Concurrent Secure Computation with Optimal Query Complexity

Ran Canetti * Vipul Goyal† Abhishek Jain‡

Abstract

The multiple ideal query (MIQ) model [Goyal, Jain, and Ostrovsky, Crypto'10] offers a relaxed notion of security for concurrent secure computation, where the simulator is allowed to query the ideal functionality *multiple times per session* (as opposed to just once in the standard definition). The model provides a quantitative measure for the degradation in security under concurrent self-composition, where the degradation is measured by the number of ideal queries. However, to date, all known MIQ-secure protocols guarantee only an overall *average* bound on the number of queries per session throughout the execution, thus allowing the adversary to potentially fully compromise some sessions of its choice. Furthermore, [Goyal and Jain, Eurocrypt'13] rule out protocols where the simulator makes only an adversary-independent constant number of ideal queries per session.

We show the first MIQ-secure protocol with worst-case per-session guarantee. Specifically, we show a protocol for any functionality that matches the [GJ13] bound: The simulator makes only a *constant* number of ideal queries in *every* session. The constant depends on the adversary but is independent of the security parameter.

An immediate corollary of our main result is in extending the password authenticated key exchange (PAKE) protocol of [GJO10] from the case of a single password to the general case of multiple arbitrary passwords. Specifically, we give the first PAKE protocol for the fully concurrent, multiple password setting in the standard model with no set-up assumptions.

1 Introduction

General feasibility results for secure computation were established nearly three decades ago in the seminal works of [Yao86, GMW87]. However, these results only promise security for a protocol if it is executed in isolation, "unplugged" from any network activity. In particular, these results are not suitable for the Internet setting where multiple protocol executions may occur *concurrently* under the control of a common adversary.

A brief history of concurrent security. Towards that end, an ambitious effort to understand and design concurrently secure protocols kicked into gear with early works such as [GK96, DDN00a],

*Boston University and Tel Aviv University. Email: canetti@bu.edu. Supported by the Check Point Institute for Information Security, ISF grant 1523/14, and NSF Frontier CNS 1413920 and 1218461 grants.

†Microsoft Research India. Email: vipul@microsoft.com

‡Johns Hopkins University. Email: abhishek@cs.jhu.edu. Work done in part while visiting Microsoft Research, India, and at Boston University and MIT, where the author was supported in part by NSF 1218461 and DARPA FA8750-11-2-0225.

and later the study of the *concurrent zero knowledge* setting [DNS98, RK99, CKPR01, KP01, PRS02]. For other functionalities and in more general settings, however, far-reaching impossibility results were established [CF01, CKL03, Lin03, BPS06, Goy12, AGJ+12, GKOV12]. These results refer to the "plain model" where the participating parties have no trusted set-up, and hold even if the parties have access to pairwise authenticated communication and a broadcast channel.

Two main lines of research have emerged in order to circumvent these impossibility results. The first concerns with the use of *trusted setup assumptions* such as a common random string, strong public key infrastructure or tamper-proof hardware tokens (see, e.g. [CLOS02, BCNP04, Kat07]).

The second line of research is dedicated to the study of weaker security definitions that allow for positive results in the plain model, without additional trust assumptions. The most notable examples of this include security w.r.t. super-polynomial time simulation [Pas03, PS04, BS05, CLP10, GGJS12] and input-indistinguishable computation [MPR06, GGJS12]. One main drawback in this line of research is that it is not always clear by "how much" is the definition of security relaxed, or in other words "how much security" is being lost due to concurrent attacks.

The multiple ideal query model and its applications. The multiple ideal query model (or, the MIQ model in short) of Goyal, Jain and Ostrovsky [GJO10] takes a different approach to the problem of quantifying the security loss. In this model, the simulator is allowed to query the ideal functionality *multiple times per session* (as opposed to just once in the standard definition). On the technical side, allowing the simulator multiple queries indeed facilitates proofs of security in a concurrent setting. On the conceptual side, this model allows for a natural quantification of the "security loss" incurred by concurrent attack: the more ideal queries, the weaker the security guarantee. Furthermore, the effect of multiple ideal queries strongly depends on the task at hand, thus allowing for more fine-tuned notions of security for a given problem or setting.

One functionality where this approach proved very effective is that of password-based key exchange (namely the two-party function that outputs a secret random value to both parties if the inputs provided by the two parties are equal). When the number of queries made by the simulator per session is a constant, the security guarantees of the MIQ model actually imply fully concurrent password-based authenticated key exchange (see [GL01, GL06, GJO10]). This fact was exploited by Goyal et. al [GJO10] to get the first concurrent PAKE in the plain model — albeit with the significant restriction that the *same* password is to be used as input in every session. This restriction results from a weakness in their modeling and analysis - a weakness that we overcome in this work.

The central question: how many queries? So, how to best bound the number of ideal queries made by the simulator? Intuitively, if we allow a large number of queries, then the security guarantee may quickly degrade and become meaningless; in particular, if enough queries are allowed, then the adversary may be able to completely learn the inputs of the honest parties. On the other hand, if the number of allowed queries is very small (say only $1 + \epsilon$ per session) then the security guarantee is very close to that of the standard definition. To exemplify this further, let us recall the following example from [GJ13]: consider two parties who wish to jointly evaluate a polynomial over a point [NP06, NP99]. The input of party P_1 is a polynomial Q, while the input of P_2 is a point α. At the end of the protocol, the party P_2 gets $Q(\alpha)$ as the output. This is a natural functionality with applications to list intersection, mutual authentication, metering on the web, etc (see [NP06] for more details on these).

Now, observe that if we only allow, say, 2 queries to a malicious P_2 in the ideal world (per real world session), then as long as Q is a high-degree polynomial, the security guarantee for P_1 is still quite meaningful. Instead of a single point, now a malicious adversary may learn the output on two points of its choice (from an exponential domain of points). However, the adversary still does not learn any information about what the polynomial evaluates to on rest of the domain. On the other hand, if we allow too many queries (exceeding the degree of the polynomial), then the ideal world adversary may be able to learn the entire polynomial Q!

Another related example is 1-out-of-m OT. Here, as long as λ, the simulator's query complexity, is smaller than m, MIQ provides meaningful security which degrades gracefully with λ. More generally, the remaining security for any session i in concurrently secure computation of function f is proportional to the "**level of unlearnability**" of $f(\cdot, x_i)$ after q queries, where x_i is the secret input of the honest party in session i. Password-based key exchange is an extreme case of an unlearnable function. Ideally, we would like to bring λ as close as possible to 1.

Prior work: Average case vs. worst case guarantees. The best positive result in the MIQ model is due to Goyal, Gupta, and Jain [GGJ13] (improving upon [GJO10]). They provide a construction where the number of ideal queries in a session are $(1 + \frac{\log^6 n}{n})$, where n is the security parameter. However, this is only an *average-case* guarantee over the sessions that provides very weak security. In particular, it does not preclude the ideal adversary from making an arbitrarily large number of queries in some chosen sessions (while keeping the number of queries low in the other sessions). In cases of interest, such as the PAKE functionality or the above oblivious polynomial evaluation functionality, this means that the security in some sessions may be *completely compromised*!

Furthermore, Goyal and Jain [GJ13] recently proved an unconditional lower bound on the number of ideal queries per session. Specifically, they show that there exists a two-party functionality that cannot be securely realized in the MIQ model with any (adversary independent) constant number of ideal queries per session. A natural and important question is thus what is the best worst-case bound we can give on the number of ideal queries asked per session?

1.1 Our Results

In this work, we fully settle the question of worst-case number of per session ideal queries in the context of general function evaluation. Our main result is stated below.

Theorem 1 (Main result (informally stated))**.** *Under standard cryptographic assumptions, for every PPT functionality f, there exists a protocol in the MIQ model where the simulator makes only a constant number of ideal queries in every session. The aforementioned constant is dependent upon the adversary, and, in particular on the number of sessions (rather than being universal).*

If the number of concurrent sessions being executed by the adversary is n^c, then the constant in the above theorem will be derived from c. A more detailed discussion on this can be found at the end of this subsection.

We stress that due to the worst-case guarantee of our result, we are able to achieve, for the *first* time in the study of the MIQ model, meaningful security for *all sessions*, which is much closer to standard security for secure computation. Interestingly, our protocol is the same as the [GGJ13] protocol. Still, we provide a significantly better analysis of its security. We stress that prior to this work, no approach for obtaining a worst-case bound on the ideal query complexity was known.

Our upper bound tightly matches the lower bound of Goyal and Jain [GJ13] which rule out protocols where the simulator makes a constant number of ideal queries per session for any universal constant. Taken together, this fully resolves the central problem in the study of the MIQ problem: a (adversary dependent) constant number of ideal queries per session is both necessary and sufficient for simulation. Thus, our work can be viewed as the *final step* in understanding the simulator query complexity of the MIQ model.

Fully concurrent PAKE without setup. Say that a password-based key exchange protocol is *fully concurrent* if it remains secure in a setting where unboundedly many executions of the protocol run concurrently, on potentially different passwords. An immediately corollary of our main result is the resolution of the long standing open problem of designing a fully concurrent PAKE protocol in the standard model and with no setup assumptions:

Theorem 2 (Concurrent PAKE (informally stated))**.** *Under standard cryptographic assumptions, there exists a fully concurrent Password-based Key Exchange protocol in the standard model and with no trusted set-up. The security of the exchanged keys is $c/|D|$, where D is the password dictionary and c is an adversary-dependent constant.*

A discussion on adversary dependent constants. In the above theorems, if the number of concurrent sessions being executed by the adversary is n^c, then the number of ideal world queries made by the simulator (in Theorem 1) or the distinguishing probabilty for the exchanged keys (in Theorem 2) is a function of c alone. We call this as an *adversary dependent* constant. Consider any adversary which runs in polynomial time. For such an adversary, there must exist a constant c such that n^c bounds its running time (and hence the number of concurrent sessions). Then if the number of ideal queries is a function of c alone, it does *not* grow with n. Thus overall, the number of ideal queries is a fixed constant for every polynomial time adversary (although this constant could be different for different polynomial time adversaries).

We remark that this is reminiscent of how we define and treat the running time of simulator in zero-knowledge (and other cryptographic protocols). The running time of the simulator may depend upon the running time of the adversary (and in particular upon the number of sessions in the concurrent setting), and, hence is not an a priori fixed polynomial. However for every adversary, there is a (possibly different) polynomial in the security parameter which describes the running time of the simulator.

1.2 Technical Overview

Simulator Query Complexity and Precise Simulation. The question of simulator query complexity in the MIQ model is intimately connected to the notion of precise simulation introduced by Micali and Pass [MP06]. Recall that traditional simulator strategies allow for the simulator's running time to be an arbitrary polynomial factor of the (worst-case) running time of the real adversary. The notion of precise simulation concerns with the study of how low this polynomial can be. This idea is, in fact, much more general and can also be used in the context of resources other than running time, such as memory, etc. Thus, in the most general sense, the goal of precise simulation is to develop simulation strategies whose resource utilization is "close" to the resource utilization of the real adversary.

As observed in [GJO10], the study of simulator query complexity in the MIQ model can also be cast as a precise simulation problem by viewing the trusted party queries as the resource of the simulator. Therefore, advances in precise simulation strategies go hand in hand with improvements in the simulator query complexity in the MIQ model. Indeed, prior works in the MIQ model [GJO10, GGJ13] have relied upon sophisticated precise simulation strategies in order to obtain their positive results. We note, however, that till date, all precise simulation strategies only focus on minimizing the *total cost* of the simulator across all the sessions. Indeed, this is why these works only yield an *average-case* bound on the simulator query complexity.

In this work, we are interested in minimizing the *worst-case* simulator query complexity per session. In other words, we are interested in simulation strategies that guarantee **local precision for every session**.

Our approach in a nutshell. Towards that end, our starting observation is that the problem of bounding the simulator query complexity per session can be reduced to bounding the number of times the output message of a session appears in the entire simulation transcript.[1] In other words, we need a precise (concurrent) simulation strategy where the output message of every session appears only a constant number of times across the *entire* simulation transcript.[2] For this purpose, we revisit existing precise simulation strategies. Concretely, we show that a slight variant of the "sparse" rewinding strategy of Goyal, Gupta and Jain [GGJ13] (that we henceforth refer to as the GGJ simulation strategy) satisfies our desired property. We prove this by a novel, purely combinatorial analysis. Our final secure computation protocol remains essentially identical to those in the prior works in the MIQ model.

We now give an overview of the steps involved in our proof. Say that we wish to analyze the number of queries in session i. Consider the specific point in the protocol execution of session i where, the simulator actually makes a query to the ideal functionality: call this point p_i (for example, this may be the 5th message of the protocol execution in session i). This means that whenever the simulator reaches the point p_i (in the overall concurrent execution), it will have to call the trusted functionality for session i to compute the next outgoing message. Thus, now the problem reduces to *simply counting* how many times the point p_i occurs in the entire rewinding schedule. Observe that in each thread of execution, point p_i only occurs once. However, there could be multiple threads of execution resulting because of rewinding. Therefore, p_i may also occur multiple times in the rewinding schedule.

While a direct (full) analysis of the GGJ rewinding strategy [GGJ13] turns out to be complex, we are able to break it down into three different steps. Each step builds upon the previous one, with the final step yielding us the desired bound on the simulator query complexity. Below, we provide an informal overview of each of the three steps and refer the reader to the later sections for details.

Step 1. Lazy-KP with *static* scheduling: We first consider the warm-up case when scheduling of messages by the adversary is static. This means that the ordering of the messages of different sessions is decided by the adversary ahead of time and is fixed (and does not change upon rewinding

[1] More concretely, we wish to bound the first message in the protocol where the simulator is forced to query the trusted party in order to obtain the function output.

[2] Note that the output message of a session may appear more than once in the simulation transcript if the simulator employs rewinding.

by the simulator). Further, instead of directly analyzing the GGJ simulator [GGJ13], here we will analyze the query complexity of the (simpler) "lazy-KP" simulator [PTV14, PRS02, KP01] for the case where the simulator uses a splitting factor of n for rewinding. That is, during simulation, each thread is divided into n equal parts, and, each resulting part is rewound individually (resulting in different threads of execution).

In this case, we are able to prove that the simulator makes at most $O(1)$ queries to the ideal functionality in any given session. This is done by relying on the following fact. Say that the point p_i does *not* occur in a given thread. Then, since the adversary only employs static scheduling, this would mean that the point p_i also cannot occur in any threads resulting from rewinding this thread. Thus, the proof reduces to a counting argument on the number of threads resulting from rewinding the part of the main thread containing p_i. If d is the depth of recursion for our recursive rewinding schedule, then we are able to show that there are at most $O(2^d)$ threads containing point p_i. However, the depth d will be a constant for lazy-KP simulation with splitting factor n.

Step 2. Lazy-KP with *dynamic* scheduling: Now we analyze a general adversary that may dynamically change the ordering of the messages across different sessions upon being rewound. Hence, different threads of execution may have different ordering of the messages. We shall continue to analyze the lazy-KP simulation strategy with splitting factor n.

In this case, we are able to prove that the simulator makes at most $O(\log(n))$ queries to the ideal functionality in any given session. The key difficulty in this case is that even if a given thread does *not* contain the point p_i, the threads resulting from its rewinding may still have p_i. Hence, it seems hard to rule out the possibility that p_i may show up in a large number of threads throughout the simulation.

To overcome this problem, we rely on the following fact: once the point p_i is seen in the main thread of execution, it cannot occur in any thread arising out of the main thread *after* that point. We also observe that *before* this point is seen in the main thread, there seems hope to rule out its occurrence in a "large" number of look ahead threads. This relies on the symmetry of the main and the look-ahead threads, and, on the fact that this point has roughly equal probability of occurring first in the main thread vs occurring first in any given look ahead thread. Indeed, this step of the proof is much more involved than the first step and we refer the reader to Section 4 for more details.

Step 3. *Sparsifying* the lazy-KP simulation: In the final step, we analyze the *sparse* rewinding strategy of [GGJ13]. Very roughly speaking, the sparse rewinding strategy of [GGJ13] aims to rewind the adversary in "as few places as possible" while still solving all the sessions. More specifically, there is a cost associated with creating each look ahead, and, the goal of the rewinding strategy is to solve all sessions while minimizing the cost.

The sparse rewinding strategy of [GGJ13] builds upon the lazy-KP simulator with splitting factor n. Very roughly, [GGJ13] pick a subset of the total threads resulting out of the lazy-KP simulation, and choose to execute only the threads in the subset (while ignoring the remaining threads by aborting them at their start). In more detail, at each level of recursion, [GGJ13] randomly chooses $\frac{\text{polylog}(n)}{n}$ fraction of the total threads and execute them while ignoring the rest. Interestingly, Goyal et. al [GGJ13] show that, if one uses protocols with somewhat higher round complexity, all the session will still be solved even though most of the look-ahead threads are never executed.

The key idea of our final step is to leverage this sparsification in order the reduce the number of queries from $O(\log(n))$ from the previous step to $O(1)$. Recall from above that if we were to use the full lazy-KP simulation, the point p_i would have occurred at $O(\log(n))$ places in the entire simulation. However, now, in the GGJ rewinding strategy, it will occur only $O(1)$ times because most of the threads will never be executed. More details are given in section 5.

2 Our Model

We define our security model by extending the standard real/ideal paradigm for secure computation. Roughly speaking, we consider a relaxed notion of concurrently secure computation where the ideal world adversary is allowed to make an a priori fixed λ number of output queries to the ideal functionality for each session. Note that in contrast, the standard definition for concurrently secure computation only allows for *one* output query per session to the ideal adversary. We now give more details.

Notation. Let n denote the security parameter. We denote computational indistinguishability by $\stackrel{c}{\equiv}$. In this work, we consider malicious, static adversaries that choose whom to corrupt before the start of any protocol. Further, we work in the static input setting, i.e., we assume that the inputs of the honest parties in all sessions are fixed at the beginning. We do not require fairness.

Ideal model. We first define the ideal world experiment, where there is a trusted party for computing the desired two-party functionality f. Let there be two parties P_1 and P_2 that are involved in multiple, say $m = m(n)$, evaluations of f.[3] Let $\mathcal{S}$ denote the adversary. The ideal world execution (parameterized by λ) proceeds as follows.

I. Inputs: P_1 and P_2 obtain a vector of m inputs, denoted $\vec{x}$ and $\vec{y}$ respectively. The adversary is given auxiliary input z, and chooses a party to corrupt. Without loss of generality, we assume that the adversary corrupts P_2 (when the adversary controls P_1, the roles are simply reversed). The adversary receives the input vector $\vec{y}$ of the corrupted party.

II. Session initiation: The adversary initiates a new session by sending a start-session message to the trusted party. The trusted party then sends $(\mathsf{start\text{-}session}, i)$ to P_1, where i is the index of the session.

III. Honest parties send inputs to trusted party: Upon receiving $(\mathsf{start\text{-}session}, i)$ from the trusted party, honest party P_1 sends (i, x_i) to the trusted party, where x_i denotes P_1's input for session i.

IV. Adversary sends input to trusted party and receives output: Whenever the adversary wishes, it may send a message $(i, \ell, y'_{i,\ell})$ to the trusted party for any $y'_{i,\ell}$ of its choice. Upon sending this pair, it receives back $(i, \ell, f(x_i, y'_{i,\ell}))$ where x_i is the input value that P_1 previously sent to the trusted party for session i. The only limitation is that for any i, the trusted party accepts at most λ tuples indexed by i from the adversary.

V. Adversary instructs trusted party to answer honest party: When the adversary sends a message of the type $(\mathsf{output}, i, \ell)$ to the trusted party, the trusted party sends $(i, f(x_i, y'_{i,\ell}))$ to P_1, where x_i and $y'_{i,\ell}$ denote the respective inputs sent by P_1 and adversary for session i.

[3] Note that there is no a priori bound assumed on m.

VI. Outputs: The honest party P_1 always outputs the values $f(x_i, y'_{i,\ell})$ that it obtained from the trusted party. The adversary may output an arbitrary (probabilistic polynomial-time computable) function of its auxiliary input z, input vector $\vec{y}$ and the outputs obtained from the trusted party.

The ideal execution of a function $\mathcal{F}$ with security parameter n, input vectors $\vec{x}$, $\vec{y}$ and auxiliary input z to $\mathcal{S}$, denoted $\mathsf{Ideal}_{\mathcal{F},\mathcal{S}}(n, \vec{x}, \vec{y}, z)$, is defined as the output pair of the honest party and $\mathcal{S}$ from the above ideal execution.

Definition 1 (λ-Ideal Query Simulator)**.** *Let $\mathcal{S}$ be a non-uniform probabilistic (expected)* PPT *machine representing the ideal-model adversary. We say that $\mathcal{S}$ is a λ-ideal query simulator if it makes at most λ output queries per session in the above ideal experiment.*

Real model. Let Π be a two-party protocol for computing $\mathcal{F}$. Let $\mathcal{A}$ denote a non-uniform probabilistic polynomial-time adversary that controls either P_1 or P_2. The parties run concurrent executions of the protocol Π, where the honest party follows the instructions of Π in all executions. The honest party initiates a new session i with input x_i whenever it receives a start-session message from $\mathcal{A}$. The scheduling of all messages throughout the executions is controlled by the adversary. At the conclusion of the protocol, an honest party computes its output as prescribed by the protocol. Without loss of generality, we assume the adversary outputs exactly its entire view of the execution of the protocol.

The real concurrent execution of Π with security parameter n, input vectors $\vec{x}$, $\vec{y}$ and auxiliary input z to $\mathcal{A}$, denoted $\mathsf{Real}_{\Pi,\mathcal{A}}(n, \vec{x}, \vec{y}, z)$, is defined as the output pair of the honest party and $\mathcal{A}$, resulting from the above real-world process.

Definition 2 (λ-Secure Concurrent Computation in the MIQ Model)**.** *A protocol Π is said to λ-*securely realize *a functionality $\mathcal{F}$ under concurrent self composition in the MIQ model if for every real model non-uniform* PPT *adversary $\mathcal{A}$, there exists a non-uniform (expected)* PPT *λ-ideal query simulator $\mathcal{S}$ such that for all polynomials $m = m(n)$, every pair of input vectors $\vec{x} \in X^m$, $\vec{y} \in Y^m$, every $z \in \{0,1\}^*$,*

$$\{\mathsf{Ideal}_{\mathcal{F},\mathcal{S}}(n, \vec{x}, \vec{y}, z)\}_{n\in\mathbb{N}} \stackrel{c}{\equiv} \{\mathsf{Real}_{\Pi,\mathcal{A}}(n, \vec{x}, \vec{y}, z)\}_{n\in\mathbb{N}}$$

3 Framework for Concurrent Extraction

The Setting. Consider the following two-party computation protocol $\Pi = (P_1, P_2)$:

- **Stage 1:** First, P_1 and P_2 interact in the commit phase of an execution of an extractable commitment scheme $\langle C, R\rangle$ (described below) where P_2 acts as the committer, committing to a random string, and, P_1 acts as the receiver.
- **Stage 2:** At the end of the commitment protocol, P_1 sends a special message msg to P_2.

Now, consider the scenario where P_1 and P_2 are interacting in multiple concurrent executions of Π. Suppose that P_2 is corrupted. Our goal is to design a simulator algorithm $\mathcal{S}$ that satisfies the following two properties:

- **Extraction in all sessions:** $\mathcal{S}$ must successfully extract the value committed by adversarial P_2^* in each execution of Π.

- **Minimize the query parameter:** Let λ denote the upper bound on the number of times the special message msg_s of any session s appears in the entire simulation transcript. We refer to λ as the *query parameter*. Then, the goal of $\mathcal{S}$ is to minimize the query parameter.

In the next subsection, we describe the extractable commitment scheme $\langle C, R\rangle$ from [PRS02]. Later, in Sections 4 and 5, we analyze the "lazy-KP" rewinding strategy [PTV14, PRS02, KP01] and the "sparse" rewinding strategy of Goyal, Gupta and Jain (GGJ) [GGJ13].

3.1 Extractable Commitment Protocol $\langle C, R\rangle$

Let $\textsc{com}(\cdot)$ denote the commitment function of a non-interactive perfectly binding string commitment scheme. Let n denote the security parameter. Let $\ell = \omega(\log n)$. Let $N = N(n)$ which will be determined later depending on the extraction strategy. The commitment scheme $\langle C, R\rangle$ between the committer C and the receiver R is described as follows.

Commit Phase: This consists of two stages, namely, the Init stage and the Challenge-Response stage, described below:

Init: To commit to a n-bit string σ, C chooses $(\ell \cdot N)$ independent random pairs of n-bit strings $\{\alpha^0_{i,j}, \alpha^1_{i,j}\}^{\ell,N}_{i,j=1}$ such that $\alpha^0_{i,j} \oplus \alpha^1_{i,j} = \sigma$ for all $i \in [\ell], j \in [N]$. C commits to all these strings using $\textsc{com}$, with fresh randomness each time. Let $B \leftarrow \textsc{com}(\sigma)$, and $A^0_{i,j} \leftarrow \textsc{com}(\alpha^0_{i,j})$, $A^1_{i,j} \leftarrow \textsc{com}(\alpha^1_{i,j})$ for every $i \in [\ell], j \in [N]$.

Challenge-Response: For every $j \in [N]$, do the following:

- $\mathsf{Challenge}$: R sends a random ℓ-bit challenge string $v_j = v_{1,j}, \ldots, v_{\ell,j}$.
- $\mathsf{Response}$: $\forall i \in [\ell]$, if $v_{i,j} = 0$, C opens $A^0_{i,j}$, else it opens $A^1_{i,j}$ by sending the decommitment information.

Open Phase: C opens all the commitments by sending the decommitment information for each one of them. R verifies the consistency of the revealed values. This completes the description of $\langle C, R\rangle$.

Notation. We introduce some terminology that will be used in the remainder of this paper. We refer to the committed value σ as the *preamble secret*. A slot_i of the commitment scheme consists of the i'th $\mathsf{Challenge}$ message from R and the corresponding $\mathsf{Response}$ message from C. Thus, in the above protocol, there are N slots.

4 Lazy-KP Extraction Strategy

In this section, we discuss the "lazy-KP" rewinding strategy[4] [PTV14, PRS02, KP01] with a "splitting factor" of n. We note that the idea of using a large splitting factor was first used in [PPS+08].

For this strategy, we will first prove that $\lambda = \mathcal{O}(1)$ for *static* adversarial schedules. Next, we will prove that for *dynamic* schedules, $\lambda = \mathcal{O}(\log n)$. In both of these results, the constants in $\mathcal{O}$ depend on number of sessions started by the concurrent adversary.

[4]The term "lazy-KP" originates in [PTV14].

Lazy-KP Simulator. The rewinding strategy of the lazy-KP simulator is specified by the Lazy-KP-Simulate procedure. Very roughly, the simulator divides the current thread (given as input) into n equal parts and then rewinds each part individually and recursively. The input to the Lazy-KP-Simulate procedure consists of a triplet $(\ell, \mathsf{hist}, \mathcal{T})$. The parameter ℓ denotes the adversary's messages to be explored, the string hist is a transcript of the *current* thread of execution, and $\mathcal{T}$ is a table containing the contents of all the adversary's messages explored so far (to extract the preamble secrets and for sending the Stage 2 special message in protocol Π in any session).

The simulation is performed by invoking the procedure Lazy-KP-Simulate with appropriate parameters. Let $m = \mathsf{poly}(n)$ denote the number of concurrent sessions in the adversarial schedule. Then, the Lazy-KP-Simulate procedure is invoked with input $(m\,(N+1)\,, \emptyset, \emptyset)$, where $m(N+1)$ is the total number of adversary's messages in a schedule of m sessions. The Lazy-KP-Simulate procedure is described in Figure 1. Note that here (similar to [PPS+08]) we divide each thread into n parts. In other words, we consider a splitting factor of n.

Lazy-KP-Simulate$(\ell, \mathsf{hist}, \mathcal{T})$:

Bottom level $(\ell = 1)$:

- Run P_1's algorithm to choose the next message α_1 and feed P_2^* with $(\mathsf{hist}, \alpha_1)$. Let α_2 be the answer of P_2^*.
- Output $((\alpha_1, \alpha_2), \alpha_2)$.

Recursive step $(\ell > 1)$:

1. Initialize $\widetilde{\mathsf{hist}} = \emptyset$, $\widetilde{\mathcal{T}} = \emptyset$.
2. For every $i \in [n]$:
   (a) Compute $(\widetilde{\mathsf{hist}}_{i,1}, \widetilde{\mathcal{T}}_{i,1}) \leftarrow$ Lazy-KP-Simulate $\left(\ell/n, \left(\mathsf{hist}, \widetilde{\mathsf{hist}}\right), \left(\mathcal{T}, \widetilde{\mathcal{T}}\right)\right)$.
   (b) Compute $(\widetilde{\mathsf{hist}}_{i,2}, \widetilde{\mathcal{T}}_{i,2}) \leftarrow$ Lazy-KP-Simulate $\left(\ell/n, \left(\mathsf{hist}, \widetilde{\mathsf{hist}}\right), \left(\mathcal{T}, \widetilde{\mathcal{T}}\right)\right)$.
   (c) Update $\widetilde{\mathsf{hist}} = (\widetilde{\mathsf{hist}}, \widetilde{\mathsf{hist}}_{i,1})$ and $\widetilde{\mathcal{T}} = (\widetilde{\mathcal{T}}, \widetilde{\mathcal{T}}_{i,1}, \widetilde{\mathcal{T}}_{i,2})$.
3. Output $(\widetilde{\mathsf{hist}}, \widetilde{\mathcal{T}})$.

Figure 1: Lazy-KP Simulator with splitting factor n. Even though the messages in $\{\widetilde{\mathsf{hist}}_{i,2}\}$ do not appear in the output, some of them do appear in $\widetilde{\mathcal{T}}$.

For every session s consisting of an execution of Π, the goal of the simulator is to find two instances of any slot $i \in [N]$ of the commitment protocol $\langle C, R\rangle$ where the simulator's challenges are different and adversary responds with a valid response to each challenge. Note that in this case, the simulator can extract the preamble secret of $\langle C, R\rangle$ from the two responses of the adversary. On the other hand, if the simulation reaches Stage 2 in Π at any time, without having extracted the preamble secret from the adversary, then it gives up the simulation and outputs $\perp$. In this case, we say the simulator *gets stuck*.

It follows from [PTV14] that the lazy-KP simulator (as described above) gets stuck with only

negligible probability.

Theorem 3 ([PTV14]). *Let $N = \mathcal{O}(n)$. Then, for any concurrent schedule of $m = \mathsf{poly}(n)$ sessions, the lazy-KP simulator gets stuck with only $\mathsf{negl}(n)$ probability.*

4.1 Terminology for Concurrent Simulation

Here we introduce some terminology and definitions regarding concurrent simulation that will be used in the rest of the paper.

Execution Thread. Consider any adversary that starts $m = \mathsf{poly}(n)$ number of concurrent sessions of Π. In order to extract the preamble secret in every session, the simulator creates multiple execution threads, where a thread of execution is a simulation of (part of) the protocol messages in the m sessions. We differentiate between the following:

Main Thread vs Look-ahead Thread: The *main thread* is a simulation of a complete execution of the m sessions, and this is the execution thread that is output by the simulator. In addition, from any execution thread, the simulator may create other threads by rewinding the adversary to a previous state and continuing the execution from that state. Such a thread is called a *look-ahead thread*. Note that a look-ahead thread can be created from another look-ahead thread.

Complete vs Partial Thread: We say that an execution thread T is a *complete* thread if it shares a prefix with the main thread: it starts where the main thread starts, and, continues until it is terminated by the simulator. Other threads that start from intermediary points of the simulation are called *partial* threads. Note that by definition, the main thread is a complete thread. In general, a complete thread may consist of various partial threads. Various complete threads may overlap with each other. For simplicity of exposition, unless necessary, we will not distinguish between complete and partial threads in the sequel.

Simulation Transcript. The simulation transcript is the set of all the messages between the simulator and the adversary during the simulation of all the concurrent sessions. In particular, this includes the messages that appear on the main thread as well as all the look-ahead threads.

Simulation Index. Consider $m = \mathsf{poly}(n)$ concurrent executions of Π. Let $M = m(2N + 2)$, where $2N + 2$ is the round complexity of Π. Then, a simulation index i denotes the point where the i'th message (out of a maximum of M messages) is sent on any complete execution thread in the simulation transcript.

Note that a simulation index i may appear *multiple* times on over various threads in the simulation transcript. However, a simulation index i can appear at most once on any given thread (complete or partial). In particular, every simulation index $i \in [M]$ appears on the main thread (unless the main thread is aborted prematurely). Further, if a look-ahead thread T was created from a thread at simulation index i, then only simulation indices $j > i$ can appear on T.

Static vs Dynamic Scheduling. Consider the concurrent execution of $m = \mathsf{poly}(n)$ instances of Π. Recall that the adversary controls the scheduling of the protocol messages across the m sessions. We say that a concurrent schedule is *static* if the scheduling of the protocol messages is decided by the adversary ahead of time and does not change upon rewindings. Thus, in a static

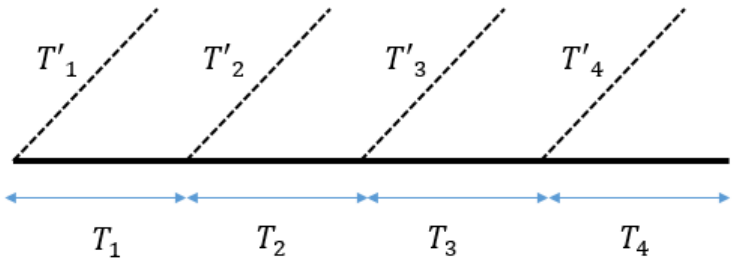

Figure 2: One recursion step for splitting factor 4. Every T_i and T'_i are sibling threads.

schedule, protocol messages appear in the *same* order on every complete thread. In particular, for every $i \in [M]$, every instance of a simulation index i in the simulation transcript corresponds to the *same* message index $j \in [2N+2]$ of the *same* session s (out of the m sessions). However note that the actual content of the j'th message may differ on every execution thread.

We say that a concurrent schedule is *dynamic* if at any point during the execution, the adversary may decide which message to schedule next based on the protocol messages received so far. Therefore, in a dynamic schedule, the ordering of messages may be *different* on different execution threads in the simulation. In particular, each instance of a simulation index i may correspond to a *different* message j_i of a *different* session s_i.

Recursion Levels. We define recursion levels of simulation and count the number of threads at each recursion level for the lazy-KP simulator. We say that the main thread is at recursion level 0 of simulation. Note that the Lazy-KP-Simulate divides the main thread of execution into n parts and executes each part twice. This results in $2n$ execution threads, n of which are part of the main thread, while the remaining n are look-ahead threads. All of these $2n$ threads are said to be at recursion level 1. Now, each of these threads at recursion level 1 is divided into n parts and each part is executed twice. This creates $2n$ threads at recursion level 2. Since there are $2n$ threads at recursion level 1, in total, we have $(2n)^2$ threads at recursion level 2. (Again, out of these $(2n)^2$ threads, $2n^2$ threads actually lie on the $2n$ threads at level 1.) This process is continued recursively. At recursion level ℓ, there are $(2n)^\ell$ threads. Since there are total $m(2N+2)$ messages across the m sessions, the depth of the recursion is a constant c', where $c' = c + \log(2N+2)$ when $m = n^c$. Then, at recursion level c', there are $(2n)^{c'}$ threads.

Sibling Threads. Consider Figure 2 where a thread T at some recursion level ℓ is divided into $n = 4$ parts, which leads to the creation of 8 threads at recursion level $\ell + 1$. Each pair of threads (T_i, T'_i) that are started from the same point are referred to as *sibling* threads.

4.2 Analysis of λ for Static Schedules

We start by analyzing the lazy-KP extraction strategy for static schedules. Let $\lambda_{\mathsf{lazy\text{-}KP}}$ denote the query parameter for the lazy-KP simulator. We claim the following:

Theorem 4. *For any constant c and any concurrent execution of $m = n^c$ instances of Π where the scheduling of messages is static, $\lambda_{\mathsf{lazy\text{-}KP}} = 2^{c'}$, where $c' = c + \log(2N + 2)$.*

In order to prove Theorem 4, we will make use of the following lemma.

Lemma 1. *For any constant c and any concurrent execution of $m = n^c$ instances of Π, the simulation transcript generated by the lazy-KP simulator is such that every simulation index $i \in [M]$ appears $2^{c'}$ times, where $c' = c + \log(2N + 2)$.*

Proof. Fix any simulation index $i \in [M]$. We will count the number of threads where i appears in the simulation transcript. We will use the definition of recursion levels for our analysis.

- First note that simulation index i appears exactly once on the main thread. Since main thread is the only thread at recursion level 0, we have that i appears once at recursion level 0.
- Now, recall that there are $2n$ threads at recursion level 1. Then, the simulation index i appears on exactly 2 threads at recursion level 1 that are siblings of each other. To see this, recall that the Lazy-KP-Simulate procedure divides the main thread into n equal parts $T_1, \ldots, T_n$. Note that each of these parts corresponds to a thread at recursion level 1. Further, Lazy-KP-Simulate creates n look-ahead threads $T'_1, \ldots, T'_n$, one from each thread T_i, which contribute to the remaining n threads at recursion level 1. Now, since simulation index i appears at most once on the main thread, let k be such that index i appears on T_k (on the main thread). Then, note that amongst the set of look-ahead threads $\{T'_j\}$, simulation index i can only appear on T'_k (which is a sibling of T_k). Thus, in total, simulation index i appears on 2 threads at recursion level 1.
- Now, suppose by induction hypothesis that the simulation index i appears 2^ℓ times at recursion level ℓ. Let $T_1, \ldots, T_{2^\ell}$ denote these 2^ℓ threads at recursion level ℓ where i appears. Now, note that each of these threads T_j leads to $2n$ threads at recursion level $\ell + 1$, out of which exactly 2 contain the simulation index i. Thus, in total simulation index i appears $2^{\ell+1}$ times at recursion level $\ell + 1$.
- Finally, by induction, there are $2^{c'}$ appearances of simulation index i at the last recursion level $c' = c + \log(2N + 2)$.

Now, note that in order the count the total number of different threads where the simulation index i appears in the simulation transcript, we only need to count the number of times it appears at recursion level c'. This is because half of the $2^{c'}$ appearances of simulation index i at recursion level c' are on threads that are part of the threads at recursion level $c' - 1$. In particular, this is true for every recursion level ℓ.

From the above, we have that each simulation index i appears on $2^{c'}$ different threads in the simulation transcript. □

Proof of Theorem 4. Consider any session s. From the definition of static scheduling, we have that for every $j \in [2N+2]$, if the j'th message of session s appears at simulation index i on any thread, then *every* instance of simulation index i in the simulation transcript corresponds to the j'th message of session s. Now, from Lemma 1, since each simulation index appears $2^{c'}$ times in the simulation transcript, we have that the special message of every session s appears $2^{c'}$ times in the simulation. Thus, we have that $\lambda_{\mathsf{lazy\text{-}KP}} = 2^{c'}$ for static schedules.

4.3 Analysis of λ for Dynamic Schedules

We now analyze the query parameter $\lambda_{\mathsf{lazy\text{-}KP}}$ for the lazy-KP extraction strategy for *dynamic* schedules. We claim the following:

Theorem 5. *For any polynomial $m = \mathsf{poly}(n)$, for any concurrent execution of m instances of Π (with possibly dynamic scheduling of messages), $\lambda_{\mathsf{lazy\text{-}KP}} = \mathcal{O}(\log n)$ except with negligible probability.*

Proof of Theorem 5. Fix any session s out of the $m = n^c$ sessions. Note that the special message msg_s of session s appears exactly once on the main thread. Let i_{main} denote the simulation index where msg_s appears on the main thread. Now, we will count:

1. The number of times msg_s appears in the simulation transcript *before* i_{main}. Let δ_1 denote this number.

2. The number of times msg_s appears in the simulation transcript at i_{main} or *after* i_{main}. Let δ_2 denote this number.

Thus, the total number of times msg_s appears in the simulation transcript is $\delta_1 + \delta_2$. In the rest of the proof, we will compute δ_1 and δ_2. In particular, we will show that (for every session s) $\delta_1 + \delta_2$ is bounded by $\mathcal{O}(\log n)$ except with negligible probability. Note that this implies that $\lambda_{\mathsf{lazy\text{-}KP}} = \mathcal{O}(\log n)$.

Let $i_1, \ldots, i_k$ be the *distinct* simulation indices where msg_s appears in the simulation transcript. Let $i_1, \ldots, i_k$ be ordered, i.e., for every $\ell \in [k-1]$, $i_\ell < i_{\ell+1}$. Let $k_1 \leq k$ be such that $i_{k_1} < i_{\mathsf{main}}$ and $i_{k_1+1} \geq i_{\mathsf{main}}$. We first make the following claim:

Lemma 2. *For any $\ell \in [k]$, the probability that msg_s does not appear on the main thread at simulation index i_ℓ is at most $(1 - \frac{1}{c'})$.*

Proof. Consider the simulation index i_1. From Lemma 1, we have that i_1 appears on $2^{c'}$ threads in the simulation transcript. Let $T[i_1] = T_1, \ldots, T_{2^{c'}}$ denote these threads. Now, let q be such that the special message msg_s appears at simulation index i_1 on q of these $2^{c'}$ threads. Let $T^*[i_1] = T^*_1, \ldots, T^*_q$ denote these q threads. Let T_{main} denote the main thread. Then, we have that:

$$\Pr\left[T_{\mathsf{main}} \in T^*\left[i_1\right]\right] = \frac{q}{2^{c'}} \tag{1}$$

To see this, recall that the Lazy-KP-Simulate procedure uses uniformly random coins on each execution thread, and follows the same strategy. Thus, the view of the adversary is indistinguishable on each thread. In particular, if p is the probability that a message α appears on a thread T and m' appears on its sibling thread T' with, then with probability $p - \mathsf{negl}(n)$, m' appears on T and m

appears on T'. (This is the "symmetry" property for threads in the lazy-KP simulation.) Therefore, Equation 1 follows.

From Equation 1, we have that:

$$\Pr\left[T_{\mathsf{main}} \notin T^*\left[i_1\right]\right] = 1 - \frac{q}{2^{c'}}$$

Note that the above probability is maximum when $q = 1$. Hence, we have that:

$$\Pr[\mathsf{msg}_s \text{ does not occur on main thread at } i_1] \leq 1 - \frac{1}{2^{c'}}. \tag{2}$$

Now, consider simulation index i_2. Again, from Lemma 1, we have that i_2 appears on $2^{c'}$ threads. Let $T[i_2]$ denote the set of these threads. Now, note that msg_s cannot appear on the look-ahead threads $T \in T^*[i_1] \cap T[i_2]$. Thus, following Equation 2, we have that:

$$\Pr[\mathsf{msg}_s \text{ does not occur on main thread at } i_2] \leq 1 - \frac{1}{2^{c''}}.$$

where $c'' \leq c'$. Continuing the same argument, we have that for every $\ell \in [k-1]$,

$$\Pr[\mathsf{msg}_s \text{ does not occur on main thread at } i_{\ell+1}] \leq \Pr[\mathsf{msg}_s \text{ does not occur on main thread at } i_\ell]$$

Thus, for every i_ℓ, we have that the probability that msg_s does not occur on main thread at i_ℓ is at most $1 - \frac{1}{c'}$. □

Computing δ_1. Now, note that $(1 - \frac{1}{c'})^t = \mathsf{negl}(n)$ for $t = \omega(\log n)$. Therefore, we have that $k_1 = \mathcal{O}(\log n)$. Now, since each of the simulation indices $i_1, \ldots, i_{k_1}$ appears $2^{c'}$ times in the simulation transcript, we have that:

$$\delta_1 \leq 2^{c'}\mathcal{O}(\log n) \tag{3}$$

Computing δ_2. We now compute the value of γ_2. Towards this, let us suppose that for every simulation index $i \in [\ell]$, the Lazy-KP-Simulate procedure runs all threads starting from simulation index i in *parallel*. That is, Lazy-KP-Simulate performs one step of execution on each of these threads. It then performs the next execution step on each of these threads, and so on. Note that this is without loss of generality since the Lazy-KP-Simulate procedure runs all such threads *independently*.

Now, we first observe that msg_s cannot appear on a look-ahead thread that starts at a simulation index $i > i_{\mathsf{main}}$. Thus, to compute δ_2, we only need to consider the look-ahead threads that started at simulation indices $i < i_{\mathsf{main}}$ and did not finish before reaching i_{main}. Let T_{good} denote the set of such threads.

Then, we claim that:

Lemma 3. $|T_{\mathsf{good}}| \leq 2^{c'}$.

Proof. Suppose for contradiction that $|T_{\mathsf{good}}| > 2^{c'}$. Now, by definition, each thread $T \in T_{\mathsf{good}}$ is such that a simulation index $i \geq i_{\mathsf{main}}$ appears on it. In other words, simulation index i_{main} appears on each thread $T \in T_{\mathsf{main}}$. However, from Lemma 1, simulation index i_{main} appears on at most $2^{c'}$ threads. This is a contradiction. □

Now, assuming the worst case where msg_s appears on each thread $T \in T_{\mathsf{good}}$, we have that:

$$\delta_2 \leq 2^{c'} \tag{4}$$

Completing the Proof of Theorem 5. From Equation 3, we have that $\delta_1 = \mathcal{O}(\log n)$. From Equation 4, we have that $\delta_2 = \mathcal{O}(1)$. Thus, summing up δ_1 and δ_2, we have that for every session s, number of times msg_s appears in the simulation transcript is $\mathcal{O}(\log n)$. Thus, we have that $\lambda_{\mathsf{lazy\text{-}KP}} = \mathcal{O}(\log n)$. This completes the proof.

5 GGJ Extraction Strategy

In this section, we discuss the GGJ extraction strategy [GGJ13] and analyze the query complexity parameter for the same. Unlike [GGJ13] that used a splitting factor of 2, we will work with n as the splitting factor. For this strategy, we will prove that for every concurrent schedule of polynomial number of sessions, the query parameter $\lambda = \mathcal{O}(1)$. Here, the constant in $\mathcal{O}$ depends on the number of concurrent sessions.

We start by providing a brief overview of the GGJ extraction strategy. We then describe the GGJ strategy more formally and then proceed to analyze the query parameter λ for the same.

Overview. Roughly speaking, the GGJ rewinding strategy can be viewed as a "stripped down" version of the lazy-KP simulation strategy. In particular, unlike lazy-KP that executes *every* thread at every recursion level, here we only execute a small fraction of them. The actual threads that are to be executed are chosen uniformly at random, at every level. It is shown in GGJ that by slightly increasing the round complexity – (roughly) $N = n^2$ from $N = n$, executing a $\frac{\mathsf{polylog} n}{N}$ fraction of threads at every level is sufficient to extract the preamble secret in every session.[5]

Below, we describe the GGJ rewinding strategy in two main steps:

1. We first describe an algorithm $\mathsf{Sparsify}$ that essentially selects which threads to execute in the lazy-KP recursion tree (Section 5.1).

2. Next, we describe the actual GGJ simulation procedure GGJ-Simulate that is essentially the same as the Lazy-KP-Simulate strategy, except that it only executes the threads selected by $\mathsf{Sparsify}$ (Section 5.2).

5.1 The Sparsification Procedure

We first describe the lazy-KP simulation tree and give a coloring scheme for the same. Next, we describe the $\mathsf{Sparsify}$ algorithm that takes the lazy-KP simulation tree as input and outputs a "trimmed" version of it that will correspond to the GGJ simulation tree.

Lazy-KP Simulation Tree. Let $m = n^c$ be the total number of concurrent sessions of Π started by an adversary $\mathcal{A}$. Then, the Lazy-KP-Simulate strategy for $\mathcal{A}$ can be described by a $2n$-ary tree $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$ of constant depth c' where $c' = c + \log(2N + 2)$. The nodes in $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$ are colored *white* or *black* as per the following strategy:

[5] We do not attempt to optimize the round complexity parameter here since our focus is on minimizing the parameter λ.

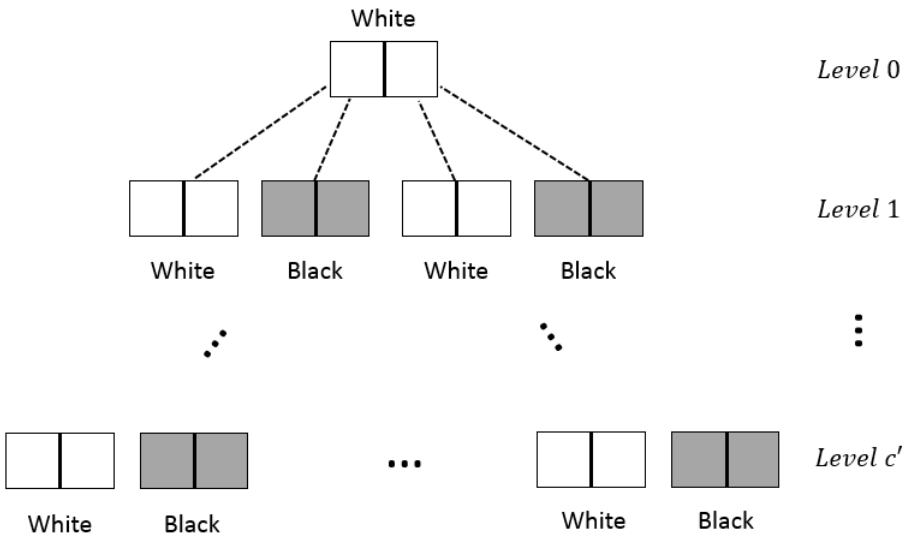

Figure 3: The lazy-KP simulation tree for splitting factor 2.

- The root node is colored white.
- Consider the $2n$ child nodes of any parent node. The odd numbered nodes are colored white and the even numbered nodes are colored black.

Let us explain our coloring strategy. The root node (which is colored white) corresponds to the main thread of execution. Each black colored node Node corresponds to a look-ahead thread that was forked from the thread corresponding to node $\mathsf{Parent}(\mathsf{Node})$. A white colored node Node (except the root node) corresponds to a thread T' that is a part of the thread T corresponding to node $\mathsf{Parent}(\mathsf{Node})$.

Figure 3 denotes the lazy-KP simulation tree for splitting factor $n = 2$ with white boxes representing white nodes and grey boxes representing black nodes.

Node Labeling. To facilitate the description of the GGJ simulation strategy, we first describe a simple tree node labeling strategy for $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$. The root node is labeled 1. The i'th child (out of $2n$ children) of the root node is labeled $(1, i)$. More generally, consider a node Node at level $\ell \in [c']$. Let path be its label. Then the i'th child of Node is labeled (path, i).

Below, whenever necessary, we shall refer to the nodes by their associated labels.

The Sparsify Procedure. Let p be such that $\frac{1}{p} = \frac{\mathsf{polylog}(n)}{N}$. The $\mathsf{Sparsify}$ function transforms the lazy-KP simulation tree $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$ into a "sparse" tree $\mathsf{Tree}_{\mathsf{sp}}$ in the following manner.

Let the root node correspond to level 0 and the leaf nodes correspond to level c'. The Sparsify procedure starts at level 0 and traverses down $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$, stopping at level c'. It performs the following steps at every level $\ell \in [c']$:

1. Choose $\frac{1}{p}$ fraction of the total black nodes at level ℓ, uniformly at random. Let B_ℓ denote the set of these nodes.

2. Delete from $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$, every black node Node at level ℓ that is not present in set B_ℓ. Further, delete the entire subtree of Node from $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$.

The resultant tree is denoted as $\mathsf{Tree}_{\mathsf{sp}}$. Looking ahead, we will describe the GGJ rewinding strategy as essentially a modification of Lazy-KP-SIMULATE in that it only executes the threads corresponding to the nodes in $\mathsf{Tree}_{\mathsf{sp}}$.

5.2 The GGJ-Simulate Procedure

The rewinding strategy of the GGJ simulator is specified by the GGJ-SIMULATE procedure. The input to the GGJ-SIMULATE procedure consists of a tuple $(\mathsf{path}, \ell, \mathsf{hist}, \mathcal{T})$. The parameter path denotes the label of the node in $\mathsf{Tree}_{\mathsf{sp}}$ that is to be explored, ℓ denotes the number of adversary's messages to be explored (on the thread corresponding to the node labeled with path), the string hist is a transcript of the *current* thread of execution, $\mathcal{T}$ is a table containing the contents of all the adversary's messages explored so far (to extract the preamble secrets and for sending the Stage 2 special message in Π in any session).

The simulation is performed by invoking the procedure GGJ-SIMULATE with appropriate parameters. Let $m = \mathsf{poly}(n)$ denote the number of concurrent sessions in the adversarial schedule. Then, the GGJ-SIMULATE procedure is invoked with input $(1, m\,(N+1)\,, \emptyset, \emptyset)$, where $m(N+1)$ is the total number of adversary's messages in a schedule of m sessions. The GGJ-SIMULATE procedure is described in Figure 4. Note that unlike [GGJ13], where each thread is recursively divided into two parts, here we divide each thread into n parts. In other words, we consider a splitting factor of n. For every session s consisting of an execution of Π, the goal of the simulator is to find two instances of any slot $i \in [N]$ of the commitment protocol $\langle C, R\rangle$ where the simulator's challenges are different and adversary responds with a valid response to each challenge. Note that in this case, the simulator can extract the preamble secret of $\langle C, R\rangle$ from the two responses of the adversary. On the other hand, if the simulation reaches Stage 2 in Π at any time, without having extracted the preamble secret from the adversary, then it gives up the simulation and outputs $\perp$. In this case, we say the simulator *gets stuck*.

It is implicit in [GGJ13] that the GGJ simulator (as described above) gets stuck with only negligible probability when $N = \mathcal{O}(n^2)$.

Theorem 6 ([GGJ13])**.** *Let $N = \mathcal{O}(n^2)$ be the number of slots in $\langle C, R\rangle$. Then, for any concurrent schedule of $m = \mathsf{poly}(n)$ sessions of Π, the GGJ simulator gets stuck with only $\mathsf{negl}(n)$ probability.*

We now analyze the query parameter λ_{GGJ} for the GGJ simulation strategy. We claim the following:

Theorem 7. *For every constant c, every $m = n^c$ number of concurrent executions of Π, the query parameter $\lambda_{\mathsf{GGJ}} = \mathcal{O}(1)$, where the constant depends on c.*

Proof. Fix any session s. We will show that the special message msg_s can appear at most $\mathcal{O}(1)$ times at each recursion level RL_ℓ. Then, since there are only a constant number of recursion levels, it will follow that $\lambda_{\mathsf{GGJ}} = \mathcal{O}(1)$.

Towards that end, lets fix a recursion level ℓ. First recall from Theorem 5 that for the lazy-KP simulation strategy, $\lambda_{\mathsf{lazy\text{-}KP}} = \mathcal{O}(\log n)$. In particular, this implies that at every recursion level ℓ in

GGJ-SIMULATE($\mathsf{path}, \ell, \mathsf{hist}, \mathcal{T}$):

Bottom level ($\ell = 1$):

- Run P_1's algorithm to choose the next message α_1 and feed P_2^* with $(\mathsf{hist}, \alpha_1)$. Let α_2 be the answer of P_2^*.
- Output $((\alpha_1, \alpha_2), \alpha_2)$.

Recursive step ($\ell > 1$):

1. Initialize $\widetilde{\mathsf{hist}} = \emptyset$, $\widetilde{\mathcal{T}} = \emptyset$.
2. For every $i \in [n]$:
   - If node $(\mathsf{path}, 2i-1) \notin \mathsf{Tree}_{\mathsf{sp}}$, set $\widetilde{\mathsf{hist}}_{i,1} = \emptyset$, $\widetilde{\mathcal{T}}_{i,1} = \emptyset$.
     Else, compute:
     $(\widetilde{\mathsf{hist}}_{i,1}, \widetilde{\mathcal{T}}_{i,1}) \leftarrow$ GGJ-SIMULATE $\left((\mathsf{path}, 2i-1), \ell/n, \left(\mathsf{hist}, \widetilde{\mathsf{hist}}\right), \left(\mathcal{T}, \widetilde{\mathcal{T}}\right)\right)$.
   - If node $(\mathsf{path}, 2i) \notin \mathsf{Tree}_{\mathsf{sp}}$, set $\widetilde{\mathsf{hist}}_{i,2} = \emptyset$, $\widetilde{\mathcal{T}}_{i,2} = \emptyset$.
     Else, compute:
     $(\widetilde{\mathsf{hist}}_{i,2}, \widetilde{\mathcal{T}}_{i,2}) \leftarrow$ GGJ-SIMULATE $\left((\mathsf{path}, 2i), \ell/n, \left(\mathsf{hist}, \widetilde{\mathsf{hist}}\right), \left(\mathcal{T}, \widetilde{\mathcal{T}}\right)\right)$.
   - Update $\widetilde{\mathsf{hist}} = (\widetilde{\mathsf{hist}}, \widetilde{\mathsf{hist}}_{i,1})$ and $\widetilde{\mathcal{T}} = (\widetilde{\mathcal{T}}, \widetilde{\mathcal{T}}_{i,1}, \widetilde{\mathcal{T}}_{i,2})$.
3. Output $(\widetilde{\mathsf{hist}}, \widetilde{\mathcal{T}})$.

Figure 4: GGJ Simulator with splitting factor n. Even though the messages in $\{\widetilde{\mathsf{hist}}_{i,2}\}$ do not appear in the output, some of them do appear in $\widetilde{\mathcal{T}}$.

the lazy-KP simulation, msg_s for a session s appears on at most $\mathcal{O}(\log n)$ threads. Using the tree terminology as introduced earlier, we have that msg_s appears on (the threads corresponding to) at most $\mathcal{O}(\log n)$ black nodes at level ℓ in $\mathsf{Tree}_{\mathsf{lazy\text{-}KP}}$. Now, recall that at every level ℓ, the Sparsify procedure selects only $\frac{1}{p} = \frac{\mathsf{polylog}n}{N}$ fraction of black nodes, uniformly at random, and deletes the rest of the black nodes. Using Chernoff bound, we now show that the probability that Sparsify selects $\omega(1)$ black nodes containing msg_s is negligible.

Towards that end, first note that the expected number of black nodes selected by Sparsify that contain the heavy message msg_s is $\mu = \frac{\mathsf{polylog}n}{N} \cdot \mathcal{O}(\log n)$. Let γ denote the *actual* number of black nodes at level ℓ containing msg_s that are selected by Sparsify. Then, we have that:

$$\Pr[\lambda \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \tag{5}$$

Setting $(1+\delta)\mu = \omega(1)$, and ignoring the $\mathcal{O}(\log n)$ in μ, we have that $(1+\delta) = \frac{\omega(1)\cdot N}{\mathsf{polylog}n}$. Now, using

the fact that $1+\delta \approx \delta$ and substituting values in Equation 5, we have:

$$\begin{aligned}
\Pr[\gamma \geq \omega(1)] &\leq \left(\frac{e^{\frac{\omega(1)\cdot N}{\mathsf{polylog}n}}}{\left(\frac{\omega(1)\cdot N}{\mathsf{polylog}n}\right)^{\frac{\omega(1)\cdot N}{\mathsf{polylog}n}}} \right)^{\frac{\mathsf{polylog}n}{N}} \\
&\leq \left(\frac{\mathsf{polylog}n}{N}\right)^{\omega(1)} \\
&= \mathsf{negl}(n)
\end{aligned}$$

when $N = \mathcal{O}(n)$. Thus, for every level ℓ, we have $\gamma = \mathcal{O}(1)$. It then follows that $\lambda_{\mathsf{GGJ}} = \mathcal{O}(1)$. □

6 From Concurrent Extraction to Concurrently Secure Computation

Theorem 8. *Assuming* 1*-out-of-*2 *oblivious transfer, for any efficiently computable functionality* f *there exists a protocol* Π *that* $\mathcal{O}(1)$*-securely realizes* f *in the MIQ model.*

We construct such a protocol by following the exact recipe of [GJO10, GGJ13]. We note that the works of [GJO10, GGJ13] show how to compile a semi-honest secure computation protocol Π_{sh} for any functionality f into a new protocol Π that securely realizes f in the MIQ model (we discuss the query parameter λ shortly). The core ingredient of their compiler is a concurrently extractable commitment $\langle C, R\rangle$, which in turn is used inside a concurrent non-malleable zero-knowledge protocol. In particular, it follows from these works that if there exists a concurrent simulator for $\langle C, R\rangle$ with query parameter λ, then the resultant (compiled) protocol Π λ-securely realizes f.

Then, in order to prove Theorem 8, we construct such a protocol Π by simply plugging in our $O(n^2)$-round extractable commitment scheme in the construction of [GJO10, GGJ13]. Then, it follows from Theorem 7 that protocol Π $\mathcal{O}(1)$-securely realizes f in the MIQ model, where the constant in $\mathcal{O}$ depends on c, where n^c is the number of sessions opened by the concurrent adversary. For completeness, we provide a description of protocol Π in Appendix A (which remain identical to these prior works except for the concurrently extractable commitment scheme being used).

Concurrent PAKE in the plain model. Consider the PAKE functionality: it takes a password as input from each party, and, if they match, outputs a randomly generated key to both of them. The above protocol, when executed for the PAKE functionality gives a PAKE construction in the MIQ model where the simulator makes a constant number of queries per session in the ideal world. We then plug in Lemma 7 in [GJO10] which shows that a PAKE construction in the MIQ model for a constant number of queries implies a concurrent PAKE as per the definition of Goldreich and Lindell [GL01] (with the modification that the constant in big O is adversary dependent). Put together, this gives us a construction of concurrent password-authenticated key exchange in the plain model.

References

[AGJ+12] Shweta Agrawal, Vipul Goyal, Abhishek Jain, Manoj Prabhakaran, and Amit Sahai. New impossibility results on concurrently secure computation and a non-interactive completeness theorem for secure computation. In *CRYPTO*, 2012.

[BCNP04] B. Barak, R. Canetti, J.B. Nielsen, and R. Pass. Universally composable protocols with relaxed set-up assumptions. In *FOCS*, 2004.

[Blu87] Manual Blum. How to prove a theorem so no one else can claim it. In *International Congress of Mathematicians*, pages 1444–1451, 1987.

[BPS06] Boaz Barak, Manoj Prabhakaran, and Amit Sahai. Concurrent non-malleable zero knowledge. In *FOCS*, 2006.

[BS05] Boaz Barak and Amit Sahai. How to play almost any mental game over the net - concurrent composition using super-polynomial simulation. In *Proc. 46th FOCS*, 2005.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, 2001.

[CKL03] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In *Eurocrypt*, 2003.

[CKPR01] Ran Canetti, Joe Kilian, Erez Petrank, and Alon Rosen. Black-box concurrent zero-knowledge requires $\widetilde{\Omega}(\log n)$ rounds. In *STOC*, pages 570–579, 2001.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC*, 2002.

[CLP10] Ran Canetti, Huijia Lin, and Rafael Pass. Adaptive hardness and composable security in the plain model from standard assumptions. In *FOCS*, 2010.

[DDN00a] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437 (electronic), 2000. Preliminary version in STOC 1991.

[DDN00b] Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM J. Comput.*, 30(2):391–437, 2000.

[DNS98] Cynthia Dwork, Moni Naor, and Amit Sahai. Concurrent zero-knowledge. In *STOC*, pages 409–418, 1998.

[GGJ13] Vipul Goyal, Divya Gupta, and Abhishek Jain. What information is leaked under concurrent composition. In *CRYPTO*, 2013.

[GGJS12] Sanjam Garg, Vipul Goyal, Abhishek Jain, and Amit Sahai. Concurrently secure computation in constant rounds. In *Eurocrypt*, 2012.

[GJ13] Vipul Goyal and Abhishek Jain. On concurrently secure computation in the multiple ideal query model. In *Eurocrypt*, 2013.

[GJO10] Vipul Goyal, Abhishek Jain, and Rafail Ostrovsky. Password-authenticated session-key generation on the internet in the plain model. CRYPTO, 2010. Full version available online.

[GK96] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM Journal on Computing*, 25(1):169–192, February 1996. Preliminary version appeared in ICALP' 90.

[GKOV12] Sanjam Garg, Abishek Kumarasubramanian, Rafail Ostrovsky, and Ivan Visconti. Impossibility results for static input secure computation. In *CRYPTO*, 2012.

[GL01] Oded Goldreich and Yehuda Lindell. Session-key generation using human passwords only. In *CRYPTO*, pages 408–432, 2001.

[GL06] Oded Goldreich and Yehuda Lindell. Session-key generation using human passwords only. *J. Cryptology*, 19(3):241–340, 2006.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, 1987.

[Goy12] Vipul Goyal. Positive results for concurrently secure computation in the plain model. In *FOCS*, 2012.

[HHK+05] Iftach Haitner, Omer Horvitz, Jonathan Katz, Chiu-Yuen Koo, Ruggero Morselli, and Ronen Shaltiel. Reducing complexity assumptions for statistically-hiding commitment. In *Eurocrypt*, pages 58–77, 2005.

[Kat07] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *Eurocrypt*, 2007.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *STOC*, pages 20–31, 1988.

[KP01] Joe Kilian and Erez Petrank. Concurrent and resettable zero-knowledge in poly-loalgorithm rounds. In *STOC*, 2001.

[Lin03] Yehuda Lindell. Bounded-concurrent secure two-party computation without setup assumptions. In *STOC*, pages 683–692. ACM, 2003.

[MP06] Silvio Micali and Rafael Pass. Local zero knowledge. In *STOC*, 2006.

[MPR06] Silvio Micali, Rafael Pass, and Alon Rosen. Input-indistinguishable computation. In *FOCS*, 2006.

[Nao91] Moni Naor. Bit commitment using pseudorandomness. *J. Cryptology*, 4(2):151–158, 1991.

[NOVY98] Moni Naor, Rafail Ostrovsky, Ramarathnam Venkatesan, and Moti Yung. Perfect zero-knowledge arguments for *NP* using any one-way permutation. *J. Cryptology*, 11(2):87–108, 1998.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer with adaptive queries. In *Crypto '99*, pages 573–590, 1999.

[NP06] Moni Naor and Benny Pinkas. Oblivious polynomial evaluation. *SIAM J. Comput.*, 35(5):1254–1281, 2006.

[Pas03] Rafael Pass. Simulation in quasi-polynomial time, and its application to protocol composition. In *Eurocrypt*, 2003.

[PPS+08] Omkant Pandey, Rafael Pass, Amit Sahai, Wei-Lung Dustin Tseng, and Muthuramakrishnan Venkitasubramaniam. Precise concurrent zero knowledge. In *Eurocrypt*, 2008.

[PRS02] Manoj Prabhakaran, Alon Rosen, and Amit Sahai. Concurrent zero knowledge with logarithmic round-complexity. In *FOCS*, 2002.

[PS04] Manoj Prabhakaran and Amit Sahai. New notions of security: achieving universal composability without trusted setup. In *STOC*, 2004.

[PTV14] Rafael Pass, Wei-Lung Dustin Tseng, and Muthuramakrishnan Venkitasubramaniam. Concurrent zero knowledge, revisited. *Journal of Cryptology*, 27(1):45–66, 2014.

[RK99] Ransom Richardson and Joe Kilian. On the concurrent composition of zero-knowledge proofs. In *Eurocrypt*, 1999.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, 1986.

A The Protocol

In this section, we describe our concurrently secure computation protocol Π in the MIQ model for a general functionality $\mathcal{F}$. We remark that this protocol is exactly the same as the one presented in [GJO10, GGJ13] except that we shall use an $N = O(n^2)$-round version of the extractable commitment scheme $\langle C, R \rangle$ (Section 3.1), while [GJO10, GGJ13] require different round-complexities for $\langle C, R \rangle$.

We start by recalling the building blocks used in the protocol.

A.1 Building Blocks

A.1.1 Statistically Binding String Commitments

In our protocol, we will use a (2-round) statistically binding string commitment scheme, e.g., a parallel version of Naor's bit commitment scheme [Nao91] based on one-way functions. For simplicity of exposition, in the presentation of our results, we will actually use a non-interactive perfectly binding string commitment.[6] Such a scheme can be easily constructed based on a 1-to-1 one way function. Let $\text{COM}(\cdot)$ denote the commitment function of the string commitment scheme. For simplicity of exposition, in the sequel, we will assume that random coins are an implicit input to the commitment function.

[6] It is easy to see that the construction given in Section A does not necessarily require the commitment scheme to be non-interactive, and that a standard 2-round scheme works as well. As noted above, we choose to work with non-interactive schemes only for simplicity of exposition.

A.1.2 Statistically Witness Indistinguishable Arguments

In our construction, we shall use a statistically witness indistinguishable (SWI) argument $\langle P_{\mathsf{swi}}, V_{\mathsf{swi}} \rangle$ for proving membership in any **NP** language with perfect completeness and negligible soundness error. Such a scheme can be constructed by using $\omega(\log k)$ copies of Blum's Hamiltonicity protocol [Blu87] in parallel, with the modification that the prover's commitments in the Hamiltonicity protocol are made using a statistically hiding commitment scheme [NOVY98, HHK+05].

A.1.3 Semi-Honest Two Party Computation

We will also use a semi-honest two party computation protocol $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$ that emulates the functionality $\mathcal{F}$ (as described in section 2) in the stand-alone setting. The existence of such a protocol $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$ follows from [Yao86, GMW87, Kil88].

A.1.4 Concurrent Non-Malleable Zero Knowledge Argument

Concurrent non-malleable zero knowledge (CNMZK) considers the setting where a man-in-the-middle adversary is interacting with several honest provers and honest verifiers in a concurrent fashion: in the "left" interactions, the adversary acts as verifier while interacting with honest provers; in the "right" interactions, the adversary tries to prove some statements to honest verifiers. The goal is to ensure that such an adversary cannot take "help" from the left interactions in order to succeed in the right interactions. This intuition can be formalized by requring the existence of a machine called the simulator-extractor that generates the view of the man-in-the-middle adversary and additionally also outputs a witness from the adversary for each "valid" proof given to the verifiers in the right sessions.

Barak, Prabhakaran and Sahai [BPS06] gave the first construction of a concurrent non-malleable zero knowledge (CNMZK) argument for every language in **NP** with perfect completeness and negligible soundness error.

In our main construction, we will use a specific CNMZK protocol, denoted $\langle P, V \rangle$, based on the CNMZK protocol of Barak et al. [BPS06] to guarantee non-malleability. Specifically, we will make the following two changes to Barak et al's protocol: (a) Instead of using an $\omega(\log n)$-round extractable commitment scheme [PRS02], we will use the N-round extractable commitment scheme $\langle C, R \rangle$ (described in Section 3.1). (b) Further, we require that the non-malleable commitment scheme being used in the protocol be public-coin w.r.t. receiver[7]. We now describe the protocol $\langle P, V \rangle$.

Protocol $\langle P, V \rangle$. Let P and V denote the prover and the verifier respectively. Let L be an NP language with a witness relation R. The common input to P and V is a statement $x \in L$. P additionally has a private input w (witness for x). Protocol $\langle P, V \rangle$ consists of two main phases: (a) the *preamble phase*, where the verifier commits to a random secret (say) σ via an execution of $\langle C, R \rangle$ with the prover, and (b) the *post-preamble phase*, where the prover proves an NP statement. In more detail, protocol $\langle P, V \rangle$ proceeds as follows.

[7] The original NMZK construction only required a public-coin extraction phase inside the non-malleable commitment scheme. We, however, require that the entire commitment protocol be public-coin. We note that the non-malleable commitment protocol of [DDN00b] only consists of standard perfectly binding commitments and zero knowledge proof of knowledge. Therefore, we can easily instantiate the DDN construction with public-coin versions of these primitives such that the resultant protocol is public-coin.

PREAMBLE PHASE.

1. P and V engage in execution of $\langle C, R\rangle$ (Section 3.1) where V commits to a random string σ.

POST-PREAMBLE PHASE.

2. P commits to 0 using a statistically-hiding commitment scheme. Let c be the commitment string. Additionally, P proves the knowledge of a valid decommitment to c using a statistical zero-knowledge argument of knowledge (SZKAOK).

3. V now reveals σ and sends the decommitment information relevant to $\langle C, R\rangle$ that was executed in step 1.

4. P commits to the witness w using a public-coin non-malleable commitment scheme.

5. P now proves the following statement to V using SZKAOK:

   (a) *either* the value committed to in step 4 is a valid witness to x (i.e., $R(x, w) = 1$, where w is the committed value), *or*

   (b) the value committed to in step 2 is the trapdoor secret σ.

   P uses the witness corresponding to the first part of the statement.

A.1.5 Modified Extractable Commitment Scheme $\langle C', R'\rangle$

Due to technical reasons, in our secure computation protocol, we will also use a minor variant, denoted $\langle C', R'\rangle$, of the extractable commitment scheme given in Section 3.1. Protocol $\langle C', R'\rangle$ is the same as $\langle C, R\rangle$, except that for a given receiver challenge string, the committer does not "open" the commitments, but instead simply reveals the appropriate committed values (without revealing the randomness used to create the corresponding commitments). More specifically, in protocol $\langle C', R'\rangle$, on receiving a challenge string $v_j = v_{1,j}, \ldots, v_{\ell,j}$ from the receiver, the committer uses the following strategy: for every $i \in [\ell]$, if $v_{i,j} = 0$, C' sends $\alpha^0_{i,j}$, otherwise it sends $\alpha^1_{i,j}$ to R'. Note that C' does not reveal the decommitment values associated with the revealed shares.

When we use $\langle C', R'\rangle$ in our main construction, we will require the committer C' to prove the "correctness" of the values (i.e., the secret shares) it reveals in the last step of the commitment protocol. In fact, due to technical reasons, we will also require the the committer to prove that the commitments that it sent in the first step are "well-formed".

We remark that the extraction proofs for the Lazy-KP-SIMULATE procedure (Section 4) and GGJ-SIMULATE procedure (Section 5) also hold for the $\langle C', R'\rangle$ commitment scheme.

A.2 Protocol Description

Notation. Let COM$(\cdot)$ denote the commitment function of a non-interactive perfectly binding commitment scheme. Let $\langle C, R\rangle$ denote the N-round extractable commitment scheme and $\langle C', R'\rangle$ be its modified version as described in Section A.1.5. Let $\langle P, V\rangle$ denote the modified version of the CNMZK argument of Barak et al. [BPS06] as described in Section A.1.4. Further, let $\langle P_{\mathsf{swi}}, V_{\mathsf{swi}}\rangle$ denote a SWI argument (Section A.1.2) and let $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}}\rangle$ denote a semi-honest two party computation protocol $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}}\rangle$ that securely computes $\mathcal{F}$ in the stand-alone setting as per the standard definition of secure computation (Section A.1.3).

Let P_1 and P_2 be two parties with inputs x_1 and x_2. Let n be the security parameter. Protocol $\Pi = \langle P_1, P_2\rangle$ proceeds as follows.

I. Trapdoor Creation Phase.

1. $P_1 \Rightarrow P_2 : P_1$ creates a commitment $com_1 = \text{COM}(0)$ to bit 0 and sends com_1 to P_2. P_1 and P_2 now engage in the execution of $\langle P, V \rangle$ where P_1 proves that com_1 is a commitment to 0.

2. $P_2 \Rightarrow P_1 : P_2$ now acts symmetrically. That is, it creates a commitment $com_2 = \text{COM}(0)$ to bit 0 and sends com_2 to P_1. P_2 and P_1 now engage in the execution of $\langle P, V \rangle$ where P_2 proves that com_2 is a commitment to 0.

Informally speaking, the purpose of this phase is to aid the simulator in obtaining a "trapdoor" to be used during the simulation of the protocol.

II. Input Commitment Phase. In this phase, the parties commit to their inputs and random coins (to be used in the next phase) via the commitment protocol $\langle C', R' \rangle$.

1. $P_1 \Rightarrow P_2 : P_1$ first samples a random string r_1 (of appropriate length, to be used as P_1's randomness in the execution of $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$ in Phase III) and engages in an execution of $\langle C', R' \rangle$ (denoted as $\langle C', R' \rangle_{1 \to 2}$) with P_2, where P_1 commits to $x_1 \| r_1$. Next, P_1 and P_2 engage in an execution of $\langle P_{\mathsf{swi}}, V_{\mathsf{swi}} \rangle$ where P_1 proves the following statement to P_2: (a) *either* there exist values $\hat{x}_1$, $\hat{r}_1$ such that the commitment protocol $\langle C', R' \rangle_{1 \to 2}$ is *valid* with respect to the value $\hat{x}_1 \| \hat{r}_1$, *or* (b) com_1 is a commitment to bit 1.

2. $P_2 \Rightarrow P_1 : P_2$ now acts symmetrically. Let r_2 (analogous to r_1 chosen by P_1) be the random string chosen by P_2 (to be used in the next phase).

Informally speaking, the purpose of this phase is aid the simulator in extracting the adversary's input and randomness.

III. Secure Computation Phase. In this phase, P_1 and P_2 engage in an execution of $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$ where P_1 plays the role of P_1^{sh}, while P_2 plays the role of P_2^{sh}. Since $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$ is secure only against semi-honest adversaries, we first enforce that the coins of each party are truly random, and then execute $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$, where with every protocol message, a party gives a proof using $\langle P_{\mathsf{swi}}, V_{\mathsf{swi}} \rangle$ of its honest behavior "so far" in the protocol. We now describe the steps in this phase.

1. $P_1 \leftrightarrow P_2 : P_1$ samples a random string r_2' (of appropriate length) and sends it to P_2. Similarly, P_2 samples a random string r_1' and sends it to P_1. Let $r_1'' = r_1 \oplus r_1'$ and $r_2'' = r_2 \oplus r_2'$. Now, r_1'' and r_2'' are the random coins that P_1 and P_2 will use during the execution of $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$.

2. Let t be the number of rounds in $\langle P_1^{\mathsf{sh}}, P_2^{\mathsf{sh}} \rangle$, where one round consists of a message from P_1^{sh} followed by a reply from P_2^{sh}. Let transcript $T_{1,j}$ (resp., $T_{2,j}$) be defined to contain all the messages exchanged between P_1^{sh} and P_2^{sh} before the point P_1^{sh} (resp., P_2^{sh}) is supposed to send a message in round j. For $j = 1, \ldots, t$:

   (a) $P_1 \Rightarrow P_2$: Compute $\beta_{1,j} = P_1^{\mathsf{sh}}(T_{1,j}, x_1, r_1'')$ and send it to P_2. P_1 and P_2 now engage in an execution of $\langle P_{\mathsf{swi}}, V_{\mathsf{swi}} \rangle$, where P_1 proves the following statement:

      i. *either* there exist values $\hat{x}_1$, $\hat{r}_1$ such that (a) the commitment protocol $\langle C', R' \rangle_{1 \to 2}$ is *valid* with respect to the value $\hat{x}_1 \| \hat{r}_1$, and (b) $\beta_{1,j} = P_1^{\mathsf{sh}}(T_{1,j}, \hat{x}_1, \hat{r}_1 \oplus r_1')$

      ii. *or*, com_1 is a commitment to bit 1.

   (b) $P_2 \Rightarrow P_1 : P_2$ now acts symmetrically.

This completes the description of the protocol $\Pi = \langle P_1, P_2 \rangle$.