\emptyset}^{\text{Sim}}\rangle$. Next, given any query of the form $|t, x, u, b\rangle_Q$, the simulator does the following:

1. Increase: Map

$$|\psi_{k,s}^{\text{Sim}}\rangle \mapsto |\psi_{k+2,s}^{\text{Sim}}\rangle$$

2. Decomp^{Sim}: For any sequence $s \in (\mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda)^*$ of length $t-1$, swap the following states:

$$|\psi_{k,s}^{\text{Sim}}\rangle \leftrightarrow |\phi_{k,s,x,0,0}^{\text{Sim}}\rangle$$

and act as the identity on all states orthogonal to the ones above.

3. Copy^{Sim}:

- For any sequence $s \in (\mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda)^*$ of length t for which $s_t = (x, *, *)$, swap the following states:

$$|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle \leftrightarrow |t, x, u \oplus (y_t, r_{2,t}), b \oplus 1\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$$

where $y_t, r_{2,t}$ are computed from s as described in definition 5.4.

Act as the identity on all states orthogonal to the ones above.

4. (Decomp^{Sim})[†]: Uncompute step 2.

The following definition defines the space of reachable states of hybrid 3 and 4's internal registers.

Definition 5.5 (Reachable States of the Simulator). Let $\mathcal{S}^{\text{Sim}}$ be the span of states of the form $|\psi_{k,s}^{\text{Sim}}\rangle$ for all (potentially empty) sequences $s \in (\mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda)^*$ and all $k \geq |s|$.

The following will help us show that step 3 of hybrid 3 can be replaced with $\text{Decomp}^{\text{Sim}}$, step 2 of hybrid 4.

Lemma 5.6. For any state of the form $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$, where $(t, x) = (|s|, x_{|s|})$ or $t = |s| + 1$, step 3 of hybrid 3 acts on this state equivalently to step 2 of hybrid 4 ($\text{Decomp}^{\text{Sim}}$) and produces a state in S^{Sim} .

Proof.

Case 1: The input is $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$, where $|s| = t - 1$.

Let us step through the action of steps 3a - 3e of hybrid 3 on this state. Initially, $C_t \times \mathcal{G}_t \times \mathcal{A}_t = |0\rangle_{C_t} \times |0\rangle_{\mathcal{G}_t} \times |0\rangle_{\mathcal{A}_t}$, and $(t, x, *) \notin D$. Step 3a (ReflectAndCopy) maps:

$$|0\rangle_{C_t} \times |0\rangle_{\mathcal{G}_t} \times |0\rangle_{\mathcal{A}_t} \mapsto |t, x, 0, 0\rangle_{C_t} \times |0\rangle_{\mathcal{G}_t} \times |0\rangle_{\mathcal{A}_t}$$

Step 3b acts as the identity because $C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t$ is in the span of the states

$$|t, x, u, b\rangle_{C_t} \otimes |\text{st}\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{G}_t} \otimes |0\rangle_{\mathcal{A}_t}, \quad \forall (u, b, \text{st}).$$

Step 3c (Query O_P^{LEQ}) verifies that $(t, x, *) \notin D$ and then maps:

$$|t, x, 0, 0\rangle_{C_t} \otimes |\text{st}_{|s|+1}\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{G}_t} \otimes |0\rangle_{\mathcal{A}_t} \mapsto |\phi_{t,x,0,0,\text{st}_{|s|+1}}^{\text{LEQ}}\rangle$$

At the start of step 3d ($\text{Decomp}_{t,x}$), $\mathcal{D}$ is in the span of databases D such that $(t, x, *) \notin D$. Step 3d maps:

$$|D\rangle_{\mathcal{D}} \mapsto \frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

Step 3e acts as the identity because $g'_t = 1$.

The final state of the simulator's internal registers at the end of step 3d is:

$$\frac{1}{\sqrt{|R| \cdot 2^\lambda}} \cdot \sum_{(r_1, r_2) \in \mathcal{R} \times \{0,1\}^\lambda} |\psi_{k,s||x,r_1,r_2}^{\text{Sim}}\rangle = |\phi_{k,s,x,0,0}^{\text{Sim}}\rangle$$

Next, let us examine the action of $\text{Decomp}^{\text{Sim}}$ (step 2 of hybrid 4) on $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$. $\text{Decomp}^{\text{Sim}}$ maps:

$$|\psi_{k,s}^{\text{Sim}}\rangle \mapsto |\phi_{k,s,x,0,0}^{\text{Sim}}\rangle$$

In this case, $\text{Decomp}^{\text{Sim}}$ acts equivalently to steps 3a - 3e of hybrid 3, and the final state is in S^{Hyb} .

Case 2: The input is $|t, x, u, b\rangle_Q \otimes |\phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle$, where $\tilde{r}_1 = 0$, $\tilde{r}_2 = 0$, and $|s| = t - 1$.

Let us step through the action of steps 3a - 3e of hybrid 3 on this state. Initially, since $\tilde{r}_1 = 0$,

$$C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t = |\phi_{t,x,0,0,\text{st}_{|s|+1}}^{\text{LEQ}}\rangle,$$

and since $\tilde{r}_2 = 0$, $\mathcal{D}$ is in the span of states of the form:

$$\frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

for all databases D that do not contain $(t, x, *)$.

Step 3a (ReflectAndCopy) verifies that $g'_t = 1$ and then acts as the identity.

Step 3b acts as the identity because $C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t$ is in the span of the states

$$\left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle, \quad \forall (u, b, st)$$

Step 3c (Query O_P^{LEQ}) maps:

$$\left| \phi_{t,x,0,0,st_{|s|+1}}^{\text{LEQ}} \right\rangle \mapsto |t, x, 0, 0\rangle_{C_t} \otimes |st_{|s|+1}\rangle_S \otimes |0\rangle_{\mathcal{G}_t} \otimes |\mathbf{0}\rangle_{\mathcal{A}_t}$$

Step 3d ($\text{Decomp}_{t,x}$) maps:

$$\frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle \mapsto |D\rangle_{\mathcal{D}}$$

where $(t, x, *) \notin D$.

Step 3e verifies that $g'_t = 0$ and then maps:

$$|t, x, 0, 0\rangle_{C_t} \mapsto |0, 0, 0, 0\rangle_{C_t}$$

The final state of the simulator's internal registers at the end of step 3e is $|\psi_{k,s}^{\text{Sim}}\rangle$.

Next, let us examine the action of $\text{Decomp}^{\text{Sim}}$ (step 2 of hybrid 4) on the initial state $|t, x, u, b\rangle_Q \otimes |\phi_{k,s,x,0,0}^{\text{Sim}}\rangle$. $\text{Decomp}^{\text{Sim}}$ maps:

$$|\phi_{k,s,x,0,0}^{\text{Sim}}\rangle \mapsto |\psi_{k,s}^{\text{Sim}}\rangle$$

In this case, $\text{Decomp}^{\text{Sim}}$ acts equivalently to steps 3a - 3e of hybrid 3, and the final state is in S^{Hyb} .

Case 3: The input is $|t, x, u, b\rangle_Q \otimes |\phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle$, where $\tilde{r}_1 = 0$, $\tilde{r}_2 \neq 0$, and $|s| = t - 1$.

Let $D = \{d_i\}_{i \in [|s|]}$, where d_i is defined in Definition 5.4; in particular, D does not contain $(t, x, *)$. By that definition, the state on $C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t \times \mathcal{D}$ factors as

$$\left| \phi_{t,x,0,0,st_t}^{\text{LEQ}} \right\rangle \otimes \frac{1}{\sqrt{2^\lambda}} \sum_{r_2 \in \{0,1\}^\lambda} (-1)^{\langle r_2, \tilde{r}_2 \rangle} |D \cup \{(t, x, r_2)\}\rangle.$$

All the other internal registers are fixed by k, s . The database component is orthogonal to every database state with no $(t, x, *)$ entry. Its inner product with

$$\frac{1}{\sqrt{2^\lambda}} \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

is

$$\frac{1}{2^\lambda} \sum_{r_2 \in \{0,1\}^\lambda} (-1)^{\langle r_2, \tilde{r}_2 \rangle} = 0,$$

since $\tilde{r}_2 \neq 0$. The uniform states associated with different residual databases are orthogonal as well.

Step 3a (ReflectAndCopy) acts as the identity because $g'_t = 1$.

Step 3b acts as the identity because the RAM component is $|\phi_{t,x,0,0,st_t}^{\text{LEQ}}\rangle$, one of the states excluded by its control condition.

Step 3c acts as the identity because, by the two orthogonality calculations above, the database component is orthogonal to the entire subspace specified in its control condition.

Step 3d ($\text{Decomp}_{t,x}$) acts as the identity because the database component is orthogonal to both states that this decomposition swaps, for every fixed residual database.

Step 3e (ReflectAndCopy) acts as the identity because g'_t is still 1.

Therefore steps 3a–3e leave $\left| \phi_{k,s,x,0,\tilde{r}_2}^{\text{Sim}} \right\rangle$ unchanged. The operator $\text{Decomp}^{\text{Sim}}$ also leaves it unchanged: by Definition 5.4,

$$\left\langle \psi_{k,s}^{\text{Sim}} \left| \phi_{k,s,x,0,\tilde{r}_2}^{\text{Sim}} \right. \right\rangle = 0, \quad \left\langle \phi_{k,s,x,0,0}^{\text{Sim}} \left| \phi_{k,s,x,0,\tilde{r}_2}^{\text{Sim}} \right. \right\rangle = 0,$$

and the states belonging to different predecessor sequences are orthogonal. Thus the input is orthogonal to every state swapped by $\text{Decomp}^{\text{Sim}}$ for this query. The two operations agree in Case 3, and the final state remains in S^{Sim} .

Case 4: The input is $|t, x, u, b\rangle_Q \otimes \left| \phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}} \right\rangle$, where $\tilde{r}_1 \neq 0$, $\tilde{r}_2 = 0$, and $|s| = t - 1$.

Let us step through the action of steps 3a - 3e of hybrid 3 on this state. Initially, since $\tilde{r}_1 \neq 0$, $C_t \times S \times \mathcal{G}_t \times \mathcal{A}_t$ is orthogonal to states of the form:

$$\begin{aligned} &|t, x, u, b\rangle_{C_t} \otimes |st\rangle_S \otimes |0\rangle_{\mathcal{G}_t} \otimes |0\rangle_{\mathcal{A}_t}, \quad \forall (u, b, st) \\ &\left| \phi_{t,x,u,b,st}^{\text{LEQ}} \right\rangle, \quad \forall (u, b, st), \end{aligned}$$

and since $\tilde{r}_2 = 0$, $\mathcal{D}$ is in the span of states of the form:

$$\frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

for all databases D that do not contain $(t, x, *)$.

Step 3a (ReflectAndCopy) verifies that $g'_t = 1$ and then acts as the identity.

Step 3b applies $\text{Decomp}_{t,x}$ to $\mathcal{D}$, which maps:

$$\frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle \mapsto |D\rangle$$

Step 3c ($\text{Query } O_P^{\text{LEQ}}$) verifies that $\mathcal{D}$ is in the span of databases D that do not contain $(t, x, *)$. Then the step acts on $C_t \times S \times \mathcal{G}_t \times \mathcal{A}_t$ as the identity because $C_t \times S \times \mathcal{G}_t \times \mathcal{A}_t$ is orthogonal to the states that this step swaps.

Step 3d ($\text{Decomp}_{t,x}$) maps:

$$|D\rangle \mapsto \frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle$$

Step 3e verifies that $g'_t = 1$ and then acts as the identity.

The final state of the simulator's internal registers at the end of step 3e is the initial state $\left| \phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}} \right\rangle$.

Next, let us examine the action of $\text{Decomp}^{\text{Sim}}$ (step 2 of hybrid 4) on the initial state $|t, x, u, b\rangle_Q \otimes \left| \phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}} \right\rangle$. $\text{Decomp}^{\text{Sim}}$ also acts as the identity. Since $\tilde{r}_1 \neq 0$, then $\left| \phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}} \right\rangle$ is orthogonal to the states that $\text{Decomp}^{\text{Sim}}$ swaps.

In this case, $\text{Decomp}^{\text{Sim}}$ acts equivalently to steps 3a - 3e of hybrid 3, and the final state is in S^{Hyb} .

Case 5: The input is $|t, x, u, b\rangle_Q \otimes |\phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle$, where $\tilde{r}_1 \neq 0$, $\tilde{r}_2 \neq 0$, and $|s| = t - 1$.

Let us step through the action of steps 3a - 3e of hybrid 3 on this state. Initially, since $\tilde{r}_1 \neq 0$, $C_t \times \mathcal{S} \times \mathcal{G}_t \times \mathcal{A}_t$ is orthogonal to states of the form:

$$\begin{aligned} &|t, x, u, b\rangle_{C_t} \otimes |st\rangle_{\mathcal{S}} \otimes |0\rangle_{\mathcal{G}_t} \otimes |0\rangle_{\mathcal{A}_t}, \quad \forall(u, b, st) \\ &|\phi_{t,x,u,b,st}^{\text{LEQ}}\rangle, \quad \forall(u, b, st), \end{aligned}$$

and since $\tilde{r}_2 \neq 0$, $\mathcal{D}$ is orthogonal to states of the form:

$$\begin{aligned} &|D\rangle \\ &\frac{1}{\sqrt{2^\lambda}} \cdot \sum_{r_2 \in \{0,1\}^\lambda} |D \cup \{(t, x, r_2)\}\rangle \end{aligned}$$

for all databases D that do not contain $(t, x, *)$.

Step 3a (ReflectAndCopy) verifies that $g'_t = 1$ and then acts as the identity.

Step 3b applies $\text{Decomp}_{t,x}$ to $\mathcal{D}$, which also acts as the identity because $\mathcal{D}$ is orthogonal to the states that $\text{Decomp}_{t,x}$ swaps.

Step 3c (Query O_P^{LEQ}) acts as the identity because $\mathcal{D}$ is orthogonal to the states that this step checks.

Step 3d ($\text{Decomp}_{t,x}$) acts as the identity because $\mathcal{D}$ is orthogonal to the states that $\text{Decomp}_{t,x}$ swaps.

Step 3e verifies that $g'_t = 1$ and then acts as the identity.

The final state of the simulator's internal registers at the end of step 3e is the initial state $|\phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle$.

Next, let us examine the action of $\text{Decomp}^{\text{Sim}}$ (step 2 of hybrid 4) on the initial state $|t, x, u, b\rangle_Q \otimes |\phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle$. $\text{Decomp}^{\text{Sim}}$ also acts as the identity. Since $\tilde{r}_1 \neq 0$ and $\tilde{r}_2 \neq 0$, then $|\phi_{k,s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle$ is orthogonal to the states that $\text{Decomp}^{\text{Sim}}$ swaps.

In this case, $\text{Decomp}^{\text{Sim}}$ acts equivalently to steps 3a - 3e of hybrid 3, and the final state is in S^{Hyb} . $\square$

Lemma 5.7. At the start of step 4 of hybrid 3, the state of the hybrid

Lemma 5.8. Hybrids 3 and 4 are perfectly indistinguishable after any number of quantum queries.

Proof. We will prove that (1) at the start of any query, the state of hybrid 3 or 4's internal registers is in S^{Sim} , and (2) that a query to hybrids 3 or 4 acts equivalently if the initial state is in S^{Sim} . We will prove these claims inductively. For the base case, note that the initial state of the simulator's registers in hybrids 3 or 4 is $|\psi_{0,\emptyset}^{\text{Sim}}\rangle \in S^{\text{Sim}}$. Next, let us assume as the inductive hypothesis that at the start of any query, the state of the simulator's internal registers is in S^{Sim} . We will show that then hybrids 3 and 4 act equivalently on this state, and the state at the end of the query is also in S^{Sim} .

It suffices to consider the action of each hybrid on every state of the form: $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$.

Case 1: The input is $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$, where $(t, x) \neq (|s|, x_{|s|})$ and $t \neq |s| + 1$.

Let us step through the action of hybrid 3 on this state. Step 1 appends registers to the state, mapping:

$$|\psi_{k,s}^{\text{Sim}}\rangle \mapsto |\psi_{k+2,s}^{\text{Sim}}\rangle$$

Step 2 (Validate Query) computes $(k_{\max}, x'_{k_{\max}}) = (|s|, x_{|s|})$ and then leaves $\mathcal{B}' = |0\rangle$ because $(t, x) \neq (k_{\max}, x'_{k_{\max}})$ and $t \neq k_{\max} + 1$.

Next, each step from 3a to 3e is an involution (it is its own inverse). Step 4 (Answer Query) acts as the identity because $\mathcal{B}' = |0\rangle$. Then steps 5a - 5e uncompute steps 3a - 3e because they apply steps 3a - 3e in reverse order.

Finally, step 5f acts as the identity, the same as step 2. The final state is $|t, x, u, b\rangle_Q \otimes |\psi_{k+2,s}^{\text{Sim}}\rangle$.

Now let us step through the action of hybrid 4 on $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$. Step 1 appends registers, mapping:

$$|\psi_{k,s}^{\text{Sim}}\rangle \mapsto |\psi_{k+2,s}^{\text{Sim}}\rangle$$

Step 2 (Decomp^{Sim}) acts as the identity because $|s| \neq t - 1$. Step 3 (Copy^{Sim}) acts as the identity because $(t, x) \neq (|s|, x_{|s|})$ and $|s| \neq t - 1$. Step 4 (Decomp^{Sim}) acts as the identity as well. The final state is $|t, x, u, b\rangle_Q \otimes |\psi_{k+2,s}^{\text{Sim}}\rangle$, which is in S^{Sim} . Furthermore, we've shown that hybrids 3 and 4 act equivalently in this case.

Case 2: The input is $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$, where $t = |s| + 1$.

Let us step through the action of hybrid 3 on this state. Step 1 appends registers to the state, mapping:

$$|\psi_{k,s}^{\text{Sim}}\rangle \mapsto |\psi_{k+2,s}^{\text{Sim}}\rangle$$

which acts equivalently to step 1 of hybrid 4. Step 2 (Validate Query) computes $(k_{\max}, x'_{k_{\max}}) = (|s|, x_{|s|})$ and then maps

$$|0\rangle_{\mathcal{B}'} \mapsto |1\rangle_{\mathcal{B}'}$$

because $t = k_{\max} + 1 = |s| + 1$.

Next, steps 3a - 3e act equivalently to Decomp^{Sim}. This follows from lemma 5.6. At the end of step 3e, the state is:

$$|t, x, u, b\rangle_Q \otimes |\phi_{k+2,s,x,0,0}^{\text{Sim}}\rangle$$

Note that $|\phi_{k+2,s,x,0,0}^{\text{Sim}}\rangle$ is a superposition over states of the form $|\psi_{k+2,s||x,r_1,r_2}^{\text{Sim}}\rangle \in S^{\text{Sim}}$.

Step 4 (Answer Query) verifies that $(t, x, *) \in D$, then reads (y, r_2) from $C_t \times D_t$, and finally maps:

$$|t, x, u, b\rangle_Q \otimes |\psi_{k+2,s||x,r_1,r_2}^{\text{Sim}}\rangle \mapsto |t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q \otimes |\psi_{k+2,s||x,r_1,r_2}^{\text{Sim}}\rangle$$

Note that step 4 acts the same as Copy^{Sim}.

Steps 5a - 5e uncompute steps 3a - 3e and act equivalently to $(\text{Decomp}^{\text{Sim}})^\dagger = \text{Decomp}^{\text{Sim}}$.

Finally, step 5f (Validate Query) computes $(k_{\max}, x'_{k_{\max}}) = (|s| + 1, x) = (t, x)$ and then maps:

$$|1\rangle_{\mathcal{B}'} \mapsto |0\rangle_{\mathcal{B}'}$$

This shows that in this case, hybrid 3 acts equivalently to hybrid 4, and the final state is in S^{Sim} .

Case 3: The input is $|t, x, u, b\rangle_Q \otimes |\psi_{k,s}^{\text{Sim}}\rangle$, where $(t, x) = (|s|, x_{|s|})$.

Let us step through the action of hybrid 3 on this state. Step 1 appends registers to the state, mapping:

$$|\psi_{k,s}^{\text{Sim}}\rangle \mapsto |\psi_{k+2,s}^{\text{Sim}}\rangle$$

which acts equivalently to step 1 of hybrid 4. Step 2 (Validate Query) computes $(k_{\max}, x'_{k_{\max}}) = (|s|, x_{|s|})$ and then maps

$$|0\rangle_{\mathcal{B}'} \mapsto |1\rangle_{\mathcal{B}'}$$

because $(t, x) = (|s|, x_{|s|}) = (k_{\max}, x'_{k_{\max}})$.

At the start of step 3a, the state of the simulator is $|\psi_{k+2,s}^{\text{Sim}}\rangle$, which is a superposition over states of the form

$$|\phi_{k+2,s_{[t-1]},x,\tilde{r}_1,\tilde{r}_2}^{\text{Sim}}\rangle, \quad \forall (\tilde{r}_1, \tilde{r}_2) \in \mathcal{R} \times \{0,1\}^\lambda$$

where $s_{[t-1]} = (s_1, \dots, s_{t-1})$.

Next, steps 3a - 3e act equivalently to $\text{Decomp}^{\text{Sim}}$. This follows from lemma 5.6. At the end of step 3e, the state is in a superposition over states of the form:

$$\begin{aligned} & |\psi_{k+2,s_{[t-1]}}^{\text{Sim}}\rangle \\ & |\psi_{k+2,s_{[t-1]}||(x,r_1,r_2)}^{\text{Sim}}\rangle, \quad \forall (r_1, r_2) \end{aligned}$$

Step 4 (Answer Query) acts equivalently to Copy^{Sim} . If the simulator's state is $|\psi_{k+2,s_{[t-1]}}^{\text{Sim}}\rangle$, then $(t, x, *) \notin D$, so step 4 maps

$$|t, x, u, b\rangle_Q \mapsto |t, x, u, b \oplus 1\rangle_Q,$$

which Copy^{Sim} does as well. If the simulator's state is $|\psi_{k+2,s_{[t-1]}||(x,r_1,r_2)}^{\text{Sim}}\rangle$, then $(t, x, *) \in D$, so step 4 acts the same as it did in case 2 above.

Steps 5a - 5e uncompute steps 3a - 3e and act equivalently to $(\text{Decomp}^{\text{Sim}})^\dagger = \text{Decomp}^{\text{Sim}}$.

Finally, step 5f (Validate Query) computes either $(k_{\max}, x'_{k_{\max}}) = (|s|, x) = (t, x)$ or $k_{\max} = t - 1$. In either case, the step maps:

$$|1\rangle_{\mathcal{B}'} \mapsto |0\rangle_{\mathcal{B}'}$$

This shows that in this case, hybrid 3 acts equivalently to hybrid 4, and the final state is in S^{Sim} . $\square$

State Transition Function of O_{Hyb}^P .

Let us define the reachable states of O_{Hyb}^P 's internal registers.

Definition 5.6 (States of Sim in Hybrid 5). Let $s \in (\mathcal{X} \times \mathcal{R} \times \{0,1\}^\lambda)^*$ be a (potentially empty) tuple, and for each $i \in [|s|]$, let

$$s_i = (x_i, r_{1,i}, r_{2,i}) \in \mathcal{X} \times \mathcal{R} \times \{0,1\}^\lambda.$$

Next, let $\text{st}_1 = \perp$, and for each $i \in [|s|]$, let

$$\begin{aligned} d_i &= (i, s_i) = (i, x_i, r_{1,i}, r_{2,i}) \\ (\text{st}_{i+1}, y_i) &= P(\text{st}_i, x_i; r_{1,i}) \end{aligned}$$

Let us define a computational-basis state as follows:

$$|\psi_s^{\text{Hyb}}\rangle = |\{d_i\}_{i \in [|s|]}\rangle_{\mathcal{D}} \otimes |0,0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}}$$

Let us define a Fourier basis state as follows. For any $x \in \mathcal{X}$, $\tilde{r}_1 \in \mathcal{R}$ and $\tilde{r}_2 \in \{0,1\}^\lambda$, let:

$$\begin{aligned} |\phi_{s,x,\tilde{r}_1,\tilde{r}_2}^{\text{Hyb}}\rangle &= \frac{1}{\sqrt{|R| \cdot 2^\lambda}} \cdot \sum_{(r_1,r_2) \in \mathcal{R} \times \{0,1\}^\lambda} (-1)^{\langle r_1, \tilde{r}_1 \rangle + \langle r_2, \tilde{r}_2 \rangle} |\psi_{s|||(x,r_1,r_2)}^{\text{Hyb}}\rangle \\ &= \frac{1}{\sqrt{|R| \cdot 2^\lambda}} \cdot \sum_{(r_1,r_2) \in \mathcal{R} \times \{0,1\}^\lambda} (-1)^{\langle r_1, \tilde{r}_1 \rangle + \langle r_2, \tilde{r}_2 \rangle} |\{d_i\}_{i \in [|s|]} \cup \{|s|+1, x, r_1, r_2\}\rangle_{\mathcal{D}} \otimes |0,0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}} \end{aligned}$$

Hybrid 5: The initial state of the simulator is $|\psi_0^{\text{Hyb}}\rangle$. Next, given any query of the form $|t, x, u, b\rangle_Q$, the simulator does the following:

1. Increase: (omitted)
2. Decomp^{Hyb}: For any sequence $s \in (\mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda)^*$ of length $t-1$, swap the following states:

$$|\psi_s^{\text{Hyb}}\rangle \leftrightarrow |\phi_{s,x,0,0}^{\text{Hyb}}\rangle$$

and act as the identity on all states orthogonal to the ones above.

3. Copy^{Hyb}:
 - For any sequence $s \in (\mathcal{X} \times \mathcal{R} \times \{0, 1\}^\lambda)^*$ of length t for which $s_t = (x, *, *)$, swap the following states:

$$|t, x, u, b\rangle_Q \otimes |\psi_s^{\text{Hyb}}\rangle \leftrightarrow |t, x, u \oplus (y_t, r_{2,t}), b \oplus 1\rangle_Q \otimes |\psi_s^{\text{Hyb}}\rangle$$

where $y_t, r_{2,t}$ are computed from s as described in [definition 5.6](#).

Act as the identity on all states orthogonal to the ones above.

4. (Decomp^{Hyb})[†]: Uncompute step 2.

Lemma 5.9. Hybrids 4 and 5 are perfectly indistinguishable after any number of quantum queries.

Proof. Hybrids 4 and 5 implement essentially the same state transition function with different eigenbases. We will construct a unitary that maps between the bases of hybrids 4 and 5.

Hybrid 4 works with states of the form $|\psi_{k,s}^{\text{Sim}}\rangle$ ([definition 5.4](#)). k refers to the length of registers $C, \mathcal{G}, \mathcal{A}$. Note that after any number of queries $q \in \{0\} \cup \mathbb{N}$, $k = 2q$. This is because before any queries have been made, $k = 0$, and Increase (step 1) is the only step where k changes, and this step increases k by 2 during each query.

Hybrid 5 works with states of the form $|\psi_s^{\text{Hyb}}\rangle$ ([definition 5.6](#)). These states do not depend on k .

For any query number $q \in \{0\} \cup \mathbb{N}$, let us construct a unitary U_q that maps the state in hybrid 4 after q queries to the state in hybrid 5 after q queries. For each sequence s of length $|s| \leq 2q$, let U_q map:

$$|\psi_{2q,s}^{\text{Sim}}\rangle \mapsto |\psi_s^{\text{Hyb}}\rangle$$

Let U_q act as the identity on all states orthogonal to the ones above.

Note that

$$U_0 |\psi_{0,\emptyset}^{\text{Sim}}\rangle = |\psi_\emptyset^{\text{Hyb}}\rangle$$

So U_0 maps the initial state of hybrid 4 to the initial state of hybrid 5. Additionally, U_q acts on the simulator's internal registers only.

Let us use $\{U_q\}_{q \in \{0\} \cup \mathbb{N}}$ to transform hybrid 4 into hybrid 5. Let us start with hybrid 4. Then change the initial state to be $|\psi_\emptyset^{\text{Hyb}}\rangle$. Then on each query $q \in \mathbb{N}$, the simulator applies $U_{q-1}^\dagger$ at the beginning and applies U_q at the end. The state of the simulator after q queries is computed as follows (from right to left):

$$(U_q \cdot \text{Hybrid 4, Query } q \cdot U_{q-1}^\dagger) \cdots (U_2 \cdot \text{Hybrid 4, Query 2} \cdot U_1^\dagger) \cdot (U_1 \cdot \text{Hybrid 4, Query 1} \cdot U_0^\dagger) \cdot |\psi_\emptyset^{\text{Hyb}}\rangle \quad (12)$$

Procedure 12 is perfectly indistinguishable from hybrid 4. The procedure computes the same state as the following procedure:

$$U_q \cdot \text{Hybrid 4, Query } q \cdot \dots \cdot \text{Hybrid 4, Query 2} \cdot \text{Hybrid 4, Query 1} \cdot |\psi_{0,\emptyset}^{\text{Sim}}\rangle \quad (13)$$

This is because the unitaries between queries, such as $U_1^\dagger \cdot U_1$, cancel out, and $U_0^\dagger \cdot |\psi_0^{\text{Hyb}}\rangle = |\psi_{0,\emptyset}^{\text{Sim}}\rangle$. Procedure 13 computes hybrid 4 and then applies U_q after the final query. The application of U_q is undetectable to the party that queries the simulator because U_q acts on the simulator's internal registers only.

Additionally, procedure 12 is equivalent to hybrid 5. Just like hybrid 5, the procedure starts with the state $|\psi_0^{\text{Hyb}}\rangle$. Additionally, each operation of the form $(U_q \cdot \text{Hybrid 4, Query } q \cdot U_{q-1}^\dagger)$ is equivalent to hybrid 5 because hybrids 4 and 5 implement the same state transition function, just on different eigenbases. This shows that hybrids 4 and 5 are perfectly indistinguishable. $\square$

Next, we show that hybrid 5 is equivalent to the hybrid oracle $O_{\text{Hyb}}^P(1^\lambda)$. Hybrid 5 simply describes the abstract state transition function of $O_{\text{Hyb}}^P(1^\lambda)$.

Hybrid 6: The oracle $O_{\text{Hyb}}^P(1^\lambda)$ in fig. 9.

Given some basis state $|t, x, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}} \otimes |0, 0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}}$, the oracle does the following:

1. **Validate Query.** If either $t = |D| \wedge x = x_{|D|}$ or if $t = |D| + 1$, then apply X to register $\mathcal{B}$ to obtain $|1\rangle_{\mathcal{B}}$.
2. **Answer Query.** If $\mathcal{B}$ contains 1, then answer the query as follows.

- (a) CNOT (t, x) from $\mathcal{Q}$ to $\mathcal{W}$.

$$|0, 0, 0\rangle_{\mathcal{W}} \mapsto |t, x, 0\rangle_{\mathcal{W}}$$

Then query $|(t, x), 0\rangle_{\mathcal{W}}$ to the compressed oracle by applying

$$\text{Decomp}_{t,x} \cdot \text{CO}' \cdot \text{Decomp}_{t,x}$$

to registers $\mathcal{W} \times \mathcal{D}$.

- (b) Let $\text{st}_t = \text{State}_t(D)$ and parse $r = (r_1, r_2) \in \mathcal{R} \times \{0, 1\}^\lambda$ from register $\mathcal{W}$. Then compute $(\text{st}_{t+1}, y) = P(\text{st}_t, x; r_1)$.
- (c) Using CNOTs, write the value (y, r_2) to the sub-register of $\mathcal{Q}$ containing $|u\rangle$ and flip $|b\rangle$:

$$|t, x, u, b\rangle_Q \otimes |(t, x), r\rangle_{\mathcal{W}}$$

$$\mapsto$$

$$|t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q \otimes |(t, x), r\rangle_{\mathcal{W}}$$

- (d) Uncompute step 2a. First apply

$$\text{Decomp}_{t,x} \cdot \text{CO}' \cdot \text{Decomp}_{t,x}$$

to registers $\mathcal{W} \times \mathcal{D}$. Then map

$$|t, x, 0\rangle_{\mathcal{W}} \mapsto |0, 0, 0\rangle_{\mathcal{W}}$$

3. Uncompute step 1.

Lemma 5.10. Hybrids 5 and 6 are perfectly indistinguishable after any number of quantum queries.

Proof. Let us first simplify hybrid 6, steps 2a - 2d, which query the compressed oracle and copy the output to $\mathcal{Q}$. Hybrid 6 is equivalent to the following:

Hybrid 5.c: Given some basis state $|t, x, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}} \otimes |0, 0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}}$, the oracle does the following:

1. **Validate Query.** If either $t = |D| \wedge x = x_{|D|}$ or if $t = |D| + 1$, then apply X to register $\mathcal{B}$ to obtain $|1\rangle_{\mathcal{B}}$.
2. **Answer Query.** If $\mathcal{B}$ contains 1, then answer the query as follows.

(a) Apply $\text{Decomp}_{t,x}$ to register $\mathcal{D}$.

(b) CNOT (t, x) from Q to $\mathcal{W}$.

$$|0, 0, 0\rangle_{\mathcal{W}} \mapsto |t, x, 0\rangle_{\mathcal{W}}$$

(c) Apply CO' to registers $\mathcal{W} \times \mathcal{D}$.

(d) Let $\text{st}_t = \text{State}_t(D)$ and parse $r = (r_1, r_2) \in \mathcal{R} \times \{0, 1\}^\lambda$ from register $\mathcal{W}$. Then compute $(\text{st}_{t+1}, y) = P(\text{st}_t, x; r_1)$.

(e) Using CNOTs, write the value (y, r_2) to the sub-register of Q containing $|u\rangle$ and flip $|b\rangle$:

$$\begin{aligned} |t, x, u, b\rangle_Q \otimes |(t, x), r\rangle_{\mathcal{W}} \\ \mapsto \\ |t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q \otimes |(t, x), r\rangle_{\mathcal{W}} \end{aligned}$$

(f) Apply CO' to registers $\mathcal{W} \times \mathcal{D}$.

(g) CNOT (t, x) from Q to $\mathcal{W}$:

$$|t, x, 0\rangle_{\mathcal{W}} \mapsto |0, 0, 0\rangle_{\mathcal{W}}$$

(h) Apply $(\text{Decomp}_{t,x})^\dagger$ to register $\mathcal{D}$.

3. Uncompute step 1.

The only changes we have made are to the $\text{Decomp}_{t,x}$ operations. The first $\text{Decomp}_{t,x}$ operation has been moved to step 2a because it commutes with step 2b. Likewise, the fourth $\text{Decomp}_{t,x}$ operation has been moved to step 2h because it commutes with step 2g. Note that we've also replaced the fourth $\text{Decomp}_{t,x}$ with $(\text{Decomp}_{t,x})^\dagger$. This is just a notational change because $\text{Decomp}_{t,x} = (\text{Decomp}_{t,x})^\dagger$. Finally, the second and third $\text{Decomp}_{t,x}$ operations have been removed from steps 2a and 2d of hybrid 6. We have cancelled them out. First, $\text{Decomp}_{t,x}$ commutes with the steps in the middle (2b and 2c of hybrid 6). These steps compute st_t by reading the first $t - 1$ entries of $\mathcal{D}$, and they read $r = (r_1, r_2)$ from $\mathcal{W}$. In contrast, $\text{Decomp}_{t,x}$ acts only on the t -th entry of $\mathcal{D}$, so it commutes with steps 2b and 2c of hybrid 6. Next, $\text{Decomp}_{t,x}$ is its own inverse, so the second and third applications of $\text{Decomp}_{t,x}$ in hybrid 6 cancel out with each other.

Next, hybrid 5.c is equivalent to the following hybrid:

Hybrid 5.b: Given some basis state $|t, x, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}} \otimes |0, 0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}}$, the oracle does the following:

1. **Validate Query.** If either $t = |D| \wedge x = x_{|D|}$ or if $t = |D| + 1$, then apply X to register $\mathcal{B}$ to obtain $|1\rangle_{\mathcal{B}}$.
2. **Answer Query.** If $\mathcal{B}$ contains 1, then answer the query as follows.

(a) Apply $\text{Decomp}_{t,x}$ to $\mathcal{D}$.

- (b) Let $\text{st}_t = \text{State}_t(D)$, read r from the entry $(t, x, r) \in D$, and parse $r = (r_1, r_2) \in \mathcal{R} \times \{0, 1\}^\lambda$. Then compute $(\text{st}_{t+1}, y) = P(\text{st}_t, x; r_1)$.
- (c) Using CNOTs, write the value (y, r_2) to the sub-register of Q containing $|u\rangle$ and flip $|b\rangle$:

$$\begin{aligned} |t, x, u, b\rangle_Q &\mapsto \\ |t, x, u \oplus (y, r_2), b \oplus 1\rangle_Q \end{aligned}$$

- (d) Apply $(\text{Decomp}_{t,x})^\dagger$ to register $\mathcal{D}$.

3. Uncompute step 1.

The only difference is that we have eliminated all actions on $\mathcal{W}$ (we've cut out the middle man). Now we read r directly from the t -th entry of $\mathcal{D}$. It is clear by inspection that hybrid 5.c is equivalent to hybrid 5.b.

Next, hybrid 5.b is equivalent to the following hybrid:

Hybrid 5.a: Given some basis state $|t, x, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}} \otimes |0, 0\rangle_{\mathcal{W}} \otimes |0\rangle_{\mathcal{B}}$, the oracle does the following:

1. **Validate Query.** If either $t = |D| \wedge x = x_{|D|}$ or if $t = |D| + 1$, then apply X to register $\mathcal{B}$ to obtain $|1\rangle_{\mathcal{B}}$.
2. **Answer Query.** If $\mathcal{B}$ contains 1, then answer the query as follows.
 - (a) Apply $\text{Decomp}^{\text{Hyb}}$.
 - (b) Apply Copy^{Sim} .
 - (c) Apply $(\text{Decomp}^{\text{Hyb}})^\dagger$.
3. Uncompute step 1.

We have replaced $\text{Decomp}_{t,x}$ with $\text{Decomp}^{\text{Hyb}}$. These two operations compute the same function. We have also replaced steps 2b and 2c with Copy^{Sim} . These two operations compute the same function as long as the state at the start of the operation is a superposition over databases D with $(|D|, x_{|D|}) = (t, x)$. This property is enforced by the query validation in step 1 and the properties of the compressed oracle.

Finally, we claim that hybrid 5.a is equivalent to hybrid 5. The only difference is that hybrid 5 always executes $(\text{Decomp}^{\text{Hyb}})^\dagger \cdot \text{Copy}^{\text{Sim}} \cdot \text{Decomp}^{\text{Hyb}}$, whereas hybrid 5.a executes these steps only when the initial state passes query validation in step 1. Hybrids 5.a and 5 are equivalent because $(\text{Decomp}^{\text{Hyb}})^\dagger \cdot \text{Copy}^{\text{Sim}} \cdot \text{Decomp}^{\text{Hyb}}$ acts as the identity on any state that fails query validation. $\square$

In summary, we have shown that $\text{Sim}^{O_P^{\text{LEQ}}}(1^\lambda)$ (hybrid 1) and $O_{\text{Hyb}}^P(1^\lambda)$ (hybrid 6) are perfectly indistinguishable. This completes the proof of proposition 2. $\square$

6 AI Disclosure

LLM tools have been used to check correctness of the proofs and provide feedback on our writing.

References

- [AB26] Prabhanjan Ananth and Divyanshu Bhardwaj. Quantum one time programs: Less assumptions, more feasibility and one message 2pc. *Cryptology ePrint Archive*, 2026.
- [ABG26] Prabhanjan Ananth, Divyanshu Bhardwaj, and Aparna Gupte. Non interactive MPC, (quantumly) revisited. *Cryptology ePrint Archive*, Paper 2026/302, 2026.
- [AGKZ20] Ryan Amos, Marios Georgiou, Aggelos Kiayias, and Mark Zhandry. One-shot signatures and applications to hybrid quantum/classical authentication. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, page 255–268. Association for Computing Machinery, 2020.
- [AMRS20] Gorjan Alagic, Christian Majenz, Alexander Russell, and Fang Song. Quantum-access-secure message authentication via blind-unforgeability. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 788–817. Springer, Cham, May 2020.
- [BDS23] Shalev Ben-David and Or Sattath. Quantum tokens for digital signatures. *Quantum*, 7:901, jan 2023.
- [BGS13] Anne Broadbent, Gus Gutoski, and Douglas Stebila. Quantum one-time programs. In *Advances in Cryptology – CRYPTO 2013*, pages 344–360. Springer Berlin Heidelberg, 2013.
- [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.
- [CLLZ21] Andrea Coladangelo, Jiahui Liu, Qipeng Liu, and Mark Zhandry. Hidden cosets and applications to unclonable cryptography. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part I* 41, pages 556–584. Springer, 2021.
- [Gav12] D. Gavinsky. Quantum money with classical verification. In *2012 IEEE 27th Conference on Computational Complexity*, pages 42–52, June 2012.
- [GGH⁺16] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016.
- [GHS16] Tommaso Gagliardoni, Andreas Hülsing, and Christian Schaffner. Semantic security and indistinguishability in the quantum world. In *Annual international cryptology conference*, pages 60–89. Springer, 2016.
- [GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. One-time programs. In *Advances in Cryptology–CRYPTO 2008: 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17–21, 2008. Proceedings* 28, pages 39–56. Springer, 2008.
- [GLQ⁺26] Aparna Gupte, Jiahui Liu, Luowen Qian, Justin Raizes, Bhaskar Roberts, and Mark Zhandry. On best-possible one-time programs. *IACR Cryptol. ePrint Arch.*, 2026:413, 2026.
- [GLR⁺25] Aparna Gupte, Jiahui Liu, Justin Raizes, Bhaskar Roberts, and Vinod Vaikuntanathan. Quantum one-time programs, revisited. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pages 213–221, 2025.

- [GM24] Sam Gunn and Ramis Movassagh. Quantum one-time protection of any randomized algorithm. *private communication*, 2024.
- [HHWW19] Ariel Hamlin, Justin Holmgren, Mor Weiss, and Daniel Wichs. On the plausibility of fully homomorphic encryption for RAMs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 589–619. Springer, Cham, August 2019.
- [HV25] Andrew Huang and Vinod Vaikuntanathan. A simple and efficient one-shot signature scheme, 2025.
- [KL20] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC Cryptography and Network Security Series. CRC Press, 2020.
- [Liu23] Qipeng Liu. Depth-bounded quantum cryptography with applications to one-time memory and more. In *14th Innovations in Theoretical Computer Science Conference (ITCS 2023)*, 2023.
- [LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In Barna Saha and Rocco A. Servedio, editors, *55th ACM STOC*, pages 595–608. ACM Press, June 2023.
- [LSZ20] Qipeng Liu, Amit Sahai, and Mark Zhandry. Quantum immune one-time memories. *Cryptology ePrint Archive*, 2020.
- [RS19] Roy Radian and Or Sattath. Semi-quantum money. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT ’19, page 132–146. Association for Computing Machinery, 2019.
- [Shm22a] Omri Shmueli. Public-key quantum money with a classical bank. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 790–803, 2022.
- [Shm22b] Omri Shmueli. Semi-quantum tokenized signatures. In *Annual International Cryptology Conference*, pages 296–319. Springer, 2022.
- [Sta25a] Lev Stambler. Cryptography without long-term quantum memory and global entanglement: Classical setups for one-time programs, copy protection, and stateful obfuscation. *arXiv preprint arXiv:2504.21842*, 2025.
- [Sta25b] Lev Stambler. Information theoretic one-time programs from geometrically local QNC₀ adversaries. *arXiv preprint arXiv:2503.22016*, 2025.
- [SZ25a] Omri Shmueli and Mark Zhandry. On one-shot signatures, quantum vs classical binding, and obfuscating permutations. *Cryptology ePrint Archive*, Paper 2025/486, 2025.
- [SZ25b] Omri Shmueli and Mark Zhandry. Unclonable cryptography in linear quantum memory. *Cryptology ePrint Archive*, Paper 2025/2056, 2025.
- [WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 850–861. ACM Press, October 2015.
- [Zha18] Mark Zhandry. How to record quantum queries, and applications to quantum indistinguishability. *Cryptology ePrint Archive*, Paper 2018/276, 2018.

- [Zha19] Mark Zhandry. How to record quantum queries, and applications to quantum indistinguishability. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019*, pages 239–268, Cham, 2019. Springer International Publishing.

A Compressed Oracle Chaining

In this section, we prove the strengthened variant of [GLR⁺25]’s compressed oracle chaining lemma. We recall the statement of the lemma here for convenience.

Intuitively, the lemma considers an adversary which access to a general composition of random oracles G and H , where G is used to determine the input x_h to H in a somewhat structured manner. It states that, when G and H are implemented as compressed oracles, if H records some input/output $H(x_h) = z$ pair, then G also records a matching input/output pair that leads to x_h .

Lemma A.1 (Compressed Oracle Chaining). Let $G : \mathcal{X}_G \rightarrow \mathcal{Y}_1 \times \mathcal{Y}_2$ and $H : \mathcal{X}_H \times \mathcal{Y}_2 \rightarrow \mathcal{Z}$ be random oracles implemented by the compressed oracle technique.

Let $f : \mathcal{X}_G \times \mathcal{Y}_1 \times \mathcal{Y}_2 \rightarrow \mathcal{X}_H$ be an arbitrary function. Define the function $F_f : \mathcal{X} \rightarrow \mathcal{Z}$ by the following computation for $F(x_g)$:

1. Compute $y_1 \| y_2 := G(x_g)$.
2. Compute $x_h := f(x_g, y_1 \| y_2)$.
3. Output $H(x_h, y_2)$.

Consider running an interaction of an oracle algorithm with F_f until query t , then measuring the internal state of G and H to obtain D_G and D_H .

Let E_t be the event that after the measurement at time t , for all $(x_h \| y_2, z) \in D_H$, there exists an entry $(x_g, y_1 \| y_2) \in D_G$ such that $f(x_g, y_1 \| y_2) = x_h$. Then

$$\Pr[E_t] \geq 1 - \frac{8t^2}{|\mathcal{Y}_2|}$$

A.1 Alternative Representation of the Decompression Function

We first derive a useful alternative representation of the Decompress operation used for compressed oracles.

Let D be a database for a random function H such that $D(x) = \perp$. If one were to query H and receive result $H(x) = y$, the state of the oracle before the final decompression operation is

$|D \cup (x, y)\rangle$. The effect of the x decompression operation Decomp_x on a state $|D \cup (x, y)\rangle$ is

$$\begin{aligned}
& \text{Decomp}_x |D \cup (x, y)\rangle \\
&= \text{Decomp}_x \frac{1}{|\mathcal{Y}|} \sum_{y'} |D \cup (x, y')\rangle \sum_{r \in \mathcal{Y}} (-1)^{r \cdot (y' - y)} \\
&= \text{Decomp}_x \frac{1}{|\mathcal{Y}|} \left(\sum_{r \neq 0} (-1)^{r \cdot -y} \sum_{y'} (-1)^{r \cdot y'} |D \cup (x, y')\rangle + \sum_{y'} |D \cup (x, y')\rangle \right) \\
&= \frac{1}{|\mathcal{Y}|} \left(\sum_{r \neq 0} (-1)^{r \cdot -y} \sum_{y'} (-1)^{r \cdot y'} |D \cup (x, y')\rangle + \sqrt{|\mathcal{Y}|} |D\rangle \right) \\
&= \left(|D \cup (x, y)\rangle - \frac{1}{|\mathcal{Y}|} \sum_{y'} |D \cup (x, y')\rangle + \frac{1}{\sqrt{|\mathcal{Y}|}} |D\rangle \right) \\
&= \left(\left(1 - \frac{1}{|\mathcal{Y}|}\right) |D \cup (x, y)\rangle - \frac{1}{|\mathcal{Y}|} \sum_{y' \neq y} |D \cup (x, y')\rangle + \frac{1}{\sqrt{|\mathcal{Y}|}} |D\rangle \right)
\end{aligned}$$

We note that this only describes the state *in between* queries to x . A second query to x will apply Decomp_x to the database register before determining the response, which maps the database register back to $|D \cup (x, y)\rangle$. Thus, despite the support of the state on other databases, $G(x)$ cannot actually change in between queries to x . However, this form is relevant when looking at the database register without knowledge of x .

A.2 Proof

We first explicitly describe how F works. The internal states of H and G are stored in registers $\mathcal{D}_H$ and $\mathcal{D}_G$, respectively. In general, we can consider a basis query $|x_g, u\rangle_Q$ in register $Q = (Q_{\mathcal{X}_G}, Q_{\mathcal{Z}})$. For convenience, we insert into Q a additional work registers $Q_{\mathcal{X}_H}$ and $Q_{\mathcal{Y}} = (Q_{\mathcal{Y}_1}, Q_{\mathcal{Y}_2})$ which are initialized to $|0\rangle$, resulting in $|x_g, 0, 0, u\rangle_Q$. To compute F on this query, do the following.

1. Query $(Q_{\mathcal{X}_G}, Q_{\mathcal{Y}})$ to G to obtain $|x_g, 0, G(x_g), u\rangle_Q$, i.e. apply $\text{Decomp}_G \circ \text{CO}'_G \circ \text{Decomp}_G$ on $(Q_{\mathcal{X}_G}, Q_{\mathcal{Y}}, \mathcal{D}_G)$.²³
2. Compute f on registers $(\mathcal{X}_G, \mathcal{Y})$ and write the output to register $\mathcal{X}_H$ to obtain $|x_g, x_h, G(x_g), u\rangle_Q$. Let U_f be the corresponding unitary.
3. Query $(Q_{\mathcal{X}_H}, Q_{\mathcal{Y}_2}, Q_{\mathcal{Z}})$ to H to obtain $|x_g, x_h, G(x_g), u \oplus H(x_h, G(x_g))\rangle_Q$.
4. Compute f on registers $(\mathcal{X}_G, \mathcal{Y})$ and write the output to register $\mathcal{X}_H$ to obtain $|x_g, 0, G(x_g), u \oplus F(x_g)\rangle_Q$.
5. Query $(Q_{\mathcal{X}_G}, Q_{\mathcal{Y}})$ to G again to obtain $|x_g, 0, 0, u \oplus F(x_g)\rangle_Q$.
6. Return registers $(Q_{\mathcal{X}_G}, Q_{\mathcal{Z}})$

Observe that Decomp_G commutes with the computation of f , since Decomp_G acts on registers $(Q_{\mathcal{X}_G}, \mathcal{D}_G)$ while U_f acts on registers $(Q_{\mathcal{X}_G}, \mathcal{Y}, \mathcal{X}_H)$ and both are diagonal in the computational basis on the only overlapping register, $Q_{\mathcal{X}_G}$. Furthermore, Decomp_G commutes with the query to H , since they operate on disjoint registers $(Q_{\mathcal{X}_G}, \mathcal{D}_G)$ and $(Q_{\mathcal{X}_H}, Q_{\mathcal{Y}}, Q_{\mathcal{Z}}, \mathcal{D}_H)$, respectively. Recall

²³Since G and H have different domains and ranges, we differentiate their decompression operations, which depend on the domain/range. We also differentiate the CO'_G operation to clarify that it acts on registers corresponding to a query to G .

that $\text{Decomp}_G \circ \text{Decomp}_G = I$. Thus, if we write the operation of H as U_H , we may write the implementation of a query to F as

$$U_F = (\text{Decomp}_G \circ \text{CO}'_G) \circ (U_f \circ U_H \circ U_f) \circ (\text{CO}'_G \circ \text{Decomp}_G)$$

where Decomp_G and CO'_G act on registers $(Q_{\mathcal{X}_G}, Q_{\mathcal{Y}}, \mathcal{D}_G)$, while U_H acts on registers $(Q_{\mathcal{X}_H}, Q_{\mathcal{Y}}, Q_{\mathcal{Z}}, \mathcal{D}_H)$ and U_f operates on registers $(Q_{\mathcal{X}_G}, \mathcal{Y}, \mathcal{X}_H)$.

Now consider the interaction of an oracle algorithm with F , where the algorithm maintains an additional internal register $\mathcal{A}$. Define the projector E onto states

$$|a\rangle_{\mathcal{A}} \otimes |x_g, *, *, *\rangle_Q \otimes |D_G, D_H\rangle_{\mathcal{D}}$$

where D_G and D_H satisfy the requirements of event E_t and define $\bar{E} = I - E$. We will upper bound the norm $\|\bar{E}U_F |\psi\rangle\|$ from after the query in terms of the norm $\|\bar{E} |\psi\rangle\|$ from before the query.

We first bound the intermediate operations, in between the two Decomp_G operations.

Claim A.1.

$$\|\bar{E}(\text{CO}'_G \circ U_f) \circ U_H \circ (U_f \circ \text{CO}'_G) \circ \text{Decomp}_G |\psi\rangle\| = \|\bar{E}(\text{Decomp}_G) |\psi\rangle\|$$

Proof. First, observe that for all states $|\psi'\rangle$,

$$\|\bar{E}\text{CO}'_G |\psi'\rangle\| = \|\bar{E}U_f |\psi'\rangle\| = \|\bar{E} |\psi'\rangle\|$$

since neither CO'_G nor U_f modify the database registers.

Now consider the operation of U_H on $(\text{CO}'_G \cdot \text{Decomp}_G) |\psi\rangle$. Observe that by the definitions of CO'_G and f , the state of the query register and oracle databases when H is queried is always supported on basis states

$$|x_g, x_h, y, u\rangle_Q \otimes |D_G \cup (x_g, y)\rangle_{\mathcal{D}_G} \otimes |D_H\rangle_{\mathcal{D}_H}$$

where $f(x_g, y) = x_h$ and $y = y_1 \| y_2$. A query of (x_h, y_2) to H can only result in a new database D'_H where the only potential difference between D_H and D'_H is $D_H(x_h \| y_2) \neq D'_H(x_h \| y_2)$. Since (x_g, y) is already recorded in the contents of register $\mathcal{D}_G$ and $f(x_g, y) = x_h$, applying H always results in a valid database state with respect to the event E . In particular, if D_H and D_G already satisfied E before the query to H , they continue to do so afterwards. Thus

$$\|\bar{E}U_H \circ (U_f \circ \text{CO}'_G) \circ \text{Decomp}_G |\psi\rangle\| = \|\bar{E}(U_f \circ \text{CO}'_G) \circ \text{Decomp}_G |\psi\rangle\|.$$

Putting this together with the prior bounds on the effect of CO'_G and U_f yields the claim. $\square$

Next, we bound the effect of Decomp_G on a general state $|\psi'\rangle$. In general, both the first and the last Decomp_G operation will operate on a state of the form

$$\sum_{a, x_g, u, D_G, D_H} \alpha_{a, x_g, u, D_G, D_H} |a\rangle_{\mathcal{A}} \otimes |x_g, 0, 0, u\rangle_Q \otimes |D_G, D_H\rangle_{\mathcal{D}}$$

where $\mathcal{A}$ is the adversary's internal register, Q contains the (expanded) query, and $\mathcal{D}$ contains the oracle databases. This is clearly true for the first application; it holds for the second because CO'_G and U_f are applied twice, and U_H does not modify the registers that either on.

Claim A.2. For every normalized state $|\psi'\rangle$,

$$\|\bar{E}\text{Decomp}_G |\psi'\rangle\| \leq \|\bar{E} |\psi'\rangle\| + \sqrt{\frac{2}{|\mathcal{Y}_2|}}.$$

Proof. We first bound $\|\bar{E}\text{Decomp}_G E |\psi'\rangle\|$. Fix the queried input x_g , the database D_H , and all cells of D_G other than x_g . Write D'_G for these fixed cells, with $D'_G(x_g) = \perp$. Also fix computational-basis values of the other registers. These choices define mutually orthogonal subspaces, each preserved by both E and Decomp_G . This is a decomposition for the calculation; no intermediate measurement is performed.

For these fixed values, let T be the set of input pairs recorded in D_H that have no matching entry in D'_G :

$$T := \left\{ (h, v) \in \mathcal{X}_H \times \mathcal{Y}_2 : \begin{array}{l} D_H(h||v) \neq \perp, \text{ and there is no} \\ (x'_g, y_1||v) \in D'_G \text{ with } f(x'_g, y_1||v) = h \end{array} \right\}.$$

Every pair in T must therefore be matched by the queried cell $D_G(x_g)$ for the complete databases to satisfy E .

If $T = \emptyset$, the fixed cells already provide all required matching entries. Every value of the queried cell, including $\perp$, satisfies E , so $\bar{E}\text{Decomp}_G E = 0$ on this subspace. If $|T| \geq 2$, no value of the queried cell can make the databases satisfy E : a non- $\perp$ value $y_1||y_2$ matches only the single input pair $(f(x_g, y_1||y_2), y_2)$. Thus $E = 0$ on this subspace, so $\bar{E}\text{Decomp}_G E = 0$ here too.

It remains to consider $T = \{(h, v)\}$. Define

$$\mathcal{Y} := \mathcal{Y}_1 \times \mathcal{Y}_2, \quad N := |\mathcal{Y}|, \quad A := \{y_1||v : f(x_g, y_1||v) = h\}, \quad a := |A| \leq |\mathcal{Y}_1|.$$

Here y_1 ranges over $\mathcal{Y}_1$. In this subspace, the complete databases satisfy E exactly when $D_G(x_g) \in A$. For the following calculation, abbreviate

$$|\perp\rangle := |D'_G\rangle, \quad |y\rangle := |D'_G \cup (x_g, y)\rangle \quad (y \in \mathcal{Y}),$$

with all other fixed registers suppressed. Thus E acts as $\sum_{y \in A} |y\rangle\langle y|$ on the queried cell. By Appendix A.1,

$$\text{Decomp}_{G, x_g} |y\rangle = |y\rangle - \frac{1}{N} \sum_{w \in \mathcal{Y}} |w\rangle + \frac{1}{\sqrt{N}} |\perp\rangle \quad (y \in \mathcal{Y}).$$

For any, possibly subnormalized, vector $|\eta\rangle = \sum_{y \in A} c_y |y\rangle$, it follows that

$$\bar{E}\text{Decomp}_{G, x_g} |\eta\rangle = \left(\sum_{y \in A} c_y \right) \left(\frac{1}{\sqrt{N}} |\perp\rangle - \frac{1}{N} \sum_{w \in \mathcal{Y} \setminus A} |w\rangle \right).$$

Orthogonality of the database basis and Cauchy–Schwarz give

$$\begin{aligned} \|\bar{E}\text{Decomp}_{G, x_g} |\eta\rangle\|^2 &= \left(\frac{1}{N} + \frac{N-a}{N^2} \right) \left| \sum_{y \in A} c_y \right|^2 \\ &\leq \left(\frac{2a}{N} - \frac{a^2}{N^2} \right) \sum_{y \in A} |c_y|^2 \\ &\leq \frac{2}{|\mathcal{Y}_2|} \|\eta\rangle\|^2. \end{aligned}$$

In fact, for $a > 0$, taking $c_y = 1/\sqrt{a}$ attains the first inequality, so the squared operator norm of $\bar{E}\text{Decomp}_{G, x_g} E$ on this subspace is exactly $2a/N - (a/N)^2$. For $a = 0$, the operator is zero.

The bound is uniform over all fixed choices of x_g , D'_G , D_H , and the other registers. Since these subspaces and their images are mutually orthogonal, summing their squared contributions gives

$$\|\bar{E}\text{Decomp}_G E |\psi'\rangle\| \leq \sqrt{\frac{2}{|\mathcal{Y}_2|}} \|E |\psi'\rangle\|. \quad (14)$$

The contribution from $\bar{E}|\psi'\rangle$ satisfies

$$\left\| \bar{E} \cdot \text{Decomp}_G \cdot \bar{E} |\psi'\rangle \right\| \leq \left\| \bar{E} |\psi'\rangle \right\|. \quad (15)$$

Putting together Equations (14) and (15) with $|\psi'\rangle = E|\psi'\rangle + \bar{E}|\psi'\rangle$, we have

$$\begin{aligned} \left\| \bar{E} \text{Decomp}_G |\psi'\rangle \right\| &\leq \left\| \bar{E} \text{Decomp}_G \bar{E} |\psi'\rangle \right\| + \left\| \bar{E} \text{Decomp}_G E |\psi'\rangle \right\| \\ &\leq \left\| \bar{E} |\psi'\rangle \right\| + \sqrt{\frac{2}{|\mathcal{Y}_2|}} \|E |\psi'\rangle\| \\ &\leq \left\| \bar{E} |\psi'\rangle \right\| + \sqrt{\frac{2}{|\mathcal{Y}_2|}}, \end{aligned}$$

where the last inequality uses $\| |\psi'\rangle \| = 1$. $\square$

Putting together Claim A.1 and Claim A.2, the norm after any single query to F is bounded as

$$\begin{aligned} \left\| \bar{E} \cdot U_F |\psi\rangle \right\| &\leq \left\| \bar{E} \cdot (\text{CO}'_G \circ U_f) \circ U_H \circ (U_f \circ \text{CO}'_G) \circ \text{Decomp}_G |\psi\rangle \right\| + \sqrt{\frac{2}{|\mathcal{Y}_2|}} \\ &= \left\| \bar{E} \text{Decomp}_G |\psi\rangle \right\| + \sqrt{\frac{2}{|\mathcal{Y}_2|}} \\ &\leq \left\| \bar{E} |\psi\rangle \right\| + 2\sqrt{\frac{2}{|\mathcal{Y}_2|}}. \end{aligned}$$

Here $|\psi\rangle$ is normalized. The first and last inequalities account for the two Decomp_G operations remaining in U_F ; the equality uses Claim A.1 for the intervening operations. We may purify the adversary's randomness and measurements in $\mathcal{A}$. Its operations between complete queries then act only on $\mathcal{A}$ and $\mathcal{Q}$, not on the database registers, so they commute with E and preserve $\left\| \bar{E} |\psi\rangle \right\|$.

The norm starts at 0, so after t queries to F it is at most $2t\sqrt{\frac{2}{|\mathcal{Y}_2|}}$. The probability of seeing the event corresponding to $\bar{E}$ when we measure $\mathcal{D}$ is the square of the norm, which is at most $\frac{8t^2}{|\mathcal{Y}_2|}$. Therefore the probability of seeing the complementary event E is at least $1 - \frac{8t^2}{|\mathcal{Y}_2|}$, as claimed.

B Applications

Here we show a variety of applications of one-shot programs (section 4.4) and query-limited programs (section 5).

B.1 Pay-Per-Use Programs

A pay-per-use program can be evaluated many times, after which the user proves the total cost of the queries that they effectively made. The central subtlety is that, in the quantum setting, it is unclear how to charge for a query that is later coherently uncomputed. As in the rest of the paper, we therefore measure usage through the *effective-query* viewpoint: only queries that remain recorded in the ideal oracle count toward the final bill.

Let $\mathcal{P} = \{P : \mathcal{M} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{M} \times \mathcal{Y}\}_P$ be a family of randomized RAM programs. Throughout this subsection we assume that every output $y \in \mathcal{Y}$ contains a designated cumulative-cost field, written $\text{cost}(y) \in \mathbb{R}$, and that the hidden RAM state likewise stores the same cumulative cost. We write “the current cost of st” for this designated value.

Our construction extends a query-limited program with a special Prove-mode. In Eval-mode, the program behaves exactly like the underlying RAM program P . In Prove-mode, it outputs the current

cost together with a signature certifying that cost, and then irreversibly switches to a distinguished halt state. Intuitively, once a valid proof is produced, no additional effective evaluations should be possible.

Definition B.1 (Pay-Per-Use Program Compiler). Let $\mathcal{P} = \{P : \mathcal{M} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{M} \times \mathcal{Y}\}_P$ be a family of RAM programs whose outputs include a designated cumulative cost value $\text{cost}(y) \in \mathbb{R}$, initialized to 0 when the program starts from $\text{st}_0 = \perp$.

A **pay-per-use (PPU) program compiler** for $\mathcal{P}$ comprises the following QPT algorithms.

- **Syntax.**

- $\text{PGen}(1^\lambda, P) \rightarrow (\tilde{P}, \text{pk})$: takes a security parameter λ and a program $P \in \mathcal{P}$, and outputs a compiled program $\tilde{P}$ together with a public verification key pk .
- $\text{Eval}(\tilde{P}, x) \rightarrow (y, \tilde{P}')$: evaluates $\tilde{P}$ on input x and returns an output y and an updated program state $\tilde{P}'$.
- $\text{Prove}(\tilde{P}, c, \text{pk}) \rightarrow b \in \{0, 1\}$: a QPT interactive protocol between a prover and a verifier. The prover's inputs are a program $\tilde{P}$ and a claimed cost c . The verifier's inputs are c and pk . The verifier outputs 1 iff it accepts that the effective queries encoded by $\tilde{P}$ have total cost c .

- **Correctness.** The compiler satisfies the correctness property of definition 5.1 with respect to the underlying family $\mathcal{P}$.
- **Completeness.** For every $P \in \mathcal{P}$, every polynomially bounded $n \geq 0$, and every input sequence $(x_1, \dots, x_n) \in \mathcal{X}^n$, the following experiment accepts with probability at least $1 - \text{negl}(\lambda)$:

$$\Pr \left[\begin{array}{l} (\tilde{P}_0, \text{pk}) \leftarrow \text{PGen}(1^\lambda, P), \\ (y_i, \tilde{P}_i) \leftarrow \text{Eval}(\tilde{P}_{i-1}, x_i) \quad \text{for } i = 1, \dots, n, \\ b \leftarrow \text{Prove}(\tilde{P}_n, c, \text{pk}) \end{array} \right] \geq 1 - \text{negl}(\lambda),$$

where both parties follow the protocol honestly and the claimed cost is computed before running Prove as

$$c := \begin{cases} 0, & n = 0, \\ \text{cost}(y_n), & n \geq 1. \end{cases}$$

Any failure to produce a well-formed output in this experiment is counted as rejection. The value c is the cumulative cost of the preceding evaluations; running Prove does not add another evaluation of P to this cost. Both parties obtain c after the protocol, so its value is not read from the program state after proving.

- **Security.** There exists a QPT simulator $\mathcal{S}$ such that for every $P \in \mathcal{P}$ and every non-uniform QPT distinguisher D with auxiliary input $\text{aux}_{P, \mathcal{P}}$, there is a negligible function negl such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[D(1^\lambda, \text{PGen}(1^\lambda, P), \text{aux}_{P, \mathcal{P}}) \rightarrow 1 \right] - \Pr \left[D(1^\lambda, \mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda), \text{aux}_{P, \mathcal{P}}) \rightarrow 1 \right] \right| \leq \text{negl}(\lambda),$$

where $\mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda)$ outputs both the simulated program and the public key, and O_P^{LEQ} is the REQ oracle for P from fig. 6.

- **Soundness.** For a fixed $P \in \mathcal{P}$, let $\mathcal{S}^{O_P^{\text{LEQ}}}$ be the simulator from the security item above. Consider the following experiment.

1. The challenger sends $(\tilde{P}, \text{pk}) \leftarrow \mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda)$ to the adversary.
2. The adversary outputs a claimed cost $c \in \mathbb{R}$.
3. The adversary and the challenger engage in $\text{Prove}(\tilde{P}, c, \text{pk})$, with the adversary acting as prover and the challenger as verifier. Let b be the verifier's output.
4. The challenger reads the true effective cost c^* from the internal state of O_P^{LEQ} . Concretely, if there is no index k with $g_k = 1$, then $c^* := 0$. Otherwise, let $k_{\max}$ be the maximum index with $g_{k_{\max}} = 1$, read (st, x, r) from register $\mathcal{A}_{k_{\max}}$, compute $(*, y) = P(\text{st}, x; r)$, and set $c^* := \text{cost}(y)$.
5. The adversary wins if $b = 1$ and $c \neq c^*$.

The compiler is *sound* if every QPT adversary wins with at most negligible probability.

Construction. Let $\text{QLP}(\text{PGen}, \text{Eval})$ be the QLP compiler from fig. 8. Let $\text{BUS}(\text{KeyGen}, \text{Sign}, \text{Ver})$ be a blind-unforgeable signature scheme, and let PRF be a pseudorandom function (definition 3.1).

The compiler samples a signature key pair (sk, pk) and two PRF keys k_1, k_2 . It then defines the following two-mode RAM program P_{PPU} .

Theorem B.1 (PPU Program Compiler). The construction in figs. 14 and 15 is a pay-per-use program compiler for $\mathcal{P}$ in the sense of definition B.1.

Proof. The syntax is immediate. Correctness is proved in lemma B.1, security in lemma B.3, and soundness in lemma B.5. $\square$

Lemma B.1 (PPU Program Correctness). The construction in figs. 14 and 15 satisfies the correctness property of definition B.1.

Proof. The compiled program is simply a QLP for the two-mode RAM program P_{PPU} :

$$\tilde{P} \leftarrow \text{QLP.PGen}(1^\lambda, P_{\text{PPU}}).$$

Moreover, $\text{Eval}(\tilde{P}, x)$ invokes that QLP only on inputs of the form (x, Eval) . On such inputs, P_{PPU} behaves exactly like the underlying program P .

Therefore, by the correctness guarantee of the QLP compiler, for every polynomial-length query sequence $(x_1, \dots, x_n) \in \mathcal{X}^*$, the distributions

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} \text{st}_0 = \perp, \\ (r_1, \dots, r_n) \leftarrow \mathcal{R}^n, \\ (\text{st}_i, y_i) \leftarrow P(\text{st}_{i-1}, x_i; r_i) \quad \forall i \in [n] \end{array} \right\}$$

and $\left\{ (y_1, \dots, y_n) : \begin{array}{l} (\tilde{P}_0, \text{pk}) \leftarrow \text{PGen}(1^\lambda, P), \\ (y_i, \tilde{P}_i) \leftarrow \text{Eval}(\tilde{P}_{i-1}, x_i) \quad \forall i \in [n] \end{array} \right\}$

are statistically close. This is exactly the required correctness statement. $\square$

Lemma B.2 (PPU Program Completeness). The construction in figs. 14 and 15 satisfies the completeness property of definition B.1.

Proof. Fix $P \in \mathcal{P}$ and a polynomial-length input sequence $(x_1, \dots, x_n)$, allowing $n = 0$. The honest execution consists of n evaluations followed by one execution of Prove . By the construction, these operations invoke the same QLP for P_{PPU} on the sequence

$$(x_1, \text{Eval}), \dots, (x_n, \text{Eval}), (0, \text{Prove}).$$

$\text{PGen}(1^\lambda, P)$:

1. Sample

$$\begin{aligned} (\text{sk}, \text{pk}) &\leftarrow \text{BUS.KeyGen}(1^\lambda), \\ k_1 &\stackrel{\$}{\leftarrow} \{0, 1\}^\lambda, \\ k_2 &\stackrel{\$}{\leftarrow} \{0, 1\}^\lambda. \end{aligned} \tag{16}$$

2. Using $(P, \text{sk}, \text{pk}, k_1, k_2)$, define the RAM program P_{PPU} :

Inputs: $[\text{st}, (x, \text{mode}); r]$, where $\text{st} \in \mathcal{M}$ is the current RAM state, $x \in \mathcal{X}$ is the chosen input, $\text{mode} \in \{\text{Eval}, \text{Prove}\}$, and $r \in \mathcal{R}$ is fresh randomness.

(a) If $\text{st} = \text{Halt}$, output $(\text{st}', y) = (\text{Halt}, 0)$.

(b) If $\text{mode} = \text{Eval}$ and $\text{st} \neq \text{Halt}$, output

$$(\text{st}', y) = P(\text{st}, x; r).$$

(c) If $\text{mode} = \text{Prove}$ and $\text{st} \neq \text{Halt}$, let $c' := \text{cost}(\text{st})$, set $\text{st}' := \text{Halt}$, and compute

$$\begin{aligned} r_1 &= \text{PRF}(k_1, (c', r)), \\ r_2 &= \text{PRF}(k_2, (c', r, r_1)), \\ \sigma &= \text{BUS.Sign}(\text{sk}, (c', r, r_1); r_2), \\ y &= (c', r, r_1, \sigma). \end{aligned} \tag{17}$$

Then output (st', y) .

3. Generate a QLP for P_{PPU} :

$$\tilde{P} \leftarrow \text{QLP.PGen}(1^\lambda, P_{\text{PPU}}).$$

4. Output $(\tilde{P}, \text{pk})$.

$\text{Eval}(\tilde{P}, x)$:

1. Query the QLP in Eval-mode:

$$\begin{aligned} \text{mode} &= \text{Eval}, \\ (y, \tilde{P}') &\leftarrow \text{QLP.Eval}(\tilde{P}, (x, \text{mode})). \end{aligned}$$

2. Output $(y, \tilde{P}')$.

Figure 14: Pay-Per-Use Program Compiler Construction, Part 1

Prove($\tilde{P}, c, \text{pk}$):

1. **Input:** The prover has a QLP $\tilde{P}$ and a claimed cost c . The verifier has the same cost c and the public key pk .
2. The prover queries the QLP in Prove-mode:

$$(x, \text{mode}) = (0, \text{Prove}),$$

$$(y, \tilde{P}') \leftarrow \text{QLP.Eval}(\tilde{P}, (x, \text{mode})).$$

The prover sends y to the verifier.

3. The verifier parses $y = (c', r, r_1, \sigma)$ and outputs

$$b \leftarrow (c' = c) \wedge (1 = \text{BUS.Ver}(\text{pk}, (c', r, r_1), \sigma)).$$

Figure 15: Pay-Per-Use Program Compiler Construction, Part 2

In particular, the final Prove uses the remaining compiled program $\tilde{P}_n$, not a newly generated program.

The correctness guarantee of the QLP compiler (definition 5.1) applies to this entire sequence of $n + 1$ queries. Denote the output of the last query by y . The joint distribution of $(\text{pk}, y_1, \dots, y_n, y)$ is statistically close to the corresponding distribution obtained by directly evaluating P_{PPU} with independent uniform randomness at each step. pk is obtained by first fixing the parameters sampled in (16), applying QLP correctness to the resulting program P_{PPU} averaged over these parameters.

Consider the following execution. The first n queries are all in Eval-mode, so they evaluate the underlying RAM program P starting from $\text{st}_0 = \perp$. Let st_n be its state after these evaluations. By the designated cumulative-cost field assumed for P , the claimed cost satisfies

$$c = \text{cost}(\text{st}_n) = \begin{cases} 0, & n = 0, \\ \text{cost}(y_n), & n \geq 1. \end{cases}$$

Since no preceding query used Prove-mode, the final query executes the prove-branch in fig. 14, reads $c' := \text{cost}(\text{st}_n)$, and produces

$$\begin{aligned} r_1 &= \text{PRF}(k_1, (c', r)), \\ r_2 &= \text{PRF}(k_2, (c', r, r_1)), \\ \sigma &= \text{BUS.Sign}(\text{sk}, (c', r, r_1); r_2), \\ y &= (c', r, r_1, \sigma). \end{aligned}$$

Here r is the randomness of the final query. In particular, $c' = c$. By perfect correctness of the signature scheme (definition 3.2),

$$\text{BUS.Ver}(\text{pk}, (c', r, r_1), \sigma) = 1.$$

Perfect correctness applies to the specified signing coins r_2 as well. Therefore, the verifier in fig. 15 accepts this direct-execution transcript with probability one.

The claimed cost c is a function of the preceding outputs (or the constant 0 when $n = 0$), and the verifier's decision is a function of (pk, c, y) . Applying these functions to the two statistically close joint output distributions cannot increase their statistical distance. Consequently, the verifier accepts the honest compiled execution with probability at least $1 - \text{negl}(\lambda)$.

When $n = 0$, the only query is $(0, \text{Prove})$, which reads the initial cost $\text{cost}(\perp) = 0$ and signs it, so the same argument applies to this case. The prove-branch subsequently sets the state to Halt , but the certified cost remains in the output y and in the parties' remaining value c . $\square$

Lemma B.3 (PPU Program Security). The construction in figs. 14 and 15 satisfies the security property of definition B.1.

Proof. Let P_{PPU} be the two-mode RAM program defined in fig. 14. The real compiler outputs

$$(\tilde{P}, \text{pk}) = (\text{QLP.PGen}(1^\lambda, P_{\text{PPU}}), \text{pk}).$$

By the security of the QLP compiler (definition 5.1), there exists a QPT simulator $\mathcal{S}'$ such that

$$\left| \Pr \left[D \left(1^\lambda, \text{QLP.PGen}(1^\lambda, P_{\text{PPU}}), \text{aux}_{P, \mathcal{P}} \right) \rightarrow 1 \right] - \Pr \left[D \left(1^\lambda, \mathcal{S}'^{O_{P_{\text{PPU}}}^{\text{LEQ}}}(1^\lambda), \text{aux}_{P, \mathcal{P}} \right) \rightarrow 1 \right] \right| \leq \text{negl}(\lambda). \quad (18)$$

Thus it remains to simulate $O_{P_{\text{PPU}}}^{\text{LEQ}}$ given only O_P^{LEQ} .

Lemma B.4. For every $P \in \mathcal{P}$, there exists a QPT oracle simulator $\mathcal{S}_{P \rightarrow P_{\text{PPU}}}$ with quantum query access to O_P^{LEQ} that perfectly simulates $O_{P_{\text{PPU}}}^{\text{LEQ}}$. Moreover, after every query, the cost register maintained by $\mathcal{S}_{P \rightarrow P_{\text{PPU}}}$ equals the cumulative cost of the current effective state recorded by O_P^{LEQ} (or 0 if no effective evaluation has yet occurred).

Proof. The simulator follows the REQ procedure of fig. 6, with its own bookkeeping registers $\mathcal{A}', \mathcal{G}', \mathcal{B}'$, and augments them with a local state register $\mathcal{S}'$ storing a pair (c, h) , where c is the current cumulative cost and $h \in \{0, 1\}$ indicates whether the program is halted. Initially $\mathcal{S}'$ stores $(0, 0)$.

The only nontrivial task is to implement the REQ subroutine $U_{P_{\text{PPU}}}$.

- In Eval-mode and with $h = 0$, the program P_{PPU} simply evaluates P . The simulator therefore delegates this branch to its oracle access to O_P^{LEQ} on the same tag and input x , obtains the output y , XORs y into the outer query register exactly as $U_{P_{\text{PPU}}}$ would, and updates the cost component of $\mathcal{S}'$ to $\text{cost}(y)$.
- In Prove-mode and with $h = 0$, the hidden state of P is not needed. The simulator computes the output locally by coherently sampling $r \in \mathcal{R}$, setting $r_1 = \text{PRF}(k_1, (c, r))$, $r_2 = \text{PRF}(k_2, (c, r, r_1))$, and $\sigma = \text{BUS.Sign}(\text{sk}, (c, r, r_1); r_2)$, XORing (c, r, r_1, σ) into the output register, and flipping the halt bit to $h = 1$.
- If $h = 1$, then P_{PPU} always outputs $(\text{Halt}, 0)$, so the simulator can implement this branch locally as well.

On every computational-basis branch of the interaction, these are exactly the three branches of P_{PPU} . Hence the simulator applies the same unitary transformation as $O_{P_{\text{PPU}}}^{\text{LEQ}}$, including the inverse transformation on repeated queries. By linearity, the induced channel on arbitrary superpositions is identical, so the simulation is perfect.

The invariant on the cost register follows by induction over the sequence of effective queries. Initially both costs are 0. Each effective Eval-query updates the simulator's cost register to $\text{cost}(y)$, which is precisely the cost stored by the next effective state of P in O_P^{LEQ} . A Prove-query does not change the effective state of P and only freezes further evolution, so the invariant continues to hold. $\square$

Now define the final simulator $\mathcal{S}$ by composition:

$$\mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda) := (\mathcal{S}'^{\mathcal{S}_{P \rightarrow P_{\text{PPU}}}}(1^\lambda), \text{pk}),$$

where pk is the public key sampled by $\mathcal{S}_{P \rightarrow P_{\text{PPU}}}$. By perfect emulation of $O_{P_{\text{PPU}}}^{\text{LEQ}}$ we have

$$\Pr\left[D\left(1^\lambda, \mathcal{S}'_{O_{P_{\text{PPU}}}^{\text{LEQ}}}(1^\lambda), \text{aux}_{P, \mathcal{P}}\right) \rightarrow 1\right] = \Pr\left[D\left(1^\lambda, \mathcal{S}_P^{\text{LEQ}}(1^\lambda), \text{aux}_{P, \mathcal{P}}\right) \rightarrow 1\right]. \quad (19)$$

Combining (18) and (19) proves the claim. $\square$

Lemma B.5 (PPU Program Soundness). The construction in figs. 14 and 15 satisfies the soundness property of definition B.1.

Proof. Let A be any QPT adversary in the soundness game. By lemma B.4, the game in definition B.1 is perfectly equivalent to a game in which the prover interacts with $\mathcal{S}'_{O_{P_{\text{PPU}}}^{\text{LEQ}}}$ instead of $\mathcal{S}_P^{\text{LEQ}}$, and the challenger checks the true cost using the cost register of the simulated oracle $O_{P_{\text{PPU}}}^{\text{LEQ}}$. Thus we may treat the composition of A with $\mathcal{S}'$ as a single QPT oracle algorithm $\hat{A}$ interacting with $O_{P_{\text{PPU}}}^{\text{LEQ}}$.

We now modify the prove-branch of $O_{P_{\text{PPU}}}^{\text{LEQ}}$ through a sequence of hybrids.

Hybrid 0. This is the game above, i.e. $\hat{A}$ interacts with $O_{P_{\text{PPU}}}^{\text{LEQ}}$, where the prove-branch uses the PRFs $\text{PRF}(k_1, \cdot)$ and $\text{PRF}(k_2, \cdot)$.

Hybrid 1. Replace $\text{PRF}(k_2, \cdot)$ in the prove-branch with a truly random oracle H_2 . Standard PRF security implies that Hybrids 0 and 1 are computationally indistinguishable.

Hybrid 2. Let

$$B := \{(c, r, r_1) : r_1 \neq \text{PRF}(k_1, (c, r))\}.$$

Replace the signing procedure in the prove-branch with a *memoized blinded signing oracle* $\widehat{\text{Sign}}_B$. On a query $m = (c, r, r_1)$, the oracle behaves as follows: if $m \in B$ it outputs $\perp$; if $m \notin B$ and m has been seen before, it returns the memoized signature for m ; otherwise it queries the blind signing oracle $\text{Sign}_B(\text{sk}, m)$ once, stores the resulting signature, and returns it. Since the prove-branch of P_{PPU} only ever queries the signing procedure on messages with $r_1 = \text{PRF}(k_1, (c, r))$, the blinded branch is never taken, and Hybrids 1 and 2 are perfectly indistinguishable.

Hybrid 3. Replace $\text{PRF}(k_1, \cdot)$ with a truly random oracle H_1 both in the generation of r_1 and in the definition of the blinding set:

$$B := \{(c, r, r_1) : r_1 \neq H_1(c, r)\}.$$

Because the prove-branch and the blinding test access $\text{PRF}(k_1, \cdot)$ only as an oracle, Hybrids 2 and 3 are computationally indistinguishable by PRF security.

Hybrid 4. Implement H_1 as a compressed oracle rather than as a lazy-sampled random oracle. This does not change the adversary's view.

Fix Hybrid 4. Let E_{succ} be the event that the verifier accepts a proof for a claimed cost c while the true effective cost at the end of the game is some $c^\star \neq c$. We show that $\Pr[E_{\text{succ}}]$ is negligible.

Let the verifier-accepted message be $m = (c, r, r_1)$ with signature σ . There are two cases.

Case 1: $m \in B$, i.e. $r_1 \neq H_1(c, r)$. In this case, the signature (m, σ) is valid (because the verifier accepted), yet m lies in the blinding set of the oracle $\widehat{\text{Sign}}_B$. By construction, $\widehat{\text{Sign}}_B$ is implemented using the blind signing oracle $\text{Sign}_B(\text{sk}, \cdot)$ together with local memorization. Therefore a successful prover in this case yields a successful blind-unforgeability adversary for BUS. Hence this case occurs with negligible probability.

Case 2: $m \notin B$, i.e. $r_1 = H_1(c, r)$. Since Hybrid 4 implements H_1 as a compressed oracle, we may apply lemma 3.4 with $k = 1$. It follows that, except with negligible probability, whenever $\hat{A}$ outputs

a tuple (c, r, r_1) satisfying $r_1 = H_1(c, r)$, the compressed database of H_1 contains the recorded entry $((c, r), r_1)$ at the end of the experiment.

But H_1 is queried *only* inside the Prove-branch of P_{PPU} . Consequently, a recorded entry for (c, r) implies that a Prove-query with randomness r remains effective at the end of the interaction. By the definition of P_{PPU} , that query reads the current cost c from the program state, outputs a signature on $m = (c, r, r_1)$, and sets the program state to Halt. Once this happens, every later effective query leaves the program in the halt state and cannot change the cost. Therefore the final true effective cost must equal c , i.e. $c^* = c$, contradicting the event E_{succ} .

We have shown that both cases occur only with negligible probability in Hybrid 4, so the same is true in Hybrid 4 itself. Undoing the hybrids changes the success probability by at most a negligible amount. Hence the soundness error in the original game is negligible. $\square$

B.2 Semi-Quantum Query-Limited Programs

We construct query-limited programs that can be sent over classical channels. This is possible because in our construction, the power to evaluate the program is conferred by a one-shot program token. As with one-shot signatures, the power conferred by this token can be transferred over classical channels by using the token to sign the serial number of a new token.

Definition. A program is *semi-quantum* if the ability to evaluate the program can be transferred from a sender to a receiver using only classical communication. In definition B.2 below, we augment the definition of a QLP compiler to include an interactive protocol Transfer that allows control of a program to be transferred using classical communication.

Our correctness property is similar to the one for regular QLPs (definition 5.1) except that the program evaluations can be interleaved with program transfers. Recall that an honestly generated QLP

$$\tilde{P} \leftarrow \text{PGen}(1^\lambda, P)$$

allows the user to make a sequence of evaluations (executions of $\text{Eval}(\tilde{P}, x)$), and the correctness property of definition 5.1 ensures that $\tilde{P}$'s responses will be statistically close to the responses that P would have given. For semi-quantum QLPs, we allow the user to interleave the program evaluations with program transfers (executions of Transfer), and we require that correctness of the evaluations still holds in this setting.

Definition B.2 (Semi-Quantum Query-Limited Program Compiler). Let $\mathcal{P} = \{P : \mathcal{M} \times \mathcal{X} \times \mathcal{R} \rightarrow \mathcal{M} \times \mathcal{Y}\}_P$ be a family of RAM programs. A **semi-quantum query-limited program compiler** for $\mathcal{P}$ is a query-limited program compiler for $\mathcal{P}$ (definition 5.1) with the following additional properties.

- **Syntax:** In addition to PGen and Eval from definition 5.1, the compiler also includes the following protocol:
 - **Transfer:** An interactive protocol between a QPT sender S and a QPT receiver R . S takes as input a program $\tilde{P}$. Next, the parties may only send classical messages. If we are working in an oracle model, then the parties may also send access to classical oracles. Finally, as output R has a new program $\tilde{P}'$. We notate this protocol as $\text{Transfer}(\tilde{P}) \rightarrow \tilde{P}'$.
- **Correctness:** Let an operation sequence of length n be a sequence $[(\text{mode}_i, x_i)]_{i \in [n]}$, where for each $i \in [n]$, $\text{mode}_i \in \{\text{Eval}, \text{Transfer}\}$ and $x_i \in \mathcal{X}$.

The compiler satisfies correctness if for every $P \in \mathcal{P}$ and every polynomial function $n(\lambda) = \text{poly}(\lambda)$, there exists a negligible function $\text{negl}(\lambda)$, such that for every $\lambda \in \mathbb{N}$ and every opera-

tion sequence $[(\text{mode}_i, x_i)]_{i \in [n]}$ of length $n = n(\lambda)$, the ideal-world distribution

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} (r_1, \dots, r_n) \leftarrow \mathcal{R}^n \\ \text{st}_0 = \perp \\ (\text{st}_i, y_i) \leftarrow \begin{cases} P(\text{st}_{i-1}, x_i; r_i), & \text{mode}_i = \text{Eval} \\ (\text{st}_{i-1}, \perp), & \text{mode}_i = \text{Transfer} \end{cases}, \forall i \in [n] \end{array} \right\}$$

and the real-world distribution

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} \tilde{P}_0 \leftarrow \text{PGen}(1^\lambda, P) \\ (y_i, \tilde{P}_i) \leftarrow \begin{cases} \text{Eval}(\tilde{P}_{i-1}, x_i), & \text{mode}_i = \text{Eval} \\ (\perp, \text{Transfer}(\tilde{P}_{i-1})), & \text{mode}_i = \text{Transfer} \end{cases}, \forall i \in [n] \end{array} \right\}$$

are $\text{negl}(\lambda)$ -close in statistical distance.

- **Security** The compiler essentially satisfies the security property of definition 5.1. We will view all the senders and receivers involved in the protocol as one colluding adversary. Let the quantum distinguisher D be a non-uniform quantum algorithm that can be divided into the following stages $((S_1, R_1), (S_2, R_2), \dots, (S_L, R_L), D')$ for an arbitrary polynomial $L = L(\lambda)$, where each (S_i, R_i) consists of possibly several evaluations and then a sender-receiver Transfer protocol. D' is a final distinguisher that outputs a bit.

The scheme satisfies LEQ security if there exists a QPT simulator $\mathcal{S}$ such that for any semi-quantum query-limited program $P \in \mathcal{P}$ and any distinguisher D with (quantum) auxiliary input $\text{aux}_{P, \mathcal{P}}$ (which may depend non-uniformly on P and $\mathcal{P}$), there exists a negligible function $\text{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr [D(1^\lambda, \text{PGen}(1^\lambda, P), \text{aux}_{P, \mathcal{P}}) \rightarrow 1] - \Pr [D(1^\lambda, \mathcal{S}_P^{\text{LEQ}}(1^\lambda), \text{aux}_{P, \mathcal{P}}) \rightarrow 1] \right| \leq \text{negl}(\lambda)$$

Construction. To construct semi-quantum QLPs, we will start with the construction of (regular) QLPs given in fig. 8 (our construction is not black-boxy from any QLPs, but a simple tweak of our QLP construction). The program $\tilde{P}$ comprises $(\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}, r_s, \sigma), s, |T\rangle)$. The values $(\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}, r_s, \sigma), s)$ are classical strings, which can be sent over classical channels. (Recall that one-shot programs, such as $\tilde{P}_{\text{OSP}}$, are classical programs.) Only $|T\rangle$, the one-shot program token, is a quantum state and cannot be sent over classical channels.

Next, to transfer the program, we will make a dummy evaluation of the program. To do so, we evaluate the program in Transfer mode, in which the program outputs $y = \perp$ on every input and leaves the RAM state st unchanged. The point of running Eval in Transfer mode is to consume the program token (held by the sender) and generate a new token (held by the receiver) that can be used on the next query.

Eval includes the following steps.

1. Sample $(s', |T'\rangle) \leftarrow \text{OSP.TokenGen}(\text{crs})$.
2. Evaluate $(y, (\text{st}', r_{s'}, \sigma')) \leftarrow \text{OSP.Eval}(\tilde{P}_{\text{OSP}}, s, (x, \text{st}, r_s), \sigma, s', |T\rangle)$.
3. Update $\tilde{P}$ to $(\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}', r_{s'}, \sigma'), |T'\rangle)$.

Let us have the receiver execute step 1 to generate the new program token. Then the receiver sends s' to the sender. Next, the sender executes step 2 to evaluate the program. This step also generates a signature σ' on s' , which ensures that $|T'\rangle$ can be used to make the next query. Then the sender sends $(y, (\text{st}', r_{s'}, \sigma'))$ to the receiver, and the receiver executes step 3.

At the end of this protocol, the parties have evaluated the program in Transfer mode, which does not change the RAM state. Additionally, the sender's program token has been consumed, and the receiver's program token now has the power to evaluate the program. In this way, the power to evaluate the program has been transferred from the sender to the receiver.

Now, fig. 16 describes the construction of a semi-quantum QLP compiler in detail. Let $\text{QLP}.\text{(PGen, Eval)}$ be the QLP compiler construction from fig. 8. Let $\text{OSP}.\text{(KeyGen, TokenGen, PGen, Eval)}$ be a one-shot program compiler (definition 4.2).

Theorem B.2. The construction in fig. 16 is a semi-quantum query-limited program compiler for $\mathcal{P}$.

Proof.

Syntax. It is clear by inspection that the construction has the syntax of a semi-quantum QLP.

Correctness. First, let $[(\text{mode}_i, x_i)]_{i \in [n]}$ be a given operation sequence. It suffices to consider the case where $x_i = 0$ if $\text{mode}_i = \text{Transfer}$, for all $i \in [n]$. This is because when $\text{mode}_i = \text{Transfer}$, the value of x_i does not influence the output of the real- or ideal-world distributions, and $y_i = \perp$.

Second, the program $\tilde{P}$ output by PGen is a regular QLP for the functionality P' :

$$\tilde{P} \leftarrow \text{QLP.PGen}(1^\lambda, P')$$

$\text{Eval}(\tilde{P}, x)$ simply evaluates the program with $\text{mode} = \text{Eval}$:

$$(y, \tilde{P}') \leftarrow \text{QLP.Eval}(\tilde{P}, (x, \text{mode} = \text{Eval}))$$

Likewise, $\text{Transfer}(\tilde{P})$ is equivalent to evaluating the program with $\text{mode} = \text{Transfer}$ and $x = 0$. That is to say: if we treat the sender and receiver as one joint evaluator, then $\text{Transfer}(\tilde{P})$ is equivalent to this evaluator computing

$$(y, \tilde{P}') \leftarrow \text{QLP.Eval}(\tilde{P}, (x = 0, \text{mode} = \text{Transfer}))$$

and outputting $\tilde{P}'$. To see why, note that the Transfer protocol follows the same steps as QLP.Eval in fig. 8.

Third, the correctness of QLP (definition 5.1) implies that for any operation sequence $[(\text{mode}_i, x_i)]_{i \in [n]}$, the following distributions are statistically close:

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} (r_1, \dots, r_n) \leftarrow \mathcal{R}^n \\ \text{st}_0 = \perp \\ (\text{st}_i, y_i) \leftarrow P'(\text{st}_{i-1}, (x_i, \text{mode}_i); r_i), \forall i \in [n] \end{array} \right\} \quad (20)$$

and

$$\left\{ (y_1, \dots, y_n) : \begin{array}{l} \tilde{P}_0 \leftarrow \text{PGen}(1^\lambda, P) \\ (y_i, \tilde{P}_i) \leftarrow \text{QLP.Eval}(\tilde{P}_{i-1}, (x_i, \text{mode}_i)), \forall i \in [n] \end{array} \right\} \quad (21)$$

Fourth, eq. (20) is equivalent to the ideal-world distribution in definition B.2's correctness notion:

$$\begin{aligned} & \left\{ (y_1, \dots, y_n) : \begin{array}{l} (r_1, \dots, r_n) \leftarrow \mathcal{R}^n \\ \text{st}_0 = \perp \\ (\text{st}_i, y_i) \leftarrow P'(\text{st}_{i-1}, (x_i, \text{mode}_i); r_i), \forall i \in [n] \end{array} \right\} \\ &= \left\{ (y_1, \dots, y_n) : \begin{array}{l} (r_1, \dots, r_n) \leftarrow \mathcal{R}^n \\ \text{st}_0 = \perp \\ (\text{st}_i, y_i) \leftarrow \begin{cases} P(\text{st}_{i-1}, x_i; r_i), & \text{mode}_i = \text{Eval} \\ (\text{st}_{i-1}, \perp), & \text{mode}_i = \text{Transfer} \end{cases}, \forall i \in [n] \end{array} \right\} \end{aligned}$$

$\text{PGen}(1^\lambda, P)$:

1. Use P to construct the following RAM program P' :

Inputs: $[\text{st}, (x, \text{mode}); r]$. $\text{st} \in \mathcal{M}$ is the current program state, $x \in \mathcal{X}$ is the chosen input, $\text{mode} \in \{\text{Eval}, \text{Transfer}\}$ is the mode to run the program in, and $r \in \mathcal{R}$ is the sampled randomness.

- (a) If $\text{mode} = \text{Eval}$, then compute and output $(\text{st}', y) = P(\text{st}, x; r)$.
- (b) If $\text{mode} = \text{Transfer}$, then output $(\text{st}', y) = (\text{st}, \perp)$.

2. Generate a QLP for P' : $\tilde{P} \leftarrow \text{QLP.PGen}(1^\lambda, P')$. Here, QLP.PGen is the algorithms described in Figure 8.
3. Output $\tilde{P}$.

$\text{Eval}(\tilde{P}, x)$:

1. Query the QLP in Eval mode:

$$\begin{aligned} \text{mode} &= \text{Eval} \\ (y, \tilde{P}') &\leftarrow \text{QLP.Eval}(\tilde{P}, (x, \text{mode})) \end{aligned}$$

2. Output $(y, \tilde{P}')$.

$\text{Transfer}(\tilde{P})$:

1. **Input:** The sender S has a QLP $\tilde{P}$, which is parsed as $\tilde{P} = (\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}, r_s, \sigma), s, |T\rangle)$. The receiver R has no inputs.

2. S sends crs to R .

3. R generates a token

$$(s', |T'\rangle) \leftarrow \text{OSP.TokenGen}(\text{crs})$$

and sends s' to S .

4. S evaluates the program as follows:

$$\begin{aligned} (x, \text{mode}) &= (0, \text{Transfer}) \\ (y, (\text{st}', r_{s'}, \sigma')) &\leftarrow \text{QLP.Eval}(\tilde{P}_{\text{OSP}}, s, [(x, \text{mode}), \text{st}, r_s], \sigma, s'), |T\rangle) \end{aligned}$$

5. S sends $(\tilde{P}_{\text{OSP}}, \text{st}', r_{s'}, \sigma')$ to R .

6. R updates $\tilde{P}$ to

$$\tilde{P}' \leftarrow (\text{crs}, \tilde{P}_{\text{OSP}}, (\text{st}', r_{s'}, \sigma'), |T'\rangle).$$

R 's output is $\tilde{P}'$.

Figure 16: Construction of a Semi-Quantum QLP Compiler for $\mathcal{P}$

This follows from the definition of P' .

Fifth, eq. (21) is statistically close to the real-world distribution in definition B.2's correctness notion:

$$\begin{aligned}
& \left\{ (y_1, \dots, y_n) : \begin{array}{l} \tilde{P}_0 \leftarrow \text{PGen}(1^\lambda, P) \\ (y_i, \tilde{P}_i) \leftarrow \text{QLP.Eval}(\tilde{P}_{i-1}, (x_i, \text{mode}_i)), \forall i \in [n] \end{array} \right\} \\
& \approx_s \left\{ (y_1, \dots, y_n) : \begin{array}{l} \tilde{P}_0 \leftarrow \text{PGen}(1^\lambda, P) \\ (\tilde{y}_i, \tilde{P}_i) \leftarrow \text{QLP.Eval}(\tilde{P}_{i-1}, (x_i, \text{mode}_i)), \forall i \in [n] \\ y_i = \begin{cases} \tilde{y}_i, & \text{mode}_i = \text{Eval} \\ \perp, & \text{mode}_i = \text{Transfer} \end{cases}, \forall i \in [n] \end{array} \right\} \\
& = \left\{ (y_1, \dots, y_n) : \begin{array}{l} \tilde{P}_0 \leftarrow \text{PGen}(1^\lambda, P) \\ (y_i, \tilde{P}_i) \leftarrow \begin{cases} \text{Eval}(\tilde{P}_{i-1}, x_i), & \text{mode}_i = \text{Eval} \\ (\perp, \text{Transfer}(\tilde{P}_{i-1})), & \text{mode}_i = \text{Transfer} \end{cases}, \forall i \in [n] \end{array} \right\}
\end{aligned}$$

The second line describes the case where we evaluate $\text{QLP.Eval}(\tilde{P}_{i-1}, (x_i, \text{mode}_i))$, and if $\text{mode}_i = \text{Transfer}$, we overwrite $y_i \leftarrow \perp$. The first and second lines are statistically close because when $\text{mode}_i = \text{Transfer}$, $\text{QLP.Eval}(\tilde{P}_{i-1}, (x_i, \text{mode}_i))$ produces $y = \perp$ with overwhelming probability, due to the correctness of QLP.

In total, we have shown that the real- and ideal-world distributions in definition B.2's correctness notion are statistically close for any operation sequence. This completes the proof of correctness.

Security. The security notion of definition B.2 is the same as the one for query-limited RAM programs (definition 5.1). Furthermore, the program $\tilde{P}$ output by PGen is a regular QLP for the functionality P' :

$$\tilde{P} \leftarrow \text{QLP.PGen}(1^\lambda, P')$$

Then the security of QLP (definition 5.1) implies the following: there exists a QPT simulator $\mathcal{S}'$ such that for any randomized RAM program $P \in \mathcal{P}$ and any distinguisher D with (quantum) auxiliary input $\text{aux}_{P, \mathcal{P}}$ (which may depend non-uniformly on P and $\mathcal{P}$), there exists a negligible function $\text{negl}(\lambda)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr \left[D(1^\lambda, \text{PGen}(1^\lambda, P), \text{aux}_{P, \mathcal{P}}) \rightarrow 1 \right] - \Pr \left[D \left(1^\lambda, \mathcal{S}'^{O_{P'}^{\text{LEQ}}}(1^\lambda), \text{aux}_{P, \mathcal{P}} \right) \rightarrow 1 \right] \right| \leq \text{negl}(\lambda)$$

This is very close to what we have to prove. The only difference is that the simulator $\mathcal{S}'$ has access to $O_{P'}^{\text{LEQ}}$, but we need a simulator simulates $\text{PGen}(1^\lambda, P)$ with access to O_P^{LEQ} .

Lemma B.6 proves that there exists a simulator $\mathcal{S}_{P \rightarrow P'}$ that receives oracle access to O_P^{LEQ} and perfectly simulates the oracle $O_{P'}^{\text{LEQ}}$. Then we can compose $\mathcal{S}'$ with $\mathcal{S}_{P \rightarrow P'}$ to obtain a simulator $\mathcal{S}$ that simulates $\text{PGen}(1^\lambda, P)$ given access to O_P^{LEQ} . $\square$

Lemma B.6. There exists a simulator $\mathcal{S}_{P \rightarrow P'}$ that receives oracle access to O_P^{LEQ} and perfectly simulates the oracle $O_{P'}^{\text{LEQ}}$.

Proof. First, let us construct of $\mathcal{S}_{P \rightarrow P'}$.

At a high level, $\mathcal{S}_{P \rightarrow P'}$ simulates $O_{P'}^{\text{LEQ}}$ by following the procedure in fig. 6. Whenever $O_{P'}^{\text{LEQ}}$ needs to compute $P(\text{st}, x)$, the simulator queries O_P^{LEQ} on $|t, x, u, +\rangle$. The b -value is in the $|+\rangle$ state so that any bitflip applied to $|+\rangle$ will leave $|+\rangle$ unchanged. Additionally, the simulator does not see the state of P , so the simulator's $\mathcal{S}$ register stores nothing.

$$\underline{\mathcal{S}_{P \rightarrow P'}^{O_P^{\text{LEQ}}}}$$

1. The simulator samples (sk, pk, k_1, k_2) according to eq. (16). Let P' be the program defined in fig. 16.
2. The simulator implements $O_{P'}^{\text{LEQ}}$ by mostly following the procedure in fig. 6. Any deviations from fig. 6 are listed below.
3. **Work Registers.** The simulator stores the following registers:
 - S' : A register that stores $|c, h\rangle$, where c is the cost of all the queries so far, and h is a bit that equals 1 if and only if the program should halt. S' is initialized to $|0, 0\rangle$.
 - $\mathcal{A}', \mathcal{G}', \mathcal{B}'$. Same as $\mathcal{A}, \mathcal{G}, \mathcal{B}$ in fig. 6.
4. **Sub-Routine.** The simulator makes use of subroutine $U_{P'}$. For any $x \in \mathcal{X}$, $\text{mode} \in \{\text{Eval}, \text{Transfer}\}$, and $u \in \mathcal{Y}$ on register Q , and any $|c, h\rangle$ on register S' , $U_{P'}$ acts as follows:

(a) If $\text{mode} = \text{Eval}$, then:

i. Map

$$|x, \text{mode}, u, b\rangle_Q \otimes |0\rangle_{\mathcal{A}'_i} \leftrightarrow |x, \text{mode}, u, b\rangle_Q \otimes |i, x, \text{mode}, 0, 0\rangle_{\mathcal{A}'_i}$$

ii. Query O_P^{LEQ} on $\mathcal{A}'_i$ to map:

$$|i, x, \text{mode}, 0, 0\rangle_{\mathcal{A}'_i} \rightarrow |i, x, \text{mode}, y, b'\rangle_{\mathcal{A}'_i}$$

iii. Then map:

$$\begin{aligned} &|x, \text{mode}, u, b\rangle_Q \otimes |i, x, \text{mode}, y, b'\rangle_{\mathcal{A}'_i} \\ &\rightarrow |x, \text{mode}, u \oplus y, b \oplus b'\rangle_Q \otimes |i, x, \text{mode}, y, b'\rangle_{\mathcal{A}'_i} \end{aligned}$$

(b) If $\text{mode} = \text{Transfer}$, then map:

$$|x, \text{mode}, u, b\rangle_Q \otimes |0\rangle_{\mathcal{A}'_i} \leftrightarrow \frac{1}{\sqrt{|\mathcal{R}|}} \sum_r |x, \text{mode}, u \oplus \perp, b\rangle_Q \otimes |x, \text{mode}, r\rangle_{\mathcal{A}'_i}$$

When $\text{mode} = \text{Transfer}$, then $U_{P'}$ does *not* query O_P^{LEQ} and instead implements a reversible no-op RAM step that only outputs $\perp$ and records a dummy (transcript) coin $r \in \mathcal{R}$ in $\mathcal{A}'_i$. Concretely, it performs the following reversible maps (controlled on the mode register):

i. Map

$$|x, \text{mode}, u, b\rangle_Q \otimes |0\rangle_{\mathcal{A}'_i} \leftrightarrow |x, \text{mode}, u, b\rangle_Q \otimes |x, \text{mode}, 0\rangle_{\mathcal{A}'_i},$$

i.e., it writes the pair (x, mode) into the transcript block $\mathcal{A}'_i$.

ii. Prepare a uniform superposition over coins in the last component of $\mathcal{A}'_i$:

$$|x, \text{mode}, 0\rangle_{\mathcal{A}'_i} \mapsto \frac{1}{\sqrt{|\mathcal{R}|}} \sum_{r \in \mathcal{R}} |x, \text{mode}, r\rangle_{\mathcal{A}'_i}.$$

iii. Finally, XOR the constant output $\perp$ into the output accumulator (leaving b unchanged):

$$|x, \text{mode}, u, b\rangle_Q \otimes |x, \text{mode}, r\rangle_{\mathcal{A}'_i} \mapsto |x, \text{mode}, u \oplus \perp, b\rangle_Q \otimes |x, \text{mode}, r\rangle_{\mathcal{A}'_i}.$$

Equivalently, for $\text{mode} = \text{Transfer}$, we have $U_{P'}$ apply:

$$|x, \text{mode}, u, b\rangle_Q \otimes |0\rangle_{\mathcal{A}'_i} \leftrightarrow \frac{1}{\sqrt{|\mathcal{R}|}} \sum_{r \in \mathcal{R}} |x, \text{mode}, u \oplus \perp, b\rangle_Q \otimes |x, \text{mode}, r\rangle_{\mathcal{A}'_i}.$$

To see why $\mathcal{S}_{P \rightarrow P'}$ is a *perfect* simulator, it suffices to compare its action with the ideal oracle $O_{P'}^{\text{LEQ}}$ on each computational-basis branch of the query history. In Eval mode, the only nontrivial step of P' is to run the underlying program P on input x , and $\mathcal{S}_{P \rightarrow P'}$ does exactly this by delegating that step to its oracle access to O_P^{LEQ} , while keeping the same query-history, garbage, and bookkeeping registers that fig. 6 would maintain for $O_{P'}^{\text{LEQ}}$. In Transfer mode, P' does not access P at all: by definition it leaves the hidden RAM state unchanged and outputs the distinguished symbol $\perp$, so $\mathcal{S}_{P \rightarrow P'}$ can implement this branch entirely locally by coherently writing $\perp$ to the output register and storing the sampled randomness in its ancilla register exactly as the honest REQ oracle would. Therefore, on every basis state, $\mathcal{S}_{P \rightarrow P'}$ applies exactly the same unitary transformation as $O_{P'}^{\text{LEQ}}$ (including the corresponding inverse on repeated-query branches), and hence by linearity it induces exactly the same quantum channel on arbitrary superpositions. Thus $\mathcal{S}_{P \rightarrow P'}$ perfectly simulates $O_{P'}^{\text{LEQ}}$. $\square$

The security notion of definition B.2 is the same as the one for query-limited RAM programs (definition 5.1), except that the distinguisher D is allowed to internally run several stages of Eval and Transfer. However, this does not change the structure of the proof: the entire view of all senders and receivers is generated from the initial program state, and hence it suffices to simulate that initial state.

Fix any $P \in \mathcal{P}$ and any (non-uniform) QPT distinguisher D with auxiliary input $\text{aux}_{P,\mathcal{P}}$. Let P' be the two-mode RAM program defined in Figure 16, and observe that the real semi-quantum compiler on input P simply outputs a regular QLP for P' , i.e.,

$$\text{PGen}(1^\lambda, P) \equiv \text{QLP.PGen}(1^\lambda, P').$$

By the LEQ-security of QLP from Definition 5.1, there exists a QPT simulator $\mathcal{S}'$ such that

$$\left| \Pr \left[D \left(1^\lambda, \text{QLP.PGen}(1^\lambda, P'), \text{aux}_{P,\mathcal{P}} \right) \rightarrow 1 \right] - \Pr \left[D \left(1^\lambda, \mathcal{S}'^{O_{P'}^{\text{LEQ}}}(1^\lambda), \text{aux}_{P,\mathcal{P}} \right) \rightarrow 1 \right] \right| \leq \text{negl}(\lambda). \quad (22)$$

Next, by Lemma B.6, there exists an oracle simulator $\mathcal{S}_{P \rightarrow P'}$ that, given oracle access to O_P^{LEQ} , perfectly emulates $O_{P'}^{\text{LEQ}}$; in particular, for every QPT oracle algorithm Adv, the final joint state (i.e., the full “view”) of Adv when interacting with $O_{P'}^{\text{LEQ}}$ is identical to its final joint state when interacting with $\mathcal{S}_{P \rightarrow P'}^{O_P^{\text{LEQ}}}$. Define the desired simulator $\mathcal{S}$ by composing $\mathcal{S}'$ with this oracle emulation:

$$\mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda) := \mathcal{S}'^{\mathcal{S}_{P \rightarrow P'}^{O_P^{\text{LEQ}}}}(1^\lambda),$$

meaning that $\mathcal{S}$ runs $\mathcal{S}'$ and answers each oracle query that $\mathcal{S}'$ makes to $O_{P'}^{\text{LEQ}}$ by invoking $\mathcal{S}_{P \rightarrow P'}$, which in turn may query O_P^{LEQ} . By perfect emulation we have

$$\Pr \left[D \left(1^\lambda, \mathcal{S}'^{O_{P'}^{\text{LEQ}}}(1^\lambda), \text{aux}_{P,\mathcal{P}} \right) \rightarrow 1 \right] = \Pr \left[D \left(1^\lambda, \mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda), \text{aux}_{P,\mathcal{P}} \right) \rightarrow 1 \right]. \quad (23)$$

Combining (22) and (23), and using $\text{PGen}(1^\lambda, P) \equiv \text{QLP.PGen}(1^\lambda, P')$, we obtain

$$\left| \Pr \left[D \left(1^\lambda, \text{PGen}(1^\lambda, P), \text{aux}_{P,\mathcal{P}} \right) \rightarrow 1 \right] - \Pr \left[D \left(1^\lambda, \mathcal{S}^{O_P^{\text{LEQ}}}(1^\lambda), \text{aux}_{P,\mathcal{P}} \right) \rightarrow 1 \right] \right| \leq \text{negl}(\lambda),$$

which is exactly the security requirement of Definition 5.1 (and hence of Definition B.2).

B.3 One-Shot Signatures Based On Any Blind-Unforgeable Signature Scheme

The only constructions we have of one-shot signatures ([SZ25a, SZ25b, HV25]) produce signatures that are vectors in a subspace coset.

This limitation is not essential. We provide a generic construction of one-shot signatures with a variety of signature formats. Given any signature scheme that satisfies blind unforgeability (definition 3.3), we construct a one-shot signature scheme whose verification function simply applies the verification function of the blind-unforgeable scheme. The proof is similar to that of [GLR⁺25], who constructed *one-time* signatures from any blind-unforgeable signature scheme.

Definition. In definition B.3 below, we define the one-shot signature primitive that we construct. It is a little weaker than the primitive defined in definitions 3.4 and 3.6.

First, the correctness property says that an honestly generated signature will pass verification. Definition B.3 guarantees this property with overwhelming probability over the randomness of KeyGen and TokenGen, whereas definition 3.4's notion guarantees this property for any well-formed $(sk, pk, |T\rangle, s)$ output by KeyGen or TokenGen.

Second, definition B.3 satisfies (regular) unforgeability, which says that it is hard to produce valid signatures on two different messages with the same serial number. In contrast, definition 3.4 guarantees strong unforgeability, which says that it is also hard to find two different valid signatures on a single (s, m) pair.

Definition B.3 (One-Shot Signature Scheme).

- **Syntax:** A one-shot signature scheme comprises the following QPT algorithms:
 - $\text{KeyGen}(1^\lambda) \rightarrow (sk, pk)$: Takes a security parameter $\lambda \in \mathbb{N}$ and generates two equal keys $sk = pk$.
 - $\text{TokenGen}(sk) \rightarrow (|T\rangle, s)$: Generates a quantum signing token $|T\rangle$ and a classical serial number $s \in \mathcal{S}_\lambda \cup \{\perp\}$ associated with $|T\rangle$.
 - $\text{Sign}(|T\rangle, m) \rightarrow \sigma$: Takes a signing token $|T\rangle$ and a message m from the message space $\mathcal{M}_\lambda$ and outputs a signature $\sigma \in \Sigma_\lambda$.
 - $\text{Ver}(pk, s, m, \sigma) \rightarrow \{0, 1\}$: Verifies the signature σ on message m with serial number s . Ver outputs 1 for valid signatures and 0 for invalid signatures.

The variables $(\lambda, sk, pk, s, m, \sigma)$ are classical, and $|T\rangle$ is a quantum state.

- **Correctness:** The scheme is *correct* if there is a negligible function $\text{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$ and all $m \in \mathcal{M}_\lambda$,

$$\Pr \left[\begin{array}{l} (sk, pk) \leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) \leftarrow \text{TokenGen}(sk) \\ \sigma \leftarrow \text{Sign}(|T\rangle, m) \\ b \leftarrow \text{Ver}(pk, s, m, \sigma) \end{array} \right] \geq 1 - \text{negl}(\lambda)$$

- **(Regular) Unforgeability:** The scheme is *unforgeable* if for any QPT adversary $\mathcal{A}$, there exists a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$,

$$\Pr \left[\begin{array}{l} m_0 \neq m_1 \wedge \\ 1 \leftarrow \text{Ver}(pk, s, m_0, \sigma_0) \wedge \\ 1 \leftarrow \text{Ver}(pk, s, m_1, \sigma_1) \end{array} : \begin{array}{l} (sk, pk) \leftarrow \text{KeyGen}(1^\lambda) \\ (m_0, m_1, \sigma_0, \sigma_1) \leftarrow \mathcal{A}(1^\lambda, pk) \end{array} \right] \leq \text{negl}(\lambda)$$

Construction. The construction uses a one-shot program for the BUS.Sign functionality. Specifically, the functionality takes a message m and serial number s and outputs signature

$$\sigma = (r, r_1, \sigma_{\text{BUS}}),$$

where

$$\sigma_{\text{BUS}} = \text{BUS.Sign}(sk, (s, m, r, r_1)),$$

r is random, and r_1 is pseudorandom. Next, we verify this signature by simply running

$$\text{BUS.Ver}(\text{pk}, (s, m, r, r_1), \sigma_{\text{BUS}}).$$

Intuitively the construction is secure for the following reasons. If the adversary produces a valid message-signature tuple (s, m, σ) , then (s, m) must have been queried to the BUS signing oracle. Furthermore, σ contains r , which is highly entangled with the SEQ oracle's compressed oracle. Therefore, (s, m) will be recorded in the SEQ oracle database with overwhelming probability. Next, the security of the one-shot program says that the SEQ oracle will not record multiple queries with the same serial number s . This allows us to show that an adversary cannot produce two valid message-signature pairs that are valid with respect to the same serial number s .

Now we describe the construction in detail. First, let us define some parameters. Given a security parameter $\lambda \in \mathbb{N}$, let $\mathcal{S}_\lambda$ be the space of serial numbers of the OSS scheme, and let $\mathcal{M}_\lambda$ be the space of messages. Let ℓ be the bitlength of a tuple $(s, m, r) \in \mathcal{S}_\lambda \times \mathcal{M}_\lambda \times \{0, 1\}^\lambda$.

Second, we use the following tools:

- Let $\text{OSP}.$ (KeyGen, TokenGen, PGen, Eval) be a one-shot program compiler (definitions 4.1 and 4.2).
- Let $\text{BUS}.$ (KeyGen, Sign, Ver) be a blind-unforgeable signature scheme (definitions 3.2 and 3.3). Let us assume that the message space $\mathcal{M}_\lambda^{\text{BUS}}$ of BUS includes the set $\{0, 1\}^{2\ell}$, and BUS.Sign uses a random tape of length 2ℓ .
- Let PRF be a pseudorandom function (definition 3.1).

Third, the construction in fig. 17 is a one-shot signature scheme based on BUS. Note that the scheme's Ver function simply applies BUS.Ver to the message (s, m, r, r_1) .

Theorem B.3. The construction in fig. 17 is a one-shot signature scheme (as defined in definition B.3).

Proof. First, it is clear by inspection that the construction satisfies the syntax of a one-shot signature scheme. Note that for any $\lambda \in \mathbb{N}$ and any (sk, pk) in the support of $\text{KeyGen}(1^\lambda)$, $\text{sk} = \text{pk}$. Additionally, OSP.PGen outputs a classical program P , so $\text{sk} = \text{pk} = (\text{crs}_{\text{OSP}}, \text{pk}_{\text{BUS}}, P)$ is a classical string.

Second, lemma B.7 proves that the scheme satisfies correctness. Third, lemma B.8 shows that the scheme satisfies regular unforgeability. $\square$

Lemma B.7. The scheme constructed in fig. 17 satisfies correctness (definition B.3).

Proof. This follows from the correctness of OSP and BUS and the pseudorandomness of PRF.

For any given $\lambda \in \mathbb{N}$ and message $m \in \mathcal{M}_\lambda$, let us define a sequence of hybrids.

Hybrid 1. This hybrid is the correctness experiment:

$$\begin{aligned} (\text{sk}, \text{pk}) &\leftarrow \text{KeyGen}(1^\lambda) \\ (|T\rangle, s) &\leftarrow \text{TokenGen}(\text{sk}) \\ \sigma &\leftarrow \text{Sign}(|T\rangle, m) \\ b &= \text{Ver}(\text{pk}, s, m, \sigma) \end{aligned}$$

- **KeyGen(1^λ):**

1. Sample:

$$\begin{aligned}(k_1, k_2) &\leftarrow \{0, 1\}^\ell \times \{0, 1\}^{2\ell} \\ (\text{sk}_{\text{BUS}}, \text{pk}_{\text{BUS}}) &\leftarrow \text{BUS.KeyGen}(1^\lambda) \\ \text{crs}_{\text{OSP}} &\leftarrow \text{OSP.KeyGen}(1^\lambda)\end{aligned}$$

2. Define f to be the following randomized function, parametrized by $(k_1, k_2, \text{sk}_{\text{BUS}})$:

Inputs: $(s, m; r)$. $s \in \mathcal{S}_\lambda$ is a serial number, $m \in \mathcal{M}_\lambda$ is a message, and $r \xleftarrow{\$} \{0, 1\}^\lambda$ is a random string.

- (a) Compute

$$\begin{aligned}r_1 &= \text{PRF}(k_1, (s, m, r)) \\ r_2 &= \text{PRF}(k_2, (s, m, r, r_1)) \\ \sigma_{\text{BUS}} &= \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); r_2) \\ \sigma &= (r, r_1, \sigma_{\text{BUS}}).\end{aligned}$$

- (b) **Output:** σ

3. Compute

$$\begin{aligned}P &\leftarrow \text{OSP.PGen}(\text{crs}_{\text{OSP}}, f) \\ \text{sk} = \text{pk} &= (\text{crs}_{\text{OSP}}, \text{pk}_{\text{BUS}}, P)\end{aligned}$$

4. Output (sk, pk) .

- **TokenGen(sk):**

1. Parse $\text{sk} = (\text{crs}_{\text{OSP}}, \text{pk}_{\text{BUS}}, P)$.
2. Compute $(|T\rangle_{\text{OSP}}, s) \leftarrow \text{OSP.TokenGen}(\text{crs}_{\text{OSP}})$.
3. Set $|T\rangle = (|T\rangle_{\text{OSP}}, P, s)$.
4. Output $(|T\rangle, s)$.

- **Sign($|T\rangle, m$):**

1. Parse $|T\rangle = (|T\rangle_{\text{OSP}}, P, s)$.
2. Compute and output $\sigma \leftarrow \text{OSP.Eval}(P, s, m, |T\rangle_{\text{OSP}})$.

- **Ver(pk, s, m, σ):**

1. Parse $\text{pk} = (\text{crs}_{\text{OSP}}, \text{pk}_{\text{BUS}}, P)$ and $\sigma = (r, r_1, \sigma_{\text{BUS}})$.
2. Compute and output $b = \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m, r, r_1), \sigma_{\text{BUS}})$.

Figure 17: One-Shot Signature Scheme

Hybrid 2. This hybrid is equivalent to hybrid 1. We have just unfolded the notation.

$$\begin{aligned}
(k_1, k_2) &\leftarrow \{0, 1\}^\ell \times \{0, 1\}^{2\ell} \\
(\text{sk}_{\text{BUS}}, \text{pk}_{\text{BUS}}) &\leftarrow \text{BUS.KeyGen}(1^\lambda) \\
\text{crs}_{\text{OSP}} &\leftarrow \text{OSP.KeyGen}(1^\lambda) \\
P &\leftarrow \text{OSP.PGen}(\text{crs}_{\text{OSP}}, f) \\
(|T\rangle_{\text{OSP}}, s) &\leftarrow \text{OSP.TokenGen}(\text{crs}_{\text{OSP}}) \\
\sigma &\leftarrow \text{OSP.Eval}(P, s, m, |T\rangle_{\text{OSP}}) \\
b &= \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m, r, r_1), \sigma_{\text{BUS}})
\end{aligned}$$

Hybrid 3. This hybrid is the same as hybrid 2, except that hybrid 3 does not generate a program P , and it computes $\sigma = f(s, m; r)$ for a uniformly random r .

$$\begin{aligned}
(k_1, k_2) &\leftarrow \{0, 1\}^\ell \times \{0, 1\}^{2\ell} \\
(\text{sk}_{\text{BUS}}, \text{pk}_{\text{BUS}}) &\leftarrow \text{BUS.KeyGen}(1^\lambda) \\
\text{crs}_{\text{OSP}} &\leftarrow \text{OSP.KeyGen}(1^\lambda) \\
(|T\rangle_{\text{OSP}}, s) &\leftarrow \text{OSP.TokenGen}(\text{crs}_{\text{OSP}}) \\
r &\stackrel{\$}{\leftarrow} \{0, 1\}^\lambda \\
\sigma &= f(s, m; r) \\
b &= \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m, r, r_1), \sigma_{\text{BUS}})
\end{aligned}$$

Claim B.1. The probability that $b = 1$ is negligibly close in hybrids 2 and 3.

Proof. Hybrid 2 computes σ as follows:

$$\begin{aligned}
P &\leftarrow \text{OSP.PGen}(\text{crs}_{\text{OSP}}, f) \\
\sigma &\leftarrow \text{OSP.Eval}(P, s, m, |T\rangle_{\text{OSP}}),
\end{aligned}$$

whereas hybrid 3 computes σ as follows:

$$\begin{aligned}
r &\stackrel{\$}{\leftarrow} \{0, 1\}^\lambda \\
\sigma &= f(s, m; r)
\end{aligned}$$

These two distributions over σ are statistically close, by the correctness of OSP (definition 4.1). This is the only difference between the two hybrids, so the distribution of b is also statistically close in hybrids 2 and 3. $\square$

Hybrid 4. This hybrid is equivalent to hybrid 3. We have just unfolded the notation of $f(s, m; r)$.

$$\begin{aligned}
(k_1, k_2) &\leftarrow \{0, 1\}^\ell \times \{0, 1\}^{2\ell} \\
(\text{sk}_{\text{BUS}}, \text{pk}_{\text{BUS}}) &\leftarrow \text{BUS.KeyGen}(1^\lambda) \\
\text{crs}_{\text{OSP}} &\leftarrow \text{OSP.KeyGen}(1^\lambda) \\
(|T\rangle_{\text{OSP}}, s) &\leftarrow \text{OSP.TokenGen}(\text{crs}_{\text{OSP}}) \\
r &\xleftarrow{\$} \{0, 1\}^\lambda \\
r_1 &= \text{PRF}(k_1, (s, m, r)) \\
r_2 &= \text{PRF}(k_2, (s, m, r, r_1)) \\
\sigma_{\text{BUS}} &= \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); r_2) \\
\sigma &= (r, r_1, \sigma_{\text{BUS}}) \\
b &= \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m, r, r_1), \sigma_{\text{BUS}})
\end{aligned}$$

Hybrid 5. This hybrid is the same as hybrid 4, except that r_1, r_2 are truly random instead of pseudorandom.

$$\begin{aligned}
(\text{sk}_{\text{BUS}}, \text{pk}_{\text{BUS}}) &\leftarrow \text{BUS.KeyGen}(1^\lambda) \\
\text{crs}_{\text{OSP}} &\leftarrow \text{OSP.KeyGen}(1^\lambda) \\
(|T\rangle_{\text{OSP}}, s) &\leftarrow \text{OSP.TokenGen}(\text{crs}_{\text{OSP}}) \\
(r, r_1, r_2) &\xleftarrow{\$} \{0, 1\}^\lambda \times \{0, 1\}^\ell \times \{0, 1\}^{2\ell} \\
\sigma_{\text{BUS}} &= \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); r_2) \\
\sigma &= (r, r_1, \sigma_{\text{BUS}}) \\
b &= \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m, r, r_1), \sigma_{\text{BUS}})
\end{aligned}$$

Claim B.2. The probability that $b = 1$ is negligibly close in hybrids 4 and 5.

Proof. This follows from the security of PRF (definition 3.1). □

Claim B.3. In hybrid 5, $\Pr[b = 1] = 1$.

Proof. This follows from the correctness of BUS (definition 3.2). □

The previous analysis implies that in hybrid 1, $\Pr[b = 1]$ is overwhelming. This completes the proof of correctness. □

Lemma B.8 (Unforgeability). The scheme constructed in fig. 17 satisfies (regular) unforgeability (definition B.3).

Proof. Let us consider the following sequence of hybrids that transforms the unforgeability security game (hybrid 1) into a hybrid (hybrid 10) where the adversary has negligible probability of winning. We will show that in each pair of adjacent hybrids, the adversary's success probability is negligibly close. This shows that the adversary in the unforgeability game has negligible probability of winning.

Hybrid 1. This hybrid is the unforgeability game (definition B.3) for the OSS scheme in fig. 17. The adversary is given $(1^\lambda, \text{pk}, |T\rangle, s)$ and produces two messages with the same serial number s , and the challenger checks that the signatures on those messages are valid.

1. **Inputs:** A security parameter $\lambda \in \mathbb{N}$ and a QPT adversary $\mathcal{A}$.
2. The challenger samples

$$(\text{sk}, \text{pk}) \leftarrow \text{KeyGen}(1^\lambda)$$

and sends to a QPT adversary $\mathcal{A}$ the values $(1^\lambda, \text{pk})$.

3. The adversary outputs $(m_0, m_1, \sigma_0, \sigma_1, s)$.
4. The challenger checks that the following conditions are satisfied:

$$\begin{aligned} m_0 &\neq m_1 \\ 1 &\leftarrow \text{Ver}(\text{pk}, s, m_0, \sigma_0) \\ 1 &\leftarrow \text{Ver}(\text{pk}, s, m_1, \sigma_1) \end{aligned}$$

5. **Output:** If the checks above all pass, then the output of the hybrid is 1. Otherwise, the output is 0.

Note that during $\text{KeyGen}(1^\lambda)$, the challenger computes $P \leftarrow \text{OSP.PGen}(\text{crs}_{\text{OSP}}, f)$.

Hybrid 2. This hybrid is the same as hybrid 1, except that during $\text{KeyGen}(1^\lambda)$, the OSP P is computed as

$$P = \text{Sim}^{O_f^{\text{SEQ}}}(1^\lambda, \text{crs}_{\text{OSP}}, \text{pk}_{\text{BUS}}),$$

where Sim is the simulator guaranteed by the SEQ security of OSP (definition 4.1), and O_f^{SEQ} is the oracle defined in fig. 2.

Claim B.4. For any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $|\Pr[\text{Hybrid 1}(\lambda, \mathcal{A}) \rightarrow 1] - \Pr[\text{Hybrid 2}(\lambda, \mathcal{A}) \rightarrow 1]| \leq \text{negl}(\lambda)$.

Proof. This follows from the SEQ security of OSP (definition 4.1). $\square$

Hybrid 3. This hybrid is the same as hybrid 2, except that O_f^{SEQ} computes $\text{PRF}(k_1, \cdot)$ by querying an oracle O_1 , and computes $\text{BUS.Sign}(\text{sk}, \cdot; \text{PRF}(k_2, \cdot))$ by querying an oracle O_σ .

Let the oracle O_1 be parametrized by k_1 . O_1 takes inputs $(s, m, r) \in \mathcal{S}_\lambda \times \mathcal{M}_\lambda \times \{0, 1\}^\lambda$ and outputs

$$r_1 = \text{PRF}(k_1, (s, m, r))$$

Let the oracle O_σ be parametrized by $(k_2, \text{sk}_{\text{BUS}})$. O_σ takes inputs $(s, m, r, r_1) \in \mathcal{S}_\lambda \times \mathcal{M}_\lambda \times \{0, 1\}^\lambda \times \{0, 1\}^\ell$ and outputs

$$\sigma_{\text{BUS}} = \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); \text{PRF}(k_2, (s, m, r, r_1)))$$

Next, when O_f^{SEQ} computes f , this entails computing

$$\begin{aligned} r_1 &= \text{PRF}(k_1, (s, m, r)) \\ \sigma_{\text{BUS}} &= \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); \text{PRF}(k_2, (s, m, r, r_1))) \end{aligned}$$

In hybrid 3, O_f^{SEQ} computes r_1 by querying O_1 and computes σ_{BUS} by querying O_σ . After O_f^{SEQ} has computed $\sigma = f(s, m, r)$, O_f^{SEQ} uncomputes the values of r_1 and σ_{BUS} from its work registers.

Claim B.5. For any QPT adversary $\mathcal{A}$ and any $\lambda \in \mathbb{N}$, $\Pr[\text{Hybrid 2}(\lambda, \mathcal{A}) \rightarrow 1] = \Pr[\text{Hybrid 3}(\lambda, \mathcal{A}) \rightarrow 1]$.

Proof. O_f^{SEQ} must use a compressed oracle to choose the random input r for f . However, O_1 and O_σ compute deterministic functions of their inputs, so it is syntactically valid for O_f^{SEQ} to query these external oracles to compute $(r_1, \sigma_{\text{BUS}})$. Furthermore, the final value of σ that O_f^{SEQ} computes has the same distribution in hybrids 2 and 3. Therefore, the implementations of O_f^{SEQ} in hybrids 2 and 3 are equivalent. $\square$

Hybrid 4. This hybrid is the same as hybrid 3, except that $\text{PRF}(k_2, \cdot)$ is replaced with a (truly) random oracle H_2 .

Let H_2 be a random oracle mapping $\mathcal{S}_\lambda \times \mathcal{M}_\lambda \times \{0, 1\}^\lambda \times \{0, 1\}^\ell \rightarrow \{0, 1\}^{2\ell}$. Now O_σ is parametrized by $(H_2, \text{sk}_{\text{BUS}})$. On input (s, m, r, r_1) , O_σ outputs

$$\sigma_{\text{BUS}} = \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); H_2(s, m, r, r_1))$$

Claim B.6. For any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $|\Pr[\text{Hybrid 3}(\lambda, \mathcal{A}) \rightarrow 1] - \Pr[\text{Hybrid 4}(\lambda, \mathcal{A}) \rightarrow 1]| \leq \text{negl}(\lambda)$.

Proof. This follows from the security of PRF. $\square$

Hybrid 5. This hybrid is the same as hybrid 4, except that O_σ is blinded on a set B , as in the BUS security game.

Let the blinding set B be all inputs $(s, m, r, r_1) \in \mathcal{S}_\lambda \times \mathcal{M}_\lambda \times \{0, 1\}^\lambda \times \{0, 1\}^\ell$ such that $r_1 \neq \text{PRF}(k_1, (s, m, r))$. Equivalently, B is all inputs such that $r_1 \neq O_1(s, m, r)$.

Next, on each input (s, m, r, r_1) , O_σ queries O_1 to check that $r_1 = O_1(s, m, r)$. If the check passes (i.e. the input is $\notin B$), then O_σ answers the query with $\sigma_{\text{BUS}} = \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); H_2(s, m, r, r_1))$. If the check fails (i.e. the input is $\in B$), then O_σ responds with $\sigma_{\text{BUS}} = \perp$. Finally, O_σ acts coherently on superpositions of classical inputs.

Claim B.7. For any QPT adversary $\mathcal{A}$ and any $\lambda \in \mathbb{N}$, $\Pr[\text{Hybrid 4}(\lambda, \mathcal{A}) \rightarrow 1] = \Pr[\text{Hybrid 5}(\lambda, \mathcal{A}) \rightarrow 1]$.

Proof. The only difference between the hybrids is how O_σ acts when its input satisfies $(s, m, r, r_1) \in B$. In hybrid 4, O_σ computes BUS.Sign on such an input, whereas in hybrid 5, O_σ outputs $\perp$.

In hybrids 4 and 5, it is not possible to query O_σ on an input $(s, m, r, r_1) \in B$. O_σ is only queried by O_f^{SEQ} , which computes r_1 as $r_1 = O_1(s, m, r) = \text{PRF}(k_1, (s, m, r))$. Therefore O_σ is only queried on superpositions of inputs $\notin B$.

Therefore, the O_f^{SEQ} oracle in hybrids 4 and 5 are perfectly indistinguishable, so the two hybrids have the same probability of outputting 1. $\square$

Hybrid 6. This hybrid is the same as hybrid 5, except that the challenger additionally checks that that $m, m' \notin B$ when deciding what to output.

More specifically, after $\mathcal{A}$ outputs $(m_0, m_1, \sigma_0, \sigma_1, s)$, the challenger makes the following checks:

1. Check that

$$\begin{aligned} m_0 &\neq m_1 \\ 1 &\leftarrow \text{Ver}(\text{pk}, s, m_0, \sigma_0) \\ 1 &\leftarrow \text{Ver}(\text{pk}, s, m_1, \sigma_1) \end{aligned}$$

2. Parse $\sigma_0 = (r^0, r_1^0, \sigma_{\text{BUS}}^0)$ and $\sigma_1 = (r^1, r_1^1, \sigma_{\text{BUS}}^1)$. Check that

$$\begin{aligned} r_1^0 &= O_1(s, m_0, r^0) \\ r_1^1 &= O_1(s, m_1, r^1) \end{aligned}$$

3. **Output:** If the checks above all pass, then the output of the hybrid is 1. Otherwise, the output is 0.

Note that the checks in step 2 are new and were not present in hybrid 5.

Lemma B.9. For any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $|\Pr[\text{Hybrid 5}(\lambda, \mathcal{A}) \rightarrow 1] - \Pr[\text{Hybrid 6}(\lambda, \mathcal{A}) \rightarrow 1]| \leq \text{negl}(\lambda)$.

Proof. First, the only case when hybrids 5 and 6 output different values is when the checks in step 1 (verifying the signature) pass, but the checks in step 2 (checking that the signature is not in the blinded set B) fail. In this case, hybrid 5 outputs 1, but hybrid 6 outputs 0.

Second, the case described above occurs with only negligible probability due to the blind unforgeability of BUS (definition 3.3). Note that the oracle O_σ in hybrids 5 and 6 is equivalent to the oracle $\text{BUS.Sign}_B(\text{sk}_{\text{BUS}}, \cdot)$ from the blind unforgeability security game. Next, in the case above, we have that $1 \leftarrow \text{Ver}(\text{pk}, s, m_0, \sigma_0)$ and $1 \leftarrow \text{Ver}(\text{pk}, s, m_1, \sigma_1)$, which implies that

$$\begin{aligned} 1 &\leftarrow \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m_0, r_1^0, \sigma_{\text{BUS}}^0), \text{ and} \\ 1 &\leftarrow \text{BUS.Ver}(\text{pk}_{\text{BUS}}, (s, m_1, r_1^1, \sigma_{\text{BUS}}^1) \end{aligned}$$

Furthermore, in the case above, we have that $r_1^0 \neq O_1(s, m_0, r^0)$ or $r_1^1 \neq O_1(s, m_1, r^1)$, which implies that

$$\begin{aligned} (s, m_0, r_1^0, r_1^0) &\in B, \text{ or} \\ (s, m_1, r_1^1, r_1^1) &\in B \end{aligned}$$

Therefore, if the case above occurs, then for some $i \in \{0, 1\}$, $(s, m_i, r_1^i, \sigma_{\text{BUS}}^i)$ represent a solution to the blind unforgeability game. The probability that $\mathcal{A}$ outputs such a solution is negligible, due to the blind-unforgeability of BUS. $\square$

Hybrid 7. This hybrid is the same as hybrid 6 except that O_σ is no longer blinded on set B .

On each input (s, m, r, r_1) , O_σ always answers the query with $\sigma_{\text{BUS}} = \text{BUS.Sign}(\text{sk}_{\text{BUS}}, (s, m, r, r_1); H_2(s, m, r, r_1))$.

Claim B.8. For any QPT adversary $\mathcal{A}$ and any $\lambda \in \mathbb{N}$, $\Pr[\text{Hybrid 6}(\lambda, \mathcal{A}) \rightarrow 1] = \Pr[\text{Hybrid 7}(\lambda, \mathcal{A}) \rightarrow 1]$.

Proof. The proof is essentially the same as the proof of claim B.7.

The only difference between the hybrids is how O_σ acts when its input satisfies $(s, m, r, r_1) \in B$. In hybrid 6, O_σ outputs $\perp$ on such an input whereas in hybrid 7, O_σ computes BUS.Sign .

In hybrids 6 and 7, it is not possible to query O_σ on an input $(s, m, r, r_1) \in B$. O_σ is only queried by O_f^{SEQ} , which computes r_1 as $r_1 = O_1(s, m, r) = \text{PRF}(k_1, (s, m, r))$. Therefore O_σ is only queried on superpositions of inputs $\notin B$.

Therefore, the O_f^{SEQ} oracle in hybrids 6 and 7 are perfectly indistinguishable, so the two hybrids have the same probability of outputting 1. $\square$

Hybrid 8. This hybrid is the same as hybrid 7 except that $\text{PRF}(k_1, \cdot)$ is replaced with a (truly) random oracle $H_1(\cdot)$.

Let H_1 be a random oracle mapping $\mathcal{S}_\lambda \times \mathcal{M}_\lambda \times \{0, 1\}^\lambda \rightarrow \{0, 1\}^\ell$. Now O_1 is parametrized by H_1 . On input (s, m, r) , O_1 outputs

$$r_1 = H_1(s, m, r)$$

Claim B.9. For any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $|\Pr[\text{Hybrid 7}(\lambda, \mathcal{A}) \rightarrow 1] - \Pr[\text{Hybrid 8}(\lambda, \mathcal{A}) \rightarrow 1]| \leq \text{negl}(\lambda)$.

Proof. This follows from the security of PRF. $\square$

Hybrid 9. This hybrid is the same as hybrid 8, except that H_1 is implemented as a compressed oracle.

Claim B.10. For any QPT adversary $\mathcal{A}$ and any $\lambda \in \mathbb{N}$, $\Pr[\text{Hybrid 8}(\lambda, \mathcal{A}) \rightarrow 1] = \Pr[\text{Hybrid 9}(\lambda, \mathcal{A}) \rightarrow 1]$.

Proof. The random oracle is indistinguishable from the compressed oracle due to [Zha18] lemma 4. $\square$

Let us summarize the definition of O_f^{SEQ} in hybrid 9. O_f^{SEQ} has two compressed oracle databases D, D_1 (D is the database for the oracle H , and D_1 is the database for the oracle H_1). When queried on a basis state $|s, m, u, b\rangle_Q \otimes |D\rangle_{\mathcal{D}} \otimes |D_1\rangle_{\mathcal{D}_1}$, the oracle responds to the query as follows:

1. If the database D contains an entry of the form (s', m', r') such that $s' = s$ but $m' \neq m$, then skip the following steps. Otherwise, continue.
2. (a) Prepare the state $|s, m, 0, 0, 0\rangle$ in an ancillary register Q' . Query $H(s, m)$ by applying the compressed oracle unitary CO to $Q' \times \mathcal{D}$. This maps Q' to a superposition over states of the form $|s, m, r, 0, 0\rangle$.
- (b) Query $H_1(s, m, r)$ by applying the compressed oracle unitary CO to $Q' \times \mathcal{D}_1$. This maps Q' to a superposition over states of the form $|s, m, r, r_1, 0\rangle$.
- (c) Query O_σ to compute $\sigma_{\text{BUS}} = O_\sigma(s, m, r, r_1)$, and write the result to register Q' :

$$|s, m, r, r_1, 0\rangle_{Q'} \rightarrow |s, m, r, r_1, \sigma_{\text{BUS}}\rangle_{Q'}$$

3. Apply the following isometry to registers $Q \times Q'$:

$$|s, m, u, b\rangle_Q \otimes |s, m, r, r_1, \sigma_{\text{BUS}}\rangle_{Q'} \rightarrow |s, m, u \oplus (r, r_1, \sigma_{\text{BUS}}), b \oplus 1\rangle_Q \otimes |s, m, r, r_1, \sigma_{\text{BUS}}\rangle_{Q'}$$

4. Uncompute item 2.

Note that O_σ does not query H or H_1 .

Hybrid 10. This hybrid is the same as hybrid 9, except that right after $\mathcal{A}$ outputs $(m_0, m_1, \sigma_0, \sigma_1, s)$, the challenger checks that the compressed oracle databases (D, D_1) satisfy the following property: for every entry $(s, m, r, r_1) \in D_1$, there is the entry $(s, m, r) \in D$. If this check fails, then the output of the hybrid is 0.

Claim B.11. For any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $|\Pr[\text{Hybrid 9}(\lambda, \mathcal{A}) \rightarrow 1] - \Pr[\text{Hybrid 10}(\lambda, \mathcal{A}) \rightarrow 1]| \leq \text{negl}(\lambda)$.

Proof. This follows from the compressed oracle chaining lemma (lemma 5.2).

First, for either hybrid 9 or 10, from the beginning of the hybrid until right after $\mathcal{A}$ outputs $(m_0, m_1, \sigma_0, \sigma_1, s)$, the only times that H or H_1 are queried is during an execution of O_f^{SEQ} . Furthermore, the forward computation in O_f^{SEQ} queries H, H_1 in a chained fashion, where the inputs and outputs of H are fed as input to H_1 . Specifically, the forward computation is:

1. Compute $r = H(s, m)$.
2. Compute $r_1 = H_1(s, m, r)$.

Note that O_σ does not query H or H_1 . After computing the signature and copying the output, uncomputing step 2 first applies the inverse signing computation, then queries H_1 to uncompute r_1 , and finally queries H to uncompute r . Thus the inverse calls occur in the reverse order; they are not another forward chained computation.

Second, we verify that this complete procedure obeys the same norm estimate as the query procedure in lemma 5.2. Let us rename our variables to match the notation of lemma 5.2:

$$\begin{aligned} H &\rightarrow G, \\ H_1 &\rightarrow H, \\ ((s, m), \epsilon, r) &\rightarrow (x_g, y_1, y_2). \end{aligned}$$

Here ϵ denotes the empty string, so $\mathcal{Y}_1 = \{\epsilon\}$. Additionally, take the function f in lemma 5.2 to be

$$f(x_g, y_1 \| y_2) = f((s, m), r) = (s, m) = x_h.$$

Note that y_2 comes from a set of size $|\mathcal{Y}_2| = 2^\lambda$. Let $t = \text{poly}(\lambda)$ bound the total number of complete calls to O_f^{SEQ} before $\mathcal{A}$ outputs, including any calls made by the simulator. Each complete call uses at most two queries to G , two queries to H , and two calls to O_σ , counting both computation and uncomputation. Thus t counts complete SEQ calls, not individual calls to G or H .

With these new variable names, on the branch where the SEQ check permits the query, O_f^{SEQ} proceeds as follows:

1. Compute $y_1 \| y_2 \coloneqq G(x_g)$.
2. Compute $x_h \coloneqq f(x_g, y_1 \| y_2)$.
3. Compute $z \coloneqq H(x_h, y_2)$ in a work register. Here z is the value denoted r_1 in the original notation.
4. Compute the signature using O_σ , copy the output $(r, r_1, \sigma_{\text{BUS}})$ to the output register, and flip b , as in the definition of O_f^{SEQ} in hybrid 9.
5. Uncompute the signature, then z , then x_h , and finally $y_1 \| y_2$. The two oracle calls in this last step are to H and then G .

Let E be the projector defined in the proof of lemma 5.2: it projects onto database pairs such that every $(x_h \| y_2, z) \in D_H$ has a matching $(x_g, y_1 \| y_2) \in D_G$ with $f(x_g, y_1 \| y_2) = x_h$. Write $\bar{E} = I - E$.

The SEQ check is the first step in the description of O_f^{SEQ} in hybrid 9. For a fixed query (s, m) , it tests only whether a different cell (s, m') , with $m' \neq m$, is occupied. Its coherent computation and uncomputation therefore commute with E and with Decomp_G at $x_g = (s, m)$. The signing computation, its inverse, and the output-copying operation leave D_G, D_H and the input registers (s, m, r) unchanged. Thus they also commute with E and with Decomp_G .

Expand each of the two G queries as $\text{Decomp}_G \circ \text{CO}'_G \circ \text{Decomp}_G$, as in section 3.1. The Decomp after the forward copy and the decompression before the inverse copy commute with all operations between them and cancel, since $\text{Decomp}_G^2 = I$. Hence exactly two Decomp_G operations remain in each complete permitted query: one at the beginning and one at the end.

To check the inputs to both H queries in this representation, recall that at query boundaries each cell of a compressed database is orthogonal to its uniform non- $\perp$ state. For D_G , this holds initially and is preserved by compressed queries. It is also preserved by the SEQ check, which tests only whether cells are $\perp$ and does not distinguish their non- $\perp$ values. Consequently, the first Decomp_G leaves the queried cell non- $\perp$, and CO'_G copies its value $y_1 \| y_2$ into the work register. Throughout both the forward and inverse H queries, D_G therefore contains $(x_g, y_1 \| y_2)$, and the input to H is

$(f(x_g, y_1 \| y_2), y_2)$. Each of these H queries changes only an already-witnessed cell of D_H , so it preserves E and $\bar{E}$ on these states. The inverse call has the same property as the forward call. This is the argument of claim A.1, applied to both calls, not just the forward one. All other operations between the two remaining decompressions leave D_G and D_H unchanged.

Applying claim A.2 to these two Decomp_G therefore gives, for every normalized state $|\psi\rangle$ reached before a complete query,

$$\|\bar{E} O_f^{\text{SEQ}} |\psi\rangle\| \leq \|\bar{E} |\psi\rangle\| + 2\sqrt{\frac{2}{|\mathcal{Y}_2|}}.$$

The same estimate holds for a superposition of permitted and skipped queries: the SEQ control commutes with the operators just considered, and the skipped branch is the identity.

Third, lemma 5.2 implies that right after $\mathcal{A}$ outputs $(m_0, m_1, \sigma_0, \sigma_1, s)$, the state of the system gives squared norm at least $1 - 8t^2/|\mathcal{Y}_2|$ to databases (D_G, D_H) that satisfy the following condition: for all $(x_h \| y_2, z) \in D_H$, there exists an entry $(x_g, y_1 \| y_2) \in D_G$ such that $f(x_g, y_1 \| y_2) = x_h$. Indeed, both databases are initially empty, and operations between complete SEQ calls do not act on the databases. If $|\psi_t\rangle$ is the normalized joint state at this point, purified to include all measurement records, then

$$\|\bar{E} |\psi_t\rangle\| \leq 2t\sqrt{\frac{2}{|\mathcal{Y}_2|}}, \quad \|\bar{E} |\psi_t\rangle\|^2 \leq \frac{8t^2}{|\mathcal{Y}_2|}.$$

The above bound implies that right after $\mathcal{A}$ outputs $(m_0, m_1, \sigma_0, \sigma_1, s)$, the state of the system gives squared norm at least $1 - 8t^2/2^\lambda$ to databases (D, D_1) that satisfy the following condition: for all $(s, m, r, r_1) \in D_1$, there exists an entry $(s', m', r) \in D$ such that $(s, m) = (s', m')$.

Fifth, in hybrid 10, the challenger checks that for every entry $(s, m, r, r_1) \in D_1$, there is the entry $(s, m, r) \in D$. The argument above implies that this check passes with overwhelming probability. More precisely, the accepting component is $E |\psi_t\rangle$, and

$$\| |\psi_t\rangle - E |\psi_t\rangle \| = \|\bar{E} |\psi_t\rangle\| \leq 2t\sqrt{\frac{2}{2^\lambda}}.$$

Inserting this check and outputting 0 on failure changes the probability of any subsequent acceptance by at most twice this vector distance, which is negligible for polynomial t . Since this check is the only difference between hybrids 9 and 10, this shows that the output distributions of hybrids 9 and 10 are statistically close. $\square$

Claim B.12. For any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $\Pr[\text{Hybrid 10}(\lambda, \mathcal{A}) \rightarrow 1] \leq \text{negl}(\lambda)$.

Proof. First, parse $\sigma_i = (r^i, r_1^i, \sigma_{\text{BUS}}^i)$ for $i \in \{0, 1\}$, as in item 2; malformed outputs are rejected. The output of hybrid 10 is 1 only if both of the following events occur:

- Event E_1 : The challenger finds that for every entry $(s, m, r, r_1) \in D_1$, there is the entry $(s, m, r) \in D$.
- Event E_2 : The challenger finds that $m_0 \neq m_1$, $r_1^0 = H_1(s, m_0, r^0)$, and $r_1^1 = H_1(s, m_1, r^1)$.

We will show that

$$\Pr[E_1 \wedge E_2] \leq \frac{2}{2^\ell},$$

which suffices to prove that hybrid 10 outputs 1 with negligible probability. Here H_1 has range $\{0, 1\}^\ell$, as specified in hybrid 8. For this upper bound, we may omit the ordinary signature-verification checks in item 1, but still requiring $m_0 \neq m_1$. Those signature checks do not act on the databases and commute with the checks defining E_1 and E_2 .

Second, right after the challenger checks (D, D_1) , every database basis state with nonzero amplitude has the property that no two entries of D have the same value of s . We prove this inductively. Before any queries have been made to H , the database D is empty, so the property holds. Consider a query (s, m) to O_f^{SEQ} and assume that the property holds at the start of the query. By the SEQ check in item 1 of Figure 2, also used in the description of O_f^{SEQ} immediately before hybrid 10, the oracle skips the query if D contains an entry $(s, m', *)$ with $m' \neq m$. Otherwise, D either has no entry with serial number s or has its unique such entry at (s, m) . The queries to H , including uncomputation, modify only the cell at (s, m) . They may add or remove this entry or change its value, but cannot introduce two entries with the same serial number. The queries to H_1 and O_σ do not modify D . Thus the property holds after the complete query, also for a superposition of query inputs. The final check defining E_1 is diagonal in the database basis, so it preserves this property.

Third, the component on which the check defining E_1 succeeds is supported on databases (D, D_1) with the following properties:

1. No two entries of D have the same s -value.
2. For every entry $(s, m, r, r_1) \in D_1$, there is the entry $(s, m, r) \in D$.

These properties imply that D_1 does not contain both an entry $(s, m, *, *)$ and an entry $(s, m', *, *)$ with $m \neq m'$. Otherwise, D would contain $(s, m, *)$ and $(s, m', *)$, contrary to the first property. This statement holds on each database basis state in the component; it does not assert that the databases have been fully measured.

Fix a classical output tuple $(m_0, m_1, \sigma_0, \sigma_1, s)$ with $m_0 \neq m_1$. Let $|\varphi\rangle$ be the unnormalized component obtained by keeping the outcome E_1 of the challenger's database check, starting from the normalized state conditioned on this fixed output tuple. Include any registers needed to purify the state. Thus $\| |\varphi\rangle \|^2$ is the probability that the E_1 check succeeds for this fixed tuple, and is at most 1. Define the two fixed inputs

$$v_i := (s, m_i, r^i) \quad (i \in \{0, 1\}).$$

Since $m_0 \neq m_1$, we have $v_0 \neq v_1$. The two properties above imply that every database basis state occurring in $|\varphi\rangle$ has at least one of $D_1(v_0), D_1(v_1)$ equal to $\perp$.

Decompose $|\varphi\rangle$ as

$$|\varphi\rangle = |\varphi_0\rangle + |\varphi_1\rangle,$$

where $|\varphi_0\rangle$ consists of its database basis components satisfying $D_1(v_0) = \perp$, and $|\varphi_1\rangle$ consists of those satisfying $D_1(v_0) \neq \perp$. On the latter components, $D_1(v_1) = \perp$ necessarily holds. This is an orthogonal decomposition, so

$$\| |\varphi_0\rangle \|^2 + \| |\varphi_1\rangle \|^2 = \| |\varphi\rangle \|^2.$$

We use this decomposition only in the calculation; the challenger does not measure which component occurs.

Fourth, set $N := |\{0, 1\}^\ell| = 2^\ell$. For each $i \in \{0, 1\}$, let V_i be the linear map that appends a fresh answer register in $|0^\ell\rangle$, makes a compressed-oracle query to H_1 at the fixed input v_i , and keeps only the answer r_1^i . We suppress the answer register after projecting it onto this fixed value. Thus V_i describes the accepting part of the corresponding equality check in item 2, and $\| V_i |\chi\rangle \| \leq \| |\chi\rangle \|$ for every vector $|\chi\rangle$.

We first compute its action when $D_1(v_i) = \perp$. By the definition $\text{CO} = \text{Decomp} \circ \text{CO}' \circ \text{Decomp}$ in Section 3.1, for a database d with $d(v_i) = \perp$ the initial decompression gives

$$\text{Decomp}_{v_i} |d\rangle = \frac{1}{\sqrt{N}} \sum_{a \in \{0, 1\}^\ell} |d \cup \{(v_i, a)\}\rangle.$$

Here Decomp_{v_i} acts on the database of H_1 . Copying the value into the answer register, keeping the answer r_1^i , and applying the final decompression therefore gives

$$V_i |d\rangle = \frac{1}{\sqrt{N}} \text{Decomp}_{v_i} |d \cup \{(v_i, r_1^i)\}\rangle.$$

Different residual databases d in this calculation give orthogonal vectors before the final decompression, which is unitary. Consequently, for every vector $|\chi\rangle$ supported on $D_1(v_i) = \perp$, including vectors entangled with other registers,

$$\|V_i |\chi\rangle\| = \frac{1}{\sqrt{N}} \| |\chi\rangle \|.$$

Because $v_0 \neq v_1$, the maps V_0 and V_1 act on different database cells and commute. Applying the preceding equality to the fixed absent cell in each component gives

$$\begin{aligned} \|V_1 V_0 |\varphi_0\rangle\| &\leq \|V_0 |\varphi_0\rangle\| = \frac{\| |\varphi_0\rangle \|}{\sqrt{N}}, \\ \|V_1 V_0 |\varphi_1\rangle\| &= \|V_0 V_1 |\varphi_1\rangle\| \leq \|V_1 |\varphi_1\rangle\| = \frac{\| |\varphi_1\rangle \|}{\sqrt{N}}. \end{aligned}$$

By linearity, the triangle inequality, and Cauchy–Schwarz,

$$\begin{aligned} \|V_1 V_0 |\varphi\rangle\| &\leq \|V_1 V_0 |\varphi_0\rangle\| + \|V_1 V_0 |\varphi_1\rangle\| \\ &\leq \frac{\| |\varphi_0\rangle \| + \| |\varphi_1\rangle \|}{\sqrt{N}} \\ &\leq \sqrt{\frac{2}{N}} \sqrt{\| |\varphi_0\rangle \|^2 + \| |\varphi_1\rangle \|^2} \\ &= \sqrt{\frac{2}{N}} \| |\varphi\rangle \|. \end{aligned}$$

The squared norm on the left is the joint probability of the E_1 check and both equality checks succeeding for the fixed output tuple, with the ordinary signature checks omitted. It is at most

$$\frac{2}{N} \| |\varphi\rangle \|^2 \leq \frac{2}{N}.$$

The bound is uniform over the classical output tuple. Tuples with $m_0 = m_1$, or malformed signatures, cannot satisfy E_2 . Averaging over the output tuple and reinstating any omitted signature checks therefore gives

$$\Pr[\text{Hybrid } 10(\lambda, \mathcal{A}) \rightarrow 1] \leq \Pr[E_1 \wedge E_2] \leq \frac{2}{2^\ell}.$$

By the choice of ℓ immediately before Figure 17, ℓ is the bitlength of a tuple (s, m, r) with $r \in \{0, 1\}^\lambda$. Hence $\ell \geq \lambda$, and $2/2^\ell \leq 2^{1-\lambda}$ is negligible. This proves the claim. $\square$

The previous claims imply that for any QPT adversary $\mathcal{A}$, there is a negligible function $\text{negl}(\cdot)$ such that for any $\lambda \in \mathbb{N}$, $\Pr[\text{Hybrid } 1(\lambda, \mathcal{A}) \rightarrow 1] \leq \text{negl}(\lambda)$. Since hybrid 1 is the unforgeability game, this implies that the OSS scheme satisfies unforgeability. $\square$