Universally Composable Succinct Vector Commitments and Applications

Ran Canetti
canetti@bu.edu
Boston University

Megan Chen
megchen@bu.edu
Boston University

September 29, 2025

Abstract

We develop a toolbox for modular construction and analysis of *succinct, non-interactive* commitments and vector commitments in the random oracle model, while guaranteeing universally composable security. To demonstrate its power, we use the toolbox to construct and analyze a modular variant of the Kilian-Micali ZK-SNARK. Along the way we also propose a new UC formulation of a global random oracle, that avoids a weakness in existing formulations and also enables expressing more nuanced, session-specific abstractions. We hope that this toolbox will be useful for building secure applications in settings where both succinctness and non-interactivity are key.

Contents

1 Introduction 3
1.1 Our contributions . . . 3
1.2 Technical overview . . . 4
1.3 Additional related work . . . 9

2 Preliminaries 10
2.1 Notations . . . 11
2.2 UC with Global Subroutines . . . 11
2.3 $\mathcal{F}_{\mathsf{lib}}$: Global code library functionality . . . 11

3 $\mathcal{F}_{\mathsf{GRO}}$: Ideal functionality for global random oracle 12

4 $\mathcal{F}_{\mathsf{NC}}$: Ideal functionality for non-interactive commitment 14
4.1 The interface between $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$ . . . 15
4.2 Security guarantees provided by $\mathcal{F}_{\mathsf{NC}}$ . . . 18
4.2.1 Completeness and consistency . . . 20
4.2.2 Hiding . . . 20
4.2.3 Binding . . . 23
4.2.4 Ensuring immediate response and correctness . . . 26
4.2.5 Adaptive security . . . 27

5 Π_{NC}: The non-interactive commitment scheme 28
5.1 Proof of Theorem 5.1 . . . 28
5.1.1 The simulator's code library . . . 28
5.1.2 Validity of the simulation . . . 29

6 $\mathcal{F}_{\mathsf{NVC}}$: Non-interactive vector commitment 32
6.1 The interface between $\mathcal{F}_{\mathsf{NVC}}$ and $\mathcal{F}_{\mathsf{GRO}}$ . . . 34
6.2 Security guarantees provided by $\mathcal{F}_{\mathsf{NVC}}$ . . . 37
6.2.1 Hiding . . . 37
6.2.2 Position Binding . . . 40

7 Π_{NVC}: The non-interactive vector commitment scheme 43
7.1 Proof of Theorem 7.1 . . . 44
7.1.1 The simulator's code library . . . 44
7.1.2 Validity of the simulation . . . 50

8 $\mathcal{F}_{\mathsf{NIZK}}$: Ideal functionality for non-interactive zero knowledge (NIZK) 51
8.1 Interface between $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{GRO}}$ . . . 51
8.2 Adaptive security . . . 51

9 Π_{NIZK}: UC Zero-Knowledge SNARK from PCP 54
9.1 Probabilistically-checkable proofs (PCP) . . . 54
9.2 The protocol from Honest Verifier Zero Knowledge (HVZK) PCPs . . . 56
9.3 Proof of Theorem 9.1 . . . 56
9.3.1 The simulator's code library . . . 56
9.3.2 Validity of the simulation . . . 60
9.3.3 Efficiency measures . . . 66

Acknowledgments 67

References 67

A A weakness in earlier formulations of the global random oracle 70

1 Introduction

The Universally Composable Security (UC) framework [Can01; Can20] enables the specification of cryptographic tasks in an idealized way that precisely and robustly captures their security requirements, along with a modular design methodology where a given task is realized using idealized sub-tasks. Protocols are ultimately constructed by realizing all sub-tasks, potentially through further decomposition, until all sub-tasks are realized by concrete protocols.

Initially, the UC framework was used for analyzing the security of *interactive* protocols. Examples include [Can01; CF01; CK02; CLOS02; KLP05; DN03; MPR10]. For such protocols, ideal functionalities did not specify limits on the rounds of interaction in the realizing protocol and often require all parties to interact. In other words, an execution of such protocols consists of atomic operations, where in each atomic operation a party processes an incoming message and generates an outgoing one. This applies both to the security specifications (ideal functionalities) and to the analyzed protocols.

Another line of works use the UC framework to capture tasks that restrict the communication pattern of the realizing protocols to be *non-interactive*, i.e. protocols that consist of a single message sent by some party, plus local processing by the recipients. Examples include public-key encryption, digital signature, and non-interactive zero knowledge (NIZK), e.g. [CKN03; Can04; CSV16; GOS12; BFGMR22; GKOPTT23; CF24]. However, while many non-interactive protocols are themselves often conceived in a modular way, (namely, using other cryptographic primitives as building blocks), taking advantage of the composabilility properties of the UC framework to facilitate modular construction and analysis of composite non-interactive protocols has been progressing more slowly.

Indeed, such an endeavor is challenging, as it requires modeling the building blocks as non-interactive protocols within the UC framework, and then formulating the relevant security properties of these protocols in a way that makes them readily usable as components within other non-interactive ones. More specifically, the challenge is to formulate ideal functionalities that: (A) guarantee the desired security properties, while consisting of one party outputting a single "idealized" string that is then further processed by other non-interactive protocols, and (B) are realizable by actual non-interactive protocols. Given that realizing UC functionalities naturally requires straight-line simulation, extraction, and equivocation, this endeavor appears almost impossible.

Still, a number of works have taken on this challenge. Specifically, the UC modeling and analysis of the Signal protocol of [CJSV22] includes multiple sub-components that handle the processing of each received message and the sending of each message. Also, [CSW22] gives a UC modeling of non-interactive commitment and use it to construct non-interactive zero-knowledge protocols in the common reference string model. In both cases, the modularity enables a more general result with a simpler and more readily verifiable analysis.

1.1 Our contributions

This work aims to enable modular construction and analysis of primitives that are both non-interactive and additionally require *succinctness.* Examples include non-interactive succinct commitments, vector commitments, accumulators, ZK-SNARKs and Proof-Carrying Code (PCD) systems, delegation of computation scheme, shared database systems, and related tasks.

Given the inherent incompatibility of succinctness, non-interactivity, and straight-line black-box simulation-extraction in the plain model, we concentrate on protocols designed and analyzed in the random oracle model. This, in turn, requires appropriate modeling of of a programmable random oracle as globally shared resource within the UC framework. We thus start by providing a formulation of a programmable random oracles as a (global) ideal functionality, $\mathcal{F}_{\mathsf{GRO}}$. The new formulation avoids shortcomings of previous ones, e.g. those in [CDGLN18; CF24] (also discussed in [LR22]), and may be of independent interest. We then proceed as follows:

(a) We formulate an ideal functionality, $\mathcal{F}_{\mathsf{NC}}$, that captures succinct non-interactive commitment in the presence of $\mathcal{F}_{\mathsf{GRO}}$. This builds on the non-interactive commitment of [CSW22], which models schemes in the common reference string model.

(b) We show that $\mathcal{F}_{\mathsf{NC}}$ is realized by a simple folklore succinct commitment scheme.

(c) We formulate an ideal functionality, $\mathcal{F}_{\mathsf{NVC}}$, that captures succinct non-interactive vector commitment in the presence of $\mathcal{F}_{\mathsf{GRO}}$.

(d) We show that a variant of the Merkle tree-based vector commitment, where each node is implemented using an instance of $\mathcal{F}_{\mathsf{NC}}$, *perfectly* realizes $\mathcal{F}_{\mathsf{NVC}}$.

To demonstrate the viability of our formalism, we show that the Kilian-Micali ZK-SNARK, formulated as a protocol that uses the ideal sub-component $\mathcal{F}_{\mathsf{NVC}}$, UC-realizes $\mathcal{F}_{\mathsf{NIZK}}$, where $\mathcal{F}_{\mathsf{NIZK}}$ is an adaptation of the non-interactive zero-knowledge functionality of [GOS12] to be in the presence of $\mathcal{F}_{\mathsf{GRO}}$. This application can be viewed as a modular alternative to the recent work of Chiesa and Fenzi [CF24], which shows that the Kilian-Micali protocol UC-realizes $\mathcal{F}_{\mathsf{NIZK}}$ in the presence of a global, programmable random oracle (as formulated in [LR22]). Indeed, Chiesa and Fenzi directly analyze the entire Kilian-Micali protocol, without taking advantage of the modularity provided by the UC framework. In addition to avoiding the weakness in their formulation of $\mathcal{F}_{\mathsf{GRO}}$, our analysis obtains somewhat better concrete parameters than those obtained in [CF24], due to our modular treatment of vector commitments. (See discussion in Section 1.3.)

Additional potential uses of our formalism of succinct UC commitments and vector commitments include other constructions of ZK SNARKs and Proof-Carrying Data (PCD) protocols from these two primitives (such as, IOPs [BCS16] or the recent proposal of [BMNW25]). Additional candidate uses include verifiable and updatable databases (for secret data), dynamic accumulators (of secret data), and anonymous credentials (see [CF13]).

1.2 Technical overview

We give an overview of our formalism and technical contributions.

A weakness in existing programmable global random oracles. The main challenge in formulating the random oracle as a global ideal functionality, $\mathcal{F}_{\mathsf{GRO}}$, within the UC framework is to preserve the validity of modular composition with the oracle as a shared object, while also preserving the model's pertinent properties, namely the ability to observe the adversary's oracle queries and determine ("program") the responses provided to the adversary.

We first demonstrate a weakness in existing formalisms (e.g. those used in [CDGLN18; LR22; CF24]). Specifically, consider a protocol ρ that makes subroutine calls to some ideal functionality $\mathcal{F}$ and in addition has access to the [CDGLN18; LR22; CF24] formulation of $\mathcal{F}_{\mathsf{GRO}}$. We show how ρ can use its access to the global $\mathcal{F}_{\mathsf{GRO}}$ to tell whether or not $\mathcal{F}$ was replaced by some protocol π that realizes it - thereby "bypassing" the guarantees made by the UC theorem. We further exploit this weakness of $\mathcal{F}_{\mathsf{GRO}}$ to have ρ break down as soon as $\mathcal{F}$ is replaced by π. This demonstrates that having access to [CDGLN18; LR22; CF24] formulation of $\mathcal{F}_{\mathsf{GRO}}$ means that the UC theorem is no longer useful for security analysis. See more details within.

An alternative formulation of global programmable random oracle. We present a new $\mathcal{F}_{\mathsf{GRO}}$ formalism which avoids the weakness and also enjoys some additional attractive properties. Rather than interacting directly with the environment/adversary, our $\mathcal{F}_{\mathsf{GRO}}$ only allows ideal functionalities (i.e., machines with $\mathsf{pid} = \perp$) to observe or program. Furthermore, each ideal functionality can only observe or program the responses to queries that encode (say, start with) the session ID of that ideal functionality.

In addition to avoiding said weakness, the new $\mathcal{F}_{\mathsf{GRO}}$ is also more expressive: first, it enables ideal functionalities to determine context-specific policies for when queries can be observed and when responses can be programmed, thus capturing more nuanced security guarantees. It also allows situations where different sessions allow different observing/programming behaviors of the random oracle. Finally, it enables ideal functionalities to be explicit about the properties they expect from the random oracle.

UC non-interactive shrinking commitments. An ideal functionality that captures non-interactive commitments was recently proposed in [CSW22]. Unlike the more traditional commitment functionality of [CF01] where commitment and decommitment strings are implicit even for non-interactive commitments, here the ideal commitment functionality explicitly outputs commitment and decommitment strings to the committer in the commitment phase. This enables the modular decomposition of non-interactive protocols which involve computing on commitment strings. ([CSW22] constructs a non-interactive zero-knowledge proof in which a commitment string is input to a correlation-intractable hash function.) However, the [CSW22] formulation of commitments is geared towards realization with a common reference string. In particular, it is not realizable by protocols where the commitment string is shorter than the message.[1]

We extend the non-interactive commitment functionality of [CSW22] to interact with $\mathcal{F}_{\mathsf{GRO}}$. On the positive side, this extension enables realizations where the length of the commitment depends only on the security parameter, irrespective of the length of the message. It also allows us to use a significantly simpler formulation where each instance of the commitment functionality, $\mathcal{F}_{\mathsf{NC}}$, handles only a single commitment. (In contrast, all prior work on UC commitments have a non-global setup that is reused over multiple commitments, which in turn mandates using the more complex sub-session formalism for handling multiple commitments within the same instance of the functionality.)

On the negative side, allowing $\mathcal{F}_{\mathsf{NC}}$ to interact with the global $\mathcal{F}_{\mathsf{GRO}}$ opens the door to potential "backdoors" where $\mathcal{F}_{\mathsf{NC}}$ inadvertently leaks information to an external observer via its interaction with $\mathcal{F}_{\mathsf{GRO}}$. This danger is actually quite imminent, given that $\mathcal{F}_{\mathsf{NC}}$ uses simulator-provided code to generate strings representing ideal commitment and decommitment, and furthermore that the code can observe and program $\mathcal{F}_{\mathsf{GRO}}$. Indeed, making sure that no such "backdoors" are possible adds significant subtlety to the structure of $\mathcal{F}_{\mathsf{NC}}$. Our treatment here is inspired by that of [CDLLR24].

At high level, $\mathcal{F}_{\mathsf{NC}}$ provides the following service. When asked by a committer C to commit to a message m, $\mathcal{F}_{\mathsf{NC}}$ returns to C a commitment string cm and a decommitment string dcm. When asked by anyone to verify a triple $(m', \mathsf{cm}', \mathsf{dcm}')$, $\mathcal{F}_{\mathsf{NC}}$ accepts or rejects. Both the generation of $(\mathsf{cm}, \mathsf{dcm})$ and the accept/reject response use programs that are provided by the adversary (or, simulator) and are executed by $\mathcal{F}_{\mathsf{NC}}$ in a controlled way. Overall, the security guarantees are:

Secrecy: The commitment string cm is generated independently of m. Furthermore, even though the process of generating the decommitment string dcm depends on m, it does not leak information on m to an external observer. In fact, an external observer that interacts with $\mathcal{F}_{\mathsf{NC}}$ but is given only cm, but not

[1] Consider a UC simulator $\mathcal{S}$ for a succinct non-interactive commitment scheme in the common reference string model. Then $\mathcal{S}$ would need to win the following game with probability negligibly close to 1: $\mathcal{S}$ first generates a simulated reference string s. Next a random n-bit message m is chosen, and $\mathcal{S}$ obtains an honestly generated commitment c to m (relative to s), where c is an n'-bit string. $\mathcal{S}$ wins if it correctly guesses m. Clearly, if $n' < n$ then $\mathcal{S}$ wins with probability at most one half.

dcm, cannot guess (except for negligible probability) whether cm is a commitment to the message provided by the observer or else a commitment to some arbitrary fixed value.

Completeness and Consistency: When asked to verify a triple that was generated earlier by $\mathcal{F}_{\mathsf{NC}}$ in a response to a commitment request, $\mathcal{F}_{\mathsf{NC}}$ always accepts. Also, two requests to verify the same triple are always answered in the same way.

Binding and non-malleability: Any string cm can be opened to at most one message. Furthermore, whenever $\mathcal{F}_{\mathsf{NC}}$ accepts an opening $(m, \mathsf{cm}, \mathsf{dcm})$, we are guaranteed that there exists an explicitly determined earlier point in the execution, such that the state of the system at that point uniquely and efficiently determines m.

Immediate response: $\mathcal{F}_{\mathsf{NC}}$ responds to both commit and verify requests immediately, namely without the environment being activated in between request and response. This guarantees that both commitment and verification are local computational processes that do not involve interaction.

A few comments are in order: First, it may seem odd that $\mathcal{F}_{\mathsf{NC}}$ allows for a non-zero probability of guessing the plaintext and decommitment string. However, this is an inherent weakness of non-interactive commitments: a lucky guess of the correct decommitment string, given the commitment string and a conjectured value for the underlying message, will inevitably be recognized by the decommitment algorithm.

Second, the "extraction point" mentioned in the binding guarantee is determined as follows: If cm equals a value returned earlier by $\mathcal{F}_{\mathsf{GRO}}$, then the extraction point is at the first such event. Otherwise, the extraction point is the first activation of $\mathcal{F}_{\mathsf{NC}}$. This means that a protocol using $\mathcal{F}_{\mathsf{NC}}$ has the guarantee that, as soon as any party comes up with a string cm, cm can only be opened to a fixed (albeit hidden) plaintext.

We defer the descriptions of *how* $\mathcal{F}_{\mathsf{NC}}$ provides the above guarantees to the body of the paper, which also provides precise formulations of the above properties, along with formal proofs that these properties hold with respect to $\mathcal{F}_{\mathsf{NC}}$. This process of formulating mathematical claims regarding security properties *of ideal functionalities,* together with rigorous proofs of these claims, can be viewed as an additional contribution of this work.

Realizing $\mathcal{F}_{\mathsf{NC}}$. We show that the following protocol (which is implicit in [Mic00], and may be considered folklore), unconditionally UC-realizes $\mathcal{F}_{\mathsf{NC}}$: To commit to a message m, choose a random κ-bit string r, compute $c \leftarrow \mathcal{F}_{\mathsf{GRO}}(\mathsf{sid}, m, r)$, where sid is the relevant session ID for this particular commitment, and let c, r be the commitment and decommitment strings for m. To verify that r decommits c to message m with respect to session ID sid, check that $c = \mathcal{F}_{\mathsf{GRO}}(\mathsf{sid}, m, r)$. (The inclusion of sid in the input to $\mathcal{F}_{\mathsf{GRO}}$ cryptographically binds the commitment string to a specific instance. This binding is crucial for our modular analysis.)

Our proof of security is unconditional, and the simulation error is bounded by $t \cdot 2^{-\kappa} + (2t^2 + 3t) \cdot 2^{-\lambda}$, where t is the number of oracle queries made by the environment, λ is the length of $\mathcal{F}_{\mathsf{GRO}}$ outputs, and κ is the min-entropy of decommitments. The proof also trivially addresses the case of adaptive corruption of the committer. Indeed, the commitment stage is immediate (i.e., it takes a single activation), and no secret information is generated (other than the decommitment string itself, which is part of the output of the commitment stage).

UC non-interactive vector commitments. Vector commitments [CF13] allow committing to a vector $\mathbf{m}$ of values, then decommitting to the values in different positions in the vector at different times. To capture the relevant security properties in an abstract way, we formulate an ideal functionality $\mathcal{F}_{\mathsf{NVC}}$ that provides the following service: when asked by a committer C to commit to a plaintext vector $\mathbf{m} \in (\Sigma \cup \bot)^n$, $\mathcal{F}_{\mathsf{NVC}}$ returns to C a commitment string cm, along with a decommitment string $\mathbf{dcm}[i]$ for each index i where $\mathbf{m}[i] \in \Sigma$. When asked by anyone to verify a tuple $(\mathsf{cm}', i', m', \mathsf{dcm}')$, in which $m' \in \Sigma$ is the claimed value of entry i' in $\mathbf{m}$ and dcm' is the claimed decommitment with respect to cm', $\mathcal{F}_{\mathsf{NVC}}$ either accepts or rejects. The security guarantees of $\mathcal{F}_{\mathsf{NVC}}$ extend those of $\mathcal{F}_{\mathsf{NC}}$ to the vector setting, as follows:

Secrecy: The commitment string cm is generated independently of the committed vector $\mathbf{m}$. Furthermore, each decommitment string $\mathbf{dcm}[i]$ is generated independently of $\{\mathbf{m}[i'] : i' \neq i\}$. In addition, even though the process of generating each decommitment string $\mathbf{dcm}[i]$ depends on $\mathbf{m}[i]$, it does not leak information on $\mathbf{m}[i]$ to an external observer: an external observer that interacts with $\mathcal{F}_{\mathsf{NVC}}$ but is given only cm and $\{\mathbf{dcm}[i'] : i' \neq i\}$, but not $\mathbf{dcm}[i]$, cannot guess (except for negligible probability) whether $\mathsf{cm}[i]$ is the message provided by the observer or else some arbitrary fixed value.

Completeness and Consistency: When asked to verify a tuple $(\mathsf{cm}, i, m, \mathsf{dcm})$ that is part of the commitmentment and decommitment information generated in the commit stage, $\mathcal{F}_{\mathsf{NVC}}$ always accepts. Furthermore, two requests to verify the same tuple are always answered in the same way.

Binding and non-malleability: Any string cm can be opened to at most one message in each coordinate. More precisely, whenever $\mathcal{F}_{\mathsf{NC}}$ accepts an opening $(\mathsf{cm}, i, m, \mathsf{dcm})$ we are guaranteed that there exists an explicitly determined earlier point in the execution such that the state of the system at that point uniquely and efficiently determines an *entire vector* $\mathbf{m} \in (\Sigma \cup \bot)^n$, where $\mathbf{m}[i] = m$.

Immediate response: $\mathcal{F}_{\mathsf{NC}}$ responds to both commit and verify requests immediately, namely without the environment being activated in between request and response. This captures the fact that both commitment and verification are local computational processes that do not involve interaction.

The above discussion of $\mathcal{F}_{\mathsf{NC}}$ is applicable here as well. In addition, we emphasize that the binding and non-malleability properties are strong ones: successful decommitment of a commitment string cm to *even a single element* (i, m) implies that the *entire* plaintext vector $\mathbf{m} \in (\Sigma \cup \bot)^n$ has been fixed at the "extraction point."

Realizing $\mathcal{F}_{\mathsf{NVC}}$. We realize $\mathcal{F}_{\mathsf{NVC}}$ using a variant of the Merkle tree construction where both the leaves and the intermediary nodes are instantiated via instances of $\mathcal{F}_{\mathsf{NC}}$. The instances of $\mathcal{F}_{\mathsf{NC}}$ that correspond to intermediary nodes have plaintext that is 2λ bits long and commitment that is λ bits long. The session ID for each instance of $\mathcal{F}_{\mathsf{NC}}$ includes the session ID of $\mathcal{F}_{\mathsf{NVC}}$, together with an identifier that indicates the position of this instance in the commitment tree.

Note that this variant is somewhat different than the standard Kilian-Micali construction, in which commitments happen only at the leaves and internal nodes are implemented via a simple shrinking hash function. Our construction incurs a small cost in commitment size, but has the advantage of being modular, namely it only needs access to $\mathcal{F}_{\mathsf{NC}}$. Furthermore, it *perfectly* realizes $\mathcal{F}_{\mathsf{NVC}}$, without any loss in security - a feature which more than compensates for the increase in commitment size, towards minimizing the overall overhead needed to obtain a given security level.

Like for plain commitment, the proof of security of this protocol trivially addresses the case of adaptive corruption of the committer: The commitment stage is immediate (i.e., it takes a single activation), and no secret information is generated other than the decommitment string itself, which is part of the output of the commitment stage.

Constructing a simulator for this protocol encounters the following technical difficulty: In order to provide the environment with a view that is identical to an execution of the protocol, the simulator code within $\mathcal{F}_{\mathsf{NVC}}$

should somehow cause $\mathcal{F}_{\mathsf{NVC}}$ to program points that correspond to the reserved spaces of the individual commitments; however, these spaces are outside the domain that $\mathcal{F}_{\mathsf{NVC}}$ is allowed to program. We get around this issue by devising a mechanism for $\mathcal{F}_{\mathsf{NVC}}$ to spawn gateway functionalities that perform the appropriate programming, while still keeping the simulator code appropriately "sandboxed". The same mechanism is used also for proving security of the application to UC SNARKS below.

Application to UC ZK SNARKs. To exhibit the usefulness of $\mathcal{F}_{\mathsf{NVC}}$, we use it to provide modular analysis of the Kilian-Micali ZK-SNARK. To this end, we first adapt the non-interactive zero knowledge functionality of [CF24] to our modeling, where the ideal functionality is the one interacting with $\mathcal{F}_{\mathsf{GRO}}$ for observing and programming.

Next, we show how to turn any honest verifier zero knowledge extractable PCP (e.g. [KPT97; IMS12]) into a ZK-SNARK along the lines of the Kilian-Micali construction.

The proof is unconditional and relatively straightforward. It makes crucial use of the fact that even a single successful opening of an element in a vector commitment implies that the entire vector has been fixed, and furthermore extracted. There are four sources of error, all inevitable: The probability of guessing a decommitment of an ideal vector commitment, the probability of finding an ideal vector commitment string that yields a bad Fiat-Shamir challenge, the soundness (or, extraction) error of the underlying PCP, and the zero knowledge error of the underlying PCP.

We also extend the analysis to provide security in case of adaptive corruption of the prover. Specifically, we define a natural extension of the notion of zero-knowledge PCP to handle adaptive corruption of the PCP prover, and argue that most existing Zero Knowledge PCPs are already adaptive zero-knowledge. We then show that, as long as the underlying PCP has the adaptive zero-knowledge property, the constructed protocol UC realizes $\mathcal{F}_{\mathsf{NIZK}}$ in the presence of adaptive corruptions.

More potential uses of $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{NVC}}$. While the analysis of the Kilian-Micali ZK-SNARK demonstrates the validity of the formalism, we hope that $\mathcal{F}_{\mathsf{NVC}}$ will be valuable for more modular analysis of applications that use vector commitments, or more generally Merkle trees – especially in contexts that require the combination of soundness, secrecy, and succinctness.

One natural target is modular analysis of ZK-SNARKs that are built from PCPs/IOPs and *polynomial commitments,* which is a special case of vector commitments in which the committed vector has polynomial structure. Indeed, prior work analyzes ZK-SNARKs based on polynomial commitments monolithically.

Additionally, a substantial body of literature validates [CF13]'s proposed uses of vector commitments, including: verifiable and updatable databases (for secret data), dynamic accumulators (of secret data), and anonymous credentials.

Additional schemes that make crucial use of Merkle trees and could potentially benefit from the $\mathcal{F}_{\mathsf{NVC}}$ abstraction include constructions of proofs of sequential work [MMV13; CP18], key verification for end users [MBBFF15], proofs of space (or storage) [DFKP15; Pro17], and private blockchains such as Zerocash [Ben+14].

A more recent and exciting application of vector commitments is constructing incrementally verifiable computation (IVC) and proof-carrying data (PCD). The ability to model vector commitments as an ideal functionality with universal composability guarantees would naturally strengthen (and potentially simplify the analysis of) any of these applications. For example, the recent work of [BMNW25] shows how to generically construct PCDs from vector commitments and accumulators of low depth.

1.3 Additional related work

Other functionalities for non-interactive primitives. [CDLLR24] proposes a non-interactive (digital) signature functionality, which enables a modular analysis of the Dolev-Strong broadcast protocol.

Non-interactive zero-knowledge functionality $\mathcal{F}_{\mathsf{NIZK}}$. The $\mathcal{F}_{\mathsf{NIZK}}$ formulation in this work, as well as in [CF24; GKOPTT23; CSW22] essentially all follow that of [GOS06; GOS12]. All works have straight-line extraction; this is necessary for proving UC security. The differences are:

1. **Simulator interaction.** In [CSW22; GKOPTT23], $\mathcal{F}_{\mathsf{NIZK}}$ receives the simulated proof string and the extracted witness directly from the simulator, whereas in this work $\mathcal{F}_{\mathsf{NIZK}}$ runs code that was given to it by the simulator ahead of time. The latter formulation is somewhat stronger in that it formally guarantees that the proof generation and verification are done locally by the parties without network interaction. Similarly, in [CF24], the simulator's algorithms are fixed in the setup phase. The only difference is that we use the global code library $\mathcal{F}_{\mathsf{lib}}$ formalism.
2. **Multi vs. single session.** In [CSW22; GKOPTT23], $\mathcal{F}_{\mathsf{NIZK}}$ is multi-session (due to the use of a common reference string (CRS)), whereas in this work and in [CF24], $\mathcal{F}_{\mathsf{NIZK}}$ is single session.
3. **Interface with $\mathcal{F}_{\mathsf{GRO}}$.** In [GKOPTT23; CF24] and this work, the security of $\mathcal{F}_{\mathsf{NIZK}}$ is considered with respect to a global $\mathcal{F}_{\mathsf{GRO}}$ as discussed above.[2] Recall that [CF24] uses the observable and restricted programmable $\mathcal{F}_{\mathsf{GRO}}$ from [CDGLN18], which has an $\mathsf{isProgrammed}$ interface for "detecting" whether a point was programmed by the adversary and has a consistent sid. Thus, their $\mathcal{F}_{\mathsf{NIZK}}$ (or Π_{NIZK}) invokes $\mathsf{isProgrammed}$ to detect bad behavior. (Further discussion of this point appears in Section 3 and in Appendix A.)

Additional comparison with [CF24]. Our security analyses follow different approaches; [CF24] defines properties of the Kilian-Micali protocol that protect against particular classes of attacks in the UC setting, then uses these properties to show that the protocol UC-realizes $\mathcal{F}_{\mathsf{NIZK}}$. In contrast, we consider a version of Kilian-Micali that is composed of a PCP plus the idealized subroutine $\mathcal{F}_{\mathsf{NVC}}$, which is itself realized using instances of another idealized subroutine $\mathcal{F}_{\mathsf{NC}}$. Consequently, our UC analysis involves defining properties for the idealized subroutines, proving that the subroutines indeed satisfy them, then using these properties to prove that Kilian-Micali is UC-secure.

In fact, this work analyzes a slightly different version of the Kilian-Micali protocol than [CF24]. Specifically, [CF24] considers the "plain" Kilian-Micali protocol: a PCP/IOP plus a Merkle tree, for which nodes are computed with random oracle queries are in the same domain. i.e. are prefixed by the same session ID. In comparison, our analysis of Kilian-Micali replaces the Merkle tree with the idealized vector commitment subroutine $\mathcal{F}_{\mathsf{NVC}}$, which is, in turn, realized using a Merkle tree that has a different instance of $\mathcal{F}_{\mathsf{NC}}$ at each node. This means that each node has a different session ID, which in turn rules out collisions across different nodes. This results in a simpler and tighter analysis.[3]

[2] In [GKOPTT23], there is no explicit interaction between $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{GRO}}$. In [CF24] and this work, this interaction is made explicit. However, this is only a syntactic difference due to the fact that in [GKOPTT23] the adversary can directly ask $\mathcal{F}_{\mathsf{GRO}}$ to observe queries as opposed to routing these requests via $\mathcal{F}_{\mathsf{NIZK}}$.

[3] It is of course possible to consider a version of the Kilian-Micali protocol for which the nodes' queries in the Merkle tree are domain separated, even without resorting to UC security. This in turn enables an even tighter security analysis that combines properties of the underlying PCP to demonstrate that successful cheating would require specific patterns of collisions, as done in [CY21]. At the same time, even this tighter analysis could still benefit from the modularity provided by $\mathcal{F}_{\mathsf{NC}}$ at the nodes; indeed, one could define the tree soundness game of [CY21] with respect to schemes that use $\mathcal{F}_{\mathsf{NC}}$.

Our modular treatment of Kilian-Micali yields somewhat improved concrete parameters in the security analysis. We demonstrate degradation in security that is only linear in n, the size of the underlying PCP, and only quadratic in t, the overall number of oracle calls of the environment. This is in contrast to quadratic dependence on n and cubic dependence on t in [CF24]. We attribute this improvement to our modular treatment of vector commitments. In particular, the environment's distinguishing advantage in this work does not depend on the (knowledge) extraction error of the Merkle tree. This error term is unavoidable in the analysis of [CF24].

Simulation extractable SNARKs from polynomial IOPs. One method for constructing SNARKs is to start by constructing an (information theoretic) polynomial interactive oracle proof (PIOP), transforming it into an interactive arugment using (cryptographic) polynomial commitments, then achieving non-interactivity by applying the Fiat-Shamir transform [BCS16; CHMMVW20; BFS20; CFFQR21]. [KPT23; FFKRZ23] investigate when such SNARKs are simulation extractable. [KPT23] identifies a framework for proving simulation extractability, by isolating sufficient properties of the PIOP and polynomial commitment. [FFKRZ23] focus specifically on properties needed from the polynomial commitments. Both approaches appear to be different than ours. In particular they provide different abstactions and require using both a common reference string (or the algebraic group model) and the random oracle model; they also do not consider UC security. In comparison, this work studies how to construct a UC-secure SNARK when only relying on a global random oracle.

Other analyses of the soundness of Kilian-Micali. We list some recent (but orthogonal) works studying the soundness of Kilian-Micali in various settings.

[CY21] improves upon the soundness analysis of the (non-interactive) [Mic00] SNARG; they do this by formulating an information-theoretic game, which that captures "the prover's probability of breaking soundness, given the ability to find certain number of random oracle inversions and random oracle collisions in the Merkle tree".

[CDGS24] analyzes the security of Kilian's protocol, which is the *interactive* version of the zkSNARK considered in this work, given any PCP and any vector commitment scheme (i.e. not necessarily a Merkle tree). They also consider of the soundness of the *interactive* version of [BCS16] (i.e. public-coin IOP plus vector commitments, but no Fiat-Shamir).

Angel-based security and the shielded simulators framework. Formalizing the random oracle model as a global ideal functionality can be viewed as somewhat reminiscent of other frameworks for relaxing the UC model, specifically angel-based security and the shielded simulators framework [PS04; BDHMQN17]. However, these approaches differ from the present one in that they present the protocol in question as-is, and only modify the security experiment. In contrast (and in line with the random oracle methodology), here we only analyze an abstract version of the actual protocol (i.e. described in the presence of $\mathcal{F}_{\mathsf{GRO}}$) and say nothing about potential transitions of the analyzed protocol to fully specified ones (i.e. replacing $\mathcal{F}_{\mathsf{GRO}}$ with a protocol). In particular, general uninstantiability results for protocols in the random oracle model (such as those in [CGH04]) apply to our protocols as well.

2 Preliminaries

2.1 Notations

We notate $[n] := \{1, \ldots, n\}$.

Min-entropy. A random variable X has min-entropy h if $\max_x \Pr[X = x] = 2^{-h}$.

Conditional min-entropy. Let X, Y be random variables. Then, the min-entropy of X conditioned on Y is $-\log \max_{x,y} \Pr[X = x | Y = y] = \min_y H_\infty(X | Y = y)$.

2.2 UC with Global Subroutines

Throughout, we use the UC model from [Can20], together with the modeling from [BCHTZ20] of global subroutines within the plain UC model.

UC Theorem. Suppose a protocol Π UC-realizes an ideal functionality $\mathcal{F}$. Further suppose ρ is a protocol that uses $\mathcal{F}$ as a subroutine. Then, the UC composition theorem states that $\rho^{\mathcal{F} \to \Pi}$ UC-emulates ρ, in which $\rho^{\mathcal{F} \to \Pi}$ denotes replacing all instances of an ideal functionality $\mathcal{F}$ with a protocol Π in ρ.

UC with Global Subroutines. While the UC theorem is extremely powerful, it doesn't directly apply in cases when multiple ideal functionalities or protocols have access to a common (global) subroutine (e.g. in this work, the global random oracle). The Global Subroutines Theorem of [BCHTZ20] provides a way to apply the UC theorem even in such cases. We briefly recall the Global Subroutines theorem below.

We first define the transformation M, a.k.a. the management protocol, which takes as input two protocols $\mathcal{F}, \mathcal{G}$ (e.g., $\mathcal{G}$ is a global subroutine) that have distinct session IDs, and outputs a unified protocol $\mathsf{M}[\mathcal{F}, \mathcal{G}]$ with a unified session ID such that:

- $\mathsf{M}[\mathcal{F}, \mathcal{G}]$ acts as an interface between both $\mathcal{F}$ and $\mathcal{G}$ to all other sessions;
- the behaviors of $\mathcal{F}$ and $\mathcal{G}$ stay the same in $\mathsf{M}[\mathcal{F}, \mathcal{G}]$;
- $\mathsf{M}[\mathcal{F}, \mathcal{G}]$ supports the dynamic generation of sessions of $\mathcal{F}$ or $\mathcal{G}$; and
- the environment has access to $\mathcal{G}$.

See Section 3.1 of [BCHTZ20] for further details.

Given M, we define what it means to "UC-emulate a protocol in the presence of a global subroutine" and give the UC composition theorem in the presence of global subroutines.

Definition 2.1. *Let* $\Pi, \mathcal{F}, \mathcal{G}$ *be protocols. We say that* Π *UC-emulates* $\mathcal{F}$ in the presence *of global subroutine* $\mathcal{G}$ *if* $\mathsf{M}[\Pi, \mathcal{G}]$ *UC-emulates* $\mathsf{M}[\mathcal{F}, \mathcal{G}]$.

Theorem 2.2 (UC Composition in the presence of a global subroutine [BCHTZ20])**.** *There exists a code transformation* $\mathsf{M}[\cdot]$ *such that, given protocols* $\Pi, \mathcal{F}, \mathcal{G}$*, we have that* $\mathsf{M}[\Pi, \mathcal{G}]$ *UC-emulates* $\mathsf{M}[\mathcal{F}, \mathcal{G}]$. *Then for (essentially) any protocol* ρ*, we have* $\rho^{\mathcal{F} \to \Pi}$ *UC-emulates* ρ.

Writing convention for messages and session IDs. Throughout this work we rely on the convention from [Can20], in which input and output messages always contain the full identities of the sender and the receiver machines, as well as the session ID. For ease of reading, we keep the sender and receiver identities implicit whenever they are clear from the context.

2.3 $\mathcal{F}_{\mathsf{lib}}$: Global code library functionality

We use a variant of the global code library functionality $\mathcal{F}_{\mathsf{lib}}$ from [CJSV22] (Figure 1). The functionality provides a repository to which the simulator $\mathcal{S}$ may upload code, which is meant to be used by one or more ideal functionalities.

Functionality $\mathcal{F}_{\mathsf{lib}}$

$\mathcal{F}_{\mathsf{lib}}$ maintains an (initially empty) list $\mathsf{Library}$ of records, where each record is a tuple $(\mathsf{Identifier}, \mathsf{Code}, \mathsf{Links})$, Code is a program that potentially contains calls to unspecified subroutines whose identifiers appear in the list Links, and $\mathsf{Identifier}$ is an identifier for this program.

Obtain code.

1. Upon receiving a message $(\texttt{Store}, \mathsf{Identifier}, \mathsf{Code})$ from simulator $\mathcal{S}$, compile the list Links of all the identifiers in Code of unspecified subroutines, and add the record $(\mathsf{Identifier}, \mathsf{Code}, \mathsf{Links})$ to $\mathsf{Library}$.

Deliver code. Upon receiving $(\texttt{CodeRequest}, \mathsf{Identifier})$ from a party P, do:

1. Find the latest record Record in $\mathsf{Library}$ such that $\mathsf{Record} = (\mathsf{Identifier}, \mathsf{Code}, \mathsf{Links})$.
2. Recursively find all the records that correspond to the programs whose identifiers are listed in Links. Keep only the latest record for each such identifier. Let $\mathsf{LinkedRecords}$ be the list of all the found records.
3. Return $(\texttt{Answer}, \mathsf{LinkedRecords})$ to P.

Figure 1: The global functionality $\mathcal{F}_{\mathsf{lib}}$, representing the library of programs provided by the adversary to ideal functionalities in the system.

To simplify the description of simulator-generated code that is meant to be used in a nested way (namely, for the simulation of protocols that use ideal functionalities as subroutines, where these ideal functionalities use their own simulator code, which in turn simulates subroutine protocols that use yet other ideal functionalities that use simulator code, etc.), we view $\mathcal{F}_{\mathsf{lib}}$ as a global library of programs that serves multiple (potentially nested) protocols, and that supports dynamic linking of code. That is, the simulator for some protocol may upload code that calls subroutines that are only specified by their name (identifier). The actual code of these subroutines may then be uploaded separately in the context of a different protocol instance.

When some party asks $\mathcal{F}_{\mathsf{lib}}$ to retrieve the code associated with some identifer, $\mathcal{F}_{\mathsf{lib}}$ instantiates all relevant unspecified subroutines that exist in the library and returns the resulting linked program.

3 $\mathcal{F}_{\mathsf{GRO}}$: Ideal functionality for global random oracle

We describe our proposed ideal functionality $\mathcal{F}_{\mathsf{GRO}}$ for representing a global random oracle $\{0,1\}^* \to \{0,1\}^\lambda$. (We think of λ as the security parameter.) As with prior formulations of the global random oracle (e.g. [CJS14; CDGLN18; LR22; CF24]), the goal is to capture a globally available random oracle that enables observability and programmability and, at the same time, enables meaningful security-preserving protocol composition.

In prior works, the global programmable random oracle functionality allows anyone (in particular, the adversary) to program the oracle at any point that has not yet been queried. We first demonstrate that this modeling approach carries an inherent security weakness: it allows the simulator for a single protocol instance to "pollute" the random oracle everywhere, and thus create a situation where unsuspecting protocol instances experience a different state of the random oracle, depending on whether the "polluting" protocol instance is present.

Concretely, consider the formulation of programmable RO in [CDGLN18], which is used in [LR22; CF24]. Let π be a protocol that UC realizes an ideal functionality $\mathcal{F}$ in the presence of the global RO. We demonstrate a toy protocol ρ that uses subroutine calls to $\mathcal{F}$, and loses all security as soon as π is substituted for $\mathcal{F}$. See details in Appendix A.

To avoid "polluting simulators," we don't allow the simulator to directly program the oracle. Instead, we only allow ideal functionalities to observe and program the global oracle; furthermore, we only allow ideal functionality to observe/program the oracle at inputs that encode the session id of that functionality. In addition to preventing widespread "pollution" of programmed points, this modeling allows an ideal functionalities to set rules for how a simulator can interact with the global random oracle at points associated with them.

More specifically, we let an ideal functionality (i.e., a machine whose party ID is $\perp$) request $\mathcal{F}_{\mathsf{GRO}}$ to "control its session identifier". Now, whenever some party P asks $\mathcal{F}_{\mathsf{GRO}}$ to evaluate the oracle at a point x that decodes to a pair $x = (\mathsf{sid}, x')$ (using some pre-determined uniquely decodable encoding scheme) and sid has a registered controller P', $\mathcal{F}_{\mathsf{GRO}}$ tells the controller that P asked x. If x is a new query, then $\mathcal{F}_{\mathsf{GRO}}$ also lets P' determine the response. See Figure 2 for the description of $\mathcal{F}_{\mathsf{GRO}}$.

For one, observe that this formulation of $\mathcal{F}_{\mathsf{GRO}}$ gets around the weakness described in Appendix A, as even a rogue $\mathcal{F}$ can only program points within its own input space. Furthermore, it provides for more expressive modeling of $\mathcal{F}_{\mathsf{GRO}}$, in that it allows individual ideal functionalities to judiciously delegate the ability to observe and program points under its control to its simulator, under functionality-specific criteria.

We also note that our formalism does not interfere with the ability of ideal functionalities to guarantee immediate response of any implementing protocol. In particular the processing of an oracle query always completes without having to "yield control" to the environment. This property, which is new to the present modeling of a global random oracle, is used extensively throughout.

Functionality $\mathcal{F}_{\mathsf{GRO}}$

I **Parameters:** A security parameter λ. # Length of $\mathcal{F}_{\mathsf{GRO}}$ outputs.

II **Initialization:** Upon first activation, set the below as empty (i.e. as $\perp$):

- Hist: a list of query-response pairs (x, y), in which $x \in \{0,1\}^*$ is a query point and $y \in \{0,1\}^\lambda$ is a query answer. # Function evaluation table for the random oracle.
- Control: a list containing tuples of the form (sid, P), in which sid is a session ID and P is the party that controls sid.

III **Assigning control:** Receive input $(\texttt{Control})$ from a party $P = (\mathsf{sid}, \mathsf{pid})$. If $\mathsf{pid} = \perp$, then add sid to Control and return $(\texttt{Ok}, \mathsf{Hist}[\mathsf{sid}])$ to P, in which Hist[sid] is the subset of Hist for which queries have the prefix sid. Else, return $\perp$. # If P is an ideal functionality, record sid as controlled.

IV **Evaluate:** Receive $(\texttt{Eval}, x)$ from a party P.

(a) If x has the form $x = (\mathsf{sid}, x')$ and P' is the controller of sid, output $(\texttt{ProgramReq}, P, x)$ to P'. # Allow the party that controls sid to observe the query and program the output.

(b) Else: # No one controls sid.

i. If there is no record of the form $(x, \cdot) \in \mathsf{Hist}$, then sample $y \leftarrow \{0,1\}^\lambda$ and add (x, y) to Hist. # Lazily sample the output value y.

ii. Output $(\texttt{EvalOutput}, y)$ to P.

V **Program:** Receive $(\texttt{Program}, x, y)$ from P.

(a) If x is not of the form $x = (\mathsf{sid}, x')$ or P is not the controller of sid, then end the activation.

(b) If there is no record of the form $(x, \cdot) \in \mathsf{Hist}$, then add (x, y) to Hist. # Only program queries without a determined output value.

(c) Output $(\texttt{EvalOutput}, y)$ to P, where (x, y) is the entry in Hist corresponding to x.

Figure 2: The global random oracle functionality.

Observing and programming $\mathcal{F}_{\mathsf{GRO}}$. We note that the present formulation does not make a formal distinction between observing a query and "programming" the response. $\mathcal{F}_{\mathsf{GRO}}$ always allows the controller of a session ID sid to both observe and program the response for all the correctly formatted queries $x = (\mathsf{sid}, x')$.

This formulation is geared towards a modeling where the controller parties are ideal functionalities, so the decisions of when and how to program become an explicit part of the security specification of the task that is modelled by the ideal functionality. For any queries $x = (\mathsf{sid}, x')$ such that sid doesn't have a controller, $\mathcal{F}_{\mathsf{GRO}}$ lazily samples the output value, as per the usual formulation of the random oracle.

Finally we define shorthand notation for querying $\mathcal{F}_{\mathsf{GRO}}$.

Definition 3.1. *For a query point x, the notation $\mathsf{GRO}(x)$ denotes the following:*

1. Invoke $\mathcal{F}_{\mathsf{GRO}}$ with subroutine input $(\mathtt{Eval}, x)$.
2. Receive the subroutine output $(\mathtt{EvalOutput}, y)$ from $\mathcal{F}_{\mathsf{GRO}}$.

4 $\mathcal{F}_{\mathsf{NC}}$: Ideal functionality for non-interactive commitment

We introduce an ideal functionality for non-interactive commitment $\mathcal{F}_{\mathsf{NC}}$ in the presence of global subroutine $\mathcal{F}_{\mathsf{GRO}}$. Our motivation for defining a *non-interactive* version of the commitment functionality is the same as in [CSW22]: to describe protocols realizing the non-interactive zero-knowledge functionality $\mathcal{F}_{\mathsf{NIZK}}$. The ideal commitment functionality $\mathcal{F}_{\mathsf{Com}}$ from [CF01] does not suffice for this task because $\mathcal{F}_{\mathsf{Com}}$ never issues commitment and decommitment strings. In contrast, the ability to further process a commitment string (specifically, to hash it, along with other information) is essential for the Kilian-Micali protocol. The ideal non-interactive commitment functionality from [CSW22] also does not suffice because it does not interact with a global random oracles and hence does not generate a succinct commitment.

Functionality $\mathcal{F}_{\mathsf{NC}}$ appears in Figures 3 and 4. We overview of how it is structured.
Upon first activation with session ID sid, $\mathcal{F}_{\mathsf{NC}}$ first asks $\mathcal{F}_{\mathsf{GRO}}$ to take control of sid and aborts if the request does not succeed. $\mathcal{F}_{\mathsf{NC}}$ then creates an initially-empty list $\mathsf{Hist}_{\mathsf{GRO}}$ of queries to $\mathcal{F}_{\mathsf{GRO}}$ and the corresponding responses. $\mathsf{Hist}_{\mathsf{GRO}}$ will contain all the queries to $\mathcal{F}_{\mathsf{GRO}}$ made by entities other than $\mathcal{F}_{\mathsf{NC}}$ (namely, the environment) to values controlled by $\mathcal{F}_{\mathsf{NC}}$, specifically queries of the form $x = (\mathsf{sid}, x')$ and the query responses. Note that $\mathcal{F}_{\mathsf{NC}}$ instructs $\mathcal{F}_{\mathsf{GRO}}$ to return random values as the responses to all such external GRO queries, i.e. lazily sample; namely, it does not "program" any externally queried point.

Next, $\mathcal{F}_{\mathsf{NC}}$ uses a set of adversarially generated "simulator" algorithms SCode, which includes three algorithms:

- $\mathsf{Com}(\mathsf{sid}, r_{\mathsf{cm}}) \to (\mathsf{cm}, \mathsf{st})$: On input a session ID sid and randomness r_{cm}, Com outputs a commitment string cm and internal state st (to be handed over to Equiv).
- $\mathsf{Equiv}(\mathsf{sid}, \mathsf{cm}, \mathsf{st}, m, r_{\mathsf{dcm}}) \to (\mathsf{dcm}, \mathsf{State}^{\mathsf{NC}})$: On input a session ID sid, a commitment string cm, an internal state st from Com, a message m, and randomness r_{dcm} from the ideal functionality, Equiv outputs a decommitment string dcm and a committer state $\mathsf{State}^{\mathsf{NC}}$ which is returned in case of adaptive corruption.
- $\mathsf{Extr}(\mathsf{sid}, \mathsf{cm}, Q) \to (m, \mathsf{dcm})$: On input a session ID sid, a commitment cm, and a list of $\mathcal{F}_{\mathsf{GRO}}$ query-answer pairs Q, deterministically outputs a message m and a decommitment dcm.

Each instance of $\mathcal{F}_{\mathsf{NC}}$ generates only a single commitment. To account for fact that commitments can be generated by anyone and furthermore that verification of potential decommitments is a non-interactive process, $\mathcal{F}_{\mathsf{NC}}$ allows verifying (and possibly accepting) the decommitments of multiple commitment strings with the same sid.

To generate a commitment for an input message m, $\mathcal{F}_{\mathsf{NC}}$ first runs Com to generate the commitment string obliviously of m, and then runs Equiv to generate the decommitment string to m. Both Com and

Equiv may program $\mathcal{F}_{\mathsf{GRO}}$, as long as the query points to be programmed have sufficient min-entropy. These conditions are enforced by ensuring that these values appear as substrings of $\mathcal{F}_{\mathsf{NC}}$'s internal randomness r_{dcm}.

To verify a tuple $(\mathsf{cm}, m, \mathsf{dcm})$, $\mathcal{F}_{\mathsf{NC}}$ first checks if it has seen cm before. If so: (A) If the supplied (m, dcm) match the values used during commitment generation, $\mathcal{F}_{\mathsf{NC}}$ accepts; (B) If the entire tuple $(\mathsf{cm}, m, \mathsf{dcm})$ was previously seen, $\mathcal{F}_{\mathsf{NC}}$ responds in a manner consistent with its previous response; and (C) $\mathcal{F}_{\mathsf{NC}}$ rejects in all other cases. If cm is a value that is new to $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NC}}$ provides Extr with cm and the relevant part of the recorded history $\mathsf{Hist}_{\mathsf{GRO}}$. If Extr is able to recover m and dcm given this information, then $\mathcal{F}_{\mathsf{NC}}$ accepts; else it rejects.

4.1 The interface between $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$

We detail the interface between $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$. While this interface adds complexity to the description of $\mathcal{F}_{\mathsf{NC}}$, we note that placing this complexity with $\mathcal{F}_{\mathsf{NC}}$ (rather than with $\mathcal{F}_{\mathsf{GRO}}$) is a design choice that makes the extent of the reliance of $\mathcal{F}_{\mathsf{NC}}$ on $\mathcal{F}_{\mathsf{GRO}}$ explicit.

$\mathcal{F}_{\mathsf{NC}}$ handles internal requests by SCode algorithms to program $\mathcal{F}_{\mathsf{GRO}}$ and also enables observing queries to $\mathcal{F}_{\mathsf{GRO}}$ made by an external party (e.g. the environment).

Programming. Algorithms in SCode may program $\mathcal{F}_{\mathsf{GRO}}$ by invoking the $\mathsf{Program}$ subroutine (Figure 5), which may choose to allow or disallow the request. Specifically, this access control structure ensures that $\mathcal{F}_{\mathsf{NC}}$ is hiding and binding, even if the (adversarially-defined) algorithms in SCode program $\mathcal{F}_{\mathsf{GRO}}$. (As such, $\Phi_{\mathcal{F}}$ differs for every ideal functionality.) We formalize the checks as a predicate $\Phi_{\mathcal{F}_{\mathsf{NC}}}$ (Figure 6) that is run by $\mathsf{Program}$. Specifically, the predicate check $\Phi_{\mathcal{F}}$ has the following interface.

- $\Phi_{\mathcal{F}}(\mathsf{sid}, \mathsf{ActiveAlg}, x, y) \to \{0, 1\}$.
  On input the session ID of a calling party sid, the calling algorithm $\mathsf{ActiveAlg}$, a $\mathcal{F}_{\mathsf{GRO}}$ query point $x \in \mathrm{Dom}(\mathcal{F}_{\mathsf{GRO}})$, and a $\mathcal{F}_{\mathsf{GRO}}$ query answer $y \in \mathrm{Cod}(\mathcal{F}_{\mathsf{GRO}})$, outputs a bit indicating whether this is a valid query to program in $\mathcal{F}_{\mathsf{GRO}}$.

For $\mathcal{F}_{\mathsf{NC}}$, $\Phi_{\mathcal{F}_{\mathsf{NC}}}$ checks that:

- Only the SCode algorithms in the **Commit** stage, specifically Com and Equiv can program $\mathcal{F}_{\mathsf{GRO}}$. While it's possible to allow Com and Equiv to also query $\mathcal{F}_{\mathsf{GRO}}$ at additional points, we do not specify this functionality because it is not required for modeling the commitment protocol of Figure 7.

- $\mathsf{SCode.Equiv}$ only programs a single location in $\mathcal{F}_{\mathsf{GRO}}$. While it's possible to allow $\mathsf{SCode.Equiv}$ to program more locations in $\mathcal{F}_{\mathsf{GRO}}$, we elect not to do so for simplicity and since this condition suffices for proving Theorem 5.1.

- The SCode algorithms in the **Verify** stage (specifically, Extr) may only access adversarial queries that are made to $\mathcal{F}_{\mathsf{GRO}}$, specifically a subset Q of the list $\mathsf{Hist}_{\mathsf{GRO}}$. This prevents $\mathcal{E}$ from breaking hiding or binding, via repeatedly querying **Verify**, to either leak information about $\mathcal{F}_{\mathsf{GRO}}$ or "program" information into $\mathcal{F}_{\mathsf{GRO}}$.[4]

We note that the $\mathsf{Program}$ subroutine of Figure 5 only handles programming requests originating from the functionality itself. See Figure 12 for an extension of $\mathsf{Program}$ that additionally handles programming requests from sub-functionalities simulated by SCode, e.g. for simulating a composed protocol.

[4] We give an example that breaks binding for our Π_{NC} (Figure 7) via programming. Suppose $\mathsf{SCode.Extr}(\mathsf{sid}, \mathsf{cm}, Q)$ is specified so that if it accepts, it additionally programs a collision in $\mathcal{F}_{\mathsf{GRO}}$, i.e. a query-answer pair $((\mathsf{sid}, m', \mathsf{dcm}'),\ \mathsf{cm})$ such that $m' \neq \overline{m}$ and dcm' is known to $\mathcal{E}$. This breaks binding.

Functionality $\mathcal{F}_{\mathsf{NC}}$ (main)

```
I Parameters: A message alphabet $\Sigma$, a commitment length $h_{\mathsf{cm}} \in \mathbb{N}$, and a decommitment length $h_{\mathsf{dcm}} \in \mathbb{N}$.
II Commit: Receive input $(\texttt{Commit}, m \in \Sigma \cup \{\perp\})$ from a party $C$.
  (a) If this is not the first $(\texttt{Commit}, \cdot)$ input, output $(\texttt{Error})$. # Fail-close clause.
  (b) If this is the first activation, run the Initialization subroutine (Step V).
  (c) Compute $(\mathsf{cm}, \mathsf{st}) \leftarrow \mathsf{SCode.Com}(\mathsf{sid}, r_{\mathsf{cm}})$.
      # $\mathsf{SCode.Com}$ may program $\mathcal{F}_{\mathsf{GRO}}$ by calling Program (Figure 5).
      i. If $\mathsf{cm} \notin r_{\mathsf{cm}}$, $(\mathsf{cm}, \cdot, \cdot, \cdot, v) \in \mathsf{CState}$, or $\mathsf{SCode.Com}$ commits an $\mathsf{SCodeError}$ (Definition 4.6):
         # Error. Outputs default values when consistency is violated.
         A. Run the default committer code (Step VI) with message $m$.
         B. Output $(\texttt{Receipt}, \mathsf{cm}, \mathsf{dcm})$ to $C$.
      ii. Else, continue to next step.
  (d) If $m = \perp$: # Don't run $\mathsf{SCode.Equiv}$.
      i. Set $\mathsf{dcm} := \perp$ and $\mathsf{State}^{\mathsf{NC}} := \perp$. Append $(\mathsf{cm}, m, \mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}, 0)$ to $\mathsf{CState}$.
  (e) Else, compute $(\mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}) \leftarrow \mathsf{SCode.Equiv}(\mathsf{sid}, \mathsf{cm}, \mathsf{st}, m, r_{\mathsf{dcm}})$.
      # $\mathsf{SCode.Equiv}$ may program $\mathcal{F}_{\mathsf{GRO}}$ by calling Program (Figure 5).
      i. If $\mathsf{dcm} \notin r_{\mathsf{dcm}}$, $\mathsf{SCode.Equiv}$ or $\mathsf{SCode.Equiv}$ commits an $\mathsf{SCodeError}$ (Definition 4.6):
         A. Run the default committer code (Step VI) with message $m$.
      ii. Else, append $(\mathsf{cm}, m, \mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}, 1)$ to $\mathsf{CState}$. # No errors.
  (f) Output $(\texttt{Receipt}, \mathsf{cm}, \mathsf{dcm})$ to $C$.
III (Adaptive) Committer Corruption: On input $(\texttt{Corrupt})$ from the adversary $\mathcal{E}$.
  (a) If there exists $(\mathsf{cm}, m, \mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}, 1) \in \mathsf{CState}$, output $(\texttt{Corrupted}, \mathsf{State}^{\mathsf{NC}})$ to $\mathcal{E}$.
      # If a valid commitment was generated, outputs all internal state.
  (b) Else, output $(\texttt{Corrupted}, \perp)$ to $\mathcal{E}$.
IV Verify: Receive input $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ from a party $R$.
  (a) If this is the first activation, run the Initialization subroutine (Step V).
  (b) If $(\overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}}, \cdot, v) \in \mathsf{CState}$, output $(\texttt{VerStatus}, v)$ to $R$ (thereby ending the activation).
      # Enforces completeness and consistency.
  (c) If $(\mathsf{cm} = \overline{\mathsf{cm}}, m, \mathsf{dcm}, \cdot, 1) \in \mathsf{CState}$ for some $m \neq \overline{m}$, then output $(\texttt{VerStatus}, 0)$ to $R$.
      # Enforces binding.
  (d) If $(\mathsf{cm} = \overline{\mathsf{cm}}, m = \overline{m}, \mathsf{dcm}, \cdot, 1) \in \mathsf{CState}$ for some $\mathsf{dcm} \neq \overline{\mathsf{dcm}}$, then output $(\texttt{VerStatus}, 0)$ to $R$. # Enforces non-malleability, only allows a single decommitment.
  (e) If $(\cdot, \overline{\mathsf{cm}})$ was programmed by $\mathsf{SCode.Com}$ or $\mathsf{SCode.Equiv}$, output $(\texttt{VerStatus}, 0)$ to $R$.[a]
      # Prevents information leakage by adversarial $\mathsf{SCode.Com}$ or $\mathsf{SCode.Equiv}$.
  (f) Else:[b] # For extractability and non-malleability.
      i. Compute the set $Q \subseteq \mathsf{Hist}_{\mathsf{GRO}}$ such that $Q := \{(x, y) \mid y = \overline{\mathsf{cm}} \wedge (x, y) \in \mathsf{Hist}_{\mathsf{GRO}}\}$.
      ii. Compute $(m', \mathsf{dcm}') \leftarrow \mathsf{SCode.Extr}(\mathsf{sid}, \overline{\mathsf{cm}}, Q)$. # Only allows a single valid $\mathsf{dcm}'$ value.
      iii. If $m' = \perp$, $m' \neq \overline{m}$, $\mathsf{dcm}' \neq \overline{\mathsf{dcm}}$, $\overline{\mathsf{cm}} \notin r_{\mathsf{cm}}$, or $\mathsf{SCode.Extr}$ commits $\mathsf{SCodeTimeError}$ (Definition 4.6), set $v := 0$. Else, set $v := 1$. # Errors.
      iv. Append $(\overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}}, \perp, v)$ to $\mathsf{CState}$ and output $(\texttt{VerStatus}, v)$ to $R$.
```

[a] We are grateful to Lawrence Roy for pointing out the need for this requirement, by breaking an earlier version of $\mathcal{F}_{\mathsf{NC}}$ that did not include it.

[b] This includes the cases that (A) there is an entry $(\overline{\mathsf{cm}}, m, \mathsf{dcm}, \cdot, 0) \in \mathsf{CState}$ such that $(m, \mathsf{dcm}) \neq (\overline{m}, \overline{\mathsf{dcm}})$; and (B) there does not exist an entry $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, \cdot)$ in CState.

Figure 3: The non-interactive commitment functionality in the presence of $\mathcal{F}_{\mathsf{GRO}}$, part 1: Commit and Verify.

Functionality $\mathcal{F}_{\mathsf{NC}}$ (subroutines)

V **Initialization:** In the first activation, do: # First activation can be `Commit` or `Verify`.

- (a) Set the following variables as empty (i.e. $\perp$):
  - CState: a list of tuples $(\mathsf{cm}, m, \mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}, v)$ where cm is a commitment string, $m \in \Sigma$ is a message, dcm is a decommitment string, $\mathsf{State}^{\mathsf{NC}}$ is the committer state, and $v \in \{0,1\}$ represents whether $(\mathsf{cm}, m, \mathsf{dcm})$ is valid.
  - $\mathsf{Hist}_{\mathsf{GRO}}$: a list of tuples (x, y), in which x is an externally queried point and y is the answer.
  - $\mathsf{ActiveAlg}$: records which algorithm is currently interacting with $\mathcal{F}_{\mathsf{GRO}}$.
- (b) Take control of session sid (where sid is the local sid) in $\mathcal{F}_{\mathsf{GRO}}$ by sending $(\texttt{Control})$ to $\mathcal{F}_{\mathsf{GRO}}$, and receive a message m from $\mathcal{F}_{\mathsf{GRO}}$.# Enables observing GRO query-answer pairs and/or programming GRO responses for queries with prefix sid.
  - i. If $m = (\texttt{Ok}, \mathsf{Hist})$ such that $\mathsf{Hist} \neq \perp$, output $\perp$. # Failure.
- (c) Send $(\texttt{CodeRequest}, \mathcal{F}_{\mathsf{NC}})$ to $\mathcal{F}_{\mathsf{lib}}$ and receive $(\texttt{Answer}, \mathsf{LinkedRecords})$, where $\mathsf{LinkedRecords} = (\mathcal{F}_{\mathsf{NC}}, \mathsf{SCode}, \cdot)$ and SCode is a tuple of algorithms $\{\mathsf{Com}, \mathsf{Equiv}, \mathsf{Extr}\}$.
- (d) Let $r_{\mathsf{cm}} \leftarrow (\{0,1\}^{h_{\mathsf{cm}}})^{\lambda}$, $r_{\mathsf{dcm}} \leftarrow (\{0,1\}^{h_{\mathsf{dcm}}})^{\lambda}$, $\widehat{r_{\mathsf{cm}}} \leftarrow (\{0,1\}^{h_{\mathsf{cm}}})^{\lambda}$, $\widehat{r_{\mathsf{dcm}}} \leftarrow (\{0,1\}^{h_{\mathsf{dcm}}})^{\lambda}$, denote the random strings of $\mathcal{F}_{\mathsf{NC}}$. # The random strings $r_{\mathsf{cm}}, r_{\mathsf{dcm}}$ will be given to $\mathsf{SCode.Com}$ and $\mathsf{SCode.Equiv}$. $\widehat{r_{\mathsf{cm}}}, \widehat{r_{\mathsf{dcm}}}$ are for running default code.

VI **Default commit code:** # Generates strings in event of errors.

- (a) Set $\mathsf{cm} := \widehat{r_{\mathsf{cm}}}[0]$
- (b) If $m = \perp$, set $\mathsf{dcm} := \perp$. Else, set $\mathsf{dcm} := \widehat{r_{\mathsf{dcm}}}[0]$.
- (c) Append $(\mathsf{cm}, m, \mathsf{dcm}, \perp, 0)$ to CState. # Values generated with default code are not valid.

VII **External request to program the GRO:** Upon receiving $(\texttt{ProgramReq}, P, x)$ from $\mathcal{F}_{\mathsf{GRO}}$:
# This interface allows $\mathcal{F}_{\mathsf{NC}}$ to observe oracle queries.

- (a) Sample $y \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$ and add $(x, y) \in \mathsf{Hist}_{\mathsf{GRO}}$.
- (b) Send subroutine input $(\texttt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$.

Figure 4: The non-interactive commitment functionality in the presence of $\mathcal{F}_{\mathsf{GRO}}$, Part 2: Subroutines for initialization, default committer code, internal $\mathcal{F}_{\mathsf{GRO}}$ programming requests, and external $\mathcal{F}_{\mathsf{GRO}}$ evaluation requests.

Observing. $\mathcal{F}_{\mathsf{NC}}$ observes and records all "adversarial" queries with prefix sid in $\mathsf{Hist}_{\mathsf{GRO}}$ in Step VII of $\mathcal{F}_{\mathsf{NC}}$; we write adversarial queries to mean any queries with prefix sid that were made by the environment (and not $\mathcal{F}_{\mathsf{NC}}$). This enables extraction when **Verify** is invoked for adversarially-generated values. We emphasize that $\mathsf{Hist}_{\mathsf{GRO}}$ does *not* include any queries made or programmed in the **Commit** phase, including those made or programmed by $\mathsf{SCode.Com}$ and $\mathsf{SCode.Equiv}$, to preserve hiding and binding.

Program **subroutine** ($\mathcal{F}_{\mathsf{NC}}$ **version).**

```
• Program($\mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}}, \mathsf{sid}, \mathsf{ActiveAlg}, x, y$): # Run only if $(x, \cdot) \notin \mathsf{Hist}_{\mathsf{GRO}}$ of $\mathcal{F}$.
  1. Parse the inputs so that:
     (a) the following are honestly derived from the state of $\mathcal{F}$: $\mathsf{Hist}_{\mathsf{GRO}}$ is a list of GRO query-answer pairs,
         $\Phi_{\mathcal{F}}$ is the query-checking predicate for $\mathcal{F}$, $\mathsf{sid}$ is the calling party's session ID, $\mathsf{ActiveAlg}$ is the
         calling party/algorithm invoking Program; and
     (b) the following may be chosen by $\mathsf{ActiveAlg}$: $x$ is a GRO query in $\mathrm{Dom}(\mathcal{F}_{\mathsf{GRO}})$, and $y$ is a GRO
         answer in $\mathrm{Cod}(\mathcal{F}_{\mathsf{GRO}})$.
  2. If $(x, \cdot) \notin \mathsf{Hist}_{\mathsf{GRO}}$: # The evaluation of $x$ is undefined in $\mathcal{F}_{\mathsf{GRO}}$.
     (a) If $\Phi_{\mathcal{F}}(\mathsf{sid}, \mathsf{ActiveAlg}, x, y) = 1$: # $\Phi_{\mathcal{F}}$ is a predicate check.
         i. If $\mathsf{sid}$ is the local session ID of $\mathcal{F}$:
            A. Send subroutine input $(\mathtt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$. # $\mathcal{F}_{\mathsf{GRO}}$ records $(x, y)$ in its history.
            B. Receive subroutine output $(\mathtt{EvalOutput}, y)$ from $\mathcal{F}_{\mathsf{GRO}}$.
     (b) Else, output $(\mathtt{EvalOutput}, \perp)$ to $\mathsf{ActiveAlg}$. # Error.
  3. Else (if there exists an entry $(x, y') \in \mathsf{Hist}_{\mathsf{GRO}}$), send $(\mathtt{EvalOutput}, y')$ to $\mathsf{ActiveAlg}$.
```

Figure 5: The Program subroutine, which handles requests by simulator code to program $\mathcal{F}_{\mathsf{GRO}}$.

Maintaining consistency between $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$. Maintaining consistency between the $\mathsf{Hist}_{\mathsf{GRO}}$ of controller parties (to simplify discussion, let $\mathcal{F}_{\mathsf{NC}}$ be the controller party) and the Hist of $\mathcal{F}_{\mathsf{GRO}}$ is somewhat tricky. In particular, the query-answer histories of $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$ should stay in sync. Below, we detail a scenario in which the histories may contain discrepancies, why this scenario is undesirable, and our fix.

Consider the following scenario, in which the query-answer histories of $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$ are not identical. Initially, in $\mathcal{F}_{\mathsf{GRO}}$, Hist is empty and the session ID sid is not controlled by anyone. The environment makes a query, i.e. sends $(\mathtt{Eval}, x = (\mathsf{sid}, x'))$ to $\mathcal{F}_{\mathsf{GRO}}$. Since sid does not have a controller party, $\mathcal{F}_{\mathsf{GRO}}$ samples a random y and adds (x, y) to the query-answer history Hist of $\mathcal{F}_{\mathsf{GRO}}$. Next, an instance of $\mathcal{F}_{\mathsf{NC}}$ with session ID sid takes control of sid in $\mathcal{F}_{\mathsf{GRO}}$. At this point, the query-answer history $\mathsf{Hist}_{\mathsf{GRO}}$ of the ideal functionality is empty, whereas $(x, y) \in \mathsf{Hist}$ of $\mathcal{F}_{\mathsf{GRO}}$.

This asynchrony allows an attacker to cause query-answer transcript discrepancies, as follows. The attacker, which has session ID $\mathsf{sid}' \neq \mathsf{sid}$, sends $(\mathtt{Eval}, x = (\mathsf{sid}, x'))$ to $\mathcal{F}_{\mathsf{GRO}}$. Since $\mathcal{F}_{\mathsf{NC}}$ controls sid, $\mathcal{F}_{\mathsf{GRO}}$ sends $(\mathtt{Program}, x)$ to $\mathcal{F}_{\mathsf{NC}}$. Since $\mathsf{Hist}_{\mathsf{GRO}}$ of $\mathcal{F}_{\mathsf{NC}}$ doesn't have the entry (x, y), $\mathcal{F}_{\mathsf{NC}}$ samples y' uniformly at random, adds (x, y') to $\mathsf{Hist}_{\mathsf{GRO}}$, and sends $(\mathtt{Program}, y')$ to $\mathcal{F}_{\mathsf{GRO}}$. Ultimately, this can lead to errors in extraction.

To avoid transcript discrepancies, $\mathcal{F}_{\mathsf{GRO}}$ informs $\mathcal{F}_{\mathsf{NC}}$ if there were already queries made to the session ID sid of $\mathcal{F}_{\mathsf{NC}}$, prior to $\mathcal{F}_{\mathsf{NC}}$ taking control of sid in $\mathcal{F}_{\mathsf{GRO}}$ (See Step III of $\mathcal{F}_{\mathsf{GRO}}$ (Figure 2)). If $\mathcal{F}_{\mathsf{NC}}$ sees that queries were already made, $\mathcal{F}_{\mathsf{NC}}$ aborts.

4.2 Security guarantees provided by $\mathcal{F}_{\mathsf{NC}}$

In the coming sub-sections we argue that $\mathcal{F}_{\mathsf{NC}}$ provides the following security guarantees:

The predicate $\Phi_{\mathcal{F}_{\mathsf{NC}}}$

```
• $\Phi_{\mathcal{F}_{\mathsf{NC}}}(\mathsf{sid}, \mathsf{ActiveAlg}, x, y) \to \{0,1\}$:
  1. Parse sid as the session ID of the calling party, ActiveAlg as a calling algorithm, $x$ as a query point $(\mathsf{sid}', x')$ in $\mathrm{Dom}(\mathcal{F}_{\mathsf{GRO}})$, and $y$ as a query answer in $\mathrm{Cod}(\mathcal{F}_{\mathsf{GRO}})$.
  2. If $\mathsf{sid} \neq \mathsf{sid}'$, output $0$. # Disallow cross-domain programming.
  3. If $\mathsf{ActiveAlg} = \mathsf{SCode.Com}$: # The caller is SCode.Com.
     (a) If the $h_{\mathsf{dcm}}$-bit prefix of $x'$ is in $r_{\mathsf{dcm}}$ and the query answer $y \in r_{\mathsf{cm}}$, output $1$.[a] # Ensure programmed $\mathcal{F}_{\mathsf{GRO}}$ query and answer have sufficient min-entropy.
     (b) Else, output $0$.
  4. If $\mathsf{ActiveAlg} = \mathsf{SCode.Equiv}$: # The caller is SCode.Equiv.
     (a) If this execution of SCode.Equiv has already programmed $\mathcal{F}_{\mathsf{GRO}}$, output $0$.
         # Each execution of SCode.Equiv programs only once.
     (b) If the $h_{\mathsf{dcm}}$-bit prefix of $x'$ is in $r_{\mathsf{dcm}}$, and the query answer $y \in r_{\mathsf{cm}}$, output $1$.[b] # Ensure programmed $\mathcal{F}_{\mathsf{GRO}}$ query and answer have sufficient min-entropy.
     (c) Else, output $0$.
  5. Else, output $0$. # Allow no other SCode algorithms to program.
```

[a] For full generality, the check that "h_{dcm}-bit prefix of x' is in r_{dcm}" can be replaced by any injective function mapping between x' and r_{dcm}, which is defined in SCode. (This enables the modeling of commitments of other message formats.)

[b] For full generality, the check that "h_{dcm}-bit prefix of x' is in r_{dcm}" can be replaced by any injective function mapping between x' and r_{dcm}, which is defined in SCode. (This enables the modeling of commitments of other message formats.)

Figure 6: The predicate $\Phi_{\mathcal{F}_{\mathsf{NC}}}$, which verifies properties of $\mathcal{F}_{\mathsf{GRO}}$ programming requests originating from $\mathcal{F}_{\mathsf{NC}}$.

- **Completeness**: When asked to verify a triple $(\mathsf{cm}, \mathsf{dcm}, m)$ that was generated by $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NC}}$ always accepts.
- **Consistency**: Requests to verify a tuple $(\mathsf{cm}, \mathsf{dcm}, m)$ always have the same output.
- **Secrecy** (a.k.a. hiding): The commitment string cm is generated independently of m. Furthermore, despite the fact that dcm was generated depending on m, the value dcm doesn't leak information about m. Again, this holds for any (potentially malicious) code provided in SCode.
- **Binding, extractability, and non-malleability**: Any string cm can be opened to at most one value m. More precisely, whenever a triple $(\mathsf{cm}, \mathsf{dcm}, m)$ is accepted by $\mathcal{F}_{\mathsf{NC}}$, it is guaranteed that either cm was generated via the commit interface, or else m was the output of algorithm SCode.Extr, given only cm and information on the $\mathcal{F}_{\mathsf{GRO}}$ query where cm was the returned value (if any).
  We note that this property implies non-malleability. In particular, successful commitment (and later decommitment) to a value m' related to an honestly committed message m, where the commitment to m' was made before the commitment to m was opened, would result in a violation of the hiding property. This holds even if the decommitment to m' took place after a potential decommitment to m.
- **Immediate response**: Each call to $\mathcal{F}_{\mathsf{NC}}$ completes without the environment being activated in the process. Furthermore, this holds for any (potentially malicious) code provided in SCode.
- **Adaptive security**: An adversary that corrupts the committer and learns its internal state, at any point, learns no additional information.

4.2.1 Completeness and consistency

Completeness is the property that if $\mathcal{F}_{\mathsf{NC}}$ is asked to verify a triple $(\mathsf{cm}, \mathsf{dcm}, m)$ that was generated by $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NC}}$ always accepts. This is ensured by Step II(c)i, since tuples generated by $\mathcal{F}_{\mathsf{NC}}$ are recorded in CState.

Consistency is the property that two requests to verify a tuple $(\mathsf{cm}, \mathsf{dcm}, m)$ is always answered the same way. If the adversary first invokes $\mathcal{F}_{\mathsf{NC}}$ with **Verify**, then attempts to commit with the same information. In the **Commit** subroutine, Step II(c)i of $\mathcal{F}_{\mathsf{NC}}$ ensures consistency. In the **Verify** subroutine, Step IVb ensures consistency, because it ensures that $\mathcal{F}_{\mathsf{NC}}$'s outputs are consistent with past outputs, via CState.

4.2.2 Hiding

Hiding is the property that, as long as the decommitment information remains unknown, interacting with $\mathcal{F}_{\mathsf{NC}}$ should provide no information on the message. We begin by comparing the hiding guarantees provided by the usual (interactive) commitment functionality $\mathcal{F}_{\mathsf{Com}}$ of [CF01] and the guarantees provided by $\mathcal{F}_{\mathsf{NC}}$.

Comparing the hiding guarantees of $\mathcal{F}_{\mathsf{Com}}$ and $\mathcal{F}_{\mathsf{NC}}$. Recall that $\mathcal{F}_{\mathsf{Com}}$ (see Figure 2 of [CF01]) models perfect hiding because it never issues a commitment string cm (or decommitment string dcm).

In comparison, since $\mathcal{F}_{\mathsf{NC}}$ is non-interactive, its **Commit** subroutine, on input a message m, outputs a specific commitment string cm and decommitment string dcm. This subtle difference *inherently* gives $\mathcal{F}_{\mathsf{NC}}$ a weaker hiding guarantee compared to $\mathcal{F}_{\mathsf{Com}}$; $\mathcal{F}_{\mathsf{NC}}$ has a hiding error equal to the likelihood of guessing the valid dcm with respect to cm and m. In contrast, $\mathcal{F}_{\mathsf{Com}}$ has absolute hiding because $\mathcal{F}_{\mathsf{Com}}$ never outputs the commitment string cm, so the adversary cannot attempt to guess the correct dcm.

However, we re-emphasize that the non-interactive formulation of $\mathcal{F}_{\mathsf{NC}}$ is necessary for a modular analysis of computations in which a commitment string is used as part of the computation, e.g. for nodes in the Merkle tree of the Kilian-Micali ZK-SNARK.

Hiding in $\mathcal{F}_{\mathsf{NC}}$. To quantify the security provided by $\mathcal{F}_{\mathsf{NC}}$, we explicitly parameterize $\mathcal{F}_{\mathsf{NC}}$ with the min-entropy of the decommitment string dcm, denoted h_{dcm}. $\mathcal{F}_{\mathsf{NC}}$ enforces this min-entropy condition of dcm in the following manner: $\mathcal{F}_{\mathsf{NC}}$ provides its random tape to the simulator code SCode and checks that the simulated dcm indeed corresponds to its random tape. To model hiding, our $\mathcal{F}_{\mathsf{NC}}$ needs the following two properties:

1. The commitment string cm is distributed independently from the committed message m. In other words, seeing cm does not leak information about the committed m.
2. Given a particular commitment string cm, the decommitment string dcm is hard for the adversary to guess. To illustrate the necessity of this property, suppose $\mathcal{E}$ can efficiently recover the decommitment string dcm. We want the adversary to be unable to verify guesses, even if it receives the message as side information. Then, $\mathcal{E}$ learns the committed m upon receiving $(\texttt{VerStatus}, 1)$ from $\mathcal{F}_{\mathsf{NC}}$.

To achieve these properties, $\mathcal{F}_{\mathsf{NC}}$ receives a $(\texttt{Commit}, m \in \Sigma)$ message and does:

- For Property 1: $\mathcal{F}_{\mathsf{NC}}$ invokes $\mathsf{SCode.Com}$, which samples a commitment cm *without* access to m.
- For Property 2: $\mathcal{F}_{\mathsf{NC}}$ invokes $\mathsf{SCode.Equiv}$, which outputs each particular decommitment string dcm with min-entropy at least h_{dcm}, relative to the message m and cm. Further, $\mathcal{F}_{\mathsf{NC}}$ enforces that $\mathsf{SCode.Equiv}$ is limited to programming at most a single location in $\mathcal{F}_{\mathsf{GRO}}$.

We elaborate on the intuitions that inform how $\mathcal{F}_{\mathsf{NC}}$ achieves Property 2. Recall that in the UC framework, security is quantified over all environments $\mathcal{E}$. In particular, consider a "split" environment $\mathcal{E} = (\mathcal{E}_1, \mathcal{E}_2)$, in which $\mathcal{E}_1$ has a message m, commits to m via invoking $\mathcal{F}_{\mathsf{NC}}$ and receiving $(\mathsf{cm}, \mathsf{dcm})$ then sends cm to $\mathcal{E}_2$. Hiding does not hold if $\mathcal{E}_2$ recovers the message m. In particular, there could be leakage between $\mathcal{E}_1$ and $\mathcal{E}_2$; if there is too much, then $\mathcal{E}_2$ can recover m.

We formalize the hiding property of $\mathcal{F}_{\mathsf{NC}}$ as a distinguishing game between two interactions that the environment may have with $\mathcal{F}_{\mathsf{NC}}$: Interaction 1, in which the decommitment string dcm is generated but not disclosed, and Interaction 2, in which dcm is not generated at all. Intuitively, this game captures hiding since the environment can only distinguish if it knows dcm; verification using dcm passes in Interaction 1 but not in Interaction 2.

We first define the general notion of a ideal commitment mechanism, then define the hiding game for ideal commitment mechanisms:

Definition 4.1 (Ideal commitment mechanism C)**.** *We say that an interactive Turing machine* C *is an ideal commitment mechanism in the presence of global functionalities* $\mathcal{F}_{\mathsf{GRO}}$ *and* $\mathcal{F}_{\mathsf{lib}}$*, if* C *has the following interfaces:*

1. *On input* $(\texttt{Commit}, m)$*, generates a commitment string* cm *and decommitment string* dcm*, outputs* $(\texttt{Receipt}, \mathsf{cm}, \mathsf{dcm})$ *where* cm *is a commitment string and* dcm *is a decommitment string.*
2. *On input* $(\texttt{Verify}, \mathsf{cm}, m, \mathsf{dcm})$*, outputs* $(\texttt{VerStatus}, v)$ *where* v *is a bit indicating whether* cm *is a valid commitment to the plaintext* m *with decommitment* dcm*.*

Game 4.2 (Hiding game for C in the presence of $\mathcal{F}_{\mathsf{GRO}}$)**.** Let C be an ideal commitment mechanism (Definition 4.1) in the presence of $\mathcal{F}_{\mathsf{GRO}}$ and $\mathcal{F}_{\mathsf{lib}}$. Let Σ be the plaintext alphabet (such that $\perp \notin \Sigma$). The adversary $\mathcal{E}$ selects a message $m \in \Sigma$ and attempts to distinguish between the following interactions:

Interaction 1.
(a) C is invoked with input $(\texttt{Commit}, m)$.
(b) Receive the output $(\texttt{Receipt}, \mathsf{cm}, \mathsf{dcm})$ from C, as usual.
(c) Output cm to $\mathcal{E}$. # Though dcm may be generated, $\mathcal{E}$ only sees cm.

Interaction 2. This is identical to Interaction 1, except that the committed plaintext is $m = \perp$.
(a) C is invoked with input $(\texttt{Commit}, \perp)$.
(b) Receive the output $(\texttt{Receipt}, \mathsf{cm}, \mathsf{dcm})$ from C.
(c) Output cm to $\mathcal{E}$. # Though dcm may be generated, $\mathcal{E}$ only sees cm.

Additionally, in both interactions, $\mathcal{E}$ may: (a) access the global functionalities $\mathcal{F}_{\mathsf{GRO}}$ and $\mathcal{F}_{\mathsf{lib}}$; and (b) invoke both the **Commit** and **Verify** interfaces of C.

$\mathcal{E}$ wins if it distinguishes between Interactions 1 and 2.

Observe that our $\mathcal{F}_{\mathsf{NC}}$ is an ideal commitment mechanism. We show that no adversary can win the above hiding game against our $\mathcal{F}_{\mathsf{NC}}$, except with negligible probability.

Theorem 4.3. *Let* $h_{\mathsf{cm}}, h_{\mathsf{dcm}} \in \mathbb{N}$*. Let* $\mathcal{E}$ *be an adversary with runtime* t*. Then, for* $\mathcal{F}_{\mathsf{NC}}(h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ *running any simulator code* SCode *in the presence of* $\mathcal{F}_{\mathsf{GRO}}$ *and* $\mathcal{F}_{\mathsf{lib}}$*,* $\mathcal{E}$ *wins Game 4.2 with probability* $t \cdot 2^{-h_{\mathsf{dcm}}}$*.*

Proof of Theorem 4.3. Observe that $\mathcal{F}_{\mathsf{NC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$ is an ideal commitment mechanism, as defined in Definition 4.1. (The proof follows by inspecting the interfaces of $\mathcal{F}_{\mathsf{NC}}$.)

We argue that $\mathcal{E}$ wins Game 4.2 with probability $t \cdot 2^{-h_{\mathsf{dcm}}}$. First, we define the bad event:

Bad event. $\mathcal{E}$ guesses the decommitment string dcm generated in Interaction 1. (In Interaction 2, note that $\mathsf{dcm} = \perp$ due to Step IId of $\mathcal{F}_{\mathsf{NC}}$.)

Second, we compute the probability of the bad event: in $\mathcal{F}_{\mathsf{NC}}$, dcm is a random bitstring of length h_{dcm} (see Step IIe of $\mathcal{F}_{\mathsf{NC}}$), so $\mathcal{E}$ guesses dcm with probability $2^{-h_{\mathsf{dcm}}}$ per attempt.

Third, we examine the view of $\mathcal{E}$ when interacting with $\mathcal{F}_{\mathsf{NC}}$ in the hiding game and argue that $\mathcal{E}$ distinguishes between Interactions 1 and 2 only if the bad event occurs.

$\mathcal{E}$ only distinguishes if the output of **Verify** differs in Interactions 1 and 2. Note that **Verify** never programs $\mathcal{F}_{\mathsf{GRO}}$ since $\mathsf{SCode.Extr}$ cannot program $\mathcal{F}_{\mathsf{GRO}}$. Hence, no information is leaked via $\mathcal{F}_{\mathsf{GRO}}$ when **Verify** is invoked.

Suppose $\mathcal{E}$ invokes $\mathcal{F}_{\mathsf{NC}}$ with an input $(\mathtt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$. If **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$ is never previously invoked by $\mathcal{E}$, Interactions 1 and 2 are identical. Otherwise, if **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$ was invoked,we have cases depending on if $\overline{\mathsf{cm}} = \mathsf{cm}$, where cm is the output of the hiding game:

- $\overline{\mathsf{cm}} = \mathsf{cm}$:
  In Interaction 1, Steps IVb to IVd of $\mathcal{F}_{\mathsf{NC}}$ imply that verification passes only if $\mathcal{E}$ guesses the dcm generated during the **Commit** phase. In Interaction 2, verification always fails because $(\mathsf{cm}, \perp, \perp, \perp, 0)$ is written in CState (due to Step II(d)i of $\mathcal{F}_{\mathsf{NC}}$). Hence, $\mathcal{E}$ distinguishes when the bad event occurs.
- $\overline{\mathsf{cm}} \neq \mathsf{cm}$:
  We consider cases depending on whether $\mathcal{F}_{\mathsf{NC}}$ programs $\mathcal{F}_{\mathsf{GRO}}$ during the **Commit** phase.
  First, suppose the **Commit** phase of $\mathcal{F}_{\mathsf{NC}}$ previously programs a point $(x_{\mathsf{BAD}}, \overline{\mathsf{cm}})$ in $\mathcal{F}_{\mathsf{GRO}}$ such that $\mathsf{dcm} \in x_{\mathsf{BAD}}$. There are two possible algorithms that could program the point $(x_{\mathsf{BAD}}, \overline{\mathsf{cm}})$:
  - $\mathsf{SCode.Com}$: this is run in both Interactions 1 and 2. Hence, in both interactions, $\mathcal{E}$'s view is identical, since verification fails in both interactions due to Step IVe of $\mathcal{F}_{\mathsf{NC}}$.
  - $\mathsf{SCode.Equiv}$: In Interaction 1, $\mathsf{SCode.Equiv}$ is run during the **Commit** phase, so verification fails due to Step IVe of $\mathcal{F}_{\mathsf{NC}}$, which rejects on any $\mathcal{F}_{\mathsf{GRO}}$ queries programmed by $\mathsf{SCode.Equiv}$. In Interaction 2, since $m = \perp$, $\mathcal{F}_{\mathsf{GRO}}$ is not programmed since $\mathsf{SCode.Equiv}$ is not run. Thus, the verification of Interaction 2 potentially accepts if, prior to invoking **Verify**, $\mathcal{E}$ guesses the value dcm, then invokes $\mathsf{GRO}(x_{\mathsf{BAD}})$ and gets output $\overline{\mathsf{cm}}$. The probability $\mathcal{E}$ succeeds is upper bounded by the probability of the bad event.

  Second, suppose the **Commit** phase of $\mathcal{F}_{\mathsf{NC}}$ does not program a point $(x_{\mathsf{BAD}}, \overline{\mathsf{cm}})$ in $\mathcal{F}_{\mathsf{GRO}}$. Clearly, the interactions are identical, up to the bad event that dcm is guessed, since $\mathsf{dcm} = \perp$ in Interaction 2.

Hence, in all possible interactions, $\mathcal{E}$ distinguishes Interactions 1 and 2 with the same probability as the bad event occurring: $2^{-h_{\mathsf{dcm}}}$ per attempt. Since $\mathcal{E}$ has runtime t, its overall distinguishing probability is $t \cdot 2^{-h_{\mathsf{dcm}}}$.

□

Discussion. We can view $\mathcal{F}_{\mathsf{NC}}$ as having two sources of leakage: leakage inherent to the formulation of $\mathcal{F}_{\mathsf{NC}}$ (independent of $\mathcal{F}_{\mathsf{GRO}}$) and leakage that stems from $\mathcal{F}_{\mathsf{NC}}$'s interaction with $\mathcal{F}_{\mathsf{GRO}}$.

- **Leakage inherent to $\mathcal{F}_{\mathsf{NC}}$ (independent of $\mathcal{F}_{\mathsf{GRO}}$).**
  If the decommitment string dcm has small min-entropy (or is short), $\mathcal{E}$ can exhaustively guess it with non-trivial probability. To quantify the degree to which $\mathcal{E}$ can do so, $\mathcal{F}_{\mathsf{NC}}$ is parameterized by the min-entropy of dcm: h_{dcm}. In general, if $\mathcal{E}$ is t-query and dcm has min-entropy h_{dcm}, the hiding property of $\mathcal{F}_{\mathsf{NC}}$ breaks with probability $t \cdot 2^{-h_{\mathsf{dcm}}}$.
- **Leakage due to $\mathcal{F}_{\mathsf{GRO}}$.**
  Since our $\mathcal{F}_{\mathsf{NC}}$ which interacts with a programmable global random oracle $\mathcal{F}_{\mathsf{GRO}}$, $\mathcal{F}_{\mathsf{GRO}}$ itself is another possible source of leakage.[5]
  In particular, when invoking the UC composition theorem, the simulator code SCode is considered untrusted code, so $\mathcal{F}_{\mathsf{NC}}$ must ensure that SCode doesn't surreptitiously leak info about the committed message or the decommitment. The following is an example of a bad SCode, which leaks dcm:

  > SCode is defined so that $\mathsf{SCode.Equiv}$ programs dcm somewhere in $\mathcal{F}_{\mathsf{GRO}}$ that is easily-recoverable by $\mathcal{E}$. Specifically, consider an $\mathsf{SCode.Equiv}$ that works as described in Section 5.1.1, but additionally programs dcm as the output of location $(\mathsf{sid}, 0)$ in $\mathcal{F}_{\mathsf{GRO}}$. $\mathcal{E}$ can query location $(\mathsf{sid}, 0)$ to recover dcm.

[5] If the random oracle is local, there will be no leakage because the oracle is defined as part of the (real) protocol, whereas the ideal execution will not have access to the oracle.

To prevent such attacks, $\mathcal{F}_{\mathsf{NC}}$ only allows $\mathsf{SCode.Equiv}$ to program at most once (Step 4a) and further requires that the location programmed by $\mathsf{SCode.Equiv}$ have sufficient min-entropy (Step 4b), so that $\mathcal{E}$ cannot guess them. Note that the location programmed by $\mathsf{SCode.Equiv}$ can be correlated with dcm, since m remains hidden until dcm is revealed.

Comparison with $\mathcal{F}_{\mathsf{NC}}$ of [CSW22]. For hiding, [CSW22] does not account for the probability of guessing the decommitments string dcm, even though $\mathcal{E}$ can mount this attack against the $\mathcal{F}_{\mathsf{NC}}$ of their paper. Further, the $\mathcal{F}_{\mathsf{NC}}$ of their paper allows $\mathsf{SCode.Equiv}$ to generate decommitment strings with low entropy, since the min-entropy of dcm is never specified. Also, $\mathcal{F}_{\mathsf{NC}}$ of [CSW22] of does not interact with the global random oracle, so they don't have "hiding leakage" via the random oracle.

4.2.3 Binding

Binding is the property that, once a commitment string cm is associated with a message m, it is (computationally/statistically) hard to decommit cm to another message $m' \neq m$. As a consequence, it is hard to find two valid ways to decommit to cm.

With non-interactive commitments in the plain model (e.g. as defined in [CSW22]), one can typically associate a commitment string to a single message using an injective function. This way, one can recover the committed plaintext using a *deterministic* extraction algorithm. We call the point at which the injective function from commitment strings to messages is fixed the *binding point*.

Binding via extraction in [CSW22]. As a starting point, we review how binding works in the plain model (i.e. without random oracles), as is defined in [CSW22].

For commitments that are honestly-generated using $\mathcal{F}_{\mathsf{NC}}$, binding is enforced via $\mathcal{F}_{\mathsf{NC}}$'s internal records, specifically CState. For all other commitments, $\mathcal{F}_{\mathsf{NC}}$ runs the extraction algorithm. Explicitly, when $\mathcal{F}_{\mathsf{NC}}$ is invoked with message $(\mathtt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$:

1. If there exists an entry $(\overline{\mathsf{cm}}, m, \cdot, \cdot, 1) \in \mathsf{CState}$ such that $\overline{m} \neq m$, reject. This ensures perfect binding.
2. If there does not exist $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, \cdot) \in \mathsf{CState}$, i.e. $\overline{\mathsf{cm}}$ is adversarially-generated, $\mathcal{F}_{\mathsf{NC}}$ runs the extraction algorithm $\mathsf{SCode.Extr}$ to extract a message. Since $\mathsf{SCode.Extr}$ is deterministic and only receives $\overline{\mathsf{cm}}$ as input, there is only a single possible extracted message. (The $\mathcal{F}_{\mathsf{NC}}$ of this paper defines $\mathsf{SCode.Extr}$ to output both m and dcm. We note that having dcm as an additional output does not affect binding; we omit dcm for simplicity of ensuing exposition.)

In contrast, for $\mathcal{F}_{\mathsf{NC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$, fixing the $\mathsf{SCode.Extr}$ does not fully fix the injective function between commitments and message because $\mathsf{SCode.Extr}$ can depend on the function table of $\mathcal{F}_{\mathsf{GRO}}$, which may change over time due to programming and the fact that $\mathcal{F}_{\mathsf{NC}}$ observes the environment's oracle queries. Instead, commitment strings are bound to messages during the course of execution, at various binding points.

Towards a binding definition for $\mathcal{F}_{\mathsf{NC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$. To allow extraction depending on the observed function table of $\mathcal{F}_{\mathsf{GRO}}$, $\mathsf{SCode.Extr}$ additionally receives $\mathsf{Hist}_{\mathsf{GRO}}$, i.e. the list of GRO queries made by $\mathcal{E}$ in the domain controlled by $\mathcal{F}_{\mathsf{NC}}$ (in which queries are prefixed by the session ID sid of $\mathcal{F}_{\mathsf{NC}}$).

Unfortunately, this presents an opportunity for $\mathcal{E}$ to give auxiliary information to the extractor after a commitment string cm is issued, via querying $\mathcal{F}_{\mathsf{GRO}}$ in the domain controlled by $\mathcal{F}_{\mathsf{NC}}$. Since $\mathsf{Hist}_{\mathsf{GRO}}$ is an input to $\mathsf{SCode.Extr}$, it could affect the message output by $\mathsf{SCode.Extr}(\mathsf{cm}, \mathsf{Hist}_{\mathsf{GRO}})$, thus changing the message to which cm is bound.

Hence, for $\mathcal{F}_{\mathsf{NC}}$ to intuitively express a binding commitment, we must identify the *binding point* at which a commitment string is bound to a single message. (Different commitment strings can have different binding points.) Further, we desire that the message m to which a commitment string is bound to does not change after the binding point, despite the fact that $\mathcal{F}_{\mathsf{NC}}$ observes the oracle queries of $\mathcal{E}$; in particular, only m will be valid when $\mathcal{F}_{\mathsf{NC}}$ verifies using cm. We note that this binding property is weaker than the binding guarantee of $\mathcal{F}_{\mathsf{NC}}$ in the plain model, in the sense that the message cm is bound to is not fixed ahead of time, as in the case of $\mathcal{F}_{\mathsf{NC}}$ in the plain model.

To build towards a binding definition for $\mathcal{F}_{\mathsf{NC}}$, we first identify the *binding point(s) for* cm in the execution, i.e. when the commitment string cm is bound to a single m. For $\mathcal{F}_{\mathsf{NC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$, there are three possible binding points for cm: (1) cm is output by $\mathcal{F}_{\mathsf{NC}}$ after an honest invocation to the **Commit** subroutine with message m; or (2) cm is output by $\mathcal{F}_{\mathsf{GRO}}$ after it receives m as input; or (3) cm was neither generated by the **Commit** subroutine, nor is an an output of $\mathcal{F}_{\mathsf{GRO}}$, so binding is guaranteed by the fact that extraction algorithm $\mathsf{SCode.Extr}$ is deterministic, as in [CSW22].

Then, we can intuitively capture binding for $\mathcal{F}_{\mathsf{NC}}$ as follows: if $\mathcal{F}_{\mathsf{NC}}$ accepts (i.e. outputs $(\texttt{VerStatus}, 1)$) on input $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$, then $\overline{\mathsf{cm}}$ was already "bound" to $\overline{m}$ at the binding point for $\overline{\mathsf{cm}}$ during the execution. In other words, once the binding point occurs, $\mathcal{E}$ cannot change the message output by the extractor.

We formalize the above intuition by defining the notion of an extractable ideal commitment mechanism, then argue that no adversary wins the binding game with respect to $\mathcal{F}_{\mathsf{NC}}$ with more than negligible probability.

Definition 4.4 (Extractable ideal commitment mechanism)**.** *We say that an ideal commitment mechanism* C *(Definition 4.1) is* ϵ*-extractable if there is some extraction function* Extr *from states of an execution to the plaintext space of* C*, such that any* $\mathcal{E}$ *wins the following game with probability at most* ϵ*:*

1. *Run* $\mathcal{E}$ *in a standard execution in the UC model with* C *(with session ID* sid*),* $\mathcal{F}_{\mathsf{GRO}}$*, and* $\mathcal{F}_{\mathsf{lib}}$ *until* $\mathcal{E}$ *receives an output* $(\texttt{VerStatus}, 1)$ *in response to an input* $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ *to* C.
2. *For* $\overline{\mathsf{cm}}$*, rewind the execution to the latest of the following* binding points*, if they exist:*
   Case 1. $\overline{\mathsf{cm}}$ *is output by* C *as a commitment. Then in this case, the binding point is the state* State *of the (entire) execution immediately before the subsequent activation of* $\mathcal{E}$*.*
   Case 2. The first time $\mathcal{F}_{\mathsf{GRO}}$ *outputs* $(\texttt{EvalOutput}, \overline{\mathsf{cm}})$*, in response to an* $\texttt{Eval}$ *or* $\texttt{Program}$ *input in which* x *has the form* $(\mathsf{sid}, \cdot)$*. Then in this case, the binding point is the state* State *of the (entire) execution immediately before the subsequent activation of* $\mathcal{E}$*.*
   Case 3. $\overline{\mathsf{cm}}$ *is not output by neither* C *nor* $\mathcal{F}_{\mathsf{GRO}}$*. In this case, the binding point is the state* State *of the execution after the initialization of* C*, immediately before the next activation of* $\mathcal{E}$*.*
3. *Extract a message* $\widehat{m}$ *using the state* State *at the binding point:*
   (a) In Case 1, output the message $\widehat{m}$ *that* C *used to generate* $\overline{\mathsf{cm}}$.
   (b) In Case 2 (resp. Case 3, run $(\widehat{m}, \widehat{\mathsf{dcm}}) \leftarrow \mathsf{Extr}(\mathsf{sid}, \overline{\mathsf{cm}}, \mathsf{State})$.
4. $\mathcal{E}$ *wins if the extracted message* $\widehat{m} \neq \bot$ *and* $\widehat{m} \neq \overline{m}$*.*

Next, we show that our $\mathcal{F}_{\mathsf{NC}}$ is an extractable ideal commitment mechanism. We note that $\mathcal{F}_{\mathsf{NC}}$ has a parameter for the length or min-entropy of commitment strings: h_{cm}, which is required in the proof of binding for $\mathcal{F}_{\mathsf{NC}}$.

Theorem 4.5. *Let* $h_{\mathsf{cm}}, h_{\mathsf{dcm}} \in \mathbb{N}$*. Let* $\mathcal{E}$ *be an adversary with runtime* t*. For any (adversarially-chosen)* SCode*,* $\mathcal{E}$ *interacting with* $\mathcal{F}_{\mathsf{NC}}(h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ *(defined in Figures 3 and 4) in the presence of* $\mathcal{F}_{\mathsf{GRO}}$ *and* $\mathcal{F}_{\mathsf{lib}}$ *is an* $(2t^2 \cdot 2^{-h_{\mathsf{cm}}})$*-extractable ideal commitment mechanism (Definition 4.4) with respect to the extraction function* $\mathsf{SCode.Extr}$ *that is used to define* $\mathcal{F}_{\mathsf{NC}}(h_{\mathsf{cm}}, h_{\mathsf{dcm}})$*.*

Proof of Theorem 4.5. First, observe that $\mathcal{F}_{\mathsf{NC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$ is an ideal commitment mechanism, as defined in Definition 4.1, via inspecting the interfaces of $\mathcal{F}_{\mathsf{NC}}$.

Second, we argue that $\mathcal{F}_{\mathsf{NC}}$ is an $(2t^2 \cdot 2^{-h_{\mathsf{cm}}})$-extractable ideal commitment mechanism, i.e. that any adversary $\mathcal{E}$ wins the game defined in Definition 4.4 with respect to the extraction algorithm $\mathsf{SCode.Extr}$ with probability at most $2t^2 \cdot 2^{-h_{\mathsf{cm}}}$. To specialize the game in Definition 4.4 to the ideal commitment mechanism $\mathcal{F}_{\mathsf{NC}}$, we consider the execution state at the binding point State to include the query-answer pairs of $\mathcal{F}_{\mathsf{GRO}}$, which is used by the $\mathsf{SCode.Extr}$ of $\mathcal{F}_{\mathsf{NC}}$.

Suppose $\mathcal{E}$ interacts arbitrarily with $\mathcal{F}_{\mathsf{NC}}$ until $\mathcal{F}_{\mathsf{NC}}$ accepts on input $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$; we refer to this as the "initial execution". We define a bad event as $\mathcal{E}$ extracting a plaintext $\widehat{m}$ at the binding point such that $\widehat{m} \neq \overline{m}$ and $\widehat{m} \neq \perp$.

Next, we argue that the bad event occurs with probability $2t^2 \cdot 2^{-h_{\mathsf{cm}}}$. To do so, we consider cases depending on when the binding point for $\overline{\mathsf{cm}}$ occurs during the game.

Case 1. $\overline{\mathsf{cm}}$ is output by $\mathcal{F}_{\mathsf{NC}}$ as a commitment, after an invocation of the **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$.

This means that $\overline{\mathsf{cm}}$ is bound by an entry $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, 1) \in \mathsf{CState}$ (in Step II(e)ii of $\mathcal{F}_{\mathsf{NC}}$). We argue that there is at most a single message value m for which $(\overline{\mathsf{cm}}, m, \cdot, \cdot, 1) \in \mathsf{CState}$ of $\mathcal{F}_{\mathsf{NC}}$.

When the **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$ is invoked, Step II(c)i of $\mathcal{F}_{\mathsf{NC}}$ prevents having more than one entry of the form $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, 1)$ in CState. When the **Verify** subroutine of $\mathcal{F}_{\mathsf{NC}}$ is invoked, Step IVc of $\mathcal{F}_{\mathsf{NC}}$ prevents having more than one entry of the form $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, 1)$ in CState. Hence, the probability of the bad event is 0.

Case 2. $\overline{\mathsf{cm}}$ appears as an output in the function table Hist of $\mathcal{F}_{\mathsf{GRO}}$.

We consider cases, depending on when the binding point occurs and $\mathcal{E}$'s interactions with $\mathcal{F}_{\mathsf{NC}}$.

Note the execution of **Verify** never programs or queries $\mathcal{F}_{\mathsf{GRO}}$ due to $\mathcal{F}_{\mathsf{NC}}$'s interface with $\mathcal{F}_{\mathsf{GRO}}$ (Steps 3 and 4 of $\Phi_{\mathcal{F}_{\mathsf{NC}}}$). Hence, the binding point could not have occurred during the execution of **Verify**. Also, **Commit** of $\mathcal{F}_{\mathsf{NC}}$ can only be invoked once, due to Step IIa.

As such we give cases depending on when the binding point occurs, relative to $\mathcal{E}$'s invocation of **Commit** of $\mathcal{F}_{\mathsf{NC}}$.

Case (a). In the initial execution, $\mathcal{E}$ receives $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$, in which $x = (\mathsf{sid}, \widehat{m}, \mathsf{dcm})$, and $\mathcal{E}$ never invokes the **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$. (Hence, the message $\widehat{m}$ is extracted at the binding point.) Note that, since the **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$ is never invoked, $\mathcal{F}_{\mathsf{NC}}$ accepts on input $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ only if Step IVf of $\mathcal{F}_{\mathsf{NC}}$ yields accept, i.e. the extracted message during verification is $\overline{m}$.

Now, we argue that the probability of the bad event is $t^2 \cdot 2^{-h_{\mathsf{cm}}}$.

If verification accepts for (the same) $\overline{\mathsf{cm}}$ and $\widehat{m} \neq \overline{m}$, there must exist $(x, \overline{\mathsf{cm}}) \in Q$ such that $x = (\mathsf{sid}, \widehat{m}, \widehat{\mathsf{dcm}})$ due to Step IV(f)i of $\mathcal{F}_{\mathsf{NC}}$. This implies $\mathcal{E}$ finding a collision in $\mathcal{F}_{\mathsf{GRO}}$; this occurs with probability $t^2 \cdot 2^{-h_{\mathsf{cm}}}$ by the birthday bound.

Case (b). In the initial execution, $\mathcal{E}$ first receives $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$ in which $x = (\mathsf{sid}, \widehat{m}, \widehat{\mathsf{dcm}})$. Then, $\mathcal{E}$ invokes $\mathcal{F}_{\mathsf{NC}}$ with inputting $(\mathsf{Commit}, \overline{m} \neq \widehat{m})$ and receives $(\texttt{Receipt}, \mathsf{cm} = \overline{\mathsf{cm}}, \overline{\mathsf{dcm}})$. In this case, the "latest" binding point will be after $\mathcal{F}_{\mathsf{NC}}$ is invoked.

The bad event never occurs in this case, since Step IVc of $\mathcal{F}_{\mathsf{NC}}$ ensures that only the tuple generated by $\mathcal{F}_{\mathsf{NC}}$'s **Commit** interface will be accepted: $(\overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$.

Case (c). In the initial execution, $\mathcal{E}$ does not receive $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$, prior to invoking $\mathcal{F}_{\mathsf{NC}}$ for the first time with **Commit**.

We have cases depending on whether the binding point occurs during the execution of **Commit** of $\mathcal{F}_{\mathsf{NC}}$, or after $\mathcal{E}$ gets output from invoking **Commit**.

First, suppose the binding point occurs during the invocation of **Commit** and $\overline{\mathsf{cm}}$ is not the commitment output by $\mathcal{F}_{\mathsf{NC}}$. Then, Step IVe of $\mathcal{F}_{\mathsf{NC}}$ prevents $\mathcal{F}_{\mathsf{NC}}$ from accepting when the binding point for $\overline{\mathsf{cm}}$ occurs due to programming done by $\mathsf{SCode.Com}$ or $\mathsf{SCode.Equiv}$.

Second, (if the binding point occurs after the execution of the **Commit** subroutine of

$\mathcal{F}_{\mathsf{NC}}$ and $\overline{\mathsf{cm}}$ is not the commitment output by $\mathcal{F}_{\mathsf{NC}}$.), $\mathcal{E}$ can influence the outputs of SCode.Extr only by changing the inputs, specifically the query set $Q := \{(x, y) \mid y = \overline{\mathsf{cm}} \wedge (x, y) \in \mathsf{Hist}_{\mathsf{GRO}}\}$ computed in Step IV(f)i of $\mathcal{F}_{\mathsf{NC}}$. This can be done by finding and querying a collision in $\mathcal{F}_{\mathsf{GRO}}$. As such, the probability $\mathcal{E}$ finds a collision in $\mathcal{F}_{\mathsf{GRO}}$ is $t^2 \cdot 2^{-h_{\mathsf{cm}}}$ by the birthday bound.

Lastly, we note that $\mathcal{F}_{\mathsf{NC}}$'s output $(\mathtt{Receipt}, \mathsf{cm}, \mathsf{dcm})$ does not leak information that helps $\mathcal{E}$ find a collision, since cm (resp. dcm) must be an entry in $\mathcal{F}_{\mathsf{NC}}$'s random tape r_{cm} (resp. r_{dcm}).

Hence, the bad event occurs with probability $t^2 \cdot 2^{-h_{\mathsf{cm}}}$.

Case 3. Neither Case 1 nor Case 2 occurs, i.e. the binding point is the beginning of the execution of Fork 1. This implies $\mathsf{SCode.Extr}(\overline{\mathsf{cm}}, \perp) = \mathsf{SCode.Extr}(\overline{\mathsf{cm}}, \mathsf{Hist}_{\mathsf{GRO}})$ since Hist of $\mathcal{F}_{\mathsf{GRO}}$ does not bind $\overline{\mathsf{cm}}$. Hence, the probability of the bad event is 0.

By union bound over the cases, $\mathcal{E}$ wins the binding game with probability $2t^2 \cdot 2^{-h_{\mathsf{cm}}}$. □

4.2.4 Ensuring immediate response and correctness

Immediate response. The execution of $\mathcal{F}_{\mathsf{NC}}$ requires running code SCode that is pre-defined by the simulator, relative to a particular protocol. There are two possible ways to model "$\mathcal{F}_{\mathsf{NC}}$ receiving SCode". One way is for $\mathcal{F}_{\mathsf{NC}}$ to obtain SCode directly from the simulator when $\mathcal{F}_{\mathsf{NC}}$ is first invoked with a `Commit` message. Unfortunately, this modeling of $\mathcal{F}_{\mathsf{NC}}$ allows situations in which the commitment is never generated; the adversary can block the generation of a commitment, i.e. run a denial-of-service (DoS) attack, by never sending SCode to $\mathcal{F}_{\mathsf{NC}}$. We want to model commitment schemes in which the commitment and decommitment strings are always generated, i.e. those that only require local processing.

To do so, we use another method for modeling the delivery of SCode to $\mathcal{F}_{\mathsf{NC}}$: via the code library functionality $\mathcal{F}_{\mathsf{lib}}$ (Figure 1) defined in Section 2.3. The simulator deposits SCode in $\mathcal{F}_{\mathsf{lib}}$, which allows $\mathcal{F}_{\mathsf{NC}}$ to retrieve SCode when needed. This prevents the DoS attack because $\mathcal{F}_{\mathsf{lib}}$'s internal logic ensures it is guaranteed to respond; if the simulator never uploads SCode to $\mathcal{F}_{\mathsf{lib}}$, $\mathcal{F}_{\mathsf{lib}}$ answers with $\mathsf{SCode} = \perp$, which allows the execution to continue. Still, $\mathcal{F}_{\mathsf{NC}}$ must handle the case in which it receives $\perp$, to ensure that $\mathcal{F}_{\mathsf{NC}}$ works correctly. This is handled via running default code that preserves the desired security properties of $\mathcal{F}_{\mathsf{NC}}$, which we detail in the coming paragraphs. We note that [CJSV22] also makes use of $\mathcal{F}_{\mathsf{lib}}$ to guarantee immediate response of their ideal functionalities.

Correctness. We guarantee that $\mathcal{F}_{\mathsf{NC}}$ *always* returns an output that satisfies hiding and binding when the **Commit** interface is invoked; we call this property correctness. To ensure that $\mathcal{F}_{\mathsf{NC}}$ is correct, we employ the strategy of running default code if there is an error that prevents $\mathcal{F}_{\mathsf{NC}}$ from providing a valid output. This idea of having a default code was inspired by the work of [CDLLR24] in the context of signature functionalities, in order to ensure that the functionality provides a service that is always available. We note that while [CJSV22] makes use of $\mathcal{F}_{\mathsf{lib}}$, they do not explicitly consider this situation.

Below, we describe the default SCode behavior in $\mathcal{F}_{\mathsf{NC}}$ and list the error cases in which the default SCode is employed.

The default SCode does the following. In the **Commit** stage, the default SCode.Com chooses a random string of the appropriate length to be the commitment string, and SCode.Equiv chooses another random string with the appropriate length as the decommitment string[6] and $\perp$ as the committer state. In the **Verify** stage,

[6] This default code does not model protocols which emit low entropy commitment or decommitment strings. We do not handle this case in our formulation of default code.

since extraction is not needed for maintaining the completeness of $\mathcal{F}_{\mathsf{NC}}$, the default extraction algorithm simply returns $\perp$.

Next, the failure conditions that we protect against include:

- The adversary does not provide SCode to $\mathcal{F}_{\mathsf{lib}}$, prior to invoking $\mathcal{F}_{\mathsf{NC}}$.
- The SCode for $\mathcal{F}_{\mathsf{NC}}$ does not output a decommitment dcm with sufficient (conditional) min-entropy.
- In the commitment phase, $\mathcal{F}_{\mathsf{NC}}$ generates cm that was already submitted for verification to $\mathcal{F}_{\mathsf{NC}}$.
- In the commitment phase, $\mathsf{SCode.Com}$ or $\mathsf{SCode.Equiv}$ halts without output or outruns its time bound.

To make the description of $\mathcal{F}_{\mathsf{NC}}$ concise, we define the following errors, which an SCode algorithms may commit:

Definition 4.6. *An algorithm $\mathcal{A}$ in simulator code* SCode *may commit the following errors:*

- $\mathsf{SCodeTimeError}$*: $\mathcal{A}$ outruns its time budget; and*
- $\mathsf{SCodeError}$*: $\mathcal{A}$ halts without output (i.e. outputs $\perp$) or commits* $\mathsf{SCodeTimeError}$.

4.2.5 Adaptive security

$\mathcal{F}_{\mathsf{NC}}$ models non-interactive commitment schemes that are stateless, i.e. the commit and verify algorithms do not share any secret state.[7] Thus, any adversary that corrupts the committer, at any point, learns no information. Hence, the simulator can independently generate this empty view, so $\mathcal{F}_{\mathsf{NC}}$ has no corruption interface. Further, such commitment schemes are adaptively secure if the committer deletes its internal state, prior to outputting the commitment and associated decommitment $(\mathsf{cm}, \mathsf{dcm})$.

Protocol Π_{NC}

- **Parameters:** the decommitment length $h_{\mathsf{dcm}} \in \mathbb{N}$, the $\mathcal{F}_{\mathsf{GRO}}$ output length $h_{\mathsf{cm}} \in \mathbb{N}$, and the message alphabet Σ.
- **Commit:** The committer C does:
  1. Receive input $(\texttt{Commit}, m)$.
  2. If $m = \perp$, sample $\mathsf{cm} \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$ and set $\mathsf{dcm} := \perp$.
  3. Else:
     (a) Sample $\mathsf{dcm} \leftarrow \{0,1\}^{h_{\mathsf{dcm}}}$.
     (b) Send subroutine input $(\texttt{Eval}, (\mathsf{sid}, \mathsf{dcm}, m))$ to $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$, where sid is the session ID of Π_{NC}.
     (c) Receive subroutine output $(\texttt{EvalOutput}, y)$ from $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$.
  4. Output $(\texttt{Commit}, \mathsf{cm} := y, \mathsf{dcm})$.
- **Verify:** The receiver R verifies that the decommitment is valid, as follows:
  1. Receive input $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$.
  2. Send subroutine input $(\texttt{Eval}, (\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$ to $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$, where sid is the session ID of Π_{NC}.
  3. Receive subroutine output $(\texttt{EvalOutput}, y)$ from $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$.
  4. If $\overline{\mathsf{cm}} = y$, output $(\texttt{VerStatus}, 1)$. Else, output $(\texttt{VerStatus}, 0)$.

Figure 7: The commitment protocol, with global subroutine $\mathcal{F}_{\mathsf{GRO}}$.

[7] In comparison, in stateful commitment schemes, the verification procedure requires saved information from the commitment stage.

5 Π_{NC}: The non-interactive commitment scheme

We describe a commitment protocol Π_{NC} (see Figure 7) and prove that Π_{NC} UC-realizes $\mathcal{F}_{\mathsf{NC}}$ (Theorem 5.1); the proof appears in Section 5.1.

Theorem 5.1. *Let $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ be an ideal global random oracle, with output length $h_{\mathsf{cm}} \in \mathbb{N}$. Let $h_{\mathsf{dcm}} \in \mathbb{N}$ be a parameter denoting decommitment length.*

Then, $\Pi_{\mathsf{NC}}(h_{\mathsf{dcm}}, h_{\mathsf{cm}})$ in Figure 7 UC-realizes the ideal commitment functionality $\mathcal{F}_{\mathsf{NC}}(h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ in the presence of $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ and $\mathcal{F}_{\mathsf{lib}}$, even against adaptive corruptions. Specifically, an environment making at most t queries to $\mathcal{F}_{\mathsf{GRO}}$ has distinguishing advantage at most $t \cdot 2^{-h_{\mathsf{dcm}}} + (2t^2 + 3t) \cdot 2^{-h_{\mathsf{cm}}}$.

Furthermore, in Π_{NC}, commitments are always h_{cm}-bit strings, regardless of the length of the plaintext.

5.1 Proof of Theorem 5.1

We prove Theorem 5.1 in the following steps:

1. Define the code library SCode, which is used to simulate the execution of Π_{NC}, in Section 5.1.1.
2. Argue that $\mathsf{Exec}_{\mathcal{E},\Pi_{\mathsf{NC}}} \approx \mathsf{Exec}_{\mathcal{E},\mathcal{F}_{\mathsf{NC}},\mathsf{SCode}}$ in Section 5.1.2.

5.1.1 The simulator's code library

We define the simulator's code library $\mathsf{SCode} = \{\mathsf{Com}, \mathsf{Equiv}, \mathsf{Extr}\}$. Note that these algorithms are (non-adaptively) chosen by the adversary.

```
• $\mathsf{Com}(\mathsf{sid}, r_{\mathsf{cm}}) \to (\mathsf{cm}, \mathsf{st})$.
    1. Parse sid as a session ID and $r_{\mathsf{cm}}$ as a random tape with entries in $\{0,1\}^{h_{\mathsf{cm}}}$.
    2. Compute $\mathsf{cm} := r_{\mathsf{cm}}[0] \in \{0,1\}^{h_{\mathsf{cm}}}$.
    3. Output $(\mathsf{cm}, \mathsf{st} := \bot)$.
```

```
• $\mathsf{Equiv}(\mathsf{sid}, \mathsf{cm}, \mathsf{st}, m, r_{\mathsf{dcm}}) \to (\mathsf{dcm}, \mathsf{State}^{\mathsf{NC}})$.
    1. Parse sid as a session ID, cm as a commitment string, st as an internal state, $m$ as a message, and
       $r_{\mathsf{dcm}}$ as a random tape with entries in $\{0,1\}^{h_{\mathsf{dcm}}}$.
    2. For $i \in [\tau]$: # For each $i \in [\tau]$, attempt to program the point $((\mathsf{sid}, \mathsf{dcm}, m), \mathsf{cm})$ in $\mathcal{F}_{\mathsf{GRO}}$.
        (a) Compute the decommitment $\mathsf{dcm} := r_{\mathsf{dcm}}[i] \in \{0,1\}^{h_{\mathsf{dcm}}}$.
        (b) Run the subroutine $\mathsf{Program}(\mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}_{\mathsf{NC}}}, \mathsf{sid}, \mathsf{SCode.Equiv}, x, y)$ (Figure 5)[a], and receive
            the output $(\mathtt{EvalOutput}, y)$. # Attempt to program the point $(x, y)$ in $\mathcal{F}_{\mathsf{GRO}}$.
        (c) If $y = \mathsf{cm}$, output $(\mathsf{dcm}, \mathsf{State}^{\mathsf{NC}} := \bot)$. # Programming successful.
        (d) Else: # Errors.
            i. If $y \neq \bot$ and $y \neq \mathsf{cm}$, return to Step 2a. # $y$ was previously fixed as the output of $x$.
            ii. If $y = \bot$, output $(\mathsf{dcm} := \bot, \mathsf{State}^{\mathsf{NC}} := \bot)$. # The point $(x, y)$ fails a check in $\Phi_{\mathcal{F}_{\mathsf{NC}}}$
                (Figure 6).
```

[a] The following inputs are honestly derived from the state of $\mathcal{F}_{\mathsf{NC}}$: $\mathsf{Hist}_{\mathsf{GRO}}$ is the GRO history known to $\mathcal{F}_{\mathsf{NC}}$, $\Phi_{\mathcal{F}_{\mathsf{NC}}}$ is the query-answer predicate of Figure 6, sid is the session ID of $\mathcal{F}_{\mathsf{NC}}$, and $\mathsf{ActiveAlg} := \mathsf{SCode.Equiv}$ is the calling algorithm. The following inputs may be chosen by $\mathsf{SCode.Equiv}$: x is the query point, and y is the query answer.

```
• $\mathsf{Extr}(\mathsf{sid}, \mathsf{cm}, Q) \to (m, \mathsf{dcm})$.
  1. Parse sid as a session ID, cm as a commitment string, and $Q \subseteq \{0,1\}^* \times \{0,1\}^{h_{\mathsf{cm}}}$ as a list of
     query-answer pairs for GRO.
  2. Derive the message $m$:
     (a) If there exists an entry of the form $((\mathsf{sid}, x'), \mathsf{cm})$ in $Q$, parse $x'$ as
         $(\mathsf{dcm} \in \{0,1\}^{h_{\mathsf{dcm}}}, m \in \{0,1\}^{|x'|-h_{\mathsf{dcm}}})$.
     (b) Else, output $(m := \bot, \mathsf{dcm} := \bot)$. # Extraction fails.
  3. If there exists an entry $((\mathsf{sid}, \mathsf{dcm}, m), \mathsf{cm}) \in Q$, output $(m, \mathsf{dcm})$. # Verification check passes.
  4. Else, output $(m := \bot, \mathsf{dcm} := \bot)$. # Verification check fails.
```

Note that Extr does not query or program $\mathcal{F}_{\mathsf{GRO}}$ and only has access to $\mathsf{Hist}_{\mathsf{GRO}}$.

5.1.2 Validity of the simulation

Let $\mathcal{E}$ be an environment machine, let $\mathsf{REAL} := \mathsf{Exec}_{\mathcal{E},\Pi_{\mathsf{NC}}}$, and let $\mathsf{IDEAL} := \mathsf{Exec}_{\mathcal{E},\mathcal{F}_{\mathsf{NC}},\mathcal{S}_{\mathsf{NC}}}$. Without loss of generality, assume that $\mathcal{E}$ runs in time t and thus makes at most t queries to $\mathcal{F}_{\mathsf{GRO}}$. Let s_{NC} denote the statistical distance between REAL and IDEAL. We argue that

$$s_{\mathsf{NC}} \leq t \cdot 2^{-h_{\mathsf{dcm}}} + (2t^2 + 3t) \cdot 2^{-h_{\mathsf{cm}}} \quad .$$

First, we identify the bad events, in which the REAL execution accepts, but the IDEAL execution has an error (and outputs random strings for $\mathsf{cm}, \mathsf{dcm}$), outputs $\bot$, or rejects. We also compute the probabilities with which the bad events occur:

Bad Event 1. In IDEAL, $\mathsf{SCode.Equiv}$ fails to program $\mathcal{F}_{\mathsf{GRO}}$, i.e. it receives $\bot$ in Step 2b. This causes a bad event because there is no corresponding programming error in REAL, because commitments are computed via querying $\mathcal{F}_{\mathsf{GRO}}$.
We show that the probability that $\mathsf{SCode.Equiv}$ fails to sample dcm, i.e. outputs $\bot$, after τ attempts, is upper bounded by $t \cdot 2^{-h_{\mathsf{dcm}}}$.

Lemma 5.2. *For a* $\mathsf{cm} \in \{0,1\}^{h_{\mathsf{cm}}}$ *fixed by* Com *and* $\tau \in \mathbb{N}$ *repetitions,*

$$\Pr\left[\, \textit{Step 2b in } \mathsf{Equiv} \textit{ receives } \bot \,\right] \leq (t \cdot 2^{-h_{\mathsf{dcm}}})^{\tau} \quad .$$

Proof. The event "Step 2b in Equiv receives $\bot$" occurs if Step 2 of $\mathsf{SCode.Equiv}$ fails to program, i.e. all of the τ attempts at programming $\mathcal{F}_{\mathsf{GRO}}$ fails.
We define events $X_1, \ldots, X_\tau$, in which X_i is the event that the i-th attempt at programming $\mathcal{F}_{\mathsf{GRO}}$ (i.e. the i-th iteration of Step 2 of $\mathsf{SCode.Equiv}$) fails.
Then:

$$\Pr\left[\, \mathsf{Step\ 2b\ in\ Equiv\ outputs} \ \bot \,\right] = \Pr\left[\bigwedge_{i=1}^{\tau} X_i\right] = \prod_{i=1}^{\tau} \Pr\left[\, X_i \,\right] \quad . \tag{1}$$

Above, the final equality follows because the dcm in each iteration of Step 2 of $\mathsf{SCode.Equiv}$ is independently sampled.
We analyze $\Pr[X_i]$. In attempt i, there are at most t locations in $\mathcal{F}_{\mathsf{GRO}}$ which cannot be programmed because they were previously queried and the query bound of $\mathcal{E}$ is t. Denote this set of queries BAD. (In particular, in the worst case, every query in BAD has the format (sid, r, m) and $r \in \{0,1\}^{h_{\mathsf{dcm}}}$.) Then, since dcm is uniformly sampled, we have that

$$\Pr[\, X_i \,] = \Pr\left[\, (\mathsf{sid}, \mathsf{dcm}, m) \in \mathsf{BAD} \;\middle|\; \mathsf{dcm} \leftarrow \{0,1\}^{h_{\mathsf{dcm}}} \,\right] \leq t \cdot 2^{-h_{\mathsf{dcm}}} \quad . \tag{2}$$

Plugging Equation 2 into Equation 1 proves the lemma. □

Thus by Lemma 5.2, the programming fails w.p. $\leq t \cdot 2^{-h_{\mathsf{dcm}}}$,[8] so $\mathcal{E}$'s distinguishing advantage when this bad event occurs is $\leq t \cdot 2^{-h_{\mathsf{dcm}}}$.

Bad Event 2. $\mathcal{E}$ guesses a query-answer pair $Q^* = ((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}), \overline{\mathsf{cm}})$ in $\mathcal{F}_{\mathsf{GRO}}$ without querying $\mathcal{F}_{\mathsf{GRO}}$. This causes a bad event if $\mathcal{E}$ submits $(\mathtt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ to **Verify**. In IDEAL, the query-answer pair Q^* does not exist in $\mathsf{Hist}_{\mathsf{GRO}}$, i.e. will not be given to $\mathsf{SCode.Extr}$ in Step IV(f)ii, so Step 2b of $\mathsf{SCode.Extr}$ outputs $\perp$. However in REAL, **Verify** accepts since $\mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})) = \overline{\mathsf{cm}}$.
This bad event occurs with probability $2^{-h_{\mathsf{cm}}}$ since $2^{-h_{\mathsf{cm}}}$ is the probability any "new" query to $\mathcal{F}_{\mathsf{GRO}}$ evaluates to cm.

Bad Event 3. $\mathcal{E}$ finds a collision in $\mathcal{F}_{\mathsf{GRO}}$, i.e. $\mathcal{E}$ finds $(\mathsf{dcm}, m) \neq (\overline{\mathsf{dcm}}, \overline{m})$ such that $\mathsf{GRO}(\mathsf{sid}, \mathsf{dcm}, m) = \mathsf{GRO}(\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}) = \mathsf{cm}$. This causes a bad event when, in IDEAL, there already exists an entry of the form $(\mathsf{cm}, m, \mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}, 1)$ in CState. However in REAL, **Verify** accepts if $\mathcal{E}$ guesses and outputs $(\overline{m}, \overline{\mathsf{dcm}})$ such that

$$\mathsf{cm} = \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})) \ , \tag{3}$$

i.e. $\mathcal{E}$ finds a collision in $\mathcal{F}_{\mathsf{GRO}}$ for cm.
Since the output of $\mathcal{F}_{\mathsf{GRO}}$ has length h_{cm} and $\mathcal{E}$ is t-query, this occurs with probability $t^2 \cdot 2^{-h_{\mathsf{cm}}}$, by the birthday bound. Hence, $\mathcal{E}$'s distinguishing advantage when this bad event occurs is $t^2 \cdot 2^{-h_{\mathsf{cm}}}$.

Second, we examine $\mathcal{E}$'s execution tree when interacting with REAL or with IDEAL and compute the distinguishing advantage of $\mathcal{E}$ for all possible cases. Specifically, for each event, we argue either that the executions are identical, or else one of the bad events (listed above) occurs. We ultimately conclude that the environment's overall distinguishing advantage is $t \cdot 2^{-h_{\mathsf{dcm}}} + (2t^2 + 3t) \cdot 2^{-h_{\mathsf{cm}}}$.

Case 1. $\mathcal{E}$ first invokes the **Commit** subroutine (i.e. there were no previous invocations of **Verify**).

We show that the executions are identical, except for a bad event which occurs in IDEAL if $\mathsf{SCode.Equiv}$ fails to program $\mathcal{F}_{\mathsf{GRO}}$. We proceed by comparing the execution traces of REAL and IDEAL.

Observe that in the executions of both REAL and IDEAL, given a message m and a session ID sid, the **Commit** subroutine samples values $(\mathsf{cm}, \mathsf{dcm})$ which satisfy the invariant[9]:

$$\mathsf{cm} = \mathsf{GRO}((\mathsf{sid}, \mathsf{dcm}, m)) \ . \tag{4}$$

Note that if the invariant is satisfied, then the **Verify** subroutines of both REAL and IDEAL will accept values generated by the honest **Commit** subroutine, i.e. the executions are identical.

In REAL, the invariant is always satisfied. However, in IDEAL, the invariant won't be satisfied if programming fails, i.e. $\mathsf{SCode.Equiv}$ receives $\perp$ in Step 2b. This is precisely Bad Event 1.

[8] For simplicity of the analysis, we ignore the number of repetitions τ.

[9] In REAL, the **Commit** subroutine samples $\mathsf{dcm} \leftarrow \{0,1\}^{h_{\mathsf{dcm}}}$, then computes $\mathsf{cm} := \mathsf{GRO}((\mathsf{sid}, \mathsf{dcm}, m))$. Observe that this always satisfies Equation 4. In IDEAL, the **Commit** subroutine does (a) $\mathsf{SCode.Com}$ samples $\mathsf{cm} \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$; (b) $\mathsf{SCode.Equiv}$ samples $\mathsf{dcm} \leftarrow \{0,1\}^{h_{\mathsf{dcm}}}$; then (c) $\mathsf{SCode.Equiv}$ attempts to program the query-answer pair $((\mathsf{sid}, \mathsf{dcm}, m), \mathsf{cm})$ in $\mathcal{F}_{\mathsf{GRO}}$ (Step 2b).

Case 2. $\mathcal{E}$ invokes the **Verify** subroutine with message $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ and there exists an entry $(\mathsf{cm} = \overline{\mathsf{cm}}, m = \overline{m}, \mathsf{dcm} = \overline{\mathsf{dcm}}, \cdot, v) \in \mathsf{CState}$.

There are two main cases:

Case (a). $v = 1$.
Then, IDEAL outputs $(\texttt{VerStatus}, 1)$. Since IDEAL records $(\mathsf{cm}, m, \mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}, 1) \in \mathsf{CState}$ if $\mathsf{cm} = \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$, REAL will also accept. So, REAL and IDEAL are identical.

Case (b). $v = 0$.
Then, IDEAL rejects. This corresponds to $\mathsf{cm} \neq \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$, so REAL also rejects.

Case 3. $\mathcal{E}$ invokes the **Verify** subroutine with message $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$, and there exists an entry $(\mathsf{cm} = \overline{\mathsf{cm}}, m, \mathsf{dcm}, \cdot, 1) \in \mathsf{CState}$ such that $\overline{m} \neq m$ or $\overline{\mathsf{dcm}} \neq \mathsf{dcm}$.

In detail, this case occurs if $\mathcal{E}$ invokes the **Commit** subroutine (of either REAL or IDEAL with session ID sid) with a message m, then corrupts the committer and receives $(\overline{\mathsf{cm}}, \overline{\mathsf{dcm}})$ and no additional state. After, $\mathcal{E}$ attempts to verify $(\texttt{Verify}, \overline{\mathsf{cm}}, m, \mathsf{dcm})$. There are two possibilities: either $\overline{m} \neq m$ or $\overline{m} = m$ and $\mathcal{E}$ attempts to verify with $\overline{\mathsf{dcm}} \neq \mathsf{dcm}$:

Case (a). $\overline{m} \neq m$.
In IDEAL, Step IVc of $\mathcal{F}_{\mathsf{NC}}$ rejects since there already exists the entry $(\overline{\mathsf{cm}}, m, \mathsf{dcm}, \cdot, 1)$ in CState, i.e. $\overline{\mathsf{cm}} = \mathsf{GRO}((\mathsf{sid}, \mathsf{dcm}, m))$. However, if $\mathcal{E}$ finds a collision in $\mathcal{F}_{\mathsf{GRO}}$, i.e. $\overline{\mathsf{cm}} = \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$, REAL accepts. This is precisely Bad Event 3.

Case (b). $\overline{m} = m$ and $\overline{\mathsf{dcm}} \neq \mathsf{dcm}$.
We argue that the executions are identical, up to the bad event that $\mathcal{E}$ guesses a query $x = (\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})$ such that $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$, without querying $\mathcal{F}_{\mathsf{GRO}}$.

First, we show that the executions of REAL and IDEAL are identical in the all other cases.

- $\overline{\mathsf{cm}} \neq \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$.
  Observe that REAL rejects. Also, IDEAL rejects because there is no entry $((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}), \mathsf{cm}) \in \mathsf{Hist}_{\mathsf{GRO}}$, so Step 2b of $\mathsf{SCode.Extr}$ outputs $\perp$.
- $\overline{\mathsf{cm}} = \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$ and the query $(\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})$ was previously made to $\mathcal{F}_{\mathsf{GRO}}$.
  Observe that REAL accepts. Also, IDEAL accepts because $\mathsf{Hist}_{\mathsf{GRO}}$ contains the (observed) $\mathcal{F}_{\mathsf{GRO}}$ query-answer pair $((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}), \mathsf{cm})$.

Second, we observe that the bad event is precisely Bad Event 2. Since $\mathcal{E}$ has runtime t, it can send at most t messages $(\texttt{Verify}, \overline{\mathsf{cm}} = \mathsf{cm}, \overline{m} = m, \overline{\mathsf{dcm}} \neq \mathsf{dcm})$ to $\mathcal{F}_{\mathsf{NC}}$. Hence, $\mathcal{E}$'s overall distinguishing advantage is $t \cdot \Pr[\text{Bad Event 2}]$.

Case 4. $\mathcal{E}$ invokes the **Verify** subroutine with message $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ and there is no entry $(\mathsf{cm} = \overline{\mathsf{cm}}, \cdot, \cdot, \cdot, 1) \in \mathsf{CState}$.

There are two possible execution paths (with respect to the IDEAL execution): either there is no entry of the form $(\mathsf{cm} = \overline{\mathsf{cm}}, \cdot, \cdot, \cdot, \cdot)$ in CState, or there exists an entry $(\mathsf{cm} = \overline{\mathsf{cm}}, m, \mathsf{dcm}, \cdot, 0) \in \mathsf{CState}$ such that $(m, \mathsf{dcm}) \neq (\overline{m}, \overline{\mathsf{dcm}})$.

We analyze the environment's distinguishing advantage for both execution paths:

Case (a). In IDEAL, there is no entry $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, \cdot)$ in CState.[10]

There are two possibilities:

i. $\overline{\mathsf{cm}} \neq \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$.

In this case, both REAL and IDEAL reject because both verification procedures reject when $\overline{\mathsf{cm}} \neq \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$.[11] Hence, the executions are identical.

ii. $\overline{\mathsf{cm}} = \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$.

There are two possible execution divergences in REAL and IDEAL, depending on whether $\mathcal{E}$ previously queried $\mathcal{F}_{\mathsf{GRO}}$ at $x = (\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})$.

A. $\mathcal{E}$ queried $\mathcal{F}_{\mathsf{GRO}}$ at $x = (\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})$ and receives output $\overline{\mathsf{cm}}$. Then, **Verify** accepts both in REAL and in IDEAL, i.e. the executions are identical.

B. $\mathcal{E}$ does not query $\mathcal{F}_{\mathsf{GRO}}$ at $x = (\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})$.

Instead, $\mathcal{E}$ guesses a tuple $(\overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ that satisfies $\overline{\mathsf{cm}} = \mathsf{GRO}((\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m}))$. This is precisely Bad Event 2. Since $\mathcal{E}$ has runtime t, it can send at most t messages $(\mathtt{Verify}, \cdot, \cdot, \cdot)$ to $\mathcal{F}_{\mathsf{NC}}$. Hence, $\mathcal{E}$'s overall distinguishing advantage is $t \cdot \Pr[\text{Bad Event 2}]$.

Case (b). In IDEAL, there exists an entry $(\overline{\mathsf{cm}}, m, \mathsf{dcm}, \cdot, 0)$ in CState such that $(m, \mathsf{dcm}) \neq (\overline{m}, \overline{\mathsf{dcm}})$.

We define a bad event, in which $\mathcal{E}$ finds a collision for $\overline{\mathsf{cm}}$ in $\mathcal{F}_{\mathsf{GRO}}$. (WLOG, suppose $\mathcal{E}$ does so via querying $\mathcal{F}_{\mathsf{GRO}}$.) Then, in REAL, the **Verify** subroutine accepts both on input values corresponding to the collision. However, in IDEAL, after the first time **Verify** accepts, there will exist an entry $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, 1) \in \mathsf{CState}$. Afterwards, the analysis from Case 3 applies. Hence, by union bound $\mathcal{E}$ ultimately distinguishes with probability $t \cdot \Pr[\text{Bad Event 2}] + \Pr[\text{Bad Event 3}]$.

The theorem is proved by union bound[12] over all of the cases. Hence, $\mathcal{E}$'s distinguishing advantage is:

$$\begin{aligned} s_{\mathsf{NC}} &\leq \Pr[\text{Bad Event 1}] + 3t \cdot \Pr[\text{Bad Event 2}] + 2 \cdot \Pr[\text{Bad Event 3}] \\ &\leq t \cdot 2^{-h_{\mathsf{dcm}}} + (2t^2 + 3t) \cdot 2^{-h_{\mathsf{cm}}} \quad . \end{aligned}$$

We note that the above analysis applies also to the case of adaptive corruption of the committer, since the (real) committer does not retain any internal state.

6 $\mathcal{F}_{\mathsf{NVC}}$: Non-interactive vector commitment

We introduce the ideal functionality for a non-interactive ***vector*** commitment $\mathcal{F}_{\mathsf{NVC}}$, which has access to global subroutine $\mathcal{F}_{\mathsf{GRO}}$. Recall that a vector commitment [CF13] is a primitive that allows committing to a vector $\mathbf{m}$, then decommitting to positions in the vector at a later time. Vector commitments allow a committer

[10] This can occur several ways, including: (a) $\mathcal{E}$ outputs a message $(\mathtt{Verify}, \overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$ without previously invoking either **Commit** or **Verify**; or (b) $\mathcal{E}$ previously runs the honest **Commit** or **Verify** subroutines at total of N times to get the view $\mathsf{View} = \{(\mathsf{cm}_i, m_i, \mathsf{dcm}_i, v_i)\}_{i \in [N]}$ and $\overline{\mathsf{cm}} \neq \mathsf{cm}_i$ for all i. These cases are identical because, in IDEAL, the **Verify** subroutine runs independently of the values in CState.

[11] Note that if $\mathcal{E}$ attempts to verify $(\overline{\mathsf{cm}}, \overline{m}, \overline{\mathsf{dcm}})$, where $(\mathsf{sid}, \overline{\mathsf{dcm}}, \overline{m})$ was not a previously-made query to $\mathcal{F}_{\mathsf{GRO}}$, the IDEAL behavior is that $\mathsf{SCode.Extr}$ outputs $(\perp, \perp)$, i.e. **Verify** outputs $(\mathtt{VerStatus}, 0, C)$.

[12] We use the union bound for simplicity of analysis. However, this expression can be improved by observing that $\mathcal{E}$ can always improve its distinguishing advantage by only running the attack with the maximal distinguishing advantage, i.e. by taking the $\max$ over all cases.

to jointly and succinctly commit to a vector of plaintexts, and then individually open each element in the vector. They have the following security properties: (a) **position binding**, i.e. decommitting any position in $\mathbf{m}$ to two different values is hard; and (b) **hiding**, i.e. even when some positions of $\mathbf{m}$ are decommitted, the non-decommitted positions remain hidden.

We present the description of $\mathcal{F}_{\mathsf{NVC}}$ in Figures 8 and 9. Below, we give an intuitive overview of how $\mathcal{F}_{\mathsf{NVC}}$ generalizes $\mathcal{F}_{\mathsf{NC}}$ to the vector setting. We make use of the following notation: given a vector $\mathbf{m} \in \Sigma^n$, we denote the i-th element of $\mathbf{m}$ as $\mathbf{m}[i]$.

At a high level, the interfaces of $\mathcal{F}_{\mathsf{NVC}}$ and $\mathcal{F}_{\mathsf{NC}}$ differ as follows.

- **Committing to a vector.** $\mathcal{F}_{\mathsf{NVC}}$ receives as input a vector of values $n \in \Sigma^n$. $\mathcal{F}_{\mathsf{NVC}}$ outputs a commitment string cm and a *vector* of decommitments $\mathbf{dcm}$. In particular, $\mathbf{dcm}[i]$ is the decommitment associated with $\mathbf{m}[i]$ for every $i \in [n]$.
- **Decommitting to (i.e. verifying) positions in $\mathbf{m}$.** Each invocation of **Verify** in $\mathcal{F}_{\mathsf{NVC}}$ checks a decommitment to a single index $i \in [n]$ of the committed vector $\mathbf{m}$. Specifically, verification messages have the form $(\texttt{Verify}, \overline{\mathsf{cm}}, \bar{i}, \overline{m}, \overline{\mathsf{dcm}})$, in which (a) $\overline{\mathsf{cm}}$ is a commitment string; (b) i is the position of $\mathbf{m}$ that the caller wants to decommit; (c) $\overline{m}$ is the claimed value for $\mathbf{m}[i]$; and (d) $\overline{\mathsf{dcm}}$ is the claimed decommitment for $\overline{m}$. Further, in order to verify a set of indices $I \subseteq [n]$, the calling party must invoke $(\texttt{Verify}, \cdot)$ for every $i \in I$.

Operationally, like for $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NVC}}$ obtains a set of adversarially-generated algorithms SCode from $\mathcal{F}_{\mathsf{lib}}$ in order to generate the ideal commitment string cm and decommitments $\mathbf{dcm}$. SCode includes code for generating the commitment string cm, generating a vector of decommitments $\mathbf{dcm}$, and for extracting a committed vector from a purported commitment string and verifying the result. SCode for $\mathcal{F}_{\mathsf{NVC}}$ contains four algorithms:

- $\mathsf{Setup}(n, \mathsf{sid}_{\mathsf{VC}})$.
  On input a message length n and a session ID $\mathsf{sid}_{\mathsf{VC}}$, Setup may only take controls of sub-sessions of $\mathsf{sid}_{\mathsf{VC}}$ in $\mathcal{F}_{\mathsf{GRO}}$, as well as instantiate any subroutine machines which may interact with $\mathcal{F}_{\mathsf{GRO}}$ with their sub-session ID.[13]
- $\mathsf{Com}(n, \mathsf{sid}_{\mathsf{VC}}, r) \to (\mathsf{cm}, \mathsf{st})$.
  On input a message length n, a session ID $\mathsf{sid}_{\mathsf{VC}}$, and randomness r, outputs a commitment string cm and the internal state st of Com. Com may query or program $\mathcal{F}_{\mathsf{GRO}}$, as well as execute subroutine machines which may interact with $\mathcal{F}_{\mathsf{GRO}}$ with their sub-session ID, in ways allowed by $\mathcal{F}_{\mathsf{NVC}}$.
- $\mathsf{Equiv}(n, \mathsf{sid}_{\mathsf{VC}}, \mathsf{cm}, \mathsf{st}, (i, m), r) \to (\mathsf{dcm}, \mathsf{State})$.
  On input a message length n, a session ID $\mathsf{sid}_{\mathsf{VC}}$, a commitment string cm, an internal state st, an index-message value pair (i, m), and randomness r, outputs a decommitment dcm for the message value m at index i and an updated committer state State for index i, which is returned upon adaptive corruption. Equiv may query or program $\mathcal{F}_{\mathsf{GRO}}$, under the restrictions imposed by $\mathcal{F}_{\mathsf{NVC}}$. Equiv can also execute subroutine machines that interact with $\mathcal{F}_{\mathsf{GRO}}$ from their sub-session ID, in ways allowed by $\mathcal{F}_{\mathsf{NVC}}$.
- $\mathsf{Extr}(n, \mathsf{sid}_{\mathsf{VC}}, \mathsf{cm}, Q) \to (\mathbf{m}, \mathbf{dcm})$.
  On input a message length n, a session ID $\mathsf{sid}_{\mathsf{VC}}$, a commitment cm, a list of GRO query-answer pairs Q, deterministically outputs a message vector $\mathbf{m}$ and a decommitment vector $\mathbf{dcm}$. Note that Extr extracts an *entire* committed message vector $\mathbf{m}$ for a particular commitment cm. Extr may not query or program $\mathcal{F}_{\mathsf{GRO}}$, but it may call/invoke subroutine machines which may interact with $\mathcal{F}_{\mathsf{GRO}}$ with their sub-session

[13] We note that, in general, the ability to "spawn" subroutine machines can be given to the algorithms Setup and Com. However, Equiv should not have this ability; since Equiv knows the committed value $\mathbf{m}$, it could spawn subroutines in a manner that leaks information about $\mathbf{m}$.

ID, in ways allowed by $\mathcal{F}_{\mathsf{NVC}}$.

When simulating composed protocols, we note that SCode must emulate the behavior of any ideal subroutines. To do so operationally, the description of SCode of $\mathcal{F}_{\mathsf{NVC}}$ may contain pointers for the SCode of its subroutines. $\mathcal{F}_{\mathsf{lib}}$ handles this code linking, prior to sending the completed SCode to $\mathcal{F}_{\mathsf{NVC}}$. However, note that any linked code should not be trusted; the environment can adversarially choose and upload the code to $\mathcal{F}_{\mathsf{lib}}$. As a preview, we will make use of such code linking for simulating Π_{NVC}, i.e. a Merkle tree vector commitment, in which the nodes are implemented using $\mathcal{F}_{\mathsf{NC}}$ which, in turn, obtains its own simulator code from $\mathcal{F}_{\mathsf{lib}}$.

6.1 The interface between $\mathcal{F}_{\mathsf{NVC}}$ and $\mathcal{F}_{\mathsf{GRO}}$

Similarly to $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NVC}}$ handles internal requests by SCode algorithms to program $\mathcal{F}_{\mathsf{GRO}}$ using the Program subroutine (Figure 12) and also enables observing queries to $\mathcal{F}_{\mathsf{GRO}}$ made by an external party (e.g. the environment) (see Step VIII of $\mathcal{F}_{\mathsf{NVC}}$). The Program subroutine of Figure 12 generalizes that of Figure 5 to handle programming requests for sub-sessions of $\mathcal{F}_{\mathsf{NVC}}$; we detail this below.
We highlight differences from $\mathcal{F}_{\mathsf{NC}}$, particularly for handling internal programming requests.

Programming. Like for $\mathcal{F}_{\mathsf{NC}}$, the simulator algorithms (in SCode) may program $\mathcal{F}_{\mathsf{GRO}}$ by invoking the Program subroutine (Figure 12), subject to the restrictions enforced by a predicate $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$ (Figure 13) which maintain the security guarantees of $\mathcal{F}_{\mathsf{NVC}}$. The discussion in Section 4.1 (regarding the interaction between $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{GRO}}$) applies, with the following additions:

- **Programming bound for SCode.Equiv.** Like for $\mathcal{F}_{\mathsf{NC}}$, SCode.Equiv many only program a single time, each time it is invoked by $\mathcal{F}_{\mathsf{NVC}}$. Unlike $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NVC}}$ invokes SCode.Equiv $n = |\mathbf{m}|$ times, i.e. once for each entry in the committed vector $\mathbf{m}$. Note that each invocation of SCode.Equiv only receives access to a single entry of $\mathbf{m}$, plus its index $i \in [n]$, so the result does not depend on other entries in $\mathbf{m}$. We note that any discussion about sub-sessions may also apply to $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{NIZK}}$ if it is UC-realized by a protocols that uses ideal components.
- **Simulating subroutines in a protocol that realizes $\mathcal{F}_{\mathsf{NVC}}$.** When simulating a protocol that realizes $\mathcal{F}_{\mathsf{NVC}}$, the simulator for $\mathcal{F}_{\mathsf{NVC}}$ must emulate the behavior of any ideal sub-components in the protocol. In particular, the simulator must emulate any interactions between such sub-components and $\mathcal{F}_{\mathsf{GRO}}$.
To do so, $\mathcal{F}_{\mathsf{NVC}}$ would need to take control of the sids of these sub-components. However, $\mathcal{F}_{\mathsf{GRO}}$ (Figure 2) only allows parties to take control of their own session ID; in particular, it is unclear how to to allow the simulator to control its sub-session IDs in $\mathcal{F}_{\mathsf{GRO}}$, which is needed to emulate interactions between subroutines and $\mathcal{F}_{\mathsf{GRO}}$.
To address this, we define the formalism of a *gateway* functionality, which acts as a channel for accessing $\mathcal{F}_{\mathsf{GRO}}$ from a sub-session of $\mathcal{F}_{\mathsf{NVC}}$. During its initial activation, $\mathcal{F}_{\mathsf{NVC}}$ first takes control of its own session ID sid in $\mathcal{F}_{\mathsf{GRO}}$. Subsequently, $\mathcal{F}_{\mathsf{NVC}}$ takes control of a sub-session ID ssid via "spawning" a gateway associated with ssid, which enables observing and programming $\mathcal{F}_{\mathsf{GRO}}$ from ssid. (Note that $\mathcal{F}_{\mathsf{NVC}}$ uses a separate gateway functionality per sub-session.) As such, it's possible to simulate the interactions between any subroutines and $\mathcal{F}_{\mathsf{GRO}}$.
However, recall that programming requests to $\mathcal{F}_{\mathsf{GRO}}$, including those for sub-sessions of $\mathcal{F}_{\mathsf{NVC}}$, should not compromise the security of $\mathcal{F}_{\mathsf{NVC}}$. As such, any requests of the simulator code are subject to safety constraints. Like for programming requests from the session ID of $\mathcal{F}_{\mathsf{NVC}}$, all programming requests from sub-session IDs of $\mathcal{F}_{\mathsf{NVC}}$ are done by invoking Program (Figure 13), subject to the restrictions of $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$ (Figure 13). If a programming request for a sub-session of $\mathcal{F}_{\mathsf{NVC}}$ passes the checks of $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$, Program

Functionality $\mathcal{F}_{\mathsf{NVC}}$ (main)

```
I Parameters: An alphabet $\Sigma$, an input vector length $n$, a conditional min-entropy for commitments $h_{\mathsf{cm}}$, and a conditional min-entropy for decommitments $h_{\mathsf{dcm}}$.
II Commit: Receive input $(\texttt{Commit}, \mathbf{m} \in (\Sigma \cup \{\bot\})^n)$ from a party $C$.
  (a) If this is not the first $(\texttt{Commit}, \cdot)$ input, output $(\texttt{Error})$. # Fail-close clause.
  (b) If this is the first activation, run the Initialization subroutine (Step V).
  (c) Compute $(\mathsf{cm}, \mathsf{st}) \leftarrow \mathsf{SCode.Com}(n, \mathsf{sid}_{\mathsf{VC}}, r_{\mathsf{cm}})$.
      # SCode.Com may program $\mathcal{F}_{\mathsf{GRO}}$ by calling Program (Figure 12).
      i. If $f_0(\mathsf{cm}) \notin r_{\mathsf{cm}}[0]$, $(\mathsf{cm}, \cdot, \cdot, \cdot, v) \in \mathsf{CState}$, or SCode.Com commits SCodeError (Definition 4.6): # Error. Outputs default values when consistency is violated. Note $f_0$ is some injective function.
         A. Run the default committer code (Step VII) with message $\mathbf{m}$.
         B. Output $(\texttt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ to $C$.
      ii. Else, continue to next step.
  (d) Compute $\mathbf{dcm}$ and $\mathbf{State}_{\mathsf{VC}}$. For all $i \in [n]$:
      i. If $\mathbf{m}[i] = \bot$, set $\mathbf{dcm}[i] := \bot$ and $\mathbf{State}_{\mathsf{VC}}[i] := \bot$. # Don't run SCode.Equiv.
      ii. Else, run $(\mathbf{dcm}[i], \mathbf{State}_{\mathsf{VC}}[i]) \leftarrow \mathsf{SCode.Equiv}(n, \mathsf{sid}_{\mathsf{VC}}, \mathsf{cm}, \mathsf{st}, (i, \mathbf{m}[i]), r_{\mathsf{dcm},i})$.
          # SCode.Equiv may program $\mathcal{F}_{\mathsf{GRO}}$ by calling Program (Figure 12).
          A. If for any index $i \in [n]$, we have $f(\mathbf{dcm}[i]) \notin r_{\mathsf{dcm},i}$, SCode.Equiv commits an SCodeError (Definition 4.6), run the default committer code (Step VII) with message $\mathbf{m}$. # Error. Note $f$ is some injective function.
          B. Else, append $(\mathsf{cm}, i, \mathbf{m}[i], \mathbf{dcm}[i], \mathbf{State}_{\mathsf{VC}}[i], 1)$ to CState for all $i \in [n]$. # No errors.
  (e) Output $(\texttt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ to $C$.
III (Adaptive) Committer Corruption: On input $(\texttt{Corrupt})$ from the adversary $\mathcal{E}$.
  (a) For every $i \in [n]$, if there exists $(\mathsf{cm}, i, m, \mathsf{dcm}, \mathsf{State}, 1) \in \mathsf{CState}$, append State to a vector $\mathbf{State}_{\mathsf{VC}}$.
  (b) Output $(\texttt{Corrupted}, \mathbf{State}_{\mathsf{VC}})$ to $\mathcal{E}$. # If a valid commitment was generated, output internal state.
IV Verify: Receive input $(\texttt{Verify}, \overline{\mathsf{cm}}, \overline{i}, \overline{m}, \overline{\mathsf{dcm}})$ from a party $R$.
  (a) If this is the first activation, run the Initialization subroutine (Step V).
  (b) If $(\overline{\mathsf{cm}}, \overline{i}, \overline{m}, \overline{\mathsf{dcm}}, \cdot, v) \in \mathsf{CState}$, output $(\texttt{VerStatus}, v)$ to $R$ (thereby ending the activation).
      # Enforces completeness and consistency.
  (c) If $(\mathsf{cm} = \overline{\mathsf{cm}}, i = \overline{i}, m, \mathsf{dcm}, \cdot, 1) \in \mathsf{CState}$ for some $m \neq \overline{m}$, then output $(\texttt{VerStatus}, 0)$ to $R$.
      # Enforces position binding.
  (d) If $(\mathsf{cm} = \overline{\mathsf{cm}}, i = \overline{i}, m = \overline{m}, \mathsf{dcm}, \cdot, 1) \in \mathsf{CState}$ for some $\mathsf{dcm} \neq \overline{\mathsf{dcm}}$, then output $(\texttt{VerStatus}, 0)$ to $R$. # Enforces non-malleability, only allows a single decommitment.
  (e) If $(\cdot, \overline{\mathsf{cm}})$ was programmed by SCode.Com or SCode.Equiv, output $(\texttt{VerStatus}, 0)$ to $R$.
  (f) Else: # For extractability and non-malleability.
      i. Compute the set $Q \subseteq \mathsf{Hist}_{\mathsf{GRO}}$ such that $Q$ contains all queries before (and including) the latest query-answer pair of the form $(\cdot, \overline{\mathsf{cm}})$.
      ii. Compute $(\mathbf{m}', \mathbf{dcm}') \leftarrow \mathsf{SCode.Extr}(n, \mathsf{sid}_{\mathsf{VC}}, \overline{\mathsf{cm}}, Q)$. # Extract committed vector.
      iii. If $\mathbf{m}'[\overline{i}] \neq \overline{m}$, $\mathbf{m}'[\overline{i}] = \bot$, or SCode.Extr commits SCodeTimeError (Definition 4.6), set $v := 0$.
           # Check extraction.
      iv. Else: # $\mathbf{m}'[\overline{i}] = \overline{m}$.
          A. Set $v = 1$.
          B. If $(\overline{\mathsf{cm}}, \overline{i}, \overline{m}, \cdot, \cdot, 1) \notin \mathsf{CState}$, append $(\overline{\mathsf{cm}}, i, \mathbf{m}'[i], \mathbf{dcm}'[i], \bot, 1)$ to CState for all $i \in [n]$.
             # If $\overline{m}$ does not appear in CState, add the entire extracted vector to CState.
          C. Else, check that for all $i \in [n]$, there exists an entry $(\overline{\mathsf{cm}}, i, m = \mathbf{m}'[i], \cdot, \cdot, 1) \in \mathsf{CState}$. If not, set $v = 0$. # If $\overline{m}$ already appears in CState, only accept if $\mathbf{m}'$ is consistent with CState.
      v. Output $(\texttt{VerStatus}, v)$ to $R$.
```

Figure 8: The ideal functionality for non-interactive vector commitment in the presence of $\mathcal{F}_{\mathsf{GRO}}$.

Functionality $\mathcal{F}_{\mathsf{NVC}}$ (subroutines)

V **Initialization:** In the first activation do: # First activation can be Commit or Verify.

- (a) Set the following variables as empty (i.e. as $\bot$):
  - CState: a list of tuples $(\mathsf{cm}, i, m, \mathsf{dcm}, \mathsf{State}, v)$, in which cm is a commitment string, $i \in [n]$ is an index, $m \in \Sigma$ is an element of the message vector, dcm is a decommitment for m, State is the committer state with respect to entry i, and $v \in \{0,1\}$ is a bit representing whether $(\mathsf{cm}, i, m, \mathsf{dcm})$ is valid.
  - $\mathsf{Hist}_{\mathsf{GRO}}$: a list of tuples (x, y), in which x is a query point, and y is a query answer.
  - ActiveAlg: records which algorithm is currently interacting with $\mathcal{F}_{\mathsf{GRO}}$.
  - SubFuncs: a list of tuples $(\mathsf{ssid}, \mathsf{State})$, in which ssid is a session ID, and State is internal state associated with session ssid.
- (b) Take control of session $\mathsf{sid}_{\mathsf{VC}}$ (where $\mathsf{sid}_{\mathsf{VC}}$ is the local session ID) in $\mathcal{F}_{\mathsf{GRO}}$ by sending $(\mathtt{Control})$ to $\mathcal{F}_{\mathsf{GRO}}$, and receive a message m from $\mathcal{F}_{\mathsf{GRO}}$.# Enables observing GRO query-answer pairs and/or programming GRO responses for queries with prefix $\mathsf{sid}_{\mathsf{VC}}$.
  - i. If $m = (\mathtt{Ok}, \mathsf{Hist})$ such that $\mathsf{Hist} \neq \bot$, output $\bot$. # Failure.
- (c) Send $(\mathtt{CodeRequest}, \mathcal{F}_{\mathsf{NVC}})$ to $\mathcal{F}_{\mathsf{lib}}$ and receive $(\mathtt{Answer}, \mathsf{LinkedRecords})$, where $\mathsf{LinkedRecords} = (\mathcal{F}_{\mathsf{NVC}}, \mathsf{SCode}, \cdot)$ and SCode is a tuple of algorithms $\{\mathsf{Setup}, \mathsf{Com}, \mathsf{Equiv}, \mathsf{Extr}\}$.
- (d) Run SCode.Setup. If SCode.Setup invokes $(\mathsf{Spawn}, \mathsf{ssid})$, run Step VI. # Spawn gateway functionalities for simulating sub-functionalities.
- (e) Let $r_{\mathsf{cm}} \leftarrow (\{0,1\}^{h_{\mathsf{cm}}})^{\lambda} \times (\{0,1\}^{h_{\mathsf{dcm}}})^{\lambda}$, $r_{\mathsf{dcm},1}, \ldots, r_{\mathsf{dcm},n} \leftarrow (\{0,1\}^{h_{\mathsf{dcm}}})^{\lambda}$, $\widehat{r_{\mathsf{cm}}} \leftarrow (\{0,1\}^{h_{\mathsf{cm}}})^{\lambda}$, $\widehat{r_{\mathsf{dcm}}} \leftarrow (\{0,1\}^{h_{\mathsf{dcm}}})^{\lambda}$ denote the random strings of $\mathcal{F}_{\mathsf{NVC}}$. # The random strings $r_{\mathsf{cm}}, r_{\mathsf{dcm},i}$ will be given to SCode.Com and SCode.Equiv, respectively. $\widehat{r_{\mathsf{cm}}}$ and $\widehat{r_{\mathsf{dcm}}}$ are for running default code.

VI **Initialize gateway functionality:** When $(\mathsf{Spawn}, \mathsf{ssid})$ is invoked:

- (a) If the input ssid has the form $(\mathsf{sid}_{\mathsf{VC}}, \cdot)$ and there is no entry of the form $(\mathsf{ssid}, \cdot) \in \mathsf{SubFuncs}$: # Ensure ssid is a valid sub-session ID of $\mathsf{sid}_{\mathsf{VC}}$.
  - i. Send $(\mathsf{Spawn}, \mathsf{ssid})$ to the gateway functionality GF (Figure 10), and receive a message m from GF.
    - A. If $m = (\mathtt{Ok}, \mathsf{Hist})$ and $\mathsf{Hist} = \bot$, add $(\mathsf{ssid}, \mathsf{State} := \bot)$ to the list SubFuncs, where State is the state associated with ssid.
    - B. Else, output $\bot$. # Failure.

VII **Default commit code:** # Generates strings in event of errors.

- (a) Set $\mathsf{cm} := \widehat{r_{\mathsf{cm}}}[0]$.
- (b) Compute $\mathbf{dcm}$ such that for all $i \in [n]$:

$$\mathbf{dcm}[i] := \begin{cases} \bot & \text{if } \mathbf{m}[i] = \bot \\ \widehat{r_{\mathsf{dcm}}} & \text{o.w.} \end{cases} .$$

VIII **External request to program the GRO:** Upon receiving $(\mathtt{ProgramReq}, P, x)$ from $\mathcal{F}_{\mathsf{GRO}}$: # This interface allows $\mathcal{F}_{\mathsf{NVC}}$ to observe oracle queries.

- (a) Sample $y \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$ and add $(x, y) \in \mathsf{Hist}_{\mathsf{GRO}}$.
- (b) Send subroutine input $(\mathtt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$.

Figure 9: The ideal functionality for non-interactive vector commitment in the presence of $\mathcal{F}_{\mathsf{GRO}}$, specifically the subroutines for initialization, for default committer code and for the interactions between $\mathcal{F}_{\mathsf{NVC}}$ and the global subroutine $\mathcal{F}_{\mathsf{GRO}}$.

Gateway functionality GF

- **Spawn.** Upon the invocation of $(\mathsf{Spawn}, \mathsf{ssid})$ by a caller $\mathcal{F}$ with session ID sid:
  1. Take control of session ssid in $\mathcal{F}_{\mathsf{GRO}}$ by sending $(\texttt{Control})$ to $\mathcal{F}_{\mathsf{GRO}}$.
  2. Receive a message from $\mathcal{F}_{\mathsf{GRO}}$ and forward it to $\mathcal{F}_{\mathsf{NVC}}$.
- **Internal request to program the GRO:** Upon receiving $(\texttt{Program}, x, y)$ from an ideal functionality $\mathcal{F}$:
  1. Send $(\texttt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$.
  2. Receive subroutine output $(\texttt{EvalOutput}, y)$ from $\mathcal{F}_{\mathsf{GRO}}$.
  3. Forward $(\texttt{EvalOutput}, y)$ to $\mathcal{F}$.
- **External request to program the GRO:** Upon receiving $(\texttt{ProgramReq}, P, x)$ from $\mathcal{F}_{\mathsf{GRO}}$:
  # This interface allows $\mathcal{F}$ to observe oracle queries.
  1. Sample $y \leftarrow \{0,1\}^\lambda$ and add (x, y) to $\mathsf{Hist}_{\mathsf{GRO}}$.
  2. Send subroutine input $(\texttt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$.

Figure 10: Code for gateway functionalities, which are used for simulating interactions between sub-functionalities of a caller $\mathcal{F}$ and $\mathcal{F}_{\mathsf{GRO}}$.

routes the programming request through the gateway functionality with the corresponding sub-session ID. We give code of the gateway functionality in Figure 10. That is, anytime $\mathcal{F}_{\mathsf{NVC}}$ initializes a gateway functionality GF (with a sub-session ID of the form $\mathsf{ssid} = (\mathsf{sid}_{\mathsf{VC}}, \cdot)$), GF has the code given in Figure 10. Further, since $\mathcal{F}_{\mathsf{NVC}}$ (and thus GF) is not a global functionality, GF must also be subroutine respecting, i.e. it only interacts with the main party that spawned it ($\mathcal{F}_{\mathsf{NVC}}$) and $\mathcal{F}_{\mathsf{GRO}}$ but *not* the environment. (See Figure 11 for a diagram illustrating the communication channels.)

6.2 Security guarantees provided by $\mathcal{F}_{\mathsf{NVC}}$

On mechanisms for enforcing min-entropy of $\mathbf{dcm}$ strings. Recall that $\mathcal{F}_{\mathsf{NC}}$ checks that dcm has sufficient min-entropy by enforcing that the output of SCode.Equiv matches a location in the random tape r_{dcm} of $\mathcal{F}_{\mathsf{NC}}$.[14] In $\mathcal{F}_{\mathsf{NVC}}$, this technique is generalized for modeling protocols in which there is an injective mapping from decommitment strings $\mathbf{dcm}[i]$ to the i-th portion of the random tape $r_{\mathsf{dcm}}[i]$ of $\mathcal{F}_{\mathsf{NVC}}$, since injective maps preserve entropy.

6.2.1 Hiding

Like for $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NVC}}$ inherently cannot model a perfectly-hiding vector commitment. The adversary $\mathcal{E}$ can always guess the decommitment dcm for an unopened location of $\mathbf{m}$, relative to a commitment cm; this is because $\mathcal{F}_{\mathsf{NVC}}$ saves the "correct" decommitment value in CState, before outputting cm.

Similar to Game 4.2 for $\mathcal{F}_{\mathsf{NC}}$, we formalize a hiding property of $\mathcal{F}_{\mathsf{NVC}}$ in Game 6.2, which states that the environment distinguishes only with bounded probability between (1) an interaction in which all decommitment strings was generated but only a subset $I \subset [n]$ of the decommitment strings are disclosed; and (2) an interaction in which only the decommitment strings for locations I are generated. Intuitively, the adversary distinguishes if, for any $j \in [n] \setminus I$, $\mathcal{E}$ guesses the decommitment string $\mathbf{dcm}[j]$ and thus can decommit to location j in Interaction 1 but not in Interaction 2. (We note that this definition captures per-coordinate hiding when $I = [n] \setminus \{j\}$ for $j \in [n]$.)

[14] We note that this technique is compatible with the simulator code SCode.Equiv for Π_{NC} (Figure 7) and the code defined for the non-interactive version of [CF01] (described in Figure 5 of [CSW22]), since dcm is a random string in both protocols.

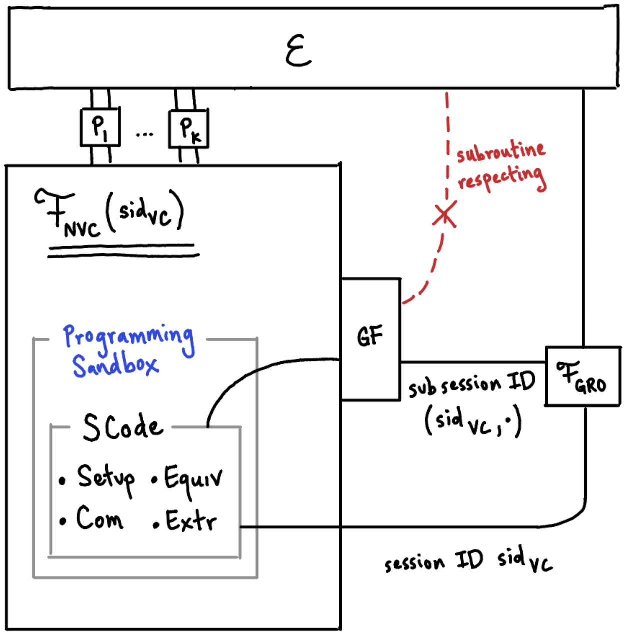

Figure 11: Sketch of a $\mathcal{F}_{\mathsf{NVC}}$ (with session ID $\mathsf{sid}_{\mathsf{VC}}$) with gateway functionality GF, which emulates interactions between a subroutine of $\mathcal{F}_{\mathsf{NVC}}$ and $\mathcal{F}_{\mathsf{GRO}}$. Solid black lines illustrate allowed communication channels. Programming Sandbox refers to running the Program subroutine (Figure 12) with predicate $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$ (Figure 13), whenever SCode initiates programming queries with $\mathcal{F}_{\mathsf{GRO}}$.

We formalize this intuition via defining the general notion of an ideal vector commitment mechanism, then defining a hiding game for ideal vector commitment mechanisms:

Definition 6.1 (Ideal vector commitment mechanism VC)**.** *We say that an interactive Turing machine* VC *is an ideal vector commitment mechanism in the presence of global functionalities* $\mathcal{F}_{\mathsf{GRO}}$ *and* $\mathcal{F}_{\mathsf{lib}}$ *for vector of length* $n \in \mathbb{N}$ *with alphabet* Σ*, if* VC *has the following interfaces:*

1. *On input* $(\texttt{Commit}, \mathbf{m} \in \Sigma^n)$*, generates a commitment string* cm *and decommitment string* $\mathbf{dcm}$*, outputs* $(\texttt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ *where* cm *is a commitment string and* $\mathbf{dcm}$ *is a vector of decommitments.*
2. *On input* $(\texttt{Verify}, \mathsf{cm}, i, m, \mathsf{dcm})$*, outputs* $(\texttt{VerStatus}, v)$ *where* v *is a bit indicating whether* cm *commits to* m *in the* i*-th position with decommitment* dcm.

Game 6.2 (Hiding game for VC in the presence of $\mathcal{F}_{\mathsf{GRO}}$)**.** Let VC be an ideal vector commitment mechanism (Definition 6.1) in the presence of $\mathcal{F}_{\mathsf{GRO}}$ and $\mathcal{F}_{\mathsf{lib}}$ for vectors of length $n \in \mathbb{N}$ with alphabet Σ (such that $\perp \notin \Sigma$). The adversary $\mathcal{E}$ selects a message $\mathbf{m} \in \Sigma^n$ and and a set of locations $\overline{I} \subset [n]$. Then, $\mathcal{E}$ attempts to distinguish between the following interactions:

Interaction 1. (a) VC is invoked with input $(\texttt{Commit}, \mathbf{m} \in \Sigma^n)$.
(b) Receive the output $(\texttt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from VC, as usual.
(c) Output $(\mathsf{cm}, \mathbf{dcm}[I])$ to $\mathcal{E}$, where $I := [n] \setminus \overline{I}$.
$\mathcal{E}$ only receives decommitments for locations with non-empty plaintext values.

Interaction 2. This is identical to Interaction 1, except that the locations $\overline{I} \subset [n]$ in $\mathbf{m}$ has value $\mathbf{m}[i] = \perp$ for all $i \in \overline{I}$.
(a) VC is invoked with input $(\texttt{Commit}, \mathbf{m} \in (\Sigma \cup \{\perp\})^n)$.
(b) Receive the output $(\texttt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from VC.

The $\mathsf{Program}$ subroutine ($\mathcal{F}_{\mathsf{NVC}}$ version, also good for $\mathcal{F}_{\mathsf{NIZK}}$).

```
• $\mathsf{Program}(\mathsf{SubFuncs}, \mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}}, \mathsf{sid}, \mathsf{ActiveAlg}, x, y)$: # Run only if $(x, \cdot) \notin \mathsf{Hist}_{\mathsf{GRO}}$ of $\mathcal{F}$.
  1. Parse the inputs so that:
     (a) the following are honestly derived from the state of $\mathcal{F}$: $\mathsf{SubFuncs}$ as the list gateway functionalities, $\mathsf{Hist}_{\mathsf{GRO}}$ is the list of GRO query-answer pairs, $\Phi_{\mathcal{F}}$ is the query-checking predicate for $\mathcal{F}$, $\mathsf{sid}$ is the calling party's session ID; and
     (b) the following may be chosen by $\mathsf{ActiveAlg}$: $x$ is a GRO query in $\mathrm{Dom}(\mathcal{F}_{\mathsf{GRO}})$, and $y$ is a GRO answer in $\mathrm{Cod}(\mathcal{F}_{\mathsf{GRO}})$.
  2. If $(x, \cdot) \notin \mathsf{Hist}_{\mathsf{GRO}}$: # The evaluation of $x$ is undefined in $\mathcal{F}_{\mathsf{GRO}}$.
     (a) If $\Phi_{\mathcal{F}}(\mathsf{sid}, \mathsf{ActiveAlg}, x, y) = 1$: # $\Phi_{\mathcal{F}}$ is a predicate check.
         i. If $\mathsf{sid}$ is the local session ID of $\mathcal{F}$:
            A. Send subroutine input $(\mathtt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$. # $\mathcal{F}_{\mathsf{GRO}}$ records $(x, y)$ in its history.
            B. Receive subroutine output $(\mathtt{EvalOutput}, y)$ from $\mathcal{F}_{\mathsf{GRO}}$.
        ii. If there exists an entry $(\mathsf{sid}, \cdot, \cdot) \in \mathsf{SubFuncs}$: # $\mathsf{sid}$ is for a sub-functionality of $\mathcal{F}$.
            A. Invoke the gateway functionality $\mathsf{GF}$ with session ID $\mathsf{sid}$ (and party ID $\mathsf{pid} = \bot$) with input $(\mathtt{Program}, x, y)$.
            B. Receive $(\mathtt{EvalOutput}, y)$ from $\mathsf{GF}$.
            C. Forward $(\mathtt{EvalOutput}, y)$ to $\mathsf{ActiveAlg}$.
     (b) Else, output $(\mathtt{EvalOutput}, \bot)$ to $\mathsf{ActiveAlg}$. # Error.
  3. Else (if there exists an entry $(x, y') \in \mathsf{Hist}_{\mathsf{GRO}}$), send $(\mathtt{EvalOutput}, y')$ to $\mathsf{ActiveAlg}$.
```

Figure 12: The $\mathsf{Program}$ subroutine, which handles $\mathcal{F}_{\mathsf{GRO}}$ programming requests originating from an ideal functionality or a subroutine.

The predicate $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$

```
• $\Phi_{\mathcal{F}_{\mathsf{NVC}}}(\mathsf{sid}, \mathsf{ActiveAlg}, x, y) \to \{0, 1\}$:
  1. Parse $\mathsf{sid}$ as the session ID of the calling party, $x$ as a query point $(\mathsf{sid}', x')$ in $\mathrm{Dom}(\mathcal{F}_{\mathsf{GRO}})$, and $y$ as a query answer in $\mathrm{Cod}(\mathcal{F}_{\mathsf{GRO}})$.
  2. If $\mathsf{sid} \neq \mathsf{sid}'$, output $0$. # Disallow cross-domain programming.
  3. If $\mathsf{ActiveAlg} = \mathsf{SCode.Com}$: # The caller is $\mathsf{SCode.Com}$.
     (a) Check that:
         i. $f(x') \in r_{\mathsf{dcm}}$. # Programmed $\mathcal{F}_{\mathsf{GRO}}$ query-answer pairs have sufficient entropy, $f$ is some injective function.
     (b) If all checks from Step 3a pass, output $1$ to $\mathsf{ActiveAlg}$.
     (c) Else, output $0$ to $\mathsf{ActiveAlg}$. # Error.
  4. If $\mathsf{ActiveAlg} = \mathsf{SCode.Equiv}$: # The caller is $\mathsf{SCode.Equiv}$.
     (a) If this execution of $\mathsf{SCode.Equiv}$ has already programmed $\mathcal{F}_{\mathsf{GRO}}$, output $0$.
         # Each execution of $\mathsf{SCode.Equiv}$ programs only once.
     (b) Check that all of the following conditions hold:
         i. This execution of $\mathsf{SCode.Equiv}$ has not already programmed $\mathcal{F}_{\mathsf{GRO}}$.
            # Each execution of $\mathsf{SCode.Equiv}$ programs only once.
        ii. $f(x') \in r_{\mathsf{dcm}}$. # Programmed $\mathcal{F}_{\mathsf{GRO}}$ query-answer pairs have sufficient entropy, $f$ is some injective function.
     (c) If Step 4b passes, output $1$ to $\mathsf{ActiveAlg}$. Else, output $0$ to $\mathsf{ActiveAlg}$.
  5. Else, output $0$ to $\mathsf{ActiveAlg}$. # No other $\mathsf{SCode}$ algorithms can program.
```

Figure 13: The predicate $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$, which verifies properties of $\mathcal{F}_{\mathsf{GRO}}$ programming requests originating from $\mathcal{F}_{\mathsf{NVC}}$.

(c) Output $(\mathsf{cm}, \mathbf{dcm}[I])$ to $\mathcal{E}$, where $I := [n] \setminus \overline{I}$.
$\mathcal{E}$ only receives decommitments for locations with non-empty plaintext values.

Additionally in both interactions, $\mathcal{E}$ may: (a) access the global functionalities $\mathcal{F}_{\mathsf{GRO}}$ and $\mathcal{F}_{\mathsf{lib}}$; and (b) invoke both the **Commit** and **Verify** interfaces of VC.

$\mathcal{E}$ wins if it distinguishes between Interactions 1 and 2.

We show that no adversary can win the above game against our $\mathcal{F}_{\mathsf{NVC}}$, except with negligible probability:

Theorem 6.3. *Let Σ be an alphabet. Let $n, h_{\mathsf{cm}}, h_{\mathsf{dcm}} \in \mathbb{N}$. Let $\mathcal{E}$ be an adversary with runtime t. Then, for $\mathcal{F}_{\mathsf{NVC}}(\Sigma, n, h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ running with any simulator code* SCode *in the presence of $\mathcal{F}_{\mathsf{GRO}}$ and $\mathcal{F}_{\mathsf{lib}}$, $\mathcal{E}$ wins Game 6.2 with probability $t \cdot 2^{-h_{\mathsf{dcm}}}$.*

Proof of Theorem 6.3. First, observe that $\mathcal{F}_{\mathsf{NVC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$ is an ideal vector commitment mechanism, as defined in Definition 6.1. (The proof follows by inspecting the interfaces of $\mathcal{F}_{\mathsf{NVC}}$.)

Second, we argue that $\mathcal{E}$ wins Game 6.2 with probability $t \cdot 2^{-h_{\mathsf{dcm}}}$. Let $I \subset [n]$. Interactions 1 and 2 are identical, up to the bad event that $\mathcal{E}$ guesses the decommitment string dcm for index $j \in \overline{I} := [n] \setminus I$ such that submitting $(\texttt{Verify}, \mathsf{cm}, j, m, \mathsf{dcm})$ to $\mathcal{F}_{\mathsf{NVC}}$ yields $(\texttt{VerStatus}, 1)$ in Interaction 1 but $(\texttt{VerStatus}, 0)$ in Interaction 2. In detail, the difference is that $\mathcal{F}_{\mathsf{NVC}}$ runs SCode.Equiv on every entry in $\mathbf{m}$ in Interaction 1 but only at entries I in Interaction 2 (see Step IId).

We compute the probability of the bad event. For a fixed index $j \in \overline{I}$, $\mathcal{E}$ has two strategies for recovering dcm:

1. $\mathcal{E}$ guesses dcm directly.
   Since $f(\mathsf{dcm}) \in r_{\mathsf{dcm}}$ and f is injective, dcm is a string with conditional min-entropy at least h_{dcm} (see Step IId of $\mathcal{F}_{\mathsf{NVC}}$). Thus, $\mathcal{E}$ guesses dcm with probability $2^{-h_{\mathsf{dcm}}}$ per attempt.
2. SCode.Equiv programs dcm somewhere in $\mathcal{F}_{\mathsf{GRO}}$, then $\mathcal{E}$ queries $\mathcal{F}_{\mathsf{GRO}}$ and recovers dcm.
   Recall that $\mathcal{F}_{\mathsf{NVC}}$ limits SCode.Equiv to only program at a single location in $\mathcal{F}_{\mathsf{GRO}}$ per index i (see Step 4(b)i of $\mathcal{F}_{\mathsf{NVC}}$). Further, the programmed location has conditional min-entropy h_{dcm} (Step 4(b)ii). Thus, $\mathcal{E}$ recovers this location with probability $2^{-h_{\mathsf{dcm}}}$ per attempt.

Hence, a single attempt of either strategy allows $\mathcal{E}$ to recover $\mathbf{dcm}[i]$ with the same probability. Since $\mathcal{E}$ has runtime t (and thus can invoke **Verify** at most t times), it recovers $\mathbf{dcm}[i]$ with overall probability $t \cdot 2^{-h_{\mathsf{dcm}}}$. □

6.2.2 Position Binding

$\mathcal{F}_{\mathsf{NVC}}$ extends the notion of binding from $\mathcal{F}_{\mathsf{NC}}$ to the vector setting. Specifically, $\mathcal{F}_{\mathsf{NVC}}$ expresses the position binding property - once a vector commitment cm is associated with a value $\mathbf{m}[i]$ at index i, it is hard to decommit cm to a different value $m \neq \mathbf{m}[i]$; this is true for all $i \in [n]$. (As a consequence, it's hard to find two valid ways to decommit to cm at any index.)

Like for $\mathcal{F}_{\mathsf{NC}}$, binding in $\mathcal{F}_{\mathsf{NVC}}$ can be characterized by an injective function from commitment strings to *entire* message vectors. Correspondingly, this means that, as part of verifying a particular index, the extractor is asked to extract an entire vector. Note that this implies position binding since if one could open a index i of $\mathbf{m}$ to two different values, then this implies opening to two different message vectors by simply switching the message value at i.

Binding definition for $\mathcal{F}_{\mathsf{NVC}}$ in the plain model.

Like in $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NVC}}$ enforces binding in cases depending on whether $\overline{\mathsf{cm}}$ was generated honestly using $\mathcal{F}_{\mathsf{NVC}}$. When $\mathcal{F}_{\mathsf{NVC}}$ is invoked with message $(\texttt{Verify}, \overline{\mathsf{cm}}, i, \overline{m}, \overline{\mathsf{dcm}})$:

1. **Honest.** If there exists an entry $(\overline{\mathsf{cm}}, i, m, \cdot, 1) \in \mathsf{CState}$ such that $\overline{m} \neq m$, $\mathcal{F}_{\mathsf{NVC}}$ rejects. This models perfect (position) binding for every index $i \in [n]$ and is analogous to the perfect binding property of $\mathcal{F}_{\mathsf{NC}}$.
2. **Adversarial.** If there does not exist $(\overline{\mathsf{cm}}, \cdot, \cdot, \cdot, \cdot) \in \mathsf{CState}$, $\mathcal{F}_{\mathsf{NVC}}$ runs the algorithm $\mathsf{SCode.Extr}$ to extract *an entire vector* $\mathbf{m}'$, such that $\mathbf{m}'[i] = \overline{m}$. To preserve binding, $\mathsf{SCode.Extr}$ is invoked only if $\overline{\mathsf{cm}}$ does not have an entry in CState. Further, $\mathcal{F}_{\mathsf{NVC}}$ requires extraction for adversarially generated commitments, to preserving the hiding property of the scheme. ($\mathsf{SCode.Extr}$ cannot extract on honest values, so hiding is preserved.)

Binding for $\mathcal{F}_{\mathsf{NVC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$. Like in the $\mathcal{F}_{\mathsf{NC}}$ case, for $\mathcal{F}_{\mathsf{NVC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$, fixing the $\mathsf{SCode.Extr}$ does not fully fix the injective function between commitments and message because $\mathsf{SCode.Extr}$ can depend on the function table of $\mathcal{F}_{\mathsf{GRO}}$, which may change over time due to programming and the fact that $\mathcal{F}_{\mathsf{NC}}$ observes the environment's oracle queries. As such, commitment strings are bound to message vectors during an execution, at various *binding points*, which mirror those for $\mathcal{F}_{\mathsf{NC}}$.

Recall that to capture "position" binding, we require that for all i, there is exactly one valid mapping from the fixed cm to $\mathbf{m}[i]$. However, the entire message vector does not have to be committed at the same time, e.g. for the Merkle tree scheme, the adversary can simulate committing to a single leaf-to-root path. To handle this case and also to get position binding, the first run of extractor for $\mathcal{F}_{\mathsf{NVC}}$ extracts the value encoded by the path, but every other message value will be $\perp$. Further, the extractor will always return the same values henceforth.

Definition 6.4 (Extractable ideal vector commitment)**.** *We say that an ideal vector commitment mechanism* VC *(Definition 6.1) is ϵ-extractable if there is some extraction function* Extr *from states of an execution to the plaintext space of* VC*, such that any $\mathcal{E}$ wins the following game with probability at most ϵ:*

1. *Run $\mathcal{E}$ in a standard execution in the UC model with* VC *(with session ID* sid*), $\mathcal{F}_{\mathsf{GRO}}$, and $\mathcal{F}_{\mathsf{lib}}$ until $\mathcal{E}$ receives an output* $(\mathtt{VerStatus}, 1)$ *in response to an input* $(\mathtt{Verify}, \overline{\mathsf{cm}}, \bar{i}, \overline{m}, \overline{\mathsf{dcm}})$ *to* VC.
2. *For $(\overline{\mathsf{cm}}, \bar{i})$, rewind the execution to the latest of the following* binding points*, if they exist:*
   Case 1. $\overline{\mathsf{cm}}$ is output by VC *as a commitment. Then in this case, the binding point is the state* State *of the (entire) execution immediately before the subsequent activation of $\mathcal{E}$.*
   Case 2. The first time $\mathcal{F}_{\mathsf{GRO}}$ outputs $(\mathtt{EvalOutput}, \overline{\mathsf{cm}})$*, in response to an* $\mathtt{Eval}$ *or* $\mathtt{Program}$ *input in which x has the form* $(\mathsf{sid}, \cdot)$*. Then in this case, the binding point is the state* State *of the (entire) execution immediately before the subsequent activation of $\mathcal{E}$.*
   Case 3. $\overline{\mathsf{cm}}$ is not output by neither VC *nor $\mathcal{F}_{\mathsf{GRO}}$. In this case, the binding point is the state* State *of the execution after the initialization of* VC*, immediately before the next activation of $\mathcal{E}$.*
3. *Extract a message $\widehat{m}$ using the state* State *at the binding point:*
   (a) In Case 1, output the message $\widehat{\mathbf{m}}$ that VC *used to generate* $\overline{\mathsf{cm}}$.
   (b) In Case 2 (resp. Case 3, run $(\widehat{\mathbf{m}}, \widehat{\mathbf{dcm}}) \leftarrow \mathsf{Extr}(\mathsf{sid}, \overline{\mathsf{cm}}, \mathsf{State})$.
4. *$\mathcal{E}$ wins if the extracted message $\widehat{\mathbf{m}}[\bar{i}] \neq \perp \widehat{\mathbf{m}}[\bar{i}] \neq \overline{m}$.*

Next, we show that our $\mathcal{F}_{\mathsf{NVC}}$ is an extractable ideal vector commitment mechanism. Recall that $\mathcal{F}_{\mathsf{NVC}}$ has parameters for length of plaintext vectors n and length of commitment strings h_{cm}.

Theorem 6.5. *Let Σ be some alphabet. Let $n, h_{\mathsf{cm}}, h_{\mathsf{dcm}} \in \mathbb{N}$. Let $\mathcal{E}$ be an adversary with runtime t. For any (adversarially-chosen)* SCode*, $\mathcal{E}$ interacting with $\mathcal{F}_{\mathsf{NVC}}(\Sigma, n, h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ (defined in Figures 8 and 9) in the presence of $\mathcal{F}_{\mathsf{GRO}}$ and $\mathcal{F}_{\mathsf{lib}}$ is an $(n \cdot 2t^2 \cdot 2^{-h_{\mathsf{cm}}})$-extractable ideal vector commitment mechanism (Definition 6.4) with respect to the extraction function* $\mathsf{SCode.Extr}$ *that is used to define $\mathcal{F}_{\mathsf{NVC}}(\Sigma, n, h_{\mathsf{cm}}, h_{\mathsf{dcm}})$.*

Proof of Theorem 6.5. First, observe that $\mathcal{F}_{\mathsf{NVC}}$ in the presence of $\mathcal{F}_{\mathsf{GRO}}$ is an ideal vector commitment mechanism, as defined in Definition 6.1. (The proof follows by inspecting the interfaces of $\mathcal{F}_{\mathsf{NVC}}$.)

Second, we argue that $\mathcal{F}_{\mathsf{NVC}}$ is an $(n \cdot 2t^2 \cdot 2^{-h_{\mathsf{cm}}})$-extractable ideal commitment mechanism, i.e. that any adversary $\mathcal{E}$ wins the game defined in Definition 6.4 with probability at most $n \cdot 2t^2 \cdot 2^{-h_{\mathsf{cm}}}$. To specialize the game in Definition 6.4 to $\mathcal{F}_{\mathsf{NVC}}$, we consider the execution state at the binding point State to include the query-answer pairs of $\mathcal{F}_{\mathsf{GRO}}$, which is used by SCode.Extr in $\mathcal{F}_{\mathsf{NVC}}$.

Suppose $\mathcal{E}$ interacts arbitrarily with $\mathcal{F}_{\mathsf{NVC}}$ until $\mathcal{F}_{\mathsf{NVC}}$ accepts on input $(\mathtt{Verify}, \overline{\mathsf{cm}}, \bar{i}, \overline{m}, \overline{\mathsf{dcm}})$; we refer to this as the "initial execution". We define a bad event as $\mathcal{E}$ extracting a plaintext vector $\widehat{\mathbf{m}}$ at the binding point such that $\widehat{\mathbf{m}}[\bar{i}] \neq \overline{m}$ and $\widehat{\mathbf{m}}[\bar{i}] \neq \bot$.

Next, we argue that the bad event occurs with probability $n \cdot 2t^2 \cdot 2^{-h_{\mathsf{cm}}}$. To do so, we consider cases depending on when the binding point for $\overline{\mathsf{cm}}$ occurs during the game.

Case 1. $\overline{\mathsf{cm}}$ is output by $\mathcal{F}_{\mathsf{NVC}}$ as a commitment, after an invocation of the **Commit** subroutine of $\mathcal{F}_{\mathsf{NVC}}$.
This means that $\overline{\mathsf{cm}}$ corresponds to a particular message vector $\mathbf{m} \in \Sigma^n$, written in CState. We argue that there is at most a single message value $m_i \in \Sigma$ for which $(\overline{\mathsf{cm}}, i, m_i, \cdot, \cdot, 1) \in$ CState of $\mathcal{F}_{\mathsf{NVC}}$.
When the **Commit** subroutine of $\mathcal{F}_{\mathsf{NVC}}$ is invoked, Step II(c)i of $\mathcal{F}_{\mathsf{NVC}}$ prevents having more than one entry of the form $(\overline{\mathsf{cm}}, i, \cdot, \cdot, 1)$ written in CState for each $i \in [n]$. When the **Verify** subroutine of $\mathcal{F}_{\mathsf{NVC}}$ is invoked, Step IVc of $\mathcal{F}_{\mathsf{NVC}}$ prevents having more than one entry of the form $(\overline{\mathsf{cm}}, i, \cdot, \cdot, 1)$ in CState for each $i \in [n]$. Hence, the probability of extracting $\mathbf{m}'$ such that $\mathbf{m}'[\bar{i}] \neq \overline{m}$ is 0.

Case 2. $\overline{\mathsf{cm}}$ appears as an output in the function table Hist of $\mathcal{F}_{\mathsf{GRO}}$ and it is possible to is able to extract an n-length vector $\widehat{\mathbf{m}}$.
We consider sub-cases, depending on when the binding point occurs and $\mathcal{E}$'s interactions with $\mathcal{F}_{\mathsf{NVC}}$. We show that the probability that the vector $\widehat{\mathbf{m}}$ extracted at the binding point is differs from the vector $\mathbf{m}'$ extracted due to invoking the **Verify** interface (and executing Step IVf) of $\mathcal{F}_{\mathsf{NC}}$.

Case (a). In the initial execution, $\mathcal{E}$ receives $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$ for some query x, a vector $\widehat{\mathbf{m}}$ with no $\bot$ entries is extracted using $\mathsf{SCode.Extr}(n, \mathsf{sid}, \overline{\mathsf{cm}}, \mathsf{Hist}^*)$, and $\mathcal{E}$ never invokes the **Commit** interface of $\mathcal{F}_{\mathsf{NVC}}$.
We argue that the probability of the bad event is $t^2 \cdot 2^{-h_{\mathsf{cm}}}$. Since the **Commit** interface of $\mathcal{F}_{\mathsf{NVC}}$ is never invoked, $\mathcal{F}_{\mathsf{NVC}}$ accepts on input $(\mathtt{Verify}, \overline{\mathsf{cm}}, \bar{i}, \overline{m}, \overline{\mathsf{dcm}})$ only if there exists a prefix $Q \subset \mathsf{Hist}_{\mathsf{GRO}}$ (the GRO table of $\mathcal{F}_{\mathsf{NVC}}$) such that extraction outputs a vector $\mathbf{m}'$ with no $\bot$ entries (i.e. Step IVf of $\mathcal{F}_{\mathsf{NVC}}$ accepts).
Since SCode.Extr is deterministic, the event $\widehat{\mathbf{m}}[\bar{i}] \neq \overline{m}$ occurs if the inputs to SCode.Extr differ at the binding point and at verification time. This occurs if $\mathcal{E}$ finds and queries a collision for $\overline{\mathsf{cm}}$, thus changing the list Q computed in Step IV(f)i of $\mathcal{F}_{\mathsf{NVC}}$; this occurs with probability $t^2 \cdot 2^{-h_{\mathsf{cm}}}$ by the birthday bound.

Case (b). In the initial execution, $\mathcal{E}$ first receives $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$ for some x. (If invoked, the output of $\mathsf{SCode.Extr}(n, \mathsf{sid}, \overline{\mathsf{cm}}, \mathsf{Hist}^*)$ is a vector $\widehat{\mathbf{m}}$ which may have $\bot$ entries.) Afterwards, $\mathcal{E}$ sends input $(\mathsf{Commit}, \widehat{\mathbf{m}})$ to $\mathcal{F}_{\mathsf{NVC}}$, such that $\mathbf{m}[\bar{i}] \neq \bot$, and receives $(\mathtt{Receipt}, \mathsf{cm} = \overline{\mathsf{cm}}, \mathbf{dcm})$.
In this case, the *latest* binding point will be immediately after $\mathcal{F}_{\mathsf{NVC}}$ invokes **Commit**. As such, the bad event never occurs because Step IVc of $\mathcal{F}_{\mathsf{NVC}}$ ensures that only the plaintext value given to **Commit** of $\mathcal{F}_{\mathsf{NVC}}$, will be accepted at verification time. Hence, the probability of the bad event is 0.

Case (c). In the initial execution, $\mathcal{E}$ does not receive $\overline{\mathsf{cm}} = \mathsf{GRO}(x)$, prior to invoking $\mathcal{F}_{\mathsf{NVC}}$ with the **Commit** interface.
First, suppose the binding point occurs during the invocation of **Commit** and $\overline{\mathsf{cm}}$ is not the commitment output by $\mathcal{F}_{\mathsf{NVC}}$. Then, Step IVe of $\mathcal{F}_{\mathsf{NVC}}$ prevents $\mathcal{F}_{\mathsf{NVC}}$ from accepting

when the binding point for $\overline{\mathsf{cm}}$ occurs due to programming done by SCode.Com or SCode.Equiv.

Second, (if the binding point occurs after the execution of the **Commit** subroutine of $\mathcal{F}_{\mathsf{NC}}$ and $\overline{\mathsf{cm}}$ is not the commitment output by $\mathcal{F}_{\mathsf{NC}}$.), $\mathcal{E}$ can influence the outputs of SCode.Extr only by changing the inputs, specifically the query set $Q \subseteq \mathsf{Hist}_{\mathsf{GRO}}$ computed in Step IV(f)i of $\mathcal{F}_{\mathsf{NVC}}$. Recall that Q contains all queries before (and including) the latest query-answer pair of the form $(\cdot, \overline{\mathsf{cm}})$. Thus, Q changes during the execution only if $\mathcal{E}$ finds and queries a collision in $\mathcal{F}_{\mathsf{GRO}}$. As such, the probability $\mathcal{E}$ finds a collision in $\mathcal{F}_{\mathsf{GRO}}$ is $t^2 \cdot 2^{-h_{\mathsf{cm}}}$ by the birthday bound.

Last, we note that the output of invoking $\mathcal{F}_{\mathsf{NVC}}$'s **Commit** interface $(\texttt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ does not leak information that helps $\mathcal{E}$ find a collision, since each execution of Com or Equiv makes at most a single query to $\mathcal{F}_{\mathsf{GRO}}$ and the **Verify** interface never accepts any $(\cdot, \mathsf{cm})$ that was programmed by those algorithms.

Hence, the bad event occurs with probability $t^2 \cdot 2^{-h_{\mathsf{cm}}}$.

Case 3. Neither Case 1 nor Case 2 occurs, i.e. the binding point is when $\mathcal{F}_{\mathsf{NVC}}$ receives the SCode.Extr algorithm.

This implies $\mathsf{SCode.Extr}(\overline{\mathsf{cm}}, \perp) = \mathsf{SCode.Extr}(\overline{\mathsf{cm}}, \mathsf{Hist}_{\mathsf{GRO}})$ since Hist of $\mathcal{F}_{\mathsf{GRO}}$ does not bind $\overline{\mathsf{cm}}$. Hence, the probability of extracting $\mathbf{m}'$ such that $\mathbf{m}'[\bar{i}] \neq \overline{m}$ is 0.

By union bound over the cases and the positions $i \in [n]$, $\mathcal{E}$ wins the binding game with probability $n \cdot 2t^2 \cdot 2^{-h_{\mathsf{cm}}}$. □

7 Π_{NVC}: The non-interactive vector commitment scheme

We construct a non-interactive vector commitment scheme Π_{NVC} (Figure 15), which is a reformulation of Merkle tree succinct commitments [Mer89] in which the internal nodes are computed using non-interactive commitments $\mathcal{F}_{\mathsf{NC}}$ (Figure 3). Subsequently, we show that Π_{NVC} *perfectly* UC-realizes $\mathcal{F}_{\mathsf{NVC}}$ (Theorem 7.1); the proof appears in Section 7.1.

Prior to describing Π_{NVC}, we first fix a method for indexing nodes in the Merkle tree and utilize these indices as session IDs to identify invocations of $\mathcal{F}_{\mathsf{NC}}$ in Π_{NVC}.

Node indices are session IDs. A node's index is a layer-position pair $(\ell,\ \mathsf{pos})$. The output node has depth $\ell = 1$, and the input layer has depth $\ell = \mathsf{d} := \log |\mathbf{m}|$. Within layer ℓ, nodes are assigned a position $\mathsf{pos} \in \left[2^{\ell-1}\right]$. Then, the session ID used to identify $\mathcal{F}_{\mathsf{NC}}$ (a.k.a. a node of the Merkle tree) is $\mathsf{sid} := (\mathsf{sid}_{\mathsf{VC}}, (\ell,\ \mathsf{pos}))$, in which $\mathsf{sid}_{\mathsf{VC}}$ is the session ID of the vector commitment scheme invoking $\mathcal{F}_{\mathsf{NC}}$.

Our construction additionally applies $\mathcal{F}_{\mathsf{NC}}$ to each entry of $\mathbf{m}$, a process we call the "initial commitment". We notate this as layer $\mathsf{d} + 1$ of the Merkle tree. This is done so that unopened entries of $\mathbf{m}$ are hidden.

We describe the vector commitment protocol Π_{NVC} in Figure 15. See Figure 14 for a diagram of the construction. In Π_{NVC}, we denote $n := |\mathbf{m}|$, in which $\mathbf{m}$ is the committed vector. In this paper, we specifically consider $n = 2^{\mathsf{d}}$ for some $\mathsf{d} \in \mathbb{N}$. Note that it's straightforward to generalize to committing to any message of length $\leq n$, via padding.

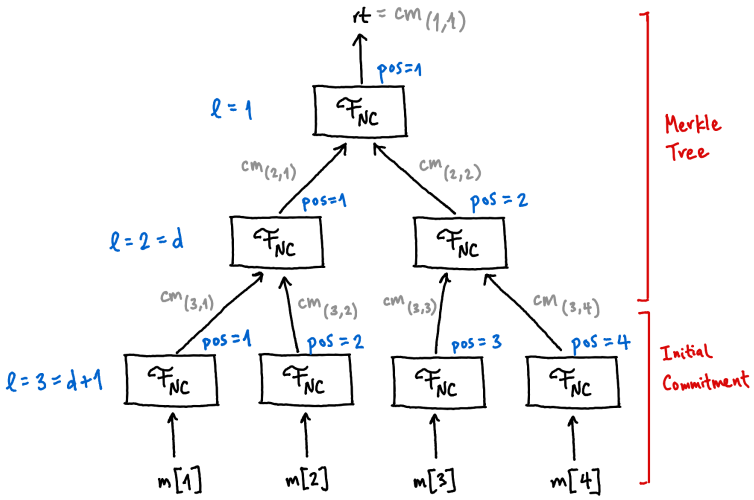

Figure 14: Vector commitment protocol for a message $\mathbf{m}$ of length $n = 4$.

Theorem 7.1. *Let $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ be an ideal global random oracle, with output length $h_{\mathsf{cm}} \in \mathbb{N}$. Let $\mathcal{F}_{\mathsf{NC}}(\Sigma, h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ (Figure 3) be an ideal commitment in the presence of $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ and $\mathcal{F}_{\mathsf{lib}}$, in which $h_{\mathsf{dcm}} \in \mathbb{N}$ and Σ is the plaintext alphabet. Let $n \in \mathbb{N}$ be a parameter denoting the length of the committed vector.*
Then, Π_{NVC} in Figure 15 perfectly UC-realizes the ideal vector commitment functionality $\mathcal{F}_{\mathsf{NVC}}(\Sigma, n, h_{\mathsf{cm}}, h_{\mathsf{dcm}})$ in the presence of $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ and $\mathcal{F}_{\mathsf{lib}}$ (described in Figure 8), even against adaptive corruptions.
Also, commitments are h_{cm}-bit strings, regardless of the alphabet for vector elements Σ or n.

7.1 Proof of Theorem 7.1

To show that Π_{NVC} UC-realizes $\mathcal{F}_{\mathsf{NVC}}$ in the presence of $\mathcal{F}_{\mathsf{lib}}$ and $\mathcal{F}_{\mathsf{GRO}}$, we first present the simulator code, $\mathcal{F}_{\mathsf{NVC}}$.SCode. To simulate Π_{NVC}, $\mathcal{F}_{\mathsf{NVC}}$.SCode runs the simulator code $\mathcal{F}_{\mathsf{NC}}$.SCode that is obtained from $\mathcal{F}_{\mathsf{lib}}$, and "plays the role" of $\mathcal{F}_{\mathsf{NC}}$ to $\mathcal{F}_{\mathsf{NC}}$.SCode, which is executed internally by $\mathcal{F}_{\mathsf{NVC}}$.SCode. This involves simulating the interactions of $\mathcal{F}_{\mathsf{NC}}$ with all global functionalities, including $\mathcal{F}_{\mathsf{lib}}$ and $\mathcal{F}_{\mathsf{GRO}}$.

It is useful to think of $\mathcal{F}_{\mathsf{NC}}$.SCode as adversarial: indeed, it is provided by the environment to $\mathcal{F}_{\mathsf{lib}}$, and there is no guarantee that $\mathcal{F}_{\mathsf{NC}}$.SCode is the code that was constructed in Section 5.1 to prove secuity of Π_{NC}.

The simulator code is constructed in Section 7.1.1. Section 7.1.2 argues that $\mathsf{Exec}_{\mathcal{E},\Pi_{\mathsf{NVC}}} \approx \mathsf{Exec}_{\mathcal{E},\mathcal{F}_{\mathsf{NVC}},\mathsf{SCode}}$.

7.1.1 The simulator's code library

We first overview how the simulator emulates instances of $\mathcal{F}_{\mathsf{NC}}$. Subsequently, we define the simulator's code library $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode} = \{\mathsf{Setup}, \mathsf{Com}, \mathsf{Equiv}, \mathsf{Extr}\}$.

Emulating instances of $\mathcal{F}_{\mathsf{NC}}$. Recall that $\mathcal{F}_{\mathsf{NVC}}$ generates $\mathsf{cm}, \mathbf{dcm}$ in two stages: (1) run SCode.Com (without knowledge of $\mathbf{m}$) to generate cm; then (2) run SCode.Equiv to generate $\mathbf{dcm}[i]$ for every position i, given cm and $\mathbf{m}[i]$. For writing the simulator for Π_{NVC}, this requires generating a commitment string cm associated with an $\mathcal{F}_{\mathsf{NC}}$ instance before the committed message is known, then generating a decommitment string dcm after a committed message is known. We describe our approach.

Protocol Π_{NVC}

- **Commit:** The committer C does:
  1. Receive input $(\texttt{Commit}, \mathbf{m})$, in which $\mathbf{m} \in (\{0,1\}^e \cup \{\perp\})^n$.
  2. Compute the Merkle tree depth $\mathsf{d} := \log n$.
  3. For all $i \in [n]$: # Initial commitment of the input.
     (a) Set session ID $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, (\mathsf{d}+1,\ i))$, in which $\mathsf{sid}_{\mathsf{VC}}$ is the session ID of Π_{NVC}.
     (b) Send input $(\texttt{Commit}, \mathbf{m}[i])$ to the $\mathcal{F}_{\mathsf{NC}}$ of session ID $\mathsf{sid}_{\mathsf{NC}}$.
     (c) Receive output $(\texttt{Receipt}, \mathsf{cm}_{(\mathsf{d}+1,i)}, \mathsf{dcm}_{(\mathsf{d}+1,i)})$ from $\mathcal{F}_{\mathsf{NC}}$.
     (d) Output $\mathsf{cm}_{(\mathsf{d}+1,i)}$ as the output wire value.
  4. For layers $\ell \leq \mathsf{d}$ and for $\mathsf{pos} \in [2^{\ell-1}]$: # Compute the Merkle tree, from leaves to root.
     (a) Set the node's session ID $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, (\ell,\ \mathsf{pos}))$.
     (b) Find the left and right input values: $x_{\mathsf{L}} := \mathsf{cm}_{(\ell+1,\ 2\cdot\mathsf{pos}-1)}$ and $x_{\mathsf{R}} := \mathsf{cm}_{(\ell+1,\ 2\cdot\mathsf{pos})}$.
     (c) Send input $(\texttt{Commit}, (x_{\mathsf{L}}, x_{\mathsf{R}}))$ to $\mathcal{F}_{\mathsf{NC}}$ of session ID $\mathsf{sid}_{\mathsf{NC}}$.
     (d) Receive output $(\texttt{Receipt}, \mathsf{cm}_{(\ell,\mathsf{pos})}, \mathsf{dcm}_{(\ell,\mathsf{pos})})$ from $\mathcal{F}_{\mathsf{NC}}$.
     (e) Output $\mathsf{cm}_{(\ell,\mathsf{pos})}$ as the output wire value.
  5. Set $\mathsf{rt} := \mathsf{cm}_{(1,1)}$. # Merkle tree root.
  6. For $i \in [n]$: # Compute Merkle tree authentication paths.
     (a) Compute the authentication path:

     $$\mathsf{ap}_i := \{((\mathsf{sid}_{\mathsf{VC}}, \mathsf{Out}), \mathsf{cm}_{\mathsf{L}}, \mathsf{cm}_{\mathsf{R}}, \mathsf{dcm}_{\mathsf{Out}}, \mathsf{cm}_{\mathsf{Out}})\}_{\mathsf{Out}\in\mathsf{A}} \quad ,$$

     in which $\mathsf{L} = (\ell+1,\ 2\cdot\mathsf{pos}-1)$, $\mathsf{R} = (\ell+1,\ 2\cdot\mathsf{pos})$, $\mathsf{Out} = (\ell,\ \mathsf{pos})$, and A is the set of all nodes on the path from $\mathsf{cm}_{(\mathsf{d}+1,i)}$ to rt.
  7. Define $\mathbf{dcm}$ as the vector such that entry $i \in [n]$ is $\mathbf{dcm}[i] := \left(\mathsf{ap}_i, \mathsf{dcm}_{(\mathsf{d}+1,\ i)}\right)$, where $\mathsf{dcm}_{(\mathsf{d}+1,\ i)}$ is decommitment from the initial commitment of the input $\mathbf{m}[i]$. # Vector of decommitments.
  8. Output $(\texttt{Commit}, \mathsf{cm} := \mathsf{rt}, \mathbf{dcm})$.
- **Verify:** The receiver R does:
  1. Receive input $(\texttt{Verify}, \overline{\mathsf{cm}}, i, \overline{m}, \overline{\mathsf{dcm}})$.
  2. Parse $\overline{\mathsf{cm}} = \overline{\mathsf{rt}}$ as a Merkle tree root.
  3. Parse $\overline{\mathsf{dcm}}$ as $\left(\overline{\mathsf{ap}}_i, \overline{\mathsf{dcm}}_{(\mathsf{d}+1,\ i)}\right)$, where $\overline{\mathsf{ap}}_i$ is a claimed authentication path and $\overline{\mathsf{dcm}}_{(\mathsf{d}+1,\ i)}$ is a decommitment associated with an initial commitment to $\overline{m}$.
  4. Check the initial commitment:
     (a) Set session ID $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, (\mathsf{d}+1,\ i))$, in which $\mathsf{sid}_{\mathsf{VC}}$ is the session ID of Π_{NVC}.
     (b) Find the commitment value cm^* associated with $\overline{m}$ in $\overline{\mathsf{ap}}_i$:
        i. Identify the entry $(\mathsf{sid}_{\mathsf{NC}}, \mathsf{cm}_{\mathsf{L}}, \mathsf{cm}_{\mathsf{R}}, \cdot, \cdot)$ in $\overline{\mathsf{ap}}_i$.
        ii. If $i \bmod 2 = 1$, set $\mathsf{cm}^* := \mathsf{cm}_{\mathsf{L}}$. Otherwise, set $\mathsf{cm}^* := \mathsf{cm}_{\mathsf{R}}$.
     (c) Send input $\left(\texttt{Verify}, \mathsf{cm}^*, (\mathsf{sid}_{\mathsf{NC}}, \overline{m}), \overline{\mathsf{dcm}}_{(\mathsf{d}+1,\ i)}\right)$ to $\mathcal{F}_{\mathsf{NC}}$.
     (d) Receive output $(\texttt{VerStatus}, v)$ from $\mathcal{F}_{\mathsf{NC}}$; this check passes if $v = 1$.
  5. Check that $\overline{\mathsf{ap}}_i$ is a valid authentication path:
     (a) For all $(\mathsf{sid}_{\mathsf{NC}}, x_{\mathsf{L}}, x_{\mathsf{R}}, \mathsf{dcm}, \mathsf{cm}) \in \overline{\mathsf{ap}}_i$,
     # Each entry of $\overline{\mathsf{ap}}_i$ corresponds to a node in the authentication path.
        i. Send input $(\texttt{Verify}, \mathsf{cm}, (\mathsf{sid}_{\mathsf{NC}}, x_{\mathsf{L}}, x_{\mathsf{R}}), \mathsf{dcm})$ to $\mathcal{F}_{\mathsf{NC}}$.
        ii. Receive output $(\texttt{VerStatus}, v)$ from $\mathcal{F}_{\mathsf{NC}}$; this check passes if $v = 1$.
     (b) The entries in $\overline{\mathsf{ap}}_i$ correspond to a path from leaf i to $\overline{\mathsf{rt}}$.
     (c) The value $\overline{\mathsf{cm}}$ is the output of the root node in $\overline{\mathsf{ap}}_i$. Specifically, there exists an entry $((\mathsf{sid}_{\mathsf{VC}}, (1,\ 1)), x_{\mathsf{L}}, x_{\mathsf{R}}, \mathsf{dcm}, \overline{\mathsf{cm}}) \in \overline{\mathsf{ap}}_i$.
  6. If all checks pass, output $(\texttt{VerStatus}, 1)$. Else, output $(\texttt{VerStatus}, 0)$.

Figure 15: The vector commitment scheme protocol, with global subroutine $\mathcal{F}_{\mathsf{GRO}}$.

Generating cm without knowing the committed plaintext is not straightforward, since $\mathsf{SCode.Com}$ cannot directly invoke the **Commit** interface of $\mathcal{F}_{\mathsf{NC}}$ on a plaintext it does not know. A partial solution is allowing SCode to run the simulator code of $\mathcal{F}_{\mathsf{NC}}$ directly in-line, i.e. $\mathsf{SCode.Com}$ outputs $\mathsf{cm} \leftarrow \mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Com}$. However, the simulator code of $\mathcal{F}_{\mathsf{NC}}$ is un-trusted code that is chosen by the environment; absent safeguards, it may cause SCode for $\mathcal{F}_{\mathsf{NVC}}$ to exhibit bad behavior such as outrunning its time bound, i.e. fail to faithfully emulate an execution of $\mathcal{F}_{\mathsf{NC}}$. In other words, in order to emulate $\mathcal{F}_{\mathsf{NC}}$, the simulator for Π_{NVC} must impose the same safeguards as in $\mathcal{F}_{\mathsf{NC}}$.

More specifically, to generate cm, $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode}$ runs the **Commit** code of $\mathcal{F}_{\mathsf{NC}}$ "in-line" with a dummy message (e.g. $m := \perp$). This gives the proper distribution of cm since cm is generated independently of m in $\mathcal{F}_{\mathsf{NC}}$, and $\mathcal{F}_{\mathsf{NC}}$ will fail if its simulator code is rogue.

The next task is generating dcm, given m. To do so, we have SCode re-run $\mathcal{F}_{\mathsf{NC}}$ with m and the same internal randomness as the first execution.

In detail, our approach for emulating an $\mathcal{F}_{\mathsf{NC}}$ instance with session ID sid is to run the code of the **Commit** stage in two stages:

Stage 1: **Generate cm.** Set the committed plaintext as $m_1 := \perp$. Run $\mathcal{F}_{\mathsf{NC}}$ with input $(\mathtt{Commit}, m_1)$ and receive $(\mathsf{cm}_1, \mathsf{dcm}_1 = \perp)$ as output. Set $\mathsf{cm} := \mathsf{cm}_1$. Denote this procedure $(\mathsf{cm}, \mathsf{st}) \leftarrow \mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Com}}(\mathsf{sid}; r)$, where r is an optional input for explicitly designating the randomness used by $\mathcal{F}_{\mathsf{NC}}$ and st contains the internal state of $\mathcal{F}_{\mathsf{NC}}$ in this execution.[15] # Since $m_1 := \perp$, $\mathcal{F}_{\mathsf{NC}}$ does not run $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Equiv}$, so dcm is not generated.

Stage 2: **Generate dcm.** At this stage, the desired plaintext $m_2 \in \Sigma$ is known. Re-run $\mathcal{F}_{\mathsf{NC}}$ with input $(\mathtt{Commit}, m_2)$ with the same random tapes as from Stage 1, and receive $(\mathsf{cm}_2, \mathsf{dcm}_2)$ as output. Set $\mathsf{dcm} := \mathsf{dcm}_2$. Denote this procedure $(\mathsf{dcm}, \mathsf{State}^{\mathsf{NC}}) \leftarrow \mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Equiv}}(\mathsf{sid}, \mathsf{st}, m_2)$, in which is the committer state $\mathsf{State}^{\mathsf{NC}}$ which is returned in case of adaptive corruption and st is the internal state of $\mathcal{F}_{\mathsf{NC}}$ in the first run, including the random tapes. # $\mathcal{F}_{\mathsf{NC}}$ runs $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Equiv}$, so dcm is generated.

Observe that $\mathsf{cm}_1 = \mathsf{cm}_2$ since both were generated using Com with the same input (the same random tape and without knowledge of the plaintext in both cases). Further observe that as part of the simulation of Π_{NVC}, only dcm_2 is released, so the switch from having no decommitment value to having one is undetectable.

Further, re-running $\mathcal{F}_{\mathsf{NC}}$ guarantees that all the checks performed by $\mathcal{F}_{\mathsf{NC}}$ regarding the adversarial $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Com}$ and $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Equiv}$ are performed here as well.

Extraction. We consider how to define the extractor $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$. Recall that $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$ needs to extract a plaintext from a commitment string cm generated by $\mathcal{F}_{\mathsf{NC}}$. This extraction will be carried out for all instances of $\mathcal{F}_{\mathsf{NC}}$, recursively down the Merkle tree of commitments, so that the vector committed by $\mathcal{F}_{\mathsf{NVC}}$ is recovered.

For extracting the committed plaintext m from cm from a particular instance of $\mathcal{F}_{\mathsf{NC}}$, $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$ runs the following procedure, denoted $\mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Extr}}(\mathsf{sid}, \mathsf{cm}, \mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}])$:

1. Play the role of $\mathcal{F}_{\mathsf{NC}}$ of session sid with input $(\mathtt{Verify}, \mathsf{cm}, m, \mathsf{dcm})$, in which m, dcm are arbitrary values and $\mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}] := \{\ (x, y) \mid (x, y) \in \mathsf{Hist}_{\mathsf{GRO}} \ \wedge\ x = (\mathsf{sid}, \cdot)\ \}$, in which $\mathsf{Hist}_{\mathsf{GRO}}$ is the GRO history of $\mathcal{F}_{\mathsf{NVC}}$. Operationally, $\mathcal{F}_{\mathsf{NVC}}\mathsf{SCode.Extr}$ does this by in-lining the code for the **Verify** subroutine of $\mathcal{F}_{\mathsf{NC}}$, which invokes $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Extr}$.[16]

[15] Note that r includes the randomness used by $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Com}$ and $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Equiv}$. However, $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode.Equiv}$ is not executed in this stage.

[16] The code linking is done by $\mathcal{F}_{\mathsf{lib}}$.

2. To extract the committed m:
   (a) If the execution of $\mathcal{F}_{\mathsf{NC}}$ does not invoke $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode}.\mathsf{Extr}$ and there exists an entry $(\mathsf{cm}, m', \mathsf{dcm}', \cdot, 1) \in \mathsf{CState}$, output (m', dcm').
   (b) If the execution of $\mathcal{F}_{\mathsf{NC}}$ invokes $\mathcal{F}_{\mathsf{NC}}.\mathsf{SCode}.\mathsf{Extr}$ to get output (m', dcm'), output (m', dcm').
   (c) Else, output $(m' := \bot, \mathsf{dcm}' := \bot)$.

SCode for $\mathcal{F}_{\mathsf{NVC}}$. We now define the $\mathcal{F}_{\mathsf{NVC}}$ simulator code library $\mathsf{SCode} = \{\mathsf{Setup}, \mathsf{Com}, \mathsf{Equiv}, \mathsf{Extr}\}$. Below, every $\mathcal{F}_{\mathsf{NC}}$ instance has randomness that is independently sampled. Note that these algorithms are (non-adaptively) chosen by the adversary. Also, whenever an algorithm $A \in \mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode}$ interacts with $\mathcal{F}_{\mathsf{GRO}}$, $\mathcal{F}_{\mathsf{NVC}}$ handles the interaction, via the interface provided in Figure 9.

```
• Setup$(n, \mathsf{sid}_{\mathsf{VC}})$.
  1. Parse $n$ as the length of the committed message and $\mathsf{sid}_{\mathsf{VC}}$ as a session ID.
  2. Set $\mathsf{d} := \log n$. # Compute Merkle tree depth.
  3. For layers $\mathsf{d} \in \{\ell, \ldots, 1\}$ and for $\mathsf{pos} \in [2^{\ell-1}]$: # Spawn gateways for Merkle tree nodes.
     (a) Define the following Merkle tree location: $\mathsf{Out} := (\ell, \mathsf{pos})$.
     (b) Set the node's session ID $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, \mathsf{Out})$.
     (c) Send $(\mathsf{Spawn}, \mathsf{sid}_{\mathsf{NC}})$ to Step VI of $\mathcal{F}_{\mathsf{NVC}}$. # Spawn a gateway for session $\mathsf{sid}_{\mathsf{NC}}$.
     (d) If $\mathsf{d} = \ell$, for $\mathsf{c} \in \{\mathsf{L} := (\ell + 1,\ 2 \cdot \mathsf{pos} - 1), \mathsf{R} := (\ell + 1,\ 2 \cdot \mathsf{pos})\}$: # Also spawn gateways for the initial commitment layer.
         i. Set $\mathsf{sid}_{\mathsf{c}} := (\mathsf{sid}_{\mathsf{VC}}, \mathsf{c})$.
        ii. Send $(\mathsf{Spawn}, \mathsf{sid}_{\mathsf{c}})$ to Step VI of $\mathcal{F}_{\mathsf{NVC}}$.
```

- $\mathsf{Com}(n, \mathsf{sid}_{\mathsf{VC}}, r) \to (\mathsf{cm}, \mathsf{st})$.
  1. Parse n as the length of the committed message, $\mathsf{sid}_{\mathsf{VC}}$ as a session ID, and $r = (r_{\mathsf{cm}}, r_{\mathsf{dcm}}) \in (\{0,1\}^{h_{\mathsf{cm}}})^{\lambda \cdot (2n-1)} \times (\{0,1\}^{h_{\mathsf{dcm}}})^{\lambda \cdot (2n-1)}$ as random tapes.
     # For simplicity of exposition, we use the notation $r_{\mathsf{cm}}[(\ell, \mathsf{pos})]$ to denote the portion of the r_{cm} tape allocated to the $\mathcal{F}_{\mathsf{NC}}$ instance at layer $\ell \in [\mathsf{d}+1]$ and position $\mathsf{pos} \in [2^{\ell-1}]$. Same for r_{dcm}.
  2. Set $\mathsf{d} := \log n$. # Compute Merkle tree depth.
  3. For $i \in [n]$: # Sample outputs of $\mathcal{F}_{\mathsf{NC}}$ nodes in initial commitment layer.
     (a) Set $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, (\mathsf{d}+1,\ i))$.
     (b) Compute $(\mathsf{cm}_{(\mathsf{d}+1,\, i)}, \mathsf{st}_{(\mathsf{d}+1,\, i)}) := \mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Com}}(\mathsf{sid}_{\mathsf{NC}}; r_{\mathsf{cm}}[(\mathsf{d}+1,\ i)], r_{\mathsf{dcm}}[(\mathsf{d}+1,\ i)])$.
     (c) If $\mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Com}}$ tries to program $\mathcal{F}_{\mathsf{GRO}}$, i.e. to send input $(\mathtt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$:
        i. Run $\mathsf{Program}(\mathsf{SubFuncs}, \mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}_{\mathsf{NVC}}}, \mathsf{sid}_{\mathsf{NC}}, \mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Com}, x, y)$. [a]
        ii. Receive the output $(\mathtt{EvalOutput}, y)$.
  4. For layers $\ell \leq \mathsf{d}$ and for $\mathsf{pos} \in [2^{\ell-1}]$: # Compute the Merkle tree, from leaves to root.
     (a) Set the node's session ID $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, (\ell,\ \mathsf{pos}))$.
     (b) Find the left and right input values: $x_{\mathsf{L}} := \mathsf{cm}_{(\ell+1,\, 2\cdot\mathsf{pos}-1)}$ and $x_{\mathsf{R}} := \mathsf{cm}_{(\ell+1,\, 2\cdot\mathsf{pos})}$.
     (c) Send input $(\mathtt{Commit}, (x_{\mathsf{L}}, x_{\mathsf{R}}))$ to $\mathcal{F}_{\mathsf{NC}}$ of session ID $\mathsf{sid}_{\mathsf{NC}}$.
     (d) Receive output $(\mathtt{Receipt}, \mathsf{cm}_{(\ell,\mathsf{pos})}, \mathsf{dcm}_{(\ell,\mathsf{pos})})$ from $\mathcal{F}_{\mathsf{NC}}$.
     (e) Output $\mathsf{cm}_{(\ell,\mathsf{pos})}$ as the output wire value.
  5. Set $\mathsf{rt} := \mathsf{cm}_{(1,1)}$. # Merkle tree root.
  6. Set the state st to contain all wire values $x_{(\cdot,\cdot)}$, decommitments $\mathsf{dcm}_{(\cdot,\cdot)}$, the internal state of $\mathcal{F}_{\mathsf{NC}}$ $\mathsf{st}_{(\mathsf{d}+1,\, i)}$[b] for all $i \in [n]$, and internal committer states $\mathsf{State}^{\mathsf{NC}}_{\mathsf{Out}}$ of $\mathcal{F}_{\mathsf{NC}}$ from Step 4 that are returned in the case of adaptive corruption.
  7. Output $(\mathsf{cm} := x_{(1,\,1)}, \mathsf{st})$. # $x_{(1,\,1)}$ is the Merkle tree root.

[a] The following inputs are honestly derived from the state of $\mathcal{F}_{\mathsf{NVC}}$: $\mathsf{Hist}_{\mathsf{GRO}}$ is the GRO history of $\mathcal{F}_{\mathsf{NVC}}$, $\Phi_{\mathcal{F}_{\mathsf{NVC}}}$ is the query-answer predicate of Figure 13, $\mathsf{sid}_{\mathsf{NC}}$ is the session ID of $\mathcal{F}_{\mathsf{NC}}$, and $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Com}$ is the calling algorithm. The following inputs may be chosen by $\mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Com}}$: x is the query point, and y is the query answer.

[b] $\mathsf{st}_{(\mathsf{d}+1,\, i)}$ contains the random tapes $r_{\mathsf{cm}}[(\mathsf{d}+1,\ i)], r_{\mathsf{dcm}}[(\mathsf{d}+1,\ i)]$

- $\mathsf{Equiv}(n, \mathsf{sid}_{\mathsf{VC}}, \mathsf{cm}, \mathsf{st}, (i, \mathbf{m}[i]), r_{\mathsf{dcm}}) \rightarrow (\mathbf{dcm}[i], \mathsf{State})$.
  1. Parse n as the length of the committed message, $\mathsf{sid}_{\mathsf{VC}}$ as a session ID, cm as a commitment string, st as the internal state from Com, $\mathbf{m}$ as a message vector for location $i \in [n]$, and $r_{\mathsf{dcm}} \in (\{0,1\}^{h_{\mathsf{dcm}}})^*$.
     # For simplicity of exposition, we use the notation $r_{\mathsf{dcm}}[(\ell, \mathsf{pos})]$ to denote the portion of r_{dcm} that is allocated to the $\mathcal{F}_{\mathsf{NC}}$ instance at layer $\ell = \mathsf{d} + 1$ and position $\mathsf{pos} \in [n]$.
  2. Set $\mathsf{d} := \log n$. # Compute Merkle tree depth.
  3. Set $\mathsf{Out} := (\mathsf{d}+1,\ i)$. # Position of Merkle tree leaf, where we equivocate the message.
  4. Set $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, \mathsf{Out})$.
  5. Compute $(\mathsf{dcm}_{\mathsf{Out}}, \mathsf{State}^{\mathsf{NC}}_{\mathsf{Out}}) := \mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Equiv}}(\mathsf{sid}_{\mathsf{NC}}, \mathsf{st}_{\mathsf{Out}}, \mathbf{m}[i])$, where
     $\mathsf{st}_{\mathsf{Out}} := (r_{\mathsf{cm}}[\mathsf{Out}], r_{\mathsf{dcm}}[\mathsf{Out}])$.
     # Equivocate the decommitment value for the initial commitments.
     (a) If $\mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Equiv}}$ tries to program $\mathcal{F}_{\mathsf{GRO}}$, i.e. to send input $(\mathtt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$:
        i. Run $\mathsf{Program}(\mathsf{SubFuncs}, \mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}_{\mathsf{NVC}}}, \mathsf{sid}_{\mathsf{NC}}, \mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode}.\mathsf{Equiv}, x, y)$.
        ii. Receive the output $(\mathtt{EvalOutput}, y)$.
  6. Compute the authentication path for decommitting to position i from the above values:
     $$\mathsf{ap}_i := \left\{\left((\mathsf{sid}_{\mathsf{VC}}, (\ell,\ \mathsf{pos})), x_{(\ell+1,\, 2\cdot\mathsf{pos}-1)}, x_{(\ell+1,\, 2\cdot\mathsf{pos})}, \mathsf{dcm}_{(\ell,\, \mathsf{pos})}, x_{(\ell,\, \mathsf{pos})}\right)\right\}_{(\ell,\, \mathsf{pos})\in\mathsf{A}} ,$$
     in which A is the set of all nodes on the path from x_{Out} to rt.
  7. Set State to include st and $\mathsf{State}^{\mathsf{NC}}_{\mathsf{Out}}$.
  8. Output $(\mathbf{dcm}[i] := (\mathsf{ap}_i, \mathsf{dcm}_{\mathsf{Out}}), \mathsf{State})$.

```
• $\mathsf{Extr}(n, \mathsf{sid}_{\mathsf{VC}}, \mathsf{cm}, Q) \to (\mathbf{m}, \mathbf{dcm})$.
  1. Parse $n$ as the length of the committed message, $\mathsf{sid}_{\mathsf{VC}}$ as a session ID, $\mathsf{cm}$ as a commitment string, $Q \subseteq \{0,1\}^* \times \{0,1\}^\lambda$ as a list of query-answer pairs for $\mathcal{F}_{\mathsf{GRO}}$.
  2. Set $\mathsf{d} := \log n$. # Compute Merkle tree depth.
  3. Extract the message vector $\mathbf{m}$, as follows:
     (a) Set $x_{(1,\,1)} := \mathsf{cm}$.
     (b) For $\ell \in [\mathsf{d}+1]$ and for $\mathsf{pos} \in [2^{\ell-1}]$, do the following at node $(\ell, \mathsf{pos})$:
         # Extract Merkle tree values from root to leaves, step by step.
         i. Define positions $\mathsf{Out} := (\ell,\ \mathsf{pos})$, $\mathsf{L} := (\ell+1,\ 2 \cdot \mathsf{pos} - 1)$, and $\mathsf{R} := (\ell+1,\ 2 \cdot \mathsf{pos})$.
         ii. Set $\mathsf{sid}_{\mathsf{NC}} := (\mathsf{sid}_{\mathsf{VC}}, \mathsf{Out})$.
         iii. If $\ell \leq \mathsf{d}$, extract input wire values: # For Merkle tree.
              – Run $((x_{\mathsf{L}}, x_{\mathsf{R}}), \mathsf{dcm}_{\mathsf{Out}}) := \mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Extr}}(\mathsf{sid}_{\mathsf{NC}}, x_{\mathsf{Out}}, \mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}_{\mathsf{NC}}])$.
         iv. Else (if $\ell = \mathsf{d}+1$), extract the committed message value: # Extract values in initial commitment layer.
              – Run $(\overline{x[i]}, \mathsf{dcm}_{\mathsf{Out}}) := \mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Extr}}(\mathsf{sid}_{\mathsf{NC}}, x_{\mathsf{Out}}, \mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}_{\mathsf{NC}}])$.
         v. For each $i \in [n]$, if $\mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Extr}}$ outputs $(\bot, \bot)$ in either Step 3(b)iii or Step 3(b)iv on the path from root to leaf $i$, then set $\overline{x[i]} := \bot$. (Note that $\mathcal{F}_{\mathsf{NC}}.\widetilde{\mathsf{Extr}}$ only produces output when the extracted tuple is valid.) # Set coordinate-wise extraction failure cases.
     (c) Set $\mathbf{m} := \left(\overline{x[i]}\right)_{i \in [n]}$.
  4. Compute the vector $\mathbf{dcm}$. For every location $i \in [n]$:
     (a) Compute the authentication path for decommitting to position $i$ from the above values:

         $$\mathsf{ap}_i := \left\{\left((\mathsf{sid}_{\mathsf{VC}}, (\ell,\ \mathsf{pos})), x_{(\ell+1,\,2\cdot\mathsf{pos}-1)}, x_{(\ell+1,\,2\cdot\mathsf{pos})}, \mathsf{dcm}_{(\ell,\,\mathsf{pos})}, x_{(\ell,\,\mathsf{pos})}\right)\right\}_{(\ell,\,\mathsf{pos}) \in \mathsf{A}},$$

         in which $\mathsf{A}$ is the set of all nodes on the path from $x_{\mathsf{Out}}$ to $\mathsf{rt}$.
     (b) Set $\mathbf{dcm}[i] := (\mathsf{ap}_i, \mathsf{dcm}_{(\mathsf{d}+1, i)})$.
  5. Output $(\mathbf{m}, \mathbf{dcm})$.
```

7.1.2 Validity of the simulation

Denote $\mathsf{REAL} := \mathsf{Exec}_{\mathcal{E}, \Pi_{\mathsf{NVC}}}$ and $\mathsf{IDEAL} := \mathsf{Exec}_{\mathcal{E}, \mathcal{F}_{\mathsf{NVC}}, \mathsf{SCode}}$. We argue that REAL and IDEAL are identical.

In both REAL and IDEAL, each node of the Merkle tree is computed using an ideal commitment $\mathcal{F}_{\mathsf{NC}}$. Note that a verification request fails in IDEAL if and only if it fails in REAL. Indeed in both executions, the verification fails exactly when $\mathcal{F}_{\mathsf{NC}}$ along the path from the queried leaf $\mathbf{m}[i]$ to the root fails. Hence, REAL and IDEAL are identical.

Remark 7.2. Since $\mathcal{E}$ "goes to sleep" during the execution of the **Commit** step of $\mathcal{F}_{\mathsf{NVC}}$ or Π_{NVC}, it cannot use a timing attack to distinguish between the cases (a) in IDEAL, $\mathcal{F}_{\mathsf{NVC}}$ fixes the Merkle tree (and programs the corresponding values in $\mathcal{F}_{\mathsf{GRO}}$), prior to computing the initial commitment layer; and (b) in REAL, Π_{NVC} starts by computing the initial commitments, then completes the Merkle tree.

The only remaining possible distinguishing strategy is if $\mathsf{SCode.Equiv}$ "programs" the $\mathcal{F}_{\mathsf{GRO}}$ programming history into $\mathcal{F}_{\mathsf{GRO}}$ for $\mathcal{E}$ to check later; this cannot be done because $\mathcal{F}_{\mathsf{NVC}}$ limits $\mathsf{SCode.Equiv}$ to programming $\mathcal{F}_{\mathsf{GRO}}$ at most once per invocation (see Step 4(b)i of $\mathcal{F}_{\mathsf{NVC}}$).

8 $\mathcal{F}_{\mathsf{NIZK}}$: Ideal functionality for non-interactive zero knowledge (NIZK)

We give the ideal functionality for non-interactive zero-knowledge for a relation $\mathcal{R}$. The present formulation builds on the formulations in [GOS12; CSW22], with two main changes: first, here $\mathcal{F}_{\mathsf{NIZK}}$ interacts with the global $\mathcal{F}_{\mathsf{GRO}}$; second, because $\mathcal{F}_{\mathsf{GRO}}$ is global, $\mathcal{F}_{\mathsf{NIZK}}$ only needs to model a single proof.[17]

$\mathcal{F}_{\mathsf{NIZK}}$ receives a tuple of algorithms $\mathsf{SCode} = \{\mathsf{Setup}, \mathsf{SimProve}, \mathsf{SimState}, \mathsf{Extr}\}$ via $\mathcal{F}_{\mathsf{lib}}$, which are defined as follows:

- $\mathsf{Setup}(\mathsf{sid}_{\mathsf{NIZK}})$.
  On input a session ID $\mathsf{sid}_{\mathsf{NIZK}}$, Setup may only take controls of sub-sessions of $\mathsf{sid}_{\mathsf{NIZK}}$ in $\mathcal{F}_{\mathsf{GRO}}$, as well as instantiate any subroutine machines which may interact with $\mathcal{F}_{\mathsf{GRO}}$ with their sub-session ID.
- $\mathsf{SimProve}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}) \to (\pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}})$.
  On input a session ID $\mathsf{sid}_{\mathsf{NIZK}}$, and an NP instance $\mathbb{x}$, outputs a (simulated) proof string π_{NIZK} and internal state $\mathsf{st}_{\mathsf{NIZK}}$. $\mathsf{SimProve}$ may program $\mathcal{F}_{\mathsf{GRO}}$.
- $\mathsf{SimState}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{w}, \pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}}) \to \mathsf{State}_{\mathsf{NIZK}}$. On input a session ID $\mathsf{sid}_{\mathsf{NIZK}}$, and an NP witness $\mathbb{w}$, a proof string π_{NIZK} and simulator state $\mathsf{st}_{\mathsf{NIZK}}$, outputs the prover's internal state $\mathsf{State}_{\mathsf{NIZK}}$. (This is for proving adaptive security.)
- $\mathsf{Extr}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \pi_{\mathsf{NIZK}}, \mathsf{Hist}_{\mathsf{GRO}}) \to \mathbb{w}$.
  On input a session ID $\mathsf{sid}_{\mathsf{NIZK}}$, an NP instance $\mathbb{x}$, a proof string π_{NIZK}, and a list of GRO query-answer pairs $\mathsf{Hist}_{\mathsf{GRO}}$, deterministically outputs an NP witness $\mathbb{w}$. Extr may not query or program $\mathcal{F}_{\mathsf{GRO}}$.

We present $\mathcal{F}_{\mathsf{NIZK}}$ in Figures 16 and 17.

8.1 Interface between $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{GRO}}$

Unlike for $\mathcal{F}_{\mathsf{NC}}$ and $\mathcal{F}_{\mathsf{NVC}}$, $\mathcal{F}_{\mathsf{NIZK}}$ maintains its (knowledge) soundness and zero-knowledge properties even if simulator algorithms may query/program $\mathcal{F}_{\mathsf{GRO}}$. Zero-knowledge holds because $\mathsf{SCode}.\mathsf{SimProve}$ never receives access to $\mathbb{w}$. Soundness holds because the **Verify** subroutine only accepts if the extractor recovers the witness

As such, the simulator algorithms $\mathsf{SCode}.\mathsf{SimProve}$ and $\mathsf{SCode}.\mathsf{Extr}$ may make arbitrary programming queries within the domain of the session ID of $\mathcal{F}_{\mathsf{NIZK}}$. Hence, these algorithms invoke the $\mathsf{Program}$ subroutine (Figure 12)) to program $\mathcal{F}_{\mathsf{GRO}}$, with the more permissive predicate $\Phi_{\mathcal{F}_{\mathsf{NIZK}}}$ (Figure 18).

8.2 Adaptive security

For $\mathcal{F}_{\mathsf{NIZK}}$, we model the corruption operation as an *input* to a particular real party (or to the ideal functionality),see Step III of $\mathcal{F}_{\mathsf{NIZK}}$. This provides tighter correspondence between the real and ideal executions than the standard modeling of party corruption in the UC framework, where the corruption event is modelded as an instruction that comes from the adversary. In addition to providing a somewhat stronger security guarantee, this formalism also simplifies the exposition: it frees the simulator from the need to communicate corruption requests from the environment to the ideal functionality, and then reporting simulated internal state back to the environment.

[17] Since the [GOS12; CSW22] formalism is aimed at capturing protocols where multiple instances use a shared common reference string, that is modeled as a non-global joint subroutine, they need to devise an ideal functionality that captures the generation and verification of multiple proofs within a single instance.

Functionality $\mathcal{F}_{\mathsf{NIZK}}$ (main)

```
I   Parameters: An NP relation $\mathcal{R}$.
II  Prove: Receive input $(\texttt{Prove}, \mathbb{x}, \mathbb{w})$ from a party $P$.
    (a) If this is not the first $(\texttt{Prove}, \cdot)$ input, end activation. # Fail-close clause.
    (b) If this the first activation, run the Initialization subroutine (Step V).
    (c) If $(\mathbb{x}, \mathbb{w}) \notin \mathcal{R}$, ignore the input.
    (d) Else if $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$:
         i. Compute $(\pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}}) \leftarrow \mathsf{SCode.SimProve}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x})$.
            # SCode.SimProve may program $\mathcal{F}_{\mathsf{GRO}}$ by calling Program (Figure 12).
            A. If $f(\pi_{\mathsf{NIZK}}) \notin r$ or SimProve commits SCodeError (Definition 4.6), output $\perp$. # $f$ is
               some injective function.
        ii. Append $(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \mathbb{w}, \pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}}, 1)$ to PState.
       iii. Send $(\texttt{Proof}, \pi_{\mathsf{NIZK}})$ to $P$.
III (Adaptive) Prover Corruption: Receive input $(\texttt{Corrupt})$ from the adversary $\mathcal{E}$.
    (a) If there exists $(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \mathbb{w}, \pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}}, 1) \in \mathsf{PState}$, in which $\mathsf{sid}_{\mathsf{NIZK}}$ is the local session ID:
        # Check if a proof was already generated.
         i. Compute $\mathsf{State}_{\mathsf{NIZK}} \leftarrow \mathsf{SCode.SimState}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{w}, \pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}})$. # Prover internal state.
        ii. Output $(\texttt{Corrupted}, \mathsf{State}_{\mathsf{NIZK}})$ to $\mathcal{E}$.
    (b) Else, output $(\texttt{Corrupted}, \perp)$ to $\mathcal{E}$.
IV  Verify: Receive input $(\texttt{Verify}, \overline{\mathbb{x}}, \overline{\pi_{\mathsf{NIZK}}})$ from a party $V$.
    (a) If this the first activation, run the Initialization subroutine (Step V).
    (b) If $(\overline{\mathbb{x}}, \mathbb{w}, \overline{\pi_{\mathsf{NIZK}}}, \cdot, v) \in \mathsf{PState}$ for any $\mathbb{w}$, output $(\texttt{VerStatus}, v)$.
        # Enforces completeness and consistency.
    (c) Else: # Enforce knowledge soundness.
         i. Extract a potential witness $\mathbb{w}' \leftarrow \mathsf{SCode.Extr}(\mathsf{sid}_{\mathsf{NIZK}}, \overline{\mathbb{x}}, \overline{\pi_{\mathsf{NIZK}}}, \mathsf{Hist}_{\mathsf{GRO}})$.
        ii. If Extr commits SCodeError (Definition 4.6), set $v := 0$.
       iii. Else, set $v := \mathcal{R}(\overline{\mathbb{x}}, \mathbb{w}')$.
        iv. Append $(\overline{\mathbb{x}}, \mathbb{w}', \overline{\pi_{\mathsf{NIZK}}}, \perp, v)$ to PState.
         v. Output $(\texttt{VerStatus}, v)$ to $V$.
```

Figure 16: The non-interactive zero knowledge functionality in the presence of $\mathcal{F}_{\mathsf{GRO}}$.

Functionality $\mathcal{F}_{\mathsf{NIZK}}$ (subroutines)

- V **Initialization:** In the first activation of session sid, do: # Can be `Prove` or `Verify`.
  - (a) Set the below as empty (i.e. as $\bot$), unless otherwise specified:
    - PState: a list of tuples $(\mathbb{x}, \mathbb{w}, \pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}}, v)$, in which $\mathbb{x}$ is an NP instance, $\mathbb{w}$ is an NP witness, $\mathsf{st}_{\mathsf{NIZK}}$ is simulator state, π_{NIZK} is a proof, and $v \in \{0,1\}$ is a bit representing whether $(\mathbb{x}, \mathbb{w}, \pi_{\mathsf{NIZK}})$ passes verification.
    - $\mathsf{Hist}_{\mathsf{GRO}}$: a list of tuples (x, y), in which x is a query point, and y is a query answer.
    - $\mathsf{ActiveAlg}$: records which algorithm is currently interacting with $\mathcal{F}_{\mathsf{GRO}}$.
    - $\mathsf{SubFuncs}$: a list of tuples $(\mathsf{ssid}, \mathsf{State})$, in which ssid is a session ID, and State is internal state associated with session ssid.
  - (b) Take control of session $\mathsf{sid}_{\mathsf{NIZK}}$ (where $\mathsf{sid}_{\mathsf{NIZK}}$ is the local sid) in $\mathcal{F}_{\mathsf{GRO}}$ by sending $(\mathtt{Control})$ to $\mathcal{F}_{\mathsf{GRO}}$, and receive a message m from $\mathcal{F}_{\mathsf{GRO}}$. # Enables observing GRO query-answer pairs and/or programming GRO responses for queries with prefix $\mathsf{sid}_{\mathsf{NIZK}}$.
    - i. If $m = (\mathtt{Ok}, \mathsf{Hist})$ such that $\mathsf{Hist} \neq \bot$, output $\bot$. # Failure.
  - (c) Send $(\mathtt{CodeRequest}, \mathcal{F}_{\mathsf{NIZK}})$ to $\mathcal{F}_{\mathsf{lib}}$ and receive $(\mathtt{Answer}, \mathsf{LinkedRecords})$, where $\mathsf{LinkedRecords} = (\mathcal{F}_{\mathsf{NIZK}}, \mathsf{SCode}, \cdot)$ and SCode is a tuple of algorithms $\{\mathsf{Setup}, \mathsf{SimProve}, \mathsf{SimState}, \mathsf{Extr}\}$.
  - (d) Run $\mathsf{SCode.Setup}$. If $\mathsf{SCode.Setup}$ invokes $(\mathsf{Spawn}, \mathsf{ssid})$, run Step VI. # Spawn gateway functionalities for simulating sub-functionalities.
  - (e) Let r denote the random string of $\mathcal{F}_{\mathsf{NIZK}}$.
- VI **Initialize gateway functionality:** When $(\mathsf{Spawn}, \mathsf{ssid})$ is invoked:
  - (a) If the input ssid has the form $(\mathsf{sid}_{\mathsf{NIZK}}, \cdot)$ and there is no entry of the form $(\mathsf{ssid}, \cdot) \in \mathsf{SubFuncs}$: # Ensure ssid is a valid sub-session ID of $\mathsf{sid}_{\mathsf{NIZK}}$.
    - i. Send $(\mathsf{Spawn}, \mathsf{ssid})$ to the gateway functionality GF (Figure 10), and receive a message m from GF.
      - A. If $m = (\mathtt{Ok}, \mathsf{Hist})$ and $\mathsf{Hist} = \bot$, add $(\mathsf{ssid}, \mathsf{State} := \bot)$ to the list $\mathsf{SubFuncs}$, where State is the state associated with ssid.
      - B. Else, output $\bot$. # Failure.
- VII **External request to program the GRO:** Upon receiving $(\mathtt{ProgramReq}, P, x)$ from $\mathcal{F}_{\mathsf{GRO}}$: # This interface allows $\mathcal{F}_{\mathsf{NVC}}$ to observe oracle queries.
  - (a) Sample $y \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$ and add $(x, y) \in \mathsf{Hist}_{\mathsf{GRO}}$.
  - (b) Send subroutine input $(\mathtt{Program}, x, y)$ to $\mathcal{F}_{\mathsf{GRO}}$.

Figure 17: The non-interactive zero knowledge functionality in the presence of $\mathcal{F}_{\mathsf{GRO}}$, for the initialization subroutine and for handling interactions between $\mathcal{F}_{\mathsf{NIZK}}$ and $\mathcal{F}_{\mathsf{GRO}}$.

The predicate $\Phi_{\mathcal{F}_{\mathsf{NIZK}}}$

- $\Phi_{\mathcal{F}_{\mathsf{NIZK}}}(\mathsf{sid}, \mathsf{ActiveAlg}, x, y) \to \{0,1\}$:
  1. Parse sid as the session ID of the calling party, x as a query point (sid', x') in $\mathrm{Dom}(\mathcal{F}_{\mathsf{GRO}})$, and y as a query answer in $\mathrm{Cod}(\mathcal{F}_{\mathsf{GRO}})$.
  2. If $\mathsf{sid} \neq \mathsf{sid}'$, output 0. # Disallow cross-domain programming.
  3. Else, output 1.

Figure 18: The predicate $\Phi_{\mathcal{F}_{\mathsf{NIZK}}}$, which checks $\mathcal{F}_{\mathsf{GRO}}$ programming requests from $\mathcal{F}_{\mathsf{NIZK}}$.

We further remark that adaptive corruption of the prover only make sense after a proof was generated, as this is the only time where the prover holds any secret internal state. Adaptive corruption of the verifier is meaningless since the verifier holds no no internal state.

section$\mathcal{F}_{\mathsf{NIZK}}$: Ideal functionality for non-interactive zero knowledge (NIZK) We give the ideal functionality for non-interactive zero-knowledge for a relation $\mathcal{R}$, which accesses the global random oracle sub-functionality $\mathcal{F}_{\mathsf{GRO}}$ in Figures 16 and 17. It is explained in Section 8.

9 Π_{NIZK}: UC Zero-Knowledge SNARK from PCP

We formulate the non-interactive Kilian-Micali argument in the UC model as Π_{NIZK} (Figure 19). Specifically, the protocol uses $\mathcal{F}_{\mathsf{NVC}}$ (Figure 8), $\mathcal{F}_{\mathsf{GRO}}$ (Figure 2), and an Adaptive Honest Verifier Zero Knowledge PCP (defined in Section 9.1). We show that Π_{NIZK} UC-realizes $\mathcal{F}_{\mathsf{NIZK}}$ (Theorem 9.1) (Figures 16 and 17); the proof appears in Section 9.3.

Theorem 9.1. *Let* $\mathsf{PCP} := (\mathbb{P}, (\mathbb{V}_1, \mathbb{V}_2))$ *be a PCP for a relation* $\mathcal{R}$ *with the following parameters:*

- Σ*: the alphabet of the PCP proof;*
- n*: PCP proof length;*
- s_{ext}*: extraction (i.e. knowledge soundness) error;*
- s_{zk}*: adaptive honest verifier zero-knowledge error.*

Let $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ *be an ideal global random oracle, with output length* $h_{\mathsf{cm}} \in \mathbb{N}$. *Let* $\mathcal{F}_{\mathsf{NVC}}$ *(Figure 8) be an ideal vector commitment in the presence of* $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ *and* $\mathcal{F}_{\mathsf{lib}}$ *with the following parameters:*

- Σ*: the plaintext alphabet;*
- n*: the length of the committed vector;*
- h_{cm}*: the length of commitment strings;*
- h_{dcm}*: the length of decommitment strings.*

Then, the resulting Π_{NIZK} *(Figure 19) UC-realizes* $\mathcal{F}_{\mathsf{NIZK}}(\mathcal{R})$ *(Figures 16 and 17) for relation* $\mathcal{R}$*, in the presence of global functionalities* $\mathcal{F}_{\mathsf{GRO}}(h_{\mathsf{cm}})$ *and* $\mathcal{F}_{\mathsf{lib}}$*, and with respect to adaptive corruptions. Specifically, an environment making* t *queries to* $\mathcal{F}_{\mathsf{GRO}}$ *has distinguishing advantage at most*

$$s_{\mathsf{zk}} + t \cdot 2^{-h_{\mathsf{dcm}}} + (1 + t + n \cdot 2t^2) \cdot 2^{-h_{\mathsf{cm}}} + t \cdot s_{\mathsf{ext}} \ .$$

Assuming that $\mathcal{F}_{\mathsf{NVC}}$ *has commitment size* p_{cm} *and decommitment size* p_{dcm} *per decommitted location,* Π_{NIZK} *has proof length* $2p_{\mathsf{cm}} + q \cdot (\log n + \log |\Sigma| + p_{\mathsf{dcm}})$.

9.1 Probabilistically-checkable proofs (PCP)

A probabilistically-checkable proof (PCP) for a relation $\mathcal{R}$ (or, equivalently, for the language $\mathcal{L}(\mathcal{R}) = \{\mathbb{x} : \exists \mathbb{w} \text{ s.t. } \mathcal{R}(\mathbb{x}, \mathbb{w}) = 1\}$) is a tuple $(\mathbb{P}, \mathbb{V})$, where $\mathbb{V}$ is probabilistic and has query-access to the proof generated by $\mathbb{P}$.

- $\mathbb{P}(\mathbb{x}, \mathbb{w}) \to \pi_{\mathsf{PCP}}$. On input an instance $\mathbb{x}$ and a corresponding witness $\mathbb{w}$, the prover $\mathbb{P}$ computes a proof $\pi_{\mathsf{PCP}} \in \Sigma^*$ that attests to the claim that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, where Σ is an alphabet of polynomial size.
- $\mathbb{V}^{\pi_{\mathsf{PCP}}}(\mathbb{x}) \to b$. On input an instance $\mathbb{x}$ and given query-access to individual Σ-symbols in a proof π_{PCP}, the verifier $\mathbb{V}$ computes a bit indicating whether π_{PCP} is a valid proof.

Alternatively, we can think of (a non-adaptive) $\mathbb{V}$ as being two algorithms $(\mathbb{V}_1, \mathbb{V}_2)$, such that:

- $\mathbb{V}_1(\mathbb{x}) \to i_1, \ldots, i_q$. $\mathbb{V}_1$ is a randomized algorithm that receives an instance $\mathbb{x}$ as input and outputs q indices $i_1, \ldots, i_q$. We sometimes write $\mathbb{V}_1(\mathbb{x}; r_{\mathbb{V}})$ to explicitly specify the internal randomness $r_{\mathbb{V}}$ that is used when running $\mathbb{V}_1$.
- $\mathbb{V}_2(\mathbb{x}, \pi_{\mathsf{PCP}}[i_1], \ldots, \pi_{\mathsf{PCP}}[i_q]) \to b$. On input an instance $\mathbb{x}$ and the Σ-symbols that correspond to the indices $i_1, \ldots, i_q$ in the proof π_{PCP}, algorithm $\mathbb{V}_2$ outputs a bit indicating whether π_{PCP} is a valid proof for the statement "there exists $\mathbb{w}$ such that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$".

We require PCP to satisfy the following completeness, extractability, and honest-verifier zero-knowledge properties.

- *Completeness.* For every $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$,

$$\Pr\left[\ \mathbb{V}^{\pi_{\mathsf{PCP}}}(\mathbb{x}) = 1 \ \middle|\ \pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w})\ \right] = 1 \ .$$

- *Extractability (a.k.a. Proof of Knowledge).* There exists a polytime algorithm $\mathbb{E}$ such that if $\mathbb{V}^{\pi_{\mathsf{PCP}}}(\mathbb{x}) = 1$ then $\mathbb{E}(\mathbb{x}, \pi_{\mathsf{PCP}})$ is a valid witness, except for negligible probability. More specifically, we say that PCP has extraction error s_{ext} if for every polytime dishonest prover $\tilde{\mathbb{P}}$,

$$\Pr\left[\begin{array}{c} \mathbb{V}^{\tilde{\pi}}(\mathbb{x}) = 1 \\ \wedge\ \mathcal{R}(\mathbb{x}, \mathbb{w}) = 0 \end{array} \ \middle|\ \begin{array}{c} (\mathbb{x}, \tilde{\pi}) \leftarrow \tilde{\mathbb{P}} \\ \mathbb{w} \leftarrow \mathbb{E}(\mathbb{x}, \tilde{\pi}) \end{array}\right] \leq s_{\mathsf{ext}} \ ,$$

where $\tilde{\pi}$ is interpreted as a vector in $(\Sigma \cup \bot)^n$. Note that extractability implies soundness: if the PCP has extraction error s_{ext} then the probability that $\mathbb{x} \notin \mathcal{L}$ and the verifier accepts is at most s_{ext}.

- *Honest Verifier Zero Knowledge.* There exists a polytime algorithm $\mathbb{S}$ that, given any $\mathbb{x} \in \mathcal{L}$, generates a distribution that is statistically-close to the view of the honest PCP verifier. More specifically, the PCP has zero knowledge error s_{zk} if, for any $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, the statistical distance between

$$\left[(i_1, \ldots, i_q, \pi_{\mathsf{PCP}}[i_1], \ldots, \pi_{\mathsf{PCP}}[i_q]) \ \middle|\ \begin{array}{r} i_1, \ldots, i_q \leftarrow \mathbb{V}_1(\mathbb{x}) \\ \pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}) \end{array}\right]$$
$$\text{and} \quad \mathbb{S}(\mathbb{x})$$

is at most s_{zk}.

To handle the case where the prover is unable to effectively erase data and can be adaptively corrupted after the proof was generated, we strengthen the zero knowledge requirement as follows. (Essentially, the simulator should be able to complete the simulated partial PCP to a full PCP, once the witness is known.)

- *Adaptive Honest Verifier Zero Knowledge.* The PCP has adaptive zero knowledge error s_{zk} if there exists a pair of polytime algorithms $\mathbb{S} = (\mathbb{S}_1, \mathbb{S}_2)$ such that the following holds for any $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$:
  - The statistical distance between

$$\left[(i_1, \ldots, i_q, \pi_{\mathsf{PCP}}[i_1], \ldots, \pi_{\mathsf{PCP}}[i_q]) \ \middle|\ \begin{array}{r} i_1, \ldots, i_q \leftarrow \mathbb{V}_1(\mathbb{x}) \\ \pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}) \end{array}\right]$$
$$\text{and} \quad [\tilde{\pi} \mid (\tilde{\pi}, \mathsf{State}_{\mathsf{PCP}}) \leftarrow \mathbb{S}_1(\mathbb{x})]$$

  is at most s_{zk}.

- Let $\tilde{\pi} = (\tilde{i}_1, \ldots, \tilde{i}_q, \sigma_1 \ldots \sigma_q) \leftarrow \mathbb{S}_1(\mathbb{x})$, let $((\tau_1 \ldots \tau_n), r_{\mathbb{P}}) \leftarrow \mathbb{S}_2(\mathsf{State}_{\mathsf{PCP}}, \mathbb{w})$, and define the reconstructed PCP as $\xi \in \Sigma^n$, in which n is the length of the PCP string, such that

$$\xi_j := \begin{cases} \sigma_j & \text{if } j = \tilde{i}_k \text{ for some } k \in [q] \\ \tau_j & \text{o.w.} \end{cases} .$$

Then, the statistical distance between the reconstructed PCP ξ and the honestly generated PCP proof $\pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}; r_{\mathbb{P}})$ is at most s_{zk}.

We have:

Theorem 9.2. *There exist adaptive honest verifier zero knowledge, extractable PCPs for all languages in NP, where: (a) both the extraction error and the zero knowledge error are negligible; (b) the prover $\mathbb{P}$ and verifier $\mathbb{V}$ are polytime; (c) $\mathbb{V}$ makes at most q queries to π_{PCP}.*

In particular, as noted in [IMS12], such PCPs are implicit in any Sigma protocol where the first prover message consists of a sequence of commitments, and the second prover message consists of openings of the commitments indicated by the verifier's challenge. ([IMS12] did not consider the adaptive version of honest verifier zero knowledge, but for many Sigma protocols, the extension is straightforward. For concrete examples see [CSW22].)

9.2 The protocol from Honest Verifier Zero Knowledge (HVZK) PCPs

We describe our UC formulation of the Kilian-Micali protocol Π_{NIZK} (Figure 19), which constructs a succinct argument given $\mathsf{PCP} = (\mathbb{P}, \mathbb{V} = (\mathbb{V}_1, \mathbb{V}_2))$, an ideal vector commitment $\mathcal{F}_{\mathsf{NVC}}$, and the global random oracle $\mathcal{F}_{\mathsf{GRO}}$. The key feature of this formulation is its simplicity: The prover uses an adaptive honest verifier zero knowledge PCP along with a single instance of $\mathcal{F}_{\mathsf{NVC}}$ and a single call to $\mathcal{F}_{\mathsf{GRO}}$ in order to obtain the Fiat-Shamir challenge. The verifier is similarly straightforward.

9.3 Proof of Theorem 9.1

We define the code library SCode in Section 9.3.1. The environment's distinguishing advantage is computed in Section 9.3.2. Efficiency measures are computed in Section 9.3.3.

9.3.1 The simulator's code library

We first overview how the simulator for $\mathcal{F}_{\mathsf{NIZK}}$ emulates an instance of $\mathcal{F}_{\mathsf{NVC}}$. After, we define the simulator's code library $\mathsf{SCode} = \{\mathsf{Setup}, \mathsf{SimProve}, \mathsf{SimState}, \mathsf{Extr}\}$.

Emulating an instance of $\mathcal{F}_{\mathsf{NVC}}$. Initially, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode}$ generates a vector commitment cm with $\mathcal{F}_{\mathsf{NVC}}$, given only a partial vector $\mathbf{m} \in (\Sigma \cup \bot)^n$, in order to simulate π_{NIZK}. Later upon adaptive corruption, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode}$ gets a completion of the vector $\mathbf{m}$ and must simulate knowing the outputs of $\mathcal{F}_{\mathsf{NVC}}$, as if it committed to the full vector initially. We describe our approach for handling this, which is a generalization of the approach used to emulate $\mathcal{F}_{\mathsf{NC}}$ in $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode}$, which solves the problem of generating cm before the committed message is known. (See Section 7.1.1.)

For initially simulating the vector commitment string cm in the proof π_{NIZK}, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode}.\mathsf{SimProve}$ invokes the **Commit** interface of an instance of $\mathcal{F}_{\mathsf{NVC}}$, where $\mathcal{F}_{\mathsf{NVC}}$ is emulated by "in-lining" its code. This is analogous to how the simulator for $\mathcal{F}_{\mathsf{NVC}}$ emulates instances of $\mathcal{F}_{\mathsf{NC}}$. In particular, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode}.\mathsf{SimProve}$

Protocol Π_{NIZK}

- **Prove:**
  1. Receive input $(\mathtt{Prove}, \mathbb{x}, \mathbb{w})$.
  2. Run $\pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w})$, where $\pi_{\mathsf{PCP}} \in \Sigma^n$ for some alphabet Σ.
  3. Commit to π_{PCP} using a vector commitment:
     (a) Set the session ID for vector commitment: $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$.[a] # $\mathsf{sid}_{\mathsf{NIZK}}$ is the local session ID.
     (b) Send subroutine input $(\mathtt{Commit}, \pi_{\mathsf{PCP}})$ to $\mathcal{F}_{\mathsf{NVC}}$ of session ID $\mathsf{sid}_{\mathsf{VC}}$.
     (c) Receive $(\mathtt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from this instance of $\mathcal{F}_{\mathsf{NVC}}$.
  4. Generate Fiat-Shamir challenge:
     (a) Sample random pad $\rho \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$. # For zero knowledge.
     (b) Send the subroutine input $(\mathtt{Eval}, (\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho))$ to $\mathcal{F}_{\mathsf{GRO}}$.
     (c) Receive the subroutine output $(\mathtt{Eval}, \beta)$ from $\mathcal{F}_{\mathsf{GRO}}$. # β is the Fiat-Shamir challenge.
  5. Derive the PCP verifier's challenge indices $I := \{i_1, \ldots, i_q\} \subseteq [|\pi_{\mathsf{PCP}}|]$ by computing $I := \mathbb{V}_1(\mathbb{x}; \beta)$.
  6. Set $\gamma := \{\ (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j]) \mid i_j \in I\ \}$.
  7. Output $(\mathtt{Prove}, \pi_{\mathsf{NIZK}} := (\mathsf{cm}, \rho, \gamma))$.
- **Verify:**
  1. Receive input $(\mathtt{Verify}, \mathsf{sid}, \mathbb{x}, \pi_{\mathsf{NIZK}})$.
  2. Parse π_{NIZK} as a tuple $(\mathsf{cm}, \rho, \gamma)$, in which cm is a vector commitment, ρ as a string in $\{0,1\}^{h_{\mathsf{cm}}}$, and $\gamma = \{\ (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j])\ \}$, i.e. is a set containing index-vector decommitment pairs.
  3. Derive the Fiat-Shamir challenge: $\beta := \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, (\mathsf{cm}, \rho))$, in which $\mathsf{sid}_{\mathsf{NIZK}}$ is the local session ID.
  4. Derive the PCP verifier's challenge indices $I := \{i_1, \ldots, i_q\}$ by computing $I := \mathbb{V}_1(\mathbb{x}; \beta)$.
  5. For all $i_j \in I$: # Check vector commitment by invoking $\mathcal{F}_{\mathsf{NVC}}$.
     (a) Send $(\mathtt{Verify}, \mathsf{cm}, i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j])$ to $\mathcal{F}_{\mathsf{NVC}}$ with session ID $\mathsf{sid}_{\mathsf{VC}} = (\mathsf{sid}_{\mathsf{NIZK}}, 1)$.
     (b) Receive $(\mathtt{VerStatus}, v_{i_j})$ from $\mathcal{F}_{\mathsf{NVC}}$.
  6. Check that:
     (a) For all $i_j \in I$, we have $v_{i_j} = 1$. # All decommitments are accepted.
     (b) The PCP verifier accepts, i.e. $\mathbb{V}_2(\mathbb{x}, \pi_{\mathsf{PCP}}[i_1], \ldots, \pi_{\mathsf{PCP}}[i_q]) = 1$.
     (c) Every index in I has an entry in γ, and γ does not contain additional entries.
  7. If all checks in Step 6 pass, output $(\mathtt{VerStatus}, 1)$. Else, output $(\mathtt{VerStatus}, 0)$.

[a] For $\mathsf{sid}_{\mathsf{VC}}$ to be a sub-session of $\mathsf{sid}_{\mathsf{NIZK}}$, $\mathsf{sid}_{\mathsf{VC}}$ must satisfy the requirement that $\mathsf{sid}_{\mathsf{VC}}$ has the form $(\mathsf{sid}_{\mathsf{NIZK}}, \cdot)$ and $\mathsf{sid}_{\mathsf{VC}} \neq \mathsf{sid}_{\mathsf{NIZK}}$.

Figure 19: The non-interactive zero-knowledge protocol from PCP, with global subroutine $\mathcal{F}_{\mathsf{GRO}}$.

invokes $\mathcal{F}_{\mathsf{NVC}}$ with a message $\mathbf{m}_1$ that contains values at the simulated positions in the PCP proof and $\perp$ everywhere else.

Then upon adaptive corruption, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode.SimState}$ learns the witness $\mathbb{w}$ and must equivocate positions in the vector committed by $\mathcal{F}_{\mathsf{NVC}}$, which did not appear in the simulated π_{NIZK}. To do so, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode.SimState}$ re-runs $\mathcal{F}_{\mathsf{NVC}}$ with input $(\mathtt{Commit}, \mathbf{m}_2 \in \Sigma^n)$ and the same random tapes as from proof generation.[18] Note that $\mathbf{m}_2$ does not have any entries with value $\perp$ and matches $\mathbf{m}_1$ at its non-$\perp$ locations. Denote this procedure $(\mathbf{dcm}, \mathbf{State}_{\mathsf{VC}}) \leftarrow \mathcal{F}_{\mathsf{NVC}}.\widetilde{\mathsf{Equiv}}(\mathsf{sid}, \mathsf{st}_{\mathsf{VC}}, \mathbf{m}_2)$, in which $\mathbf{State}_{\mathsf{VC}}$ is the committer state returned in case of adaptive corruption and $\mathsf{st}_{\mathsf{VC}}$ is the internal state of $\mathcal{F}_{\mathsf{NVC}}$ from the first run, including the random tapes.

Since the execution of **Commit** in $\mathcal{F}_{\mathsf{NVC}}$ and $\mathcal{F}_{\mathsf{NVC}}.\widetilde{\mathsf{Equiv}}$ use the same random tapes and that $\mathcal{F}_{\mathsf{NVC}}$ generates the decommitment for each position in $\mathbf{m}$ using independent random tapes, cm and the values of $\mathbf{dcm}$ associated with the simulated positions in the PCP proof do not change between executions.

Extraction $\mathcal{F}_{\mathsf{NVC}}$. The simulator also requires defining an extractor $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode.Extr}$ that outputs a witness $\mathbb{w}$, given an instance $\mathbb{x}$, a proof π_{NIZK}, and GRO history $\mathsf{Hist}_{\mathsf{GRO}}$. This involves recovering the vector committed by an instance of $\mathcal{F}_{\mathsf{NVC}}$.

For extracting the committed vector $\mathbf{m}$ from cm from an instance of $\mathcal{F}_{\mathsf{NVC}}$, $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode.Extr}$ runs the following procedure, denoted $\mathcal{F}_{\mathsf{NVC}}.\widetilde{\mathsf{Extr}}(\mathsf{sid}, \mathsf{cm}, \mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}_{\mathsf{VC}}])$:

1. Play the role of $\mathcal{F}_{\mathsf{NVC}}$ of session sid with input $(\mathtt{Verify}, \mathsf{cm}, i, m, \mathsf{dcm})$, in which i, m, dcm are arbitrary values and $\mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}]$ is the subset of $\mathsf{Hist}_{\mathsf{GRO}}$ associated with $\mathcal{F}_{\mathsf{NVC}}$.[19] Operationally, $\mathcal{F}_{\mathsf{NVC}}$ does this by in-lining the code for the **Verify** subroutine of $\mathcal{F}_{\mathsf{NVC}}$, which invokes $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$.[20]
2. To extract the committed vector $\mathbf{m}$:
   (a) If the execution of $\mathcal{F}_{\mathsf{NVC}}$ does not invoke $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$ and there exists an entry $(\mathsf{cm}, \cdot, \cdot, \cdot, \cdot, 1) \in \mathsf{CState}$, compute $\mathbf{m}$ (resp. $\mathbf{dcm}$) such that

$$\mathbf{m}[i], \mathbf{dcm}[i] := \begin{cases} m, \mathsf{dcm} & \text{if } (\mathsf{cm}, i, m, \mathsf{dcm}, \cdot, 1) \in \mathsf{CState} \\ \perp & \text{o.w.} \end{cases}$$

   Then, output $(\mathbf{m}', \mathbf{dcm}')$.
   (b) If the execution of $\mathcal{F}_{\mathsf{NVC}}$ invokes $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$ to get output $(\mathbf{m}', \mathbf{dcm}')$, output $(\mathbf{m}', \mathbf{dcm}')$.[21]
   (c) Else, output $(\mathbf{m}' := (\perp)^n, \mathbf{dcm}' := (\perp)^n)$.

SCode for $\mathcal{F}_{\mathsf{NIZK}}$. We define the $\mathcal{F}_{\mathsf{NIZK}}$ simulator code library $\mathsf{SCode} = \{\mathsf{Setup}, \mathsf{SimProve}, \mathsf{SimState}, \mathsf{Extr}\}$. Note that these algorithms are (non-adaptively) chosen by $\mathcal{E}$. Let PCP be a probabilistically checkable proof with adaptive honest verifier zero-knowledge simulator $\mathbb{S} = (\mathbb{S}_1, \mathbb{S}_2)$ and knowledge extractor $\mathbb{E}$, as defined in Section 9.1. Define the function $\Psi\colon \{0,1\}^\lambda \to [n]^q$ as mapping from the randomness of $\mathbb{V}_1$ to PCP query indices and is identical to $\mathbb{V}_1$.

```
• Setup($\mathsf{sid}_{\mathsf{NIZK}}$).
    1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID.
    2. Set $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$. # Distinct from $\mathsf{sid}_{\mathsf{NIZK}}$.
    3. Send $(\mathsf{Spawn}, \mathsf{sid}_{\mathsf{VC}})$ to Step VI of $\mathcal{F}_{\mathsf{NIZK}}$. # Spawn a gateway for session $\mathsf{sid}_{\mathsf{VC}}$.
```

[18] This is a generalization of the simulation strategy for instances of $\mathcal{F}_{\mathsf{NC}}$ emulated by $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode}$. (See Section 7.1.1.)

[19] That is, $\mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}_{\mathsf{VC}}] := \{(x, y) \mid (x, y) \in \mathsf{Hist}_{\mathsf{GRO}} \wedge x = (\mathsf{sid}, \cdot)\}$, in which $\mathsf{Hist}_{\mathsf{GRO}}$ is the GRO history of $\mathcal{F}_{\mathsf{NIZK}}$.

[20] The code linking is done by $\mathcal{F}_{\mathsf{lib}}$.

[21] Crucially, this works since $\mathcal{F}_{\mathsf{NVC}}.\mathsf{SCode.Extr}$ is defined to extract an *entire* committed vector.

- $\mathsf{SimProve}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, r) \to (\pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}})$.
  1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID, $\mathbb{x}$ as an instance for language $\mathcal{L}$, and $r = (r_{\mathsf{cm}} \in (\{0,1\}^{h_{\mathsf{cm}}})^*, r_{\mathsf{NIZK}} \in (\{0,1\}^{h_{\mathsf{cm}}})^*)$ as random tapes.
  2. Run the PCP zero-knowledge simulator: $((i_1, \dots, i_q, \pi_{\mathsf{PCP}}[i_1], \dots, \pi_{\mathsf{PCP}}[i_q]),\ \mathsf{State}_{\mathsf{PCP}}) \leftarrow \mathbb{S}_1(\mathbb{x})$.
  3. Set $I := \{i_1, \dots, i_q\}$. # Set of PCP verifier challenges.
  4. Compute $\beta := \Psi^{-1}(I)$. # Fiat-Shamir challenge.
  5. Simulate a vector commitment of π_{PCP}, restricted to I:
     (a) Generate a partial string $\Pi \in (\Sigma \cup \{\perp\})^n$, such that:
     $$\Pi[i] := \begin{cases} \pi_{\mathsf{PCP}}[i] & \text{if } i \in I \\ \perp & \text{o.w.} \end{cases} .$$
     (b) Set the session ID for vector commitment: $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$.
     (c) Emulate an execution of $\mathcal{F}_{\mathsf{NVC}}$ of session ID $\mathsf{sid}_{\mathsf{VC}}$ with input $(\mathtt{Commit}, \Pi)$.
     (d) Receive $(\mathtt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from this instance of $\mathcal{F}_{\mathsf{NVC}}$. # Only generates a decommitment for entries $i \in I$.
     (e) Record the internal state of $\mathcal{F}_{\mathsf{NVC}}$, including random tapes, in $\mathsf{st}_{\mathsf{VC}}$.
  6. Program the Fiat-Shamir challenge into $\mathcal{F}_{\mathsf{GRO}}$:
     (a) Compute $\rho := r_{\mathsf{NIZK}}[0] \in \{0,1\}^{h_{\mathsf{cm}}}$.
     (b) Set $x := (\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ and $y := \beta$.
     (c) Run $\mathsf{Program}(\mathsf{SubFuncs}, \mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}_{\mathsf{NIZK}}}, \mathsf{sid}_{\mathsf{NIZK}}, \mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode}.\mathsf{SimProve}, x, y)$.
     (d) Receive the output $(\mathtt{EvalOutput}, y)$.
        i. If $y \neq \beta$ (i.e. y was previously fixed as the output of x), output $\perp$.
        ii. Else, continue. # Programming successful.
  7. Denote $\gamma := \{\ (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j]) \mid i_j \in I\ \}$.
  8. Output $(\pi_{\mathsf{NIZK}} := \ (\mathsf{cm}, \rho, \gamma)\ , \mathsf{st}_{\mathsf{NIZK}} := (\mathsf{State}_{\mathsf{PCP}}, \mathsf{st}_{\mathsf{VC}}))$. # $\mathsf{st}_{\mathsf{NIZK}}$ is prover's internal state.

- $\mathsf{SimState}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{w}, \pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}}) \to \mathsf{State}_{\mathsf{NIZK}}$.
  1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID and $\mathbb{w}$ as a witness for language $\mathcal{L}$, proof string π_{NIZK} as $(\mathsf{cm}, \rho, \gamma)$ in which $\gamma := \{\ (i_j, \Pi[i_j], \mathbf{dcm}[i_j])\ \}$ and the prover's internal state $\mathsf{State}_{\mathsf{NIZK}}$ as $(\mathsf{State}_{\mathsf{PCP}}, \mathsf{st}_{\mathsf{VC}})$.
  2. Reconstruct PCP proof ξ:
     (a) Run $((\tau_1 \dots \tau_n), r_{\mathbb{P}}) \leftarrow \mathbb{S}_2(\mathsf{State}_{\mathsf{PCP}}, \mathbb{w})$. # $r_{\mathbb{P}}$ is simulated randomness for the PCP prover $\mathbb{P}$.
     (b) Denote the set $I := \{\ i_j \mid (i_j, \cdot, \cdot) \in \gamma\ \}$, which are the decommitted locations of π_{PCP}.
     (c) Construct $\xi \in \Sigma^n$, such that for all $j \in [n]$
     $$\xi_j := \begin{cases} \Pi[j] & \text{if } j \in I \\ \tau_j & \text{o.w.} \end{cases} .$$
  3. Generate vector decommitments for ξ by running $(\mathbf{dcm}, \mathbf{State}_{\mathsf{VC}}) \leftarrow \mathcal{F}_{\mathsf{NVC}}.\widetilde{\mathsf{Equiv}}(\mathsf{sid}_{\mathsf{VC}}, \mathsf{st}_{\mathsf{VC}}, \xi)$, where $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$.
  4. Output $\mathsf{State}_{\mathsf{NIZK}} := (r_{\mathbb{P}}, \xi, \mathbf{dcm}[[n] \setminus I], \mathbf{State}_{\mathsf{VC}})$.

- $\mathsf{Extr}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \pi_{\mathsf{NIZK}}, \mathsf{Hist}_{\mathsf{GRO}}) \to \mathbb{w}$.
  1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID, $\mathbb{x}$ as an instance for language $\mathcal{L}$, π_{NIZK} as $(\mathsf{cm}, \rho, \gamma)$ in which $\gamma := \{\, (i_j, \Pi[i_j], \mathbf{dcm}[i_j]) \,\}$, and $\mathsf{Hist}_{\mathsf{GRO}}$ as a list of query-answer pairs for GRO.
  2. Run the **Verify** code of Π_{NIZK} (Figure 19) with session ID $\mathsf{sid}_{\mathsf{NIZK}}$, on input $(\mathbb{x}, \pi_{\mathsf{NIZK}})$, and using $\mathsf{Hist}_{\mathsf{GRO}}$ to emulate the responses of $\mathcal{F}_{\mathsf{GRO}}$.
     (a) If the verifier rejects (or any of the queries made by **Verify** code of Π_{NIZK} to $\mathcal{F}_{\mathsf{GRO}}$ do not appear in $\mathsf{Hist}_{\mathsf{GRO}}$[a]), then output $\bot$.
  3. Run $(\tilde{\pi}, \cdot) \leftarrow \mathcal{F}_{\mathsf{NVC}}.\mathsf{Extr}(\mathsf{sid}_{\mathsf{VC}}, \mathsf{cm}, \mathsf{Hist}_{\mathsf{GRO}}[\mathsf{sid}_{\mathsf{VC}}])$.
  4. Run the PCP extractor: $\mathbb{w} \leftarrow \mathbb{E}(\mathbb{x}, \tilde{\pi})$. # PCP verifier already accepted in Step 2.
  5. Output $\mathbb{w}$.

[a] Explicitly, to check that the Fiat-Shamir challenge is consistent with $\mathcal{F}_{\mathsf{GRO}}$ (i.e. Step 3 of the **Verify** code of Π_{NIZK}):
i. Compute the Fiat-Shamir challenge value $\beta := \Psi^{-1}(I)$, in which $I := \{\, i_j \mid (i_j, \cdot, \cdot) \in \gamma \,\}$ are the decommitted locations of π_{PCP} and Ψ^{-1} is the inverse of $\Psi \colon \{0,1\}^{\lambda} \to [n]^q$, which is defined as identical to $\mathbb{V}_1(\mathbb{x}, \cdot)$.
ii. Check that the query-answer pair $((\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho), \mathsf{cm})$ is in $\mathsf{Hist}_{\mathsf{GRO}}$ of $\mathcal{F}_{\mathsf{NIZK}}$.

9.3.2 Validity of the simulation

Let $\mathcal{E}$ be an environment machine, let $\mathsf{REAL} := \mathsf{Exec}_{\mathcal{E},\Pi_{\mathsf{NIZK}}}$ and let $\mathsf{IDEAL} := \mathsf{Exec}_{\mathcal{E},\mathcal{F}_{\mathsf{NIZK}},\mathsf{SCode}}$. Let t be the maximum number of queries that $\mathcal{E}$ makes to $\mathcal{F}_{\mathsf{GRO}}$.

Let s_{NIZK} denote the statistical distance between REAL and IDEAL. We argue that

$$s_{\mathsf{NIZK}} \le s_{\mathsf{zk}} + t \cdot 2^{-h_{\mathsf{dcm}}} + (1 + t + n \cdot 2t^2) \cdot 2^{-h_{\mathsf{cm}}} + t \cdot s_{\mathsf{ext}} \enspace .$$

Consider the following sequence of hybrids. Informally, indistinguishability of IDEAL from HYBRID_3 covers the zero-knowledge aspect, whereas indistinguishability of HYBRID_3 and REAL covers extractability, which in turn provides both soundness and non-malleability.

- HYBRID_1 is identical to IDEAL, except that $\mathsf{SimProve}$ uses the real witness and real PCP prover to generate the PCP string. (Still, $\mathcal{F}_{\mathsf{NVC}}$ is run only for a partial PCP, which are defined at locations that are opened later in the proof.) Indistinguishability follows from the validity of the PCP ZK simulator.
- HYBRID_2 is identical to HYBRID_1, except that $\mathsf{SimProve}$ invokes $\mathcal{F}_{\mathsf{NVC}}$ with the full PCP. Indistinguishability holds based on the hiding property of $\mathcal{F}_{\mathsf{NVC}}$.
- HYBRID_3 is identical to HYBRID_2, except that $\mathcal{F}_{\mathsf{GRO}}$ is not programmed at the challenge point $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$. Instead, the Fiat-Shamir challenge is set to whatever $\mathcal{F}_{\mathsf{GRO}}$ returns. Indistinguishability follows as long as the point $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ was never queried by $\mathcal{E}$ before.
- HYBRID_3 is identical to REAL, except that in REAL the verifier runs the verifier code of Π_{NIZK} as opposed to the ideal verification. Indistinguishability follows from the binding property of $\mathcal{F}_{\mathsf{NVC}}$ and the extractability of the PCP.

Next, we write the hybrids in detail. (Below, blue indicates the difference from the previous hybrid.)

- IDEAL.

- HYBRID_1: Switch from generating the PCP proof π_{PCP} and challenge indices I using the PCP simulator $\mathbb{S}$, to generating π_{PCP} using the PCP prover $\mathbb{P}(\mathbb{x}, \mathbb{w}; r_{\mathbb{P}})$, then sampling $I \leftarrow \mathbb{V}_1(\mathbb{x}; \beta)$, in which $\beta \leftarrow \{0,1\}^{\lambda}$. (Still, only the elements in I are committed to).

- **Prove:** Same as in IDEAL, except that $\mathsf{SimProve}$ has the following code.
  $\mathsf{SimProve}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \mathbb{w}, r) \to (\pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}})$.
  1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID, $\mathbb{x}$ as an instance for language $\mathcal{L}$, $\mathbb{w}$ as a witness, and r as a random tape.
  2. Run $\pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}; r_{\mathbb{P}})$.
  3. Sample $\beta \leftarrow \{0,1\}^{\lambda}$.
  4. Set $I = \{i_1, \ldots, i_q\} := \mathbb{V}_1(\mathbb{x}; \beta)$.
  5. Simulate a vector commitment of π_{PCP}, restricted to I:
     (a) Generate a partial string $\Pi \in (\Sigma \cup \{\bot\})^n$, such that:
     $$\Pi[i] := \begin{cases} \pi_{\mathsf{PCP}}[i] & \text{if } i \in I \\ \bot & \text{o.w.} \end{cases} .$$
     (b) Set the session ID for vector commitment: $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$. # $\mathsf{sid}_{\mathsf{NIZK}}$ is the local session ID.
     (c) Emulate an execution of $\mathcal{F}_{\mathsf{NVC}}$ of session ID $\mathsf{sid}_{\mathsf{VC}}$ with input $(\mathtt{Commit}, \Pi)$.
     (d) Receive $(\mathtt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from this instance of $\mathcal{F}_{\mathsf{NVC}}$. # Only generates a decommitment for entries $i \in I$.
     (e) Record the internal state of $\mathcal{F}_{\mathsf{NVC}}$, including random tapes, in $\mathsf{st}_{\mathsf{VC}}$.
  6. Program the Fiat-Shamir challenge into $\mathcal{F}_{\mathsf{GRO}}$:
     (a) Compute $\rho := r_{\mathsf{NIZK}}[0] \in \{0,1\}^{h_{\mathsf{cm}}}$.
     (b) Set $x := (\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ and $y := \beta$.
     (c) Run $\mathsf{Program}(\mathsf{SubFuncs}, \mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}_{\mathsf{NIZK}}}, \mathsf{sid}_{\mathsf{NIZK}}, \mathsf{SimProve}, x, y)$.
     (d) Receive the output $(\mathtt{EvalOutput}, y)$.
        i. If $y \neq \beta$ (i.e. y was previously fixed as the output of x), output $\bot$.
        ii. Else, continue. # Programming successful.
  7. Denote $\gamma := \{\, (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j]) \mid i_j \in I \,\}$.
  8. Output $(\pi_{\mathsf{NIZK}} := (\mathsf{cm}, \rho, \gamma), \mathsf{st}_{\mathsf{NIZK}} := (r_{\mathbb{P}}, \pi_{\mathsf{PCP}}, \mathsf{st}_{\mathsf{VC}}))$. # $\mathsf{st}_{\mathsf{NIZK}}$ is prover's internal state.
- **(Adaptive) Prover Corruption:** Same as in IDEAL, except $\mathsf{SimState}$ runs using the actual PCP proof π_{PCP}, instead of simulating it.
- **Verify:** Same as in IDEAL.

- HYBRID_2: Generate cm by sending the entire π_{PCP} to $\mathcal{F}_{\mathsf{NVC}}$. In other words, rather than running $\mathcal{F}_{\mathsf{NVC}}$ with a partial vector, run it for the entire π_{PCP}.

- **Prove:** Same as in IDEAL, except that $\mathsf{SimProve}$ has the following code.
  $\mathsf{SimProve}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \mathbb{w}, r) \to (\pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}})$.
  1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID, $\mathbb{x}$ as an instance for language $\mathcal{L}$, $\mathbb{w}$ as a witness, and r as a random tape.
  2. Run $\pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}; r_{\mathbb{P}})$.
  3. Sample $\beta \leftarrow \{0,1\}^{\lambda}$.
  4. Set $I = \{i_1, \ldots, i_q\} := \mathbb{V}_1(\mathbb{x}; \beta)$.
  5. Commit to (the entire) π_{PCP} using a vector commitment:
     (a) Set the session ID for vector commitment: $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$. # $\mathsf{sid}_{\mathsf{NIZK}}$ is the local session ID.
     (b) Emulate an execution of $\mathcal{F}_{\mathsf{NVC}}$ of session ID $\mathsf{sid}_{\mathsf{VC}}$ with input $(\mathtt{Commit}, \pi_{\mathsf{PCP}})$.
     (c) Receive $(\mathtt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from this instance of $\mathcal{F}_{\mathsf{NVC}}$.
     (d) Record the internal state of $\mathcal{F}_{\mathsf{NVC}}$, including random tapes, in $\mathsf{st}_{\mathsf{VC}}$.
  6. Program the Fiat-Shamir challenge into $\mathcal{F}_{\mathsf{GRO}}$:
     (a) Compute $\rho := r_{\mathsf{NIZK}}[0] \in \{0,1\}^{h_{\mathsf{cm}}}$.
     (b) Set $x := (\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ and $y := \beta$.
     (c) Run $\mathsf{Program}(\mathsf{SubFuncs}, \mathsf{Hist}_{\mathsf{GRO}}, \Phi_{\mathcal{F}_{\mathsf{NIZK}}}, \mathsf{sid}_{\mathsf{NIZK}}, \mathsf{SimProve}, x, y)$.
     (d) Receive the output $(\mathtt{EvalOutput}, y)$.
        i. If $y \neq \beta$ (i.e. y was previously fixed as the output of x), output $\bot$.
        ii. Else, continue. # Programming successful.
  7. Set $\gamma := \{\, (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j]) \mid i_j \in I \,\}$.
  8. Output $(\pi_{\mathsf{NIZK}} := (\mathsf{cm}, \rho, \gamma), \mathsf{st}_{\mathsf{NIZK}} := (r_{\mathbb{P}}, \pi_{\mathsf{PCP}}, \mathsf{st}_{\mathsf{VC}}))$. # $\mathsf{st}_{\mathsf{NIZK}}$ is prover's internal state.
- **(Adaptive) Prover Corruption:** Same as in IDEAL, except $\mathsf{SimState}$ runs using the actual PCP proof π_{PCP}, instead of simulating it.
- **Verify:** Same as in IDEAL.

- HYBRID_3: Rather than sampling $\beta \leftarrow \{0,1\}^{\lambda}$, compute $\beta := \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$, i.e. via querying $\mathcal{F}_{\mathsf{GRO}}$.

- **Prove:** Same as in IDEAL, except that $\mathsf{SimProve}$ has the following code.
  $\mathsf{SimProve}(\mathsf{sid}_{\mathsf{NIZK}}, \mathbb{x}, \mathbb{w}, r) \to (\pi_{\mathsf{NIZK}}, \mathsf{st}_{\mathsf{NIZK}})$.
  1. Parse $\mathsf{sid}_{\mathsf{NIZK}}$ as a session ID, $\mathbb{x}$ as an instance for language $\mathcal{L}$, $\mathbb{w}$ as a witness, and r as a random tape.
  2. Run $\pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}; r_{\mathbb{P}})$.
  3. Commit to (the entire) π_{PCP} using a vector commitment:
     (a) Set the session ID for vector commitment: $\mathsf{sid}_{\mathsf{VC}} := (\mathsf{sid}_{\mathsf{NIZK}}, 1)$. # $\mathsf{sid}_{\mathsf{NIZK}}$ is the local session ID.
     (b) Emulate an execution of $\mathcal{F}_{\mathsf{NVC}}$ of session ID $\mathsf{sid}_{\mathsf{VC}}$ with input $(\mathtt{Commit}, \pi_{\mathsf{PCP}})$.
     (c) Receive $(\mathtt{Receipt}, \mathsf{cm}, \mathbf{dcm})$ from this instance of $\mathcal{F}_{\mathsf{NVC}}$.
     (d) Record the internal state of $\mathcal{F}_{\mathsf{NVC}}$, including random tapes, in $\mathsf{st}_{\mathsf{VC}}$.
  4. Generate Fiat-Shamir challenge:
     (a) Sample random pad $\rho \leftarrow \{0,1\}^{h_{\mathsf{cm}}}$. # For zero knowledge.
     (b) Send the subroutine input $(\mathtt{Eval}, (\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho))$ to $\mathcal{F}_{\mathsf{GRO}}$.
     (c) Receive the subroutine output $(\mathtt{Eval}, \beta)$ from $\mathcal{F}_{\mathsf{GRO}}$. # β is the Fiat-Shamir challenge.
  5. Derive the PCP verifier's challenge indices $I := \{i_1, \ldots, i_q\} \subseteq [|\pi_{\mathsf{PCP}}|]$ by computing $I := \mathbb{V}_1(\mathbb{x}; \beta)$.
  6. Set $\gamma := \{\, (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j]) \mid i_j \in I \,\}$.
  7. Output $(\pi_{\mathsf{NIZK}} := (\mathsf{cm}, \rho, \gamma), \mathsf{st}_{\mathsf{NIZK}} := (r_{\mathbb{P}}, \pi_{\mathsf{PCP}}, \mathsf{st}_{\mathsf{VC}}))$. # $\mathsf{st}_{\mathsf{NIZK}}$ is prover's internal state.
- **(Adaptive) Prover Corruption:** Same as in IDEAL, except $\mathsf{SimState}$ runs using the actual PCP proof π_{PCP}, instead of simulating it.
- **Verify:** Same as in IDEAL.

- REAL: Switch to the **Verify** of REAL.

  - **Prove:** Same as in HYBRID_3.
  - **(Adaptive) Prover Corruption:** Same as in REAL.
  - **Verify:** Same as in REAL.

Next, we argue the environment's distinguishing advantage between the hybrids. For each pair of hybrid, we argue that the executions are identical, except for when a bad event occurs.

- IDEAL and HYBRID_1 have statistical distance at most s_{zk} by the zero-knowledge property of PCP (see Section 9.1). Note that from the perspective of $\mathcal{F}_{\mathsf{NVC}}$, even in the event of an adaptive corruption (i.e. $\mathcal{E}$ sends $(\mathtt{Corrupt})$ to $\mathcal{F}_{\mathsf{NIZK}}$ after π_{NIZK} is generated), IDEAL and HYBRID_1 are identical. In both, initially in the **Prove** subroutine, only the locations in I are equivocated. Then, after corruption, all remaining entries are corrupted. We note that the distributions of β (resp. I) are identical since it is randomly sampled from $\{0,1\}^\lambda$ in both hybrids.

- HYBRID_1 and HYBRID_2 are identical, except for whether the PCP proof π_{PCP} input into $\mathcal{F}_{\mathsf{NVC}}$ is a partial (i.e. only contains values at the locations in I) or is π_{PCP} itself.

  We first identify a bad event, which allows $\mathcal{E}$ to distinguish HYBRID_1 and HYBRID_2, then compute the probability of the bad event.

  Bad event. For an index $j \in [n] \setminus I$, $\mathcal{E}$ queries the location in $\mathcal{F}_{\mathsf{GRO}}$ that is programmed by $\mathcal{F}_{\mathsf{NVC}}$ in HYBRID_2 when equivocating location j. In comparison, in HYBRID_1, no decommitment value is generated (see Step II(d)i of $\mathcal{F}_{\mathsf{NVC}}$) for indices in $[n] \setminus I$, so the corresponding location in $\mathcal{F}_{\mathsf{GRO}}$ were not queried or programmed.

We argue that the probability of the bad event is $t \cdot 2^{-h_{\mathsf{dcm}}}$, via showing that the game of distinguishing HYBRID_2 and HYBRID_1 is precisely that of distinguishing Interaction 1 and Interaction 2 of Game 6.2, i.e. the hiding game for ideal vector commitments.

Since $\mathcal{F}_{\mathsf{NVC}}$ is an ideal vector commitment mechanism (Definition 6.1), the plaintext vector in both hybrids is $\pi_{\mathsf{PCP}} \leftarrow \mathbb{P}(\mathbb{x}, \mathbb{w}; r_{\mathbb{P}})$, and that the set of locations for which $\mathcal{E}$ doesn't see the decommitment is $\overline{I} := [n] \setminus I$, we have:

- HYBRID_1 is Interaction 2 of Game 6.2.

  In HYBRID_1, $\mathsf{SimProve}$ invokes $\mathcal{F}_{\mathsf{NVC}}$ with $(\mathtt{Commit}, \Pi)$, where Π is a partial vector that matches π_{PCP} at locations I and has value $\perp$ at locations $\overline{I}$. Ultimately, $\mathsf{SimProve}$ outputs the NIZK proof π_{NIZK} containing $(\mathsf{cm}, \mathbf{dcm}[I])$, which is given to $\mathcal{E}$.
- HYBRID_2 is Interaction 1 of Game 6.2.

  In HYBRID_2, $\mathsf{SimProve}$ invokes $\mathcal{F}_{\mathsf{NVC}}$ with $(\mathtt{Commit}, \pi_{\mathsf{PCP}})$, where $\pi_{\mathsf{PCP}} \in \Sigma^n$, to get cm and $\mathbf{dcm}$; this is analogous to Interaction 1. Ultimately, $\mathsf{SimProve}$ outputs the NIZK proof π_{NIZK} containing $(\mathsf{cm}, \mathbf{dcm}[I])$, which is given to $\mathcal{E}$.

By Theorem 6.3, since $\mathcal{E}$ makes at most t queries to $\mathcal{F}_{\mathsf{GRO}}$, $\mathcal{E}$ distinguishes between the hybrids with probability $t \cdot 2^{-h_{\mathsf{dcm}}}$.

- HYBRID_2 and HYBRID_3 are identical, except for the sampling procedure of β. Specifically, HYBRID_2 samples β uniformly at random from $\{0,1\}^{h_{\mathsf{cm}}}$, then later programs $\mathcal{F}_{\mathsf{GRO}}$ so that β is the evaluation of $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$. In contrast, HYBRID_3 computes $\beta := \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$.

  First, we identify a bad event in which the executions of HYBRID_2 and HYBRID_3 diverge, and compute its probability.

  Bad event. $\mathcal{E}$ makes at most t queries of the form $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ to $\mathcal{F}_{\mathsf{GRO}}$ before invoking **Prove** in $\mathcal{F}_{\mathsf{NIZK}}$; denote this set of queries BAD. In HYBRID_2, Step 6 of $\mathsf{SimProve}$ cannot program the query-answer pair $(x = (\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho), \beta)$ in $\mathcal{F}_{\mathsf{GRO}}$ if $x \in \mathsf{BAD}$, so Step 6 of $\mathsf{SimProve}$ outputs $\perp$. In HYBRID_3, this failure never occurs.[22]

  We argue that the bad event occurs with probability $t \cdot 2^{-h_{\mathsf{cm}}}$. In HYBRID_2, Step 6a of $\mathsf{SimProve}$ samples a particular ρ with probability at most $2^{-h_{\mathsf{cm}}}$. (This is ensured since $\mathcal{F}_{\mathsf{NIZK}}$ checks that ρ has min-entropy h_{cm} in Step II(d)i.) So, the query $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho) \in \mathsf{BAD}$ with probability at most $t \cdot 2^{-h_{\mathsf{cm}}}$, since $|\mathsf{BAD}| = t$.

  Note that this difference does not cause a divergence in the execution of **Verify** because neither hybrid records the query $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ in $\mathsf{Hist}_{\mathsf{GRO}}$ — in HYBRID_2 and HYBRID_3, points programmed by $\mathsf{SCode.SimProve}$ are not appended to $\mathsf{Hist}_{\mathsf{GRO}}$, since no such instructions exists in the $\mathsf{Program}$ subroutine (Figure 12) of $\mathcal{F}_{\mathsf{NIZK}}$.

  Second, we argue HYBRID_2 and HYBRID_3 are identical given that the bad event does not occur, i.e. that ρ is sampled such that such that $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho) \notin \mathsf{BAD}$. (Note that ρ is sampled from identical distributions in Step 4a of HYBRID_3 and in Step 6a of HYBRID_2.) In particular, the only query to $\mathcal{F}_{\mathsf{GRO}}$ with prefix $\mathsf{sid}_{\mathsf{NIZK}}$ is for programming β (Step 6 of $\mathsf{SimProve}$); all other algorithms (including any from

[22] It's possible to define the bad event to exclude the case that $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho) \in \mathsf{BAD}$ but $\beta = \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$, i.e. the prover "gets lucky", in that $\mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho) = \beta$ already, without programming. For simplicity of analysis, we ignore this.

$\mathcal{F}_{\mathsf{NVC}}$) make queries to $\mathcal{F}_{\mathsf{GRO}}$ using a different session ID. (The restriction that prevents simulator code from programming $\mathcal{F}_{\mathsf{GRO}}$ at points outside their own SID is encoded in Step 2 of $\Phi_{\mathcal{F}_{\mathsf{NIZK}}}$ (Figure 18).) This implies the query $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ was not previously made to $\mathcal{F}_{\mathsf{GRO}}$, so the answer can be programmed as β.

- HYBRID_3 and REAL are identical, with the exception that in REAL the verifier runs the verifier code of Π_{NIZK} as opposed to the ideal verification. Indistinguishability follows from the binding property of $\mathcal{F}_{\mathsf{NVC}}$ and the extractability of the PCP. (We note that the NIZK proof π_{NIZK} is not necessarily generated honestly.)

Bad event. **Verify** accepts a proof $(\mathbb{x}, \pi_{\mathsf{NIZK}})$ in REAL, and yet the value $\tilde{\mathbb{w}} = \mathbb{E}(\mathbb{x}, \tilde{\pi})$ that the PCP extractor $\mathbb{E}$ generated, given the value $\tilde{\pi}$ generated by $\mathcal{F}_{\mathsf{NVC}}.\widetilde{\mathsf{Extr}}$ in this verification, fails to satisfy the corresponding NP relation, namely $\mathcal{R}(\mathbb{x}, \tilde{\mathbb{w}}) = 0$.

First we argue that, as long as the bad event did not occur, the view of $\mathcal{E}$ in REAL is the same as its view in HYBRID_3.

Indeed, HYBRID_3 and REAL are identical except for potential discrepancy in the responses to verify queries. Furthermore, in HYBRID_3, whenever **Verify** receives input $(\mathtt{Verify}, \mathsf{sid}, \mathbb{x}, \pi_{\mathsf{NIZK}} = (\mathsf{cm}, \rho, \gamma))$ there are two options:

1. $(\mathbb{x}, \cdot, \pi_{\mathsf{NIZK}}, v) \in \mathsf{PState}$. This means that $\mathcal{F}_{\mathsf{NIZK}}$ (resp. Π_{NIZK}) previously determined whether π_{NIZK} is a valid proof that $\mathbb{x} \in \mathcal{L}$. In this case the response in HYBRID_3 is the same as in REAL.
2. There is no entry $(\overline{\mathbb{x}} = \mathbb{x}, \cdot, \overline{\pi_{\mathsf{NIZK}}} = \pi_{\mathsf{NIZK}}, \cdot)$ in PState. Then, the extractor $\mathsf{SCode.Extr}$ is run (Step IV(c)i of $\mathcal{F}_{\mathsf{NIZK}}$). As long as the bad event does not occur, namely as long as $\mathcal{R}(\mathbb{x}, \tilde{\mathbb{w}}) = 1$, the response in HYBRID_3 is the same as in REAL.

Second, we argue that the probability of the bad event is $(1 + n \cdot 2t^2) \cdot 2^{-h_{\mathsf{cm}}} + t \cdot s_{\mathsf{ext}}$.

Consider the case where the verifier accepts in REAL but rejects in HYBRID_3. Without loss of generality, we restrict attention to the case in which the proof $\pi_{\mathsf{NIZK}} = (\mathsf{cm}, \rho, \gamma)$ is submitted to **Verify** for the first time. On one hand, since HYBRID_3 rejects, this implies $\mathcal{F}_{\mathsf{NIZK}}.\mathsf{SCode.Extr}$ fails to extract a witness. On the other hand, since REAL accepts, all decommitments of $\mathcal{F}_{\mathsf{NVC}}$ were successful (plus the committed messages does not contain any $\perp$ entries), and the PCP verifier accepts. There are two main cases in which the bad event may occur. For simplicity the description corresponds to HYBRID_3:

Case 1. $\mathcal{E}$ never makes the query $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ to $\mathcal{F}_{\mathsf{GRO}}$, in which cm is from π_{NIZK}.
Note that this implies there does not exist an entry $((\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho), \cdot) \in \mathsf{Hist}_{\mathsf{GRO}}$, so in HYBRID_3, Step 2 of $\mathsf{SCode.Extr}$ (specifically, Step 3 of **Verify** in Π_{NIZK}) fails. In REAL, Step 6c accepts and all other checks pass, by assumption.

Case 2. $\mathcal{E}$ makes the query $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ to $\mathcal{F}_{\mathsf{GRO}}$, prior to generating cm by committing to the PCP string. The consequent attack involves the environment "guessing" the Fiat-Shamir challenge query x, then querying the GRO at x. This causes the zero-knowledge simulator to fail because it cannot program the GRO at x.

We compute the probability of the above cases:

– $\Pr[\text{ Case 1 }]$:
Without querying $\mathcal{F}_{\mathsf{GRO}}$, $\mathcal{E}$ guesses (cm, ρ) such that $I = \mathbb{V}_1(\mathbb{x}; \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho))$. This is equivalent to $\mathcal{E}$ guessing (cm, ρ) such that $\Psi^{-1}(I) = \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$. Since the query $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}, \rho)$ was not previously made to $\mathcal{F}_{\mathsf{GRO}}$, this case occurs with probability $2^{-h_{\mathsf{cm}}}$.

- $\Pr[\,\text{Case 2}\,]$:

  Suppose $\mathcal{E}$ makes j queries (such that $j \leq t$) to $\mathcal{F}_{\mathsf{GRO}}$ of the form $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}_i, \rho_i)$ to get $\beta_i = \mathsf{GRO}(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}_i, \rho_i)$. Then, $\mathcal{E}$ attempts to complete π_{NIZK}, so that **Verify** accepts in REAL but outputs $\perp$ or rejects in HYBRID_3 (i.e. IDEAL).

  For attempt $i \in [j]$, in which $\mathcal{E}$ utilizes $(\mathsf{sid}_{\mathsf{NIZK}}, \mathsf{cm}_i, \rho_i)$, we argue that the bad event occurs with probability $n \cdot 2t_i^2 \cdot 2^{-h_{\mathsf{cm}}} + s_{\mathsf{ext}}$. $\mathcal{E}$ can attack the the binding property of cm_i or soundness of the PCP proof string, denoted Π_i, that is committed to by cm_i.

  * Suppose $\mathcal{E}$ allocates t_i queries to attacking the binding property of cm_i. (Note that $\sum_{i \in [j]} t_i \leq t$.) By the binding property of $\mathcal{F}_{\mathsf{NVC}}$ (Theorem 4.5), there is a unique vector Π_i that is bound to the commitment string cm_i, except with probability $n \cdot 2t_i^2 \cdot 2^{-h_{\mathsf{cm}}}$.
  * Suppose $\mathcal{E}$ attempts to attack the soundness of the PCP proof string Π_i that is committed to by cm_i. Given Π_i, the PCP witness extractor outputs a unique witness $\mathbb{w}_i \leftarrow \mathbb{E}(\mathbb{x}, \Pi_i)$. Also, β_i is uniformly random, since it is the output of GRO, and no party can program $\mathcal{F}_{\mathsf{GRO}}$ in $\mathsf{sid}_{\mathsf{NIZK}}$ aside from $\mathcal{F}_{\mathsf{NIZK}}$. Then, by the extractability property of PCP, the probability that the PCP verifier $\mathbb{V}^{\tilde{\pi}}(\mathbb{x}; \beta_i)$ accepts (so **Verify** accepts in REAL) and the extracted witness $\mathbb{w}_i$ satisfies $\mathcal{R}(\mathbb{x}, \mathbb{w}_i) = 0$ (so **Verify** rejects in Step IV(c)iii of $\mathcal{F}_{\mathsf{NIZK}}$) is at most s_{ext}.

  This means that for attempt i, the bad event occurs with probability $n \cdot 2t_i^2 \cdot 2^{-h_{\mathsf{cm}}} + s_{\mathsf{ext}}$.

  By union bound over the attempts ($i \in [j]$), we get that the probability of that **Verify** accepts in REAL but rejects in HYBRID_3 is:

  $$\sum_{i \in [j]} (n \cdot 2t_i^2 \cdot 2^{-h_{\mathsf{cm}}} + s_{\mathsf{ext}}) \leq n \cdot 2t^2 \cdot 2^{-h_{\mathsf{cm}}} + t \cdot s_{\mathsf{ext}} \ ,$$

  since $\sum_{i \in [j]} t_i^2 \leq t^2$ and $j \leq t$.

In summary, the bad events which allow $\mathcal{E}$ to distinguish HYBRID_3 and REAL occur with probability $\Pr[\text{Case 1}] + \Pr[\text{Case 2}] \leq (1 + n \cdot 2t^2) \cdot 2^{-h_{\mathsf{cm}}} + t \cdot s_{\mathsf{ext}}$.

To sum up, by the triangle inequality we satisfy the claim that

$$s_{\mathsf{NIZK}} \leq s_{\mathsf{zk}} + t \cdot 2^{-h_{\mathsf{dcm}}} + (1 + t + n \cdot 2t^2) \cdot 2^{-h_{\mathsf{cm}}} + t \cdot s_{\mathsf{ext}} \ .$$

9.3.3 Efficiency measures

Π_{NIZK} outputs proofs of the form $\pi_{\mathsf{NIZK}} = (\mathsf{cm}, \rho, \gamma = \{\, (i_j, \pi_{\mathsf{PCP}}[i_j], \mathbf{dcm}[i_j]) \mid i_j \in I \,\})$.

For $|\gamma|$, by PCP parameters, $\mathbb{V}$ makes q queries to π_{PCP}, so $|I| = q$. Since $|\pi_{\mathsf{PCP}}| = n$, we have $|i_j| = \log n$. Also π_{PCP} is over the alphabet Σ, so $|\pi_{\mathsf{PCP}}[i_j]| = \log |\Sigma|$. Also recall that $|\mathsf{cm}| = |\rho|$. Assuming $\mathcal{F}_{\mathsf{NVC}}$ has commitment size $|\mathsf{cm}| = p_{\mathsf{dcm}}$ and decommitment size p_{dcm} per decommitted location, we get

$$|\pi_{\mathsf{NIZK}}| = 2p_{\mathsf{cm}} + q \cdot (\log n + \log |\Sigma| + p_{\mathsf{dcm}}) \ .$$

Acknowledgments

This work is supported by NSF grant 2428488 and by DARPA under Agreement No. HR00112020023. We thank Giacomo Fenzi and Alessandro Chiesa for helpful discussions about concrete security analyses of Kilian-Micali and applications for our abstractions. We further thank Lawrence Roy, Jack Doerner, Ran Cohen, Eysa Lee, Anna Lysyanskaya and Pratik Sarkar for illuminating discussions about UC non-interactive functionalities.

References

[BCHTZ20] C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas. “Universal composition with global subroutines: Capturing global setup within plain UC”. In: *Proceedings of the 18th Theory of Cryptography Conference*. TCC ’20. 2020, pp. 1–30.

[BCS16] E. Ben-Sasson, A. Chiesa, and N. Spooner. “Interactive Oracle Proofs”. In: *Proceedings of the 14th Theory of Cryptography Conference*. TCC ’16-B. 2016, pp. 31–60.

[BDHMQN17] B. Broadnax, N. Döttling, G. Hartung, J. Müller-Quade, and M. Nagel. “Concurrently composable security with shielded super-polynomial simulators”. In: *Proceedings of the 36th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’17. 2017, pp. 351–381.

[Ben+14] E. Ben-Sasson et al. “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP ’14. 2014, pp. 459–474.

[BFGMR22] A. Bienstock, J. Fairoze, S. Garg, P. Mukherjee, and S. Raghuraman. “A more complete analysis of the signal double ratchet algorithm”. In: *Proceedings of the 42nd Annual International Cryptology Conference*. CRYPTO ’22. 2022, pp. 784–813.

[BFS20] B. Bünz, B. Fisch, and A. Szepieniec. “Transparent SNARKs from DARK Compilers”. In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’20. 2020, pp. 677–706.

[BMNW25] B. Bünz, P. Mishra, W. Nguyen, and W. Wang. “Accumulation without homomorphism”. In: *Proceedings of the 16th Innovations in Theoretical Computer Science Conference*. ITCS ’25. 2025.

[Can01] R. Canetti. “Universally Composable Security: A New Paradigm for Cryptographic Protocols”. In: *Proceedings of the 42nd Annual IEEE Symposium on Foundations of Computer Science*. FOCS ’01. 2001, pp. 136–145.

[Can04] R. Canetti. “Universally composable signature, certification, and authentication”. In: *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. CSFW ’04. 2004, pp. 219–233.

[Can20] R. Canetti. “Universally composable security”. In: *Journal of the ACM (JACM)* 67.5 (2020), pp. 1–94.

[CDGLN18] J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. “The Wonderful World of Global Random Oracles”. In: *Proceedings of the 37th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’18. 2018, pp. 280–312.

[CDGS24] A. Chiesa, M. Dall’Agnol, Z. Guan, and N. Spooner. “Concrete Security for Succinct Arguments from Vector Commitments”. In: *Proceedings of the 22nd Theory of Cryptography Conference*. TCC ’24. 2024, To appear.

[CDLLR24] R. Cohen, J. Doerner, E. Lee, A. Lysyanskaya, and L. Roy. *An Unstoppable Ideal Functionality for Signatures and a Modular Analysis of the Dolev-Strong Broadcast*. Cryptology ePrint Archive, Report 2024/1807. 2024.

[CF01] R. Canetti and M. Fischlin. “Universally Composable Commitments”. In: *Proceedings of the 21st Annual International Cryptology Conference*. CRYPTO ’01. 2001, pp. 19–40.

[CF13] D. Catalano and D. Fiore. “Vector commitments and their applications”. In: *Proceedings of the 16th International Conference on Practice and Theory in Public Key Cryptography*. PKC ’13. 2013, pp. 55–72.

[CF24] A. Chiesa and G. Fenzi. “zkSNARKs in the ROM with Unconditional UC-Security”. In: *Proceedings of the 22nd Theory of Cryptography Conference*. TCC ’24. 2024, To appear.

[CFFQR21] M. Campanelli, A. Faonio, D. Fiore, A. Querol, and H. Rodríguez. “Lunar: a toolbox for more efficient universal and updatable zkSNARKs and commit-and-prove extensions”. In: *Proceedings of the 27th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’21. 2021, pp. 3–33.

[CGH04] R. Canetti, O. Goldreich, and S. Halevi. “The random oracle methodology, revisited”. In: *Journal of the ACM* 51.4 (2004), pp. 557–594.

[CHMMVW20] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward. “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS”. In: *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’20. 2020, pp. 738–768.

[CJS14] R. Canetti, A. Jain, and A. Scafuro. “Practical UC security with a Global Random Oracle”. In: *Proceedings of the 21st ACM Conference on Computer and Communications Security*. CCS ’14. 2014, pp. 597–608.

[CJSV22] R. Canetti, P. Jain, M. Swanberg, and M. Varia. “Universally Composable End-to-End Secure Messaging: A Modular Analysis”. In: *Proceedings of the 42nd Annual International Cryptology Conference*. CRYPTO ’22. 2022, pp. 3–33.

[CK02] R. Canetti and H. Krawczyk. “Universally composable notions of key exchange and secure channels”. In: *Proceedings of the 21st Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’02. 2002, pp. 337–351.

[CKN03] R. Canetti, H. Krawczyk, and J. B. Nielsen. “Relaxing chosen-ciphertext security”. In: *Proceedings of the 23rd Annual International Cryptology Conference*. CRYPTO ’03. 2003, pp. 565–582.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. “Universally composable two-party and multi-party secure computation”. In: *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*. STOC ’02. 2002, pp. 494–503.

[CP18] B. Cohen and K. Pietrzak. “Simple Proofs of Sequential Work”. In: *Proceedings of the 37th Annual International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT ’18. 2018, pp. 451–467.

[CSV16] R. Canetti, D. Shahaf, and M. Vald. “Universally composable authentication and key-exchange with global PKI”. In: *Proceedings of the 19th International Conference on Practice and Theory in Public Key Cryptography*. PKC ’16. 2016, pp. 265–296.

[CSW22] R. Canetti, P. Sarkar, and X. Wang. “Triply Adaptive UC NIZK”. In: *Proceedings of the 28th International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT ’22. 2022, pp. 466–495.

[CY21] A. Chiesa and E. Yogev. “Tight security bounds for Micali’s SNARGs”. In: *Proceedings of the 19th Theory of Cryptography Conference*. TCC ’21. 2021, pp. 401–434.

[DFKP15] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. “Proofs of space”. In: *Proceeding of the 35st Annual Cryptology Conference*. CRYPTO ’15. 2015, pp. 585–605.

[DN03] I. Damgård and J. B. Nielsen. “Universally composable efficient multiparty computation from threshold homomorphic encryption”. In: *Proceedings of the 23rd Annual International Cryptology Conference*. CRYPTO ’03. 2003, pp. 247–264.

[FFKRZ23] A. Faonio, D. Fiore, M. Kohlweiss, L. Russo, and M. Zajac. “From polynomial IOP and commitments to non-malleable zksnarks”. In: *Proceedings of the 21st Theory of Cryptography Conference*. TCC ’23. 2023, pp. 455–485.

[GKOPTT23] C. Ganesh, Y. Kondi, C. Orlandi, M. Pancholi, A. Takahashi, and D. Tschudi. “Witness-succinct universally-composable SNARKs”. In: *Proceedings of the 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. EUROCRYPT ’23. 2023, pp. 315–346.

[GOS06] J. Groth, R. Ostrovsky, and A. Sahai. “Perfect Non-interactive Zero Knowledge for NP”. In: *Proceedings of the 25th Annual International Conference on Advances in Cryptology*. EUROCRYPT ’06. 2006, pp. 339–358.

[GOS12] J. Groth, R. Ostrovsky, and A. Sahai. “New techniques for noninteractive zero-knowledge”. In: *Journal of the ACM (JACM)* 59.3 (2012), pp. 1–35.

[IMS12] Y. Ishai, M. Mahmoody, and A. Sahai. “On Efficient Zero-Knowledge PCPs”. In: *Proceedings of the 9th Theory of Cryptography Conference on Theory of Cryptography*. TCC ’12. 2012, pp. 151–168.

[KLP05] Y. T. Kalai, Y. Lindell, and M. Prabhakaran. “Concurrent general composition of secure protocols in the timing model”. In: *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*. STOC ’05. 2005, pp. 644–653.

[KPT23] M. Kohlweiss, M. Pancholi, and A. Takahashi. “How to compile polynomial IOP into simulation-extractable snarks: a modular approach”. In: *Proceedings of the 21st Theory of Cryptography Conference*. TCC ’23. 2023, pp. 486–512.

[KPT97] J. Kilian, E. Petrank, and G. Tardos. “Probabilistically checkable proofs with zero knowledge”. In: *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*. STOC ’97. 1997, pp. 496–505.

[LR22] A. Lysyanskaya and L. N. Rosenbloom. “Universally Composable Σ-protocols in the Global Random-Oracle Model”. In: *Proceedings of the 20th Theory of Cryptography Conference*. TCC ’22. 2022, pp. 203–233.

[MBBFF15] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. “{CONIKS}: Bringing key transparency to end users”. In: *Proceedings of the 24th USENIX Security Symposium*. USENIX Security ’15. 2015, pp. 383–398.

[Mer89] R. C. Merkle. “A certified digital signature”. In: *Proceedings of the 9th Annual International Cryptology Conference*. CRYPTO ’89. 1989, pp. 218–238.

[Mic00] S. Micali. “Computationally Sound Proofs”. In: *SIAM Journal on Computing* 30.4 (2000). Preliminary version appeared in FOCS ’94., pp. 1253–1298.

[MMV13] M. Mahmoody, T. Moran, and S. P. Vadhan. “Publicly verifiable proofs of sequential work”. In: *Proceedings of the 4th Innovations in Theoretical Computer Science Conference*. ITCS ’13. 2013, pp. 373–388.

[MPR10] H. K. Maji, M. Prabhakaran, and M. Rosulek. “A Zero-One Law for Cryptographic Complexity with Respect to Computational UC Security”. In: *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*. Ed. by T. Rabin. Vol. 6223. Lecture Notes in Computer Science. Springer, 2010, pp. 595–612.

[Pro17] Protocol Labs. *Filecoin: A Decentralized Storage Network*. Whitepaper. URL: `https://www.filecoin.io/filecoin.pdf`. 2017.

[PS04] M. Prabhakaran and A. Sahai. “New notions of security: achieving universal composability without trusted setup”. In: *Proceedings of the 36th annual Annual ACM Symposium on Theory of Computing*. STOC ’04. 2004, pp. 242–251.

A A weakness in earlier formulations of the global random oracle

This section demonstrates and discusses a weakness in existing formulations of the global random oracle, namely the programmable version of the global random oracle in [CDGLN18] and other formulations that build on it, such as those of [LR22; CF24]. As explained in Section 3, this weakness serves as one of the motivations for our new formulation of $\mathcal{F}_{\mathsf{GRO}}$.

The programmable version of the global random oracle in [CDGLN18] allows anyone (in particular, the adversary) to determine the output of the oracle at any point that was not yet queried. By itself, such modeling would render the oracle useless since its value can be adversarial at programmed locations. To circumvent this "model attack", [CDGLN18] allows any party of a protocol with session ID sid to check whether any point of the form (sid, v) has been programmed. Formally, this check is done via the $\mathsf{IsProgrammed}$ interface of their global random oracle functionality. Crucially, however, *neither the environment nor the adversary have access to this capability.*

At first, this stipulation appears to resolve the issue elegantly: On the one hand, a party can tell if a point in its own "reserved input space" has been surreptitiously programmed. On the other hand, simulators (i.e. ideal-model adversaries) can still program points without the environment being aware whether programming took place. Finally, since in reality oracles are never "programmed", in the real model of execution all $\mathsf{IsProgrammed}$ queries will always be answered negatively, so they need not be made "in practice."

However, a closer look reveals that the above argument is overly optimistic. In particular, the $\mathsf{IsProgammed}$ instruction provides a mechanism for a protocol to learn, by inspecting the state of the global random oracle, whether a subroutine call to an ideal functionality is fulfilled by the ideal functionality itself or else by a protocol that realizes the functionality. This means that any security analysis of the calling protocols needs to consider the protocol's behavior in all such cases, thereby either rendering the random oracle significantly less useful, or else doing away with the benefits of secure composition.

Effectively, the $\mathsf{IsProgammed}$ instruction provides a "side channel" that bypasses the protections provided by the UC theorem.

To make this point more concrete, consider some ideal functionality (say, $\mathcal{F}_{\mathsf{NIZK}}$), and let π be a protocol that securely realizes $\mathcal{F}_{\mathsf{NIZK}}$ in the presence of the programmable random oracle of [CDGLN18; LR22; CF24]. Let S_π denote the simulator used in the proof that π realizes $\mathcal{F}_{\mathsf{NIZK}}$. Now, proceed as follows.

First, add to S_π an instruction to program the oracle at some fixed input, say $x = (\mathsf{sid}_0, 0)$, to a random value. (The session id of π is in general different than sid_0; still, this operation is allowed in the model.) Observe that the modified S_π is still a valid simulator for π, since the environment cannot tell whether the input x was programmed.

Next, consider a protocol Π with session id sid_0, that instructs party P to use $\mathcal{F}_{\mathsf{NIZK}}$ to generate a proof that $(\mathbb{x}, \mathbb{w}) \in \mathcal{R}$, where $\mathbb{w}$ is secret. In addition, protocol Π instructs P to perform the following instruction, once $\mathcal{F}_{\mathsf{NIZK}}$ returns an output:

1. Check whether the point $x = (\mathsf{sid}_0, 0)$ was programmed, via sending the query $\mathsf{IsProgrammed}(x)$.
2. If the response is negative, expose $\mathbb{w}$ and halt.

Now, let $\mathcal{E}$ be an environment machine that does not program the oracle. Observe that: (a) when P uses π and runs with environment $\mathcal{E}$ the point x is not programmed, so the instruction to output all secret information is carried out. (b) However, when the composition operation is performed (i.e., π is replaced for $\mathcal{F}_{\mathsf{NIZK}}$ along with S_π and the system remains otherwise the same), the point $x = (\mathsf{sid}, 0)$ is programmed, and so the instruction to output $\mathbb{w}$ is not carried out.

Note that this change in the behavior of Π does not violate the UC theorem, nor does it point to a flaw in the proof that π securely realizes $\mathcal{F}_{\mathsf{NIZK}}$. Rather, the $\mathsf{IsProgammed}$ operation provides Π with a "legitimate

API" that indicates whether or not $\mathcal{F}_{\mathsf{NIZK}}$ was replaced by π. This API allows the unsuspecting Π to be adversely affected by the introduction of π to the system.[23]

A natural avenue to providing better separation between protocol instances with respect to the random oracle is to only allow the adversary (and simulator) to program points that are "within the domain reserved for the relevant protocol instance". However, it is not clear how to enforce such separation while still allowing the adversary (which is a global entity that can attack multiple protocol instances at the same time) to directly interact with $\mathcal{F}_{\mathsf{GRO}}$. The formalism in Section 3 provides a way to implementing this idea.

[23] It might look surprising at first that Π is affected by the act of replacing $\mathcal{F}_{\mathsf{NIZK}}$ with a protocol that UC-realizes it, and yet the UC theorem is not violated. The reason that this is the case is that UC with global functionalities allows a global functionality (such as $\mathcal{F}_{\mathsf{GRO}}$) to be affected by the act of replacing an ideal functionality (such as $\mathcal{F}_{\mathsf{NIZK}}$) by a protocol that realizes it [BCHTZ20]. This means that it is up to the global functonality whether to make this information available to the protocol that called the ideal functionality in the first place (such as Π in the above example).