On Concurrent Security in the Client-Server Model

Ran Canetti* Abhishek Jain† Omer Paneth‡

Abstract

The traditional concurrent zero knowledge setting considers a server that proves a statement in zero-knowledge to multiple clients in multiple concurrent sessions, where the server's actions in a session are *independent* of all other sessions. Persiano and Visconti [ICALP 05] show how keeping a limited amount of global state across sessions allows the server to significantly reduce the complexity of individual sessions, while retaining the ability to interact concurrently with an unbounded number of clients. Specifically, they show a protocol that has only slightly super constant number of rounds; however the communication complexity in each session of their protocol depends on the number of other sessions. This has the drawback that the client has no a priori bound to the resources required for completing the protocol up to the moment where the protocol is completed.

We put forth a new model, namely, the *committed-server* model that avoids the main disadvantage of the protocol of Persiano-Visconti. In this model, the server must *commit* to the complexity of the protocol at the start of each session. We construct a *constant-round* (fully) concurrent zero-knowledge argument system in the committed-server model, based on collision-resistant hash functions. The main technical tool underlying our result is an adaptation of the "committed-simulator" non-black-box simulation technique.

*Boston University and Tel-Aviv University. canetti@cs.tau.ac.il

†Boston University and MIT, abhishek@csail.mit.edu

‡Boston University, omer@bu.edu

1 Introduction

Classical cryptographic protocols for fundamental tasks such as zero-knowledge proofs [GMR89] and secure computation [Yao86, GMW87] only promise "stand-alone" security. However in many real-life scenarios, such as when running protocols over the global internet, such a security guarantee is insufficient. This has motivated the study of *concurrently secure* protocols, that is protocols that remain secure even when multiple instances of the protocol are executed concurrently.

When modeling concurrent protocol executions, the standard assumption is that the honest parties act *independently* in all sessions. That is, in every session, the honest party simply follows the protocol and ignores the other sessions. While this model maximizes the flexibility in protocol deployment, designing secure protocols in this model with minimal complexity and trust assumptions is a challenging goal. For example, in the case of concurrent zero-knowledge, known protocols require logarithmic round complexity [PRS02]. The situation is even worse in the case of secure computation, where achieving concurrent security without trust assumptions is, in fact, impossible (e.g., [CF01, Lin04, AGJ$^+$12]).

The Client-Server Model. Fortunately, in many real world applications it is possible for an honest party to correlate its strategies in different sessions. A typical example of such a scenario is the client-server model, where a server interacts concurrently with multiple clients. Indeed, by allowing the server to use a correlated strategy across sessions, we can simplify the session protocol and potentially make it more efficient. As a trivial example, consider a server that refuses to start a new session until the previous session is completed. In this manner, the server can make sure that all sessions are executed sequentially, and therefore, a stand-alone secure protocol would suffice. The problem, of course, is that such a correlated strategy may hurt the *availability* (namely, the incurred overall delay) of the service, since clients may be rejected or forced to wait till other sessions end.

The focus of this work is on finding strategies that optimize the efficiency of individual sessions *as well as* the server availability.

Prior Works and their Limitations. One approach to improve the server availability in the trivial example above is to use a *bounded concurrent* protocol, that is, a protocol that remains secure when executed in m concurrent sessions for some fixed polynomial m. Now, the server only refuses (or delays) new sessions when the total number of active sessions exceed m. While constant-round bounded concurrent protocols exist [Bar01, PR03], their communication grows linearly with m. This is problematic since in many cases the "correct" bound m is unknown at the time of protocol design. Concretely, setting too large a value for m may be wasteful in communication while setting too small a value for m will hurt server availability. Ideally, we would like to *dynamically* increase the complexity of the protocol as the number of sessions increase.

Towards this end, Persiano and Visconti [PV05] proposed a model for concurrent security where the server can increase the communication and round complexity of the protocol in every session "on-the-fly." More concretely, at any point *during* the protocol execution, if the server becomes "too busy" it can add more rounds to the current execution. In this model, [PV05] construct a (fully) concurrent zero-knowledge protocol. The round complexity in every session of the protocol can be bounded by any super-constant function and the communication is proportional to the number of currently active sessions.

However, while the flexibility of dynamically increasing the protocol complexity during a session serves well for the server, it may be problematic for the clients. Specifically, when a client connects to the server, it does not know how expensive it would be to complete the interaction. As such, a client may end up spending much of its resources only to find out that its total available resources are insufficient. In fact, clients may well prefer the bounded concurrency approach where a bound on the protocol complexity is known ahead of time.

1.1 Our Contribution

In this work we put forth a new model for concurrent security in the client-server setting, where:

- The server can dynamically increase the complexity of the protocol as the number of sessions increase.
- Still, at the start of every session, the server must *commit* to the complexity of the protocol executed in the session.

We refer to this as the *committed-server* model. We believe that this model correctly captures the requirements of *both* parties in the client-server setting. Working in this model allows us to strike the right balance between protocol efficiency and server availability, as expressed by the following result:

Theorem 1.1 (Informal)**.** *Assuming collision-resistant hash functions, there exists a* constant-round *fully concurrent zero-knowledge argument in the committed-server model. The communication complexity of the protocol in each session grows at most linearly with the total number of sessions initiated so far.*

We also give a more refined analysis where we define a notion of *dependence* between sessions and then bound the communication complexity of a session to be linear in the number of sessions on which the said session depends. In some natural schedules this number is much smaller than the overall number of sessions initiated so far. See more details within.

Improvements over Prior Works. The main advantage of our protocol over [PV05] is that in our protocol the server commits to the complexity of the protocol at the *beginning* of the session. This allows the clients to evaluate their resources and decide whether or not they wish to continue the execution.

Our protocol only requires a *fixed constant* number of rounds, which improves upon the best known result for concurrent zero-knowledge in the plain model based on standard assumptions [PRS02]. In fact, even though our model is strictly more demanding than the model in [PV05], we also improve upon their protocol where the round complexity must grow with the number of sessions.

1.2 Our Techniques

Barak's bounded concurrent protocol. We start by recalling the bounded concurrent zero-knowledge protocol of [Bar01]. Barak's protocol consists of two phases, namely, a preamble phase and a proof phase. Following the Feige-Lapidot-Shamir paradigm [FLS99], the preamble phase defines a "trapdoor" that can only be obtained during simulation. Using this trapdoor, the zero-knowledge simulator is able to simulate the proof stage.

In Barak's protocol, the preamble stage consists of the prover sending a commitment c to some code and the verifier responding with a random challenge r. To obtain a trapdoor, the simulator needs to commit to a program Π that outputs the verifier's challenge r. Now, in the stand-alone setting, a non-black-box simulator can simply commit to the code of the verifier's next message function to obtain the trapdoor. However, in the concurrent setting, an adversary may schedule messages of other sessions between the "c" and "r" messages of a given session. Since r may depend upon these messages, we must now allow the program Π to also receive "auxiliary input" y that can help it predict r. In particular, the input y will account for all the messages received by the verifier in other sessions before outputting r. Note that for the protocol to be sound, it must be non-trivial for a cheating prover to find a trapdoor; therefore the length of the auxiliary input y must be shorter then the length of the challenge r. Overall, we need the total length of messages received by the verifier in other concurrent sessions to be smaller than $|r|$. Then, setting $|r| = 2\ell \cdot m$, where ℓ is the length of prover's messages in a single session, will allow simulation of m concurrent sessions.

The single server protocol of [PV05]. Let us now briefly discuss the result of [PV05]. Roughly speaking, the main idea in [PV05] for handling an unbounded number of sessions is to make the length of protocol

messages unbounded as well. Concretely, the protocol starts like the bounded concurrent protocol of Barak, where the length of the challenge r is $2\ell \cdot n$ allowing n concurrent sessions to be simulated. If the adversary starts more then n sessions during the preamble stage, the prover repeats the preamble stage, this time with a challenge of length $2\ell \cdot n^2$. This process continues until the length of incoming messages to the verifier in all concurrent sessions is less then the length of the last challenge. Since the number of concurrent sessions started by an efficient adversary is bounded by some polynomial n^c, the number of rounds in every session will be a constant that depends on c.

Towards constant-round ZK in the committed-server model. In order to construct a constant-round ZK protocol in the committed-server model, we adopt the same basic idea as [PV05]. That is, we allow the server to dynamically increase the complexity of the protocol as the total number of active sessions increases. However, note that the crucial difference between the committed-server model and the model of [PV05] is that in the former, the server cannot increase the complexity of the protocol *during* a session. In view of this, our first idea is the following: in every session, the prover sets the length of the verifier's challenge based on the number of open sessions at the *start* of the session. Concretely, if the number of open sessions is between $n^{(i-1)}$ and n^i, then the length of r is set to $2\ell \cdot n^i$.

The problem is that now, it is no longer clear how to simulate. Indeed, consider a session s for which the length of r is set to $2\ell \cdot n^i$ at the start. Now, it may happen that an adversary starts many new sessions and interleaves the messages of these sessions between the c and r messages of s. If the total length of the interleaved messages exceeds $2\ell \cdot n^i$, we cannot simulate. Therefore, a new simulation strategy is required.

Committing to the simulator's code. Our solution is based on a non-black-box simulation technique that was originally suggested in [DGS09] and later used in several other results [CLP13a, GJO$^+$13, PRT13, Goy13, CLP12]. The main observation behind this technique is that even if the auxiliary input y used by the simulator is long, it has a short representation as it simply consists of the messages generated by the simulator itself. The simulator can take advantage of this fact and commit to a version of its own code that outputs the challenge r without receiving any additional auxiliary input. The problem is that in order to prove that such a trapdoor is valid, the simulator must construct a proof arguing about its own execution. Constructing such a proof may involve multiple levels of recursive proof construction. Since the overhead of constructing proofs for computations is polynomial, the simulator's running time quickly becomes exponential.

Our main idea is to use bounded concurrent simulation technique to "flatten" the recursion tree, avoiding the blowup in the simulator's running time. Next we describe our idea in more detail. We start by assigning a *level* to each session. All sessions where the length of the challenge r is $2\ell \cdot n^i$ are assigned level i. Our protocol is defined such that for every i, the total number of sessions at all levels $\leq i$ is at most n^i. It follows that in every session at level i, the verifier's challenge is long enough to account for all the messages received by the verifier in sessions at levels $\leq i$. To deal with the messages sent in sessions at levels $> i$, the simulator will commit to a specific *part* of its own code.

Let us elaborate further. The simulator Sim is divided into multiple components $\{\mathsf{Sim}_i\}$ where the i'th component Sim_i is in charge of simulating sessions at level i. To simulate a session at level i, Sim_i will commit to a program Π_i that contains the verifier's code together with the code of all the simulator's components Sim_j for $j > i$. We can think of the program Π_i as a new verifier that simulates all sessions at levels $> i$ internally and forwards externally the messages in sessions at level $\leq i$. Since sessions at level i use a challenge of length $2\ell \cdot n^i$ and the total number of sessions at levels $< i$ is at most n^i, we have that Sim_i can encode all the messages sent to Π_i as auxiliary input.

Simulator's running time. Finally, we argue that the running time of the simulator is polynomial. Using the analysis of the bounded concurrent protocol, we have that the running time of the component Sim_i is polynomial in the running time of the program Π_i. Since Π_i simply emulates all the simulator components Sim_j for $j > i$, we have that the running time of Sim_i is only polynomially larger then the total running time of all the components Sim_j for $j > i$. Since the total number of concurrent sessions started by an efficient

adversary is bounded by some polynomial n^c, we get that the total number of levels is constant and therefore the running time of all the simulator's components is bounded by a polynomial.

Avoiding circular use of randomness. We note that by using the above leveled simulation strategy we do not only avoid the blowup in the simulator's running time, but also avoid some of the technical complications that arise when the simulator commits to its own code. For example, in [CLP13a, Goy13, CLP12], the simulator needs to commit to it's code together with the randomness that it will use to simulate the rest of the protocol. These works develop additional techniques to deal with this problem. In our setting since every component only commits to the randomness used by the *higher* level component, such circular use of randomness is avoided, resulting in simpler protocol and analysis.

Taking advantage of terminating sessions. A natural requirement from a protocol in the committed-server model is that as existing sessions terminate and the load on the server decreases, the complexity of the protocol in new sessions decreases as well. We note that extending our simulation strategy to satisfy this requirement is not straight-forward. The problem is that our simulation strategy assumes that for every session at level i, the total number of concurrent sessions at levels $\leq i$ is bounded by n^i. However, imagine that all the sessions at levels $\leq i$ terminates and a new session starts. Then, if we choose to decrease the protocol complexity in the new session, then the total number of sessions at levels $\leq i$ may exceed n^i. We show how to decrease the complexity of new sessions as sessions terminate, while preserving overall simulatability.

1.3 On Secure Computation in the Client-Server Setting

The Committed-Server Model. In light of our positive result for the zero-knowledge functionality in the committed-server model, a natural next step is to try to achieve concurrently secure computation. Indeed, in light of the strong impossibility results in the plain model [CF01, CKL03, Lin04, BPS06, Goy12, AGJ$^+$12, GKOV12], achieving feasibility results for concurrently secure computation in weaker (but still realistic) models is well motivated.

We observe that existing impossibility results for concurrently secure computation, in fact, carry over to the committed-server model. In essence, these results state that the communication complexity of a protocol that is secure with respect to m concurrent executions, must be at least m [BPS06, Goy12, AGJ$^+$12, GKOV12]. Therefore, in the committed-server model, once the server commits to executing a protocol with communication complexity m, an adversary can simply start more than m new sessions.

The Non-Committed Server Model. Due to the above impossibility, we revisit to the security model of Persiano and Visconti [PV05], where the server can dynamically increase the complexity of the protocol *during* a session. In this model, that we refer to as the *non-committed server* model, we show that it is, in fact, possible to overcome the known impossibility results and obtain positive results for concurrent two-party computation. In our eyes, this further motivates the study of concurrent security in the client-server setting.

We start by formalizing the non-commited server model. We observe that the framework of [Lin03, PR03, Pas04] for constructing concurrently-secure protocols can be easily adapted to the non-committed server model. Then, applying their framework to the zero-knowledge protocol of [PV05] yields (fully) concurrently secure two-party computation protocols. In fact, the client-server setting enables further simplification over existing protocols, since we are only interested in the concurrent security of the server. We refer the reader to Appendix A for details, and conclude the present discussion by giving some brief intuition on why the non-committed server model enables bypassing the impossibility results for concurrently secure computation in the plain model.

The main technical problem that arises in the setting of secure computation w.r.t. unbounded concurrency in the plain model is the following: Since the number of sessions is unbounded, we cannot a priori bound

the total number of protocol output messages across all sessions. Further, the session outputs cannot be "predicted" by the simulator before knowing the adversary's input. As such, known non-black-box simulation techniques that either rely on bounded-concurrency arguments (e.g., [Bar01]) or "committed-simulator" strategies (e.g., [CLP13a, Goy13, CLP12]) cannot handle these unbounded number of messages and inherently fail. In contrast, in the non-committed server model, the server's ability to dynamically increase the protocol complexity during a session naturally opens doors to the design of protocols where simulation can always be performed by relying on a bounded-concurrency argument. This is exemplified in the protocol of [PV05], and the same intuition carries over to the setting of secure computation as well.

1.4 Related Works

Optimistic Concurrent Zero Knowledge. The work of Rosen and Shelat [RS10] also studies the round complexity of concurrent zero-knowledge proofs in the client-server setting. Similarly to [PV05], Rosen and Shelat work in the non-committed server model. However, they focus on improving the round complexity of concurrent ZK w.r.t."optimistic" adversarial schedules. Another important difference from [PV05] is that their protocol has a fixed bound on the communication complexity that is independent of the adversary.

In contrast, we construct concurrent zero-knowledge protocol in the committed-server model that achieves *constant-round* complexity, even for worst-case adversarial schedules.

Concurrent ZK in the Plain Model. Improving the round-complexity of concurrent zero-knowledge proofs in the plain model has been an active area of research. Starting from [RK99], the round-complexity of concurrent zero-knowledge w.r.t. black-box simulation was improved in a series of works [KP01, PRS02], with the best known result [PRS02] achieving logarithmic round-complexity (which is essentially optimal [CKPR02]).

Recently, Chung et al. [CLP13b] give a beautiful construction of a constant-round concurrent ZK protocol in the standard model from a non-standard (but falsifiable) assumption. Gupta and Sahai [GS12] present a different constant-round construction based on a knowledge assumption. In this work, however, we focus on achieving constant-round concurrent ZK in the client-server setting based on standard hardness assumptions.

We note that our technical tools are similar to the ones in [CLP13b]. In particular, both works use the "committed-simulator" technique along with ideas from bounded-concurrency to avoid blowup in simulation time. Beyond this, we do not know of any deeper connections between our work and [CLP13b].

2 The Committed-Server Model

In this section we formally describe the committed server model. We start by describing the general syntax and the model of communication. We then consider the specific case of zero-knowledge proof systems in the committed server model and present a security definition for the same.

Let Server be interactive PPT machine that interacts with multiple clients in concurrent sessions and let $\{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}_{\ell \in \mathbb{N}}$ be a family of protocols parameterized by a *load parameter* ℓ where for every $\ell \in \mathbb{N}$, $\mathcal{S}_\ell$ and $\mathcal{C}_\ell$ are PPT machines. A protocol in the committed server model is defined by the tuple $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$.

(Honest) Protocol Execution. The execution of a protocol $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$ in the committed server model consists of a single server executing the algorithm Server while interacting with multiple clients concurrently. To initiate a new session a client sends a special *session initiation message* to the starver. In response to the session initiation message, the server chooses a load parameter ℓ for the session and sends it to the client. In the rest of the session we require that the algorithm Server follows the strategy $\mathcal{S}_\ell$ while the client follows the strategy $\mathcal{C}_\ell$.

An execution of the protocol Π with $p(n)$ sessions is defined by the randomness of all the clients and the schedule of messages across all the sessions. Even though for every fixed load parameter ℓ, the strategies $\mathcal{S}_\ell, \mathcal{C}_\ell$ are efficient, the server algorithm may choose ℓ to be very large, increasing the running time of the concurrent execution. Therefore we explicitly require the efficiency of a concurrent execution.

Definition 2.1. *A protocol* $(\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$ *in the committed server model is efficient if for every polynomial* p *there exists another polynomial* q *such that the running time of* Server *in every execution with* $p(n)$ *sessions is bounded by* $q(n)$.

Zero Knowledge in the Committed Server Model. Let $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$ be a protocol in the committed server model. We say that Π is an interactive proof system for an NP language $\mathcal{L}$ if for every $\ell \in \mathbb{N}$, the protocol $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$ is an interactive proof for $\mathcal{L}$. We now define a zero-knowledge proof system in the committed server model.

Let Π be an interactive proof for language $\mathcal{L}$. Consider a concurrent adversary V^* that, given input $x \in \mathcal{L}$, interacts with the server in an execution of Π consisting of an unbounded (polynomial) number of concurrent sessions. For simplicity we assume that the statement x is the same in all sessions. We assume that V^* controls the scheduling of the messages across all the sessions. Let $\mathsf{View}_{V^*}(x, z)$ be the random variable describing the output of V^* executed with auxiliary input z.

Definition 2.2 (Concurrent Zero Knowledge in the Committed-Server Model)**.** *Let* $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$ *be an interactive proof system for language* $\mathcal{L}$ *in the committed server model. We say that* Π *is* zero knowledge *if for every PPT concurrent adversary* V^* *there exists a PPT algorithm* $\mathcal{S}$*, such that the following ensembles are computationally indistinguishable,*

$$\{\mathsf{View}_{V^*}(x, z)\}_{x \in \mathcal{L}, z \in \{0,1\}^*} \approx_c \{\mathcal{S}(x, z)\}_{x \in \mathcal{L}, z \in \{0,1\}^*}.$$

3 Constant-Round Zero-Knowledge in the Committed-Server Model

In this section we describe a constant-round ZK protocol $\Pi_{\mathsf{zk}} = (\mathsf{Server}, \{\langle P_\ell, V_\ell \rangle\})$ in the committed-server model. We start by defining a family of protocols $\{\langle P_\ell, V_\ell \rangle\}_{\ell \in \mathbb{N}}$ where, roughly speaking, the protocol $\langle P_\ell, V_\ell \rangle$ is simply Barak's bounded-concurrent ZK protocol [Bar01] with n^ℓ as the a priori bound on the number of sessions. We then define the server algorithm Server to complete the description of Π_{zk}.

The protocol $\langle P_\ell, V_\ell \rangle$**.** The protocol will make use of the following primitives: a statistically binding commitment Com, a family $\mathcal{H} = \{\mathcal{H}_n\}_{n \in \mathbb{N}}$ of collision-resistant hash functions such that $h \in \mathcal{H}_n$ maps strings in $\{0,1\}^*$ to strings in $\{0,1\}^n$, and a witness-indistinguishable universal argument UA for an **NTIME**$(T(n))$-complete language where $T : \mathbb{N} \to \mathbb{N}$ is a "slightly" super-polynomial function, for example $T(n) = n^{\log n}$ [BG08]. In the description of the protocol, the length of the verifier's messages will depend on a parameter m that denotes the total length of the *prover's* messages in the protocol.

Remark 3.1. The relation $\mathcal{L}_U$ presented in Protocol 1 is slightly oversimplified. For this relation, we can prove the security of Protocol 1 when $\mathcal{H}$ is collision-resistant against "slightly" super-polynomial sized circuits. For simplicity of exposition, in this manuscript, we will work with this assumption. We stress, however, that as discussed in several prior works (see e.g., [BG08]), this assumption can be removed by using an appropriate error-correcting code.

The server algorithm Server**.** We start by describing a simple server algorithm that only assigns monotonically increasing values of the load parameter to new sessions. In Section 3.2, we describe a better server algorithm that decreases the load parameter when some of the concurrent sessions terminate.

The algorithm Server maintains a variable SessionCount that counts the number of concurrent sessions started so far. Whenever a client initiates a new session, Server increases the value of SessionCount. When a

Common Input: $x \in \mathcal{L}$.

Auxiliary Input to P: A witness w for $x \in \mathcal{L}$.

Initiation Stage:

V_ℓ samples $h \leftarrow \mathcal{H}_n$ and sends h to P_ℓ.

Preamble Stage:

1. P_ℓ sends $c = \mathsf{Com}(h(0^n))$ to V_ℓ.
2. V_ℓ samples $r \leftarrow \{0,1\}^{n^\ell \cdot m + n}$ and sends r to P_ℓ.

Proof Stage:

P_ℓ and V_ℓ execute the protocol UA where P_ℓ proves that $x \in \mathcal{L} \vee (h, c, r) \in \mathcal{L}_U$.

The language $\mathcal{L}_U$ is defines as follows: $(h, c, r) \in \mathcal{L}_U$ iff there exist a program $\Pi \in \{0,1\}^*$, a string $y \in \{0,1\}^*$, and randomness s for Com such that:

1. $|y| \leq |r| - n$.
2. $c = \mathsf{Com}(h(\Pi); s)$.
3. $\Pi(y)$ outputs r within $T(n)$ steps.

Figure 1: Protocol Family $\langle P_\ell, V_\ell \rangle$ for ZK in the Committed-Server Model (Protocol 1)

new clients sends a session initiation message to the server, Server sets the load parameter ℓ for that session such that $n^{\ell-1} \leq \mathsf{SessionCount} \leq n^\ell$.

3.1 Analysis of Π_{zk}

The proof that for every $\ell \in \mathbb{N}$, the protocol $\langle P_\ell, V_\ell \rangle$ is complete and sound, follows directly from the analysis of the bounded-concurrent ZK protocol in [Bar01]. In this section we first show that for every $\ell \in \mathbb{N}$, Protocol 1 is efficient according to Definition 2.1. We then show that Π_{zk} is ZK in the committed-server model.

Protocol 1 is efficient. Let p be a polynomial and let ℓ_{max} be such that for large enough values of n, $p(n) < n^{\ell_{\mathsf{max}}}$. By the definition of the server algorithm Server, in an execution with $p(n)$ sessions, the load parameter of every session is at most ℓ_{max}. Since the running time of P_ℓ only grows with ℓ, we have that the running time of Server in every session is at most the running time of $P_{\ell_{\mathsf{max}}}$ and therefore the total running time of Server is bounded by a polynomial that depends only on p.

Protocol Π_{zk} is ZK in the committed-server model. Let V^* be a malicious verifier that starts at most $n^{\ell_{\mathsf{max}}}$ sessions for some constant ℓ_{max}. By the definition of the server algorithm Server, the load parameter of every session in an honest execution is at most ℓ_{max}. We construct a simulator $\mathsf{Sim} = (\mathsf{Sim}_{\mathsf{load}}, \{\mathsf{Sim}_\ell\})$ consisting of $\mathsf{Sim}_{\mathsf{load}}$ and ℓ_{max} other components $\{\mathsf{Sim}_\ell\}_{\ell \in \ell_{\mathsf{max}}}$. Roughly speaking, the component $\mathsf{Sim}_{\mathsf{load}}$ simulates the servers responses to the clients session initiation message in all sessions. The component Sim_ℓ simulates all the executions of $\langle P_\ell, V_\ell \rangle$ in sessions with load parameter ℓ. We now give more details.

The component $\mathsf{Sim}_{\mathsf{load}}$. This component simulates the server's responses to the clients session initiation message in all sessions. This simulation involves assigning a load parameter for every session started by V^*. Since the honest server Server selects the load parameter in each session based only on the (public)

adversarial scheduling, $\mathsf{Sim}_{\mathsf{load}}$ can use the exact same algorithm as Server, resulting in a perfect simulation of these messages.

The component Sim_ℓ. This component simulates the interaction of $\langle P_\ell, V_\ell \rangle$ in all the sessions with load parameter ℓ. At a high-level, the simulation will follow the simulation strategy of Barak's bounded-concurrent ZK protocol [Bar01]. According to this strategy, the simulator sends a commitment c to the code of the verifier and then uses this code as a trapdoor witness, proving that c is commitment to a code Π that outputs the random string r sent by the verifier. All the messages simulated in concurrent sessions are given to Π as auxiliary input. The main problem is that in order to guarantee that the protocol is sound, the program Π is only allowed to get an auxiliary input of bounded length; however, the number of concurrent sessions in our setting are not bounded.

We fix this problem in the following manner. Instead of simply committing to the code of V^*, Sim_ℓ will commit to a program V^*_ℓ that includes the code of V^* as well as the code of the simulation components $\mathsf{Sim}_{\mathsf{load}}$ and $\mathsf{Sim}_{\ell+1}, \ldots, \mathsf{Sim}_{\ell_{\mathsf{max}}}$. Roughly speaking, the program V^*_ℓ will simulate all the sessions with load parameter $\ell' > \ell$ internally, and therefore Sim_ℓ will need to provide as auxiliary input only the messages of concurrent sessions where the load parameter is at most ℓ. It follows from the description of Server that the number of concurrent sessions where the load parameter is at most ℓ is bounded by some polynomial (that depends on ℓ). Therefore, it is possible to include all of these messages as an auxiliary input to V^*_ℓ.

Next we formally describe the simulator component Sim_ℓ, starting with the definition of the program V^*_ℓ.

The program V^*_ℓ. V^*_ℓ is an interactive algorithm that includes the code of V^* together with the code of the simulation components $\mathsf{Sim}_{\mathsf{load}}$ and $\mathsf{Sim}_{\ell+1}, \ldots, \mathsf{Sim}_{\ell_{\mathsf{max}}}$. V^*_ℓ uses the same randomness as Sim to execute V^* and all the other simulation components. V^*_ℓ will emulate the execution of V^*, and will use the mentioned simulator components to internally simulate the responses to the session initiation messages in all sessions as well the prover messages of the protocols $\langle P_{\ell'}, V_{\ell'} \rangle$ executed in the sessions with load parameter $\ell' > \ell$. In the sessions with load parameter $\ell' \leq \ell$, V^*_ℓ will forward the messages of the protocol $\langle P_{\ell'}, V_{\ell'} \rangle$ externally.

In every session with load parameter ℓ, Sim_ℓ will simulate the execution of $\langle P_\ell, V_\ell \rangle$ as follows:

1. Sim_ℓ receives the description of a hash function h from V^*.
2. Sim_ℓ sends a commitment c to the hash of the code of a program Π that given auxiliary input $y = (m_1, \ldots, m_t)$, emulates an execution of V^*_ℓ when receiving the messages $m_1, \ldots, m_t$, and outputs V^*_ℓ's next message.
3. Sim_ℓ receives the the random string r from V^*.
4. Sim_ℓ sends a UA proof using a trapdoor witness that contains the code of the program Π and an appropriate auxiliary input string y. The string y is a list of all the prover messages that were simulated by Sim in all sessions with load parameter at most ℓ and sent before V^* sent the random string r in the present session.

Next we show that Sim_ℓ constructs a valid witness for the statement $(h, c, r) \in \mathcal{L}_U$. This amounts to proving that $\Pi(y)$ outputs r and that $|y| \leq |r| - n$. We also need to show that the running time of $\Pi(y)$ is at most $T(n)$. We will show that the last statement is correct when we analyze the running time of the simulation.

Proof that $\Pi(y)$ outputs r. The program $\Pi(y)$ outputs the next message of V^*_ℓ given the external messages in y. V^*_ℓ emulates V^* using the same randomness as Sim. Is is left to show that the messages sent to V^* emulated by V^*_ℓ and by Sim are identical. Recall that the messages sent to V^* in the execution emulated by V^*_ℓ are as follows: in sessions with load parameter larger than ℓ, the messages are generated by the internal simulation of V^*_ℓ, and the messages sent in sessions with load parameter at most ℓ are specified in y. For sessions with load parameter larger then ℓ, the messages sent to V^* in the emulation of V^*_ℓ and of Sim are

identical since they are generated using the same simulation algorithm and using the same randomness (by the construction of V_ℓ^*). For sessions with load parameter at most ℓ, the messages sent to V^* in the emulation of V_ℓ^* and of Sim are identical by the way the auxiliary input string y is constructed.

Proof that $|y| \leq |r| - n$. The auxiliary input string y constructed by Sim_ℓ contains only prover messages in sessions with load parameter at most ℓ. By the definition of the server algorithm Server there could be at most n^ℓ such sessions, and the total length of all the prover messages in every session is bounded by the parameter m. Therefore we have $|y| \leq n^\ell \cdot m$. Since V_ℓ samples $r \in \{0,1\}^{n^\ell \cdot m + n}$ we have that $|y| \leq |r| - n$.

Proof that the simulated view and the real view are indistinguishable. For $0 \leq \ell \leq \ell_{\mathsf{max}}$, let H_i be the hybrid experiment that is identical to the execution of Sim except that the executions of protocols $\langle P_{\ell'}, V_{\ell'} \rangle$ for $\ell' \leq \ell$ follow the honest prover strategy using a valid witness for $x \in \mathcal{L}$. Since $\mathsf{Sim}_{\mathsf{load}}$ simulates the responses to the sessions initiation messages perfectly we have that:

$$H_{\ell_{\mathsf{max}}} = \mathsf{View}_{V^*}(x, z), \quad H_0 = \mathcal{S}(x, z) \ .$$

It is therefore sufficient to prove that for every $0 \leq \ell < \ell_{\mathsf{max}}$, $H_\ell \approx_c H_{\ell+1}$. By the definition of the server algorithm Server, the number of sessions with load parameter ℓ is at most n^ℓ. For $0 \leq i \leq n^\ell$, let $H_{\ell,i}$ be the hybrid experiment that is identical to H_ℓ except that the first i executions of the protocol $\langle P_\ell, V_\ell \rangle$ follow the honest prover strategy using a valid witness for $x \in \mathcal{L}$. It follows that:

$$H_{\ell,n^\ell} = H_{\ell+1}, \quad H_{\ell,0} = H_\ell \ .$$

It is therefore sufficient to prove that for every $0 \leq i < n^\ell$, $H_{\ell,i} \approx_c H_{\ell,i+1}$.

Let $H'_{\ell,i}$ be the hybrid experiment that is identical to the $H_{\ell,i}$ except that the execution of the witness-indistinguishable universal argument UA in the proof stage of the i^{th} execution of the protocol $\langle P_\ell, V_\ell \rangle$ uses a valid witness for $x \in \mathcal{L}$ instead of the trapdoor witness. Note that in an execution of Sim, the randomness of the component Sim_ℓ used for the UA prover executed in the proof stage of the protocol $\langle P_\ell, V_\ell \rangle$ is also used by the components $\mathsf{Sim}_{\ell'}$ for $\ell' < \ell$ in the construction of the program $V_{\ell'}^*$. However, in the experiment $H_{\ell,i}$, all the simulator components $\mathsf{Sim}_{\ell'}$ for $\ell' < \ell$ are replaced by executions of the honest prover. Since the randomness of the component Sim_ℓ used for the simulation of the UA prover in the protocol $\langle P_\ell, V_\ell \rangle$ is not used in any other part of the simulation, it follows from the indistinguishability property of UA that $H_{\ell,i} \approx_c H'_{\ell,i}$.

Note that the experiment $H_{\ell,i+1}$ is identical to the experiment $H'_{\ell,i}$ except that in the experiment $H_{\ell,i+1}$, the prover commitment c given in the preamble stage of the i'th execution of the protocol $\langle P_\ell, V_\ell \rangle$ is a commitment to the all zero string, following the honest prover strategy. As before, the randomness of the component Sim_ℓ used for the simulation of c sent in the protocol $\langle P_\ell, V_\ell \rangle$ is not used in any other part of the simulation and therefore it follows from the computational-hiding property of Com that $H_{\ell,i+1} \approx_c H'_{\ell,i}$.

Overall we have that for every $0 \leq \ell \leq \ell_{\mathsf{max}}, 0 \leq i \leq n^\ell$, $H_{\ell,i} \approx_c H_{\ell,i+1}$. Since $\ell \leq \ell_{\mathsf{max}}$ is a constant, n^ℓ is a polynomial and therefore we have that for every $0 \leq \ell \leq \ell_{\mathsf{max}}$, $H_{\ell+1} \approx_c H_\ell$ and also that $H_{\ell_{\mathsf{max}}} \approx_c H_0$ as required.

The simulation is polynomial time. It is enough to show that all components of Sim are polynomial time. Since $\mathsf{Sim}_{\mathsf{load}}$ just follows the honest server algorithm, the efficiency of $\mathsf{Sim}_{\mathsf{load}}$ follows from the efficiency of the protocol. For every $\ell \in [\ell_{\mathsf{max}}]$ we show that the running time of Sim_ℓ is bounded by a polynomial in the security parameter (that depends on ℓ and on V^*). Since Sim_ℓ constructs the program Sim_ℓ, commits to its code, and provides a UA proof of its execution, the running time of Sim_ℓ is polynomial in the size and running time of V_ℓ^*. Additionally, since Sim_ℓ reads the entire transcript of the execution and uses it to construct the auxiliary input y in every session it simulates, the running time of Sim_ℓ is polynomial in the total length of the transcript. Note that the total length of the transcript is always bounded by the running time of V^* which is polynomial in the security parameter.

We start by bounding the running time of $\mathsf{Sim}_{\ell_{\mathsf{max}}}$. The program $V^*_{\ell_{\mathsf{max}}}$ only consists of the code of V^* and the code of $\mathsf{Sim}_{\mathsf{load}}$ and therefore, the running time of $V^*_{\ell_{\mathsf{max}}}$ is a polynomial. It follows that the running time of $\mathsf{Sim}_{\ell_{\mathsf{max}}}$ is also a polynomial. Now, for every $\ell \in [\ell_{\mathsf{max}}]$, the program V^*_ℓ only consists of the code of V^*, the code of $\mathsf{Sim}_{\mathsf{load}}$, and the code of $\mathsf{Sim}_{\ell'}$ for every $\ell \leq \ell' < \ell_{\mathsf{max}}$. Since ℓ_{max} is a constant depending only on V^*, and assuming that for all $\ell \leq \ell' < \ell_{\mathsf{max}}$ the running time of every $\mathsf{Sim}_{\ell'}$ is polynomial, the running time of V^*_ℓ and therefore also of Sim_ℓ must be polynomial. By induction we have that for every $\ell \in [\ell_{\mathsf{max}}]$ the running time of Sim_ℓ is bounded by a polynomial, and therefore the entire simulation is polynomial time.

Using the above proof, we complete the proof that Sim_ℓ constructs a valid trapdoor witness. Sim_ℓ constructs a program Π and auxiliary input y, and we need to show that the running time of $\Pi(y)$ is bounded by some super-polynomial function $T(n)$. The running time analysis above implies that for every $\ell \in [\ell_{\mathsf{max}}]$, the running time of V^*_ℓ and the size of the auxiliary input y constructed by Sim_ℓ are polynomial. The simulator component Sim_ℓ constructs a program Π that simulates V^*_ℓ sending it messages from y. It follows that the running time of $\Pi(y)$ is polynomial and therefore bounded by $T(n)$.

3.2 Decreasing the Load Parameter

In this section, we describe a different server algorithm that takes into account the termination of sessions and decreases the load parameter for new sessions accordingly. We start by describing the new server algorithm Server', and then describe the required changes to the simulation.

We identify the technical condition required for the simulation to work, and design a server algorithm Server' that always gives new sessions the lowest possible load parameter such that the technical condition still satisfies. The validity of our simulation relies on the validity of the following technical condition: for a session with load parameter ℓ_i, the number of sessions concurrent to it with load parameters at most ℓ_i is bounded by n^i. Before describing the algorithm Server' let us first introduce some notation. Let t be the number of open sessions at the moment a new client sends its session initiation message. For $i \in [t]$, let ℓ_i be the load parameter for the i'th open session. For $i \in [t]$, let t_i be the total number of sessions with load parameters at most i that are concurrent to session i. First note that if we set the load parameter of the new session to ℓ then for every session i such that $\ell_i \geq \ell$, the value t_i increases by 1. This will contradict the technical condition only if the value of t_i was already at its maximal allowed value n^{ℓ_i}.

Using the above notation, the algorithm Server' is easy to describe: Server' will set the load parameter of a new session to be the minimal value ℓ such that for every session i with $\ell_i \geq \ell$ we have $t_i < n^{\ell_i}$. While the behavior of the server algorithm Server' is not obvious, we can prove that it satisfies some natural conditions. For example we can show that if no sessions with load parameter ℓ are currently active, then the load parameter assigned to the next session to start cannot exceed ℓ.

Modifying the simulator. Next we discuss the necessary changes to the simulator. In the current description of the simulator, every program V^*_ℓ that Sim commits to, internally emulates V^* starting from its initial state. As a result, we must give V^*_ℓ auxiliary input z that consists of the messages in all concurrent sessions with load parameter at most ℓ starting from the beginning of the concurrent execution. The problem is that the definition of the server algorithm Server' does not grantee that such auxiliary input z is sufficiently short. Instead it only gives a bound on the number sessions with load parameter at most ℓ that are executed *concurrently* to the current session. In particular, Server' does not grantee anything about the number of sessions that terminated before the current session had started. The solution is based on the observation that providing V^*_ℓ auxiliary input z that contains messages sent before the current session had started is wasteful. Instead, Sim can commit the a program $\tilde{V^*}_\ell$ that already contains these messages hardwired into it.

References

[AGJ+12] Shweta Agrawal, Vipul Goyal, Abhishek Jain, Manoj Prabhakaran, and Amit Sahai. New impossibility results for concurrent composition and a non-interactive completeness theorem for secure computation. In *CRYPTO*, pages 443–460, 2012.

[Bar01] Boaz Barak. How to go beyond the black-box simulation barrier. In *FOCS*, pages 106–115, 2001.

[BG08] Boaz Barak and Oded Goldreich. Universal arguments and their applications. *SIAM J. Comput.*, 38(5):1661–1694, 2008.

[BL02] Boaz Barak and Yehuda Lindell. Strict polynomial-time in simulation and extraction. In *STOC*, pages 484–493, 2002.

[BPS06] Boaz Barak, Manoj Prabhakaran, and Amit Sahai. Concurrent non-malleable zero knowledge. In *FOCS*, pages 345–354, 2006.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, pages 19–40, 2001.

[CKL03] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In *EUROCRYPT*, pages 68–86, 2003.

[CKPR02] Ran Canetti, Joe Kilian, Erez Petrank, and Alon Rosen. Black-box concurrent zero-knowledge requires (almost) logarithmically many rounds. *SIAM J. Comput.*, 32(1):1–47, 2002.

[CLP12] Kai-Min Chung, Huijia Lin, and Rafael Pass. Constant-round concurrent zero knowledge from falsifiable assumptions. *IACR Cryptology ePrint Archive*, 2012:563, 2012.

[CLP13a] Ran Canetti, Huijia Lin, and Omer Paneth. Public-coin concurrent zero-knowledge in the global hash model. In *TCC*, pages 80–99, 2013.

[CLP13b] Kai-Min Chung, Huijia Lin, and Rafael Pass. Constant-round concurrent zero knowledge from p-certificates. In *FOCS*, 2013.

[DGS09] Yi Deng, Vipul Goyal, and Amit Sahai. Resolving the simultaneous resettability conjecture and a new non-black-box simulation strategy. In *FOCS*, pages 251–260, 2009.

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple noninteractive zero knowledge proofs under general assumptions. *SIAM J. Comput.*, 29(1):1–28, 1999.

[GJO+13] Vipul Goyal, Abhishek Jain, Rafail Ostrovsky, Silas Richelson, and Ivan Visconti. Concurrent zero knowledge in the bounded player model. In *TCC*, pages 60–79, 2013.

[GKOV12] Sanjam Garg, Abishek Kumarasubramanian, Rafail Ostrovsky, and Ivan Visconti. Impossibility results for static input secure computation. In *CRYPTO*, pages 424–442, 2012.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, 1987.

[Goy12] Vipul Goyal. Positive results for concurrently secure computation in the plain model. In *FOCS*, pages 41–50, 2012.

[Goy13] Vipul Goyal. Non-black-box simulation in the fully concurrent setting. In *STOC*, pages 221–230, 2013.

[GS12] Divya Gupta and Amit Sahai. On constant-round concurrent zero-knowledge from a knowledge assumption. *IACR Cryptology ePrint Archive*, 2012:572, 2012.

[KP01] Joe Kilian and Erez Petrank. Concurrent and resettable zero-knowledge in poly-loalgorithm rounds. In *STOC*, pages 560–569, 2001.

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

[Lin04] Yehuda Lindell. Lower bounds for concurrent self composition. In *TCC*, pages 203–222, 2004.

[Pas04] Rafael Pass. Bounded-concurrent secure multi-party computation with a dishonest majority. In *STOC*, pages 232–241, 2004.

[PR03] Rafael Pass and Alon Rosen. Bounded-concurrent secure two-party computation in a constant number of rounds. In *FOCS*, pages 404–413, 2003.

[PRS02] Manoj Prabhakaran, Alon Rosen, and Amit Sahai. Concurrent zero knowledge with logarithmic round-complexity. In *FOCS*, pages 366–375, 2002.

[PRT13] Rafael Pass, Alon Rosen, and Wei-Lung Dustin Tseng. Public-coin parallel zero-knowledge for np. *J. Cryptology*, 26(1):1–10, 2013.

[PV05] Giuseppe Persiano and Ivan Visconti. Single-prover concurrent zero knowledge in almost constant rounds. In *ICALP*, pages 228–240, 2005.

[RK99] Ransom Richardson and Joe Kilian. On the concurrent composition of zero-knowledge proofs. In *EUROCRYPT*, pages 415–431, 1999.

[Ros04] Alon Rosen. A note on constant-round zero-knowledge proofs for np. In *TCC*, pages 191–202, 2004.

[RS10] Alon Rosen and Abhi Shelat. Optimistic concurrent zero knowledge. In *ASIACRYPT*, pages 359–376, 2010.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

A Concurrent Security in the Non-Committed Server Model

A.1 The Non-Committed Server Model

In this section we formally describe the non-committed server model. As in the committed server model, a protocol in the non-committed server model is defined by a server algorithm $\mathcal{S}$ and a protocol family $\{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}_{\ell \in L}$ parameterized by a load parameter $\ell \in L$. Additionally, the protocol definition includes a tree structure over the family $\{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}$, indicating which protocols in the family have a common prefix. Concretely, let $\mathcal{T}$ be a tree such that any leaf of $\mathcal{T}$ corresponds to a protocol $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$ in the family. The tree $\mathcal{T}$ satisfies the following two requirements:

- For every protocol $\pi \in \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}$ the round complexity of π is equal to the depth of the leaf corresponding to π in the tree $\mathcal{T}$.
- For every two protocols $\pi_i, \pi_j \in \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}$ if the leaves corresponding to π_i and π_j have a common ancestor of depth p then the protocols π_i, π_j have a common prefix of p rounds. That is, the player strategies in the first p rounds of π_i and π_j are identical.

It follows from the above requirements that a client and the server can agree on a protocol in the family $\{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}$ by specifying a path in the tree $\mathcal{T}$ from the root to one of the leaves. Furthermore, specifying only a prefix of this path of length p, already determines client and server strategies up to the p-th round. Overall, A protocol in the non-committed server model is defined by the tuple $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}, \mathcal{T})$.

(Honest) Protocol Execution. As in the committed server model, the execution of a protocol $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\}, \mathcal{T})$ in the non-committed server model consists of a single server executing the algorithm Server while interacting with multiple clients concurrently. Unlike in the committed model, the server does not send a load parameter to the client after receiving the session initiation message. Instead, the server incrementally reveals a path in the tree $\mathcal{T}$ from starting from the root and ending in a leaf that corresponds to some protocol $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$. In the first round of every session, the client sends a session initiation message and the server responds with the first edge of the path (starting from the root). Based on the server message in the fist round the client know what strategy to follow in the first round of $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$. In the p-th round of the session, the server sends the $(p-1)$-th server message of the protocol $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$ together with the p-th edge of the path in $\mathcal{T}$. Based on the server message in the p-th round the client know what strategy to follow in the p-th round of $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$. The protocol ends when the server sends the last edge of the path (an edge that ends with a leaf of $\mathcal{T}$).

We define the load parameter of the session to be the load parameter of the protocol that corresponds to the leaf sent by the server in the last round of the session. Note that since the server algorithm Server does not commit to a session's load parameter at the beginning of the execution, it is possible for Server to decide on the value of the session's load parameter dynamically during the session's execution.

Zero Knowledge in the Non-Committed Server Model. The definition of ZK protocols in the non-committed server model is analogous to Definition 2.2 of ZK in the committed server model. We omit the details.

A.2 Secure Computation in the Non-Committed Server Model

In this section, we present the security definition for secure computation in the non-committed server model, as per the standard real/ideal paradigm. The definition we present below is an adaptation of [Lin04] to the client-server setting, with static inputs (i.e., we assume that the inputs of the honest parties are fixed at the beginning).

We consider a malicious, static adversary that chooses whom to corrupt before the start of any protocol. Note that in the client-server model, one can consider two corruption models:

1. The adversary may either corrupt a subset (or all) of the clients who are interacting concurrently with a single server in multiple sessions. In this case, we are interested in the concurrent security of the server.
2. Alternatively, the adversary may corrupt the server. In this case, we are only interested in stand-alone security of each client against the dishonest server.

For clarity of exposition, we only focus on defining concurrent security of the server against adversarial clients. Stand-alone security of the clients can be defined in the standard way, we omit its discussion here.

Finally, we remark that we do not consider fairness and hence in the ideal model, we allow a corrupt party to receive its output in a session and then optionally block the output from being delivered to the honest party, in that session.

We now proceed to define concurrent security for the server. We start by describing the ideal and real world experiments and then give our security definition.

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 a server $\mathcal{S}$ that is involved in multiple evaluations of f with, say $m = m(n)$ clients $\mathcal{C}_1, \ldots, \mathcal{C}_m$.[1] Let $\mathcal{B}$ denote the adversary. The ideal world execution proceeds as follows.

I. Inputs: The server $\mathcal{S}$ receives its input vector $\vec{x}$ for all the sessions. Similarly, each client $\mathcal{C}_i$ receives its input y_i for session i. Without loss of generality, we assume that $\mathcal{B}$ corrupts all of the clients. In this case, $\mathcal{B}$ receives the input vector $\vec{y}$ of the clients.

II. Initiation: Whenever $\mathcal{B}$ wishes to initiate a session i, it sends a $(\mathsf{start\text{-}session}, i)$ message to the trusted party. Upon receiving this message, the trusted party sends $(\mathsf{start\text{-}session}, i)$ to $\mathcal{S}$.

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

IV. Adversary sends input to trusted party and receives output: Whenever $\mathcal{B}$ wishes, it may send a message (i, y_i') to the trusted party for any y_i' of its choice. Upon sending this pair, it receives back $(i, f(x_i, y_i'))$ where x_i is the input of $\mathcal{S}$ for session i.

V. Adversary instructs trusted party to answer honest party: When $\mathcal{B}$ sends a message of the type (output, i) to the trusted party, the trusted party sends $(i, f(x_i, y_i'))$ to $\mathcal{S}$, where x_i and y_i' denote the respective inputs sent by $\mathcal{S}$ and $\mathcal{B}$ for session i.

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

The output of the ideal experiment with security parameter n, input vectors $\vec{x}$, $\vec{y}$ and auxiliary input z to the adversary $\mathcal{B}$, denoted $\text{IDEAL}_{f,\mathcal{B}}(n, \vec{x}, \vec{y}, z)$, is defined as the output pair of the server and $\mathcal{B}$ from the above ideal execution.

REAL MODEL. We now consider the real model in which real two-party protocols are executed no trusted third party exists. Let $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$ be a protocol in the committed server model such that every protocol $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$ computes f. Let $\mathcal{A}$ denote a PPT adversary that interacts with the server in an execution of Π consisting of an unbounded (polynomial) number of concurrent sessions. The adversary $\mathcal{A}$ controls the clients in all of the sessions. Further, we assume that V^* controls the scheduling of the messages across all the sessions. At the conclusion of the protocol, the server 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 output of the real experiment with security parameter n, protocol $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$, input vectors $\vec{x}$, $\vec{y}$ and auxiliary input z to $\mathcal{A}$, denoted $\text{REAL}_{\Pi,\mathcal{A}}(n, \vec{x}, \vec{y}, z)$, is defined as the output pair of the server and $\mathcal{A}$, resulting from the above real-world process.

SECURITY DEFINITION. Having defined the ideal and real models, we now give our security definition:

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

Definition A.1 (Concurrently Secure Computation in the Client-Server Model). *Let* $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\})$ *be a protocol in the non-committed server model s.t. every* $\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle$ *computes* f*. We say that* Π securely computes f *under concurrent self-composition if for every real model PPT adversary* $\mathcal{A}$*, there exists a PPT ideal world adversary* $\mathcal{B}$ *such that for every polynomial* $m = m(n)$*, every pair of input vectors* $\vec{x} \in X^m$*,* $\vec{y} \in Y^m$*, every* $z \in \{0,1\}*$*,*

$$\left\{\text{IDEAL}_{f,\mathcal{B}}(n, \vec{x}, \vec{y}, z)\right\}_{n\in\mathbb{N}} \approx_c \left\{\text{REAL}_{\Pi,\mathcal{A}}(n, \vec{x}, \vec{y}, z)\right\}_{n\in\mathbb{N}}$$

A.3 Our Protocol

In this section, we give the construction of a secure two-party computation protocol in the non-committed server model. Our construction follows the framework of [Lin03, PR03, Pas04] for constructing (bounded) concurrently secure computation protocols, without setup assumptions.

We start a brief overview of the main ideas underlying the framework, adapted to the client-server model, and then proceed to the details.

Overview. Let Π_{tpc} be any (stand-alone) two-party computation protocol that is secure against semi-honest adversaries. Then, roughly speaking, the basic idea is to "compile" Π_{tpc} with two special-purpose zero-knowledge proof of knowledge protocols – $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ and $\Pi^{\mathcal{C}}_{\mathsf{zk}}$ – to obtain a new two-party computation protocol Π that provides concurrent security for the server against adversarial clients. In the compiled protocol, $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ is used by the server to give proofs to the client, while $\Pi^{\mathcal{C}}_{\mathsf{zk}}$ is used by the client to give proofs to the server. We will denote the client ZK protocol as $\Pi^{\mathcal{C}}_{\mathsf{zk}} = (\mathsf{Server}^{\mathcal{C}}_{\mathsf{zk}}, \left\{\langle P^{\mathcal{C}}_\ell, V^{\mathcal{C}}_\ell \rangle\right\}, \mathcal{T}^{\mathcal{C}}_{\mathsf{zk}})$, the server ZK protocol as $\Pi^{\mathcal{S}}_{\mathsf{zk}} = (\mathsf{Server}^{\mathcal{S}}_{\mathsf{zk}}, \left\{\langle P^{\mathcal{S}}_\ell, V^{\mathcal{S}}_\ell \rangle\right\}, \mathcal{T}^{\mathcal{S}}_{\mathsf{zk}})$, and the resultant protocol $\Pi = (\mathsf{Server}, \left\{\langle \mathcal{S}_\ell, \mathcal{C}_\ell \rangle\right\}, \mathcal{T})$.

In order to prove that Π is a secure two-party computation protocol in the non-committed server model, we require the following properties from $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ and $\Pi^{\mathcal{C}}_{\mathsf{zk}}$:

Properties of $\Pi^{\mathcal{S}}_{\mathsf{zk}}$*:*

1. Simulation: $\Pi^{\mathcal{S}}_{\mathsf{zk}} = (\mathsf{Server}^{\mathcal{S}}_{\mathsf{zk}}, \left\{\langle P^{\mathcal{S}}_\ell, V^{\mathcal{S}}_\ell \rangle\right\}, \mathcal{T}^{\mathcal{S}}_{\mathsf{zk}})$ should be, first and foremost, a (concurrent) ZK protocol in the non-committed server model. But moreover, it should be possible to simulate each session in $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ *even during the execution of* Π.[2]

2. Extraction: Further, every $\langle P^{\mathcal{S}}_\ell, V^{\mathcal{S}}_\ell \rangle$ should be a (standard) proof-of-knowledge.

Properties of $\Pi^{\mathcal{C}}_{\mathsf{zk}}$*:*

1. Simulation: Every $\langle P^{\mathcal{C}}_\ell, V^{\mathcal{C}}_\ell \rangle$ must satisfy the stand-alone ZK property.

2. Extraction: We require $\Pi^{\mathcal{C}}_{\mathsf{zk}}$ to be a *concurrent* proof-of-knowledge in the non-committed server model. More specifically, we require that $\Pi^{\mathcal{C}}_{\mathsf{zk}}$ enables extraction of the prover's witness in *every* session when used within the two-party computation protocol Π.[3]

3. Non-malleability: Finally, we want that every $\langle P^{\mathcal{C}}_\ell, V^{\mathcal{C}}_\ell \rangle$ is "non-malleable" w.r.t. every $\langle P^{\mathcal{S}}_\ell, V^{\mathcal{S}}_\ell \rangle$. More specifically, we want that $\langle P^{\mathcal{S}}_\ell, V^{\mathcal{S}}_\ell \rangle$ remains *sound* even when an adversarial prover is simultaneously receiving a simulated proof via $\langle P^{\mathcal{S}}_\ell, V^{\mathcal{S}}_\ell \rangle$ in another session.[4]

We now outline the main steps for constructing the desired ZK protocols $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ and $\Pi^{\mathcal{C}}_{\mathsf{zk}}$:

Constructing $\Pi^{\mathcal{S}}_{\mathsf{zk}}$*:*

[2] Intuitively, this property requires that $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ does not only support concurrent self-composition, but also a form of *general composition* with specific protocols – in our case, they correspond to Π.

[3] Intuitively, this property will be used to extract the input of the adversarial client in each session of Π.

[4] Note that we do not require the converse property since we are only interested in the stand-alone security of a client.

1. We first construct a zero-knowledge protocol $\Pi_{\mathsf{zk}}^{\mathcal{S},(1)}$ in the non-committed server model. Such a protocol was in fact already constructed by Visconti and Persiano [PV05]. We recall their construction below in Section A.3.1.

2. Next, we slightly modify $\Pi_{\mathsf{zk}}^{\mathcal{S},(1)}$ to derive a new protocol $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$. The protocol $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ will have the property that every session (corresponding to an execution of $\langle P_\ell, V_\ell \rangle$) in $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ can be simulated even during the execution of Π.

3. Finally, we apply the transformation of [BL02] on $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ (using a standard extractable commitment scheme [Ros04, BL02]) to derive $\Pi_{\mathsf{zk}}^{\mathcal{S}}$ such that $\Pi_{\mathsf{zk}}^{\mathcal{S}}$ is also a proof of knowledge .

Constructing $\Pi_{\mathsf{zk}}^{\mathcal{C}}$*:*

1. Our starting point will be the standard Blum's ZK protocol for Graph Hamiltonicity. We will denote it as $\langle P, V \rangle$.

2. Next, we construct a *concurrently* extractable commitment scheme $\Pi_{\mathsf{com}} = (\mathsf{Server}_{\mathsf{com}}, \{\langle C, R \rangle\}, \mathcal{T}_{\mathsf{com}})$ from $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ using the construction of [BL02]. Finally, we apply the transformation of [BL02] on $\langle P, V \rangle$, using Π_{com}, to derive $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ such that $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ is also a concurrent proof of knowledge in the non-committed server model.

We now proceed to give further details. The rest of this section is organized as follows. In Section A.3.1, we describe a zero-knowledge proof system in the non-committed server model. Then, in Section A.3.2, we discuss the remaining steps to obtain a (concurrently) secure two-party computation protocol in the non-committed server model.

A.3.1 Zero-Knowledge in the Non-Committed Server Model

We first recall the ZK protocol of [PV05] adapted to our syntax for the non-committed server model. We will denote the protocol by $\Pi_{\mathsf{zk}} = (\mathsf{Server}_{\mathsf{zk}}, \{\langle P_\ell, V_\ell \rangle\}, \mathcal{T}_{\mathsf{zk}})$. We start by defining a family of protocols $\{\langle P_\ell, V_\ell \rangle\}_{\ell \in \mathbb{N}}$. We then define the protocol tree $\mathcal{T}_{\mathsf{zk}}$ and the server algorithm $\mathsf{Server}_{\mathsf{zk}}$ to complete the description of Π_{zk}.

The protocol $\langle P_\ell, V_\ell \rangle$. The protocol will make use of a statistically binding commitment Com, a family $\mathcal{H} = \{\mathcal{H}_n\}_{n \in \mathbb{N}}$ of collision-resistant hash functions and a witness-indistinguishable universal argument UA for an **NTIME**$(T(n))$-complete language, as defined earlier for Protocol 1.

Let $\mathsf{CC}^{\mathcal{S}}(\mathrm{UA})$ be the polynomial upper bound on the length of the prover messages in UA. Further, $\forall i \in [\mathbb{N}]$, let $\mathsf{len}^i(n) = \mathsf{CC}^{\mathcal{S}}(\mathrm{UA}) \cdot n^i + n$. $\langle P_\ell, V_\ell \rangle$ is described as Protocol 2.

Protocol tree $\mathcal{T}_{\mathsf{zk}}$. From the definition of Protocol 2, we have that for every $i, j \in \mathbb{N}$ such that $i > j$, protocols $\langle P_i, V_i \rangle$ and $\langle P_j, V_j \rangle$ are identical in the initiation stage and the first j iterations of the preamble stage. Therefore, the common ancestor of the nodes corresponding to $\langle P_i, V_i \rangle$ and $\langle P_j, V_j \rangle$ in the tree $\mathcal{T}_{\mathsf{zk}}$ is of depth $2j + 1$ (one round for the initiation stage and two rounds for every iteration of the preamble stage). These constraints define the tree $\mathcal{T}_{\mathsf{zk}}$.

Note that in the non-committed model, the server is allowed to choose after every iteration of the preamble stage whether it wishes to start the proof stage or perform another iteration of the preamble stage.

The server algorithm $\mathsf{Server}_{\mathsf{zk}}$. Similar to the zero-knowledge protocol in the committed model (see Section 3), the algorithm $\mathsf{Server}_{\mathsf{zk}}$ will maintain a variable $\mathsf{SessionCount}$ that counts the number of "active" concurrent sessions. We describe the strategy of $\mathsf{Server}_{\mathsf{zk}}$ for every session.

Consider any session between the server and a client where i iterations of the preamble stage have been completed. $\mathsf{Server}_{\mathsf{zk}}$ now sends a node node in $\mathcal{T}_{\mathsf{zk}}$ to the client according to the following strategy:

Common Input: $x \in \mathcal{L}$.

Auxiliary Input to P: A witness w for $x \in \mathcal{L}$.

Initiation Stage:

V_ℓ samples $h \leftarrow \mathcal{H}_n$ and sends h to P_ℓ.

Preamble Stage:

For $i = 1$ to ℓ, do the following:

1. P_ℓ sends $c_i = \mathsf{Com}(h(0^n))$ to V_ℓ.
2. V_ℓ samples $r_i \leftarrow \{0,1\}^{\mathsf{len}^i(n)}$ and sends r_i to P_ℓ.

We refer to the above two messages as a "slot".

Proof Stage:

P_ℓ and V_ℓ execute the protocol UA where P_ℓ proves the OR of the following statements:

1. $x \in \mathcal{L}$.
2. $\exists i \in [\ell]$ s.t. $(h, c_i, r_i) \in \mathcal{L}_U$, where $\mathcal{L}_U$ is as defined in Protocol 1.

Figure 2: Protocol Family $\langle P_\ell, V_\ell \rangle$ for ZK in the Non-Committed Server Model (Protocol 2)

- If $\mathsf{SessionCount} < i$, then node indicates that the next protocol round corresponds to the first round of the proof stage.
- Otherwise, node indicates that the next protocol round corresponds to the $(i+1)^{th}$ iteration of the preamble stage.

This completes the description of $\Pi_{\mathsf{zk}} = (\mathsf{Server}_{\mathsf{zk}}, \{\langle P_\ell, V_\ell \rangle\}, \mathcal{T}_{\mathsf{zk}})$. We refer the reader to [PV05] for the proof that Π_{zk} is a ZK protocol in the non-committed server model.

A.3.2 Completing the Construction of Π

We now give a sketch of the main steps towards the construction of Π. We refer the reader to [Lin03, PR03, Pas04] for details of all the steps.

STEP I. Constructing $\Pi^{\mathcal{S}}_{\mathsf{zk}}$:

Let $\Pi^{\mathcal{S},(1)}_{\mathsf{zk}} = (\mathsf{Server}^{\mathcal{S},(1)}_{\mathsf{zk}}, \left\{\langle P^{\mathcal{S},(1)}_\ell, V^{\mathcal{S},(1)}_\ell \rangle\right\}, \mathcal{T}^{\mathcal{S},(1)}_{\mathsf{zk}})$ denote the zero-knowledge protocol in the non-committed server model, as described in Section A.3.1. We now describe the steps involved in obtaining $\Pi^{\mathcal{S}}_{\mathsf{zk}}$ from $\Pi^{\mathcal{S},(1)}_{\mathsf{zk}}$.

From $\Pi^{\mathcal{S},(1)}_{\mathsf{zk}}$ to $\Pi^{\mathcal{S},(2)}_{\mathsf{zk}}$. We first describe the necessary modifications to transform $\Pi^{\mathcal{S},(1)}_{\mathsf{zk}}$ to $\Pi^{\mathcal{S},(2)}_{\mathsf{zk}} = (\mathsf{Server}^{\mathcal{S},(2)}_{\mathsf{zk}}, \left\{\langle P^{\mathcal{S},(2)}_\ell, V^{\mathcal{S},(2)}_\ell \rangle\right\}, \mathcal{T}^{\mathcal{S},(2)}_{\mathsf{zk}})$. The server algorithm $\mathsf{Server}^{\mathcal{S},(2)}_{\mathsf{zk}}$ and the protocol tree $\mathcal{T}^{\mathcal{S},(2)}_{\mathsf{zk}}$ are defined in the same manner as $\mathsf{Server}^{\mathcal{S},(1)}_{\mathsf{zk}}$ and $\mathcal{T}^{\mathcal{S},(1)}_{\mathsf{zk}}$, respectively. Further for every load parameter ℓ, protocol $\langle P^{\mathcal{S},(2)}_\ell, V^{\mathcal{S},(2)}_\ell \rangle$ is identical to $\langle P^{\mathcal{S},(1)}_\ell, V^{\mathcal{S},(1)}_\ell \rangle$, except for the length parameters $\{\mathsf{len}^i(n)\}$, as we describe below.

Let $\mathsf{CC}^{\mathcal{S}}(\Pi_{\mathsf{tpc}})$ denote the total communication complexity of the server's messages in the semi-honest secure two-party computation protocol Π_{tpc} (that we will use in the construction of Π). Further, let $\mathsf{RC}(\Pi_{\mathsf{tpc}})$

denote the round complexity of Π_{tpc}. Let $\mathsf{CC}^{\mathcal{S}}(\langle P, V\rangle)$ denote the total communication complexity of the verifier's messages in Blum's zero-knowledge protocol $\langle P, V\rangle$ (that we will describe below). Then, we will use the following length parameters for protocol $\langle P_\ell^{\mathcal{S},(2)}, V_\ell^{\mathcal{S},(2)}\rangle$:

$$\forall i \in [\ell], \mathsf{len}^i(n) = \left(\mathsf{RC}\left(\Pi_{\mathsf{tpc}}\right) \cdot \left(2\mathsf{CC}^{\mathcal{S}}(\mathrm{UA}) + \mathsf{CC}^{\mathcal{S}}\left(\langle P, V\rangle\right)\right) + \mathsf{CC}^{\mathcal{S}}\left(\Pi_{\mathsf{tpc}}\right)\right) \cdot n^i + n$$

Intuitively, by setting the length parameter $\mathsf{len}^i(n)$ to be as above, we can now simulate a session of $\langle P_\ell^{\mathcal{S},(2)}, V_\ell^{\mathcal{S},(2)}\rangle$ even when n^i sessions of Π are contained inside the i^{th} preamble slot of $\langle P_\ell^{\mathcal{S},(2)}, V_\ell^{\mathcal{S},(2)}\rangle$.

From $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ to $\Pi_{\mathsf{zk}}^{\mathcal{S}}$. We now describe the necessary steps to transform $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ to the desired ZK protocol $\Pi_{\mathsf{zk}}^{\mathcal{S}} = (\mathsf{Server}_{\mathsf{zk}}^{\mathcal{S}}, \left\{\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle\right\}, \mathcal{T}_{\mathsf{zk}}^{\mathcal{S}})$. As discussed earlier, the only additional property required in $\Pi_{\mathsf{zk}}^{\mathcal{S}}$ (over $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$) is the (standard) proof-of-knowledge property. This is achieved by using the standard transformation of [BL02] from proof of membership to proof of knowledge, as used in [Lin03, PR03, Pas04].

Very briefly, let COM denote a standard extractable-commitment scheme [Ros04, BL02] that supports extraction of the committed value from the committer (either by means of rewinding [Ros04] or in a "straight-line" manner [BL02]) in polynomial time. Then, given COM, we simply apply the transformation of [BL02] on every $\langle P_\ell^{\mathcal{S},(2)}, V_\ell^{\mathcal{S},(2)}\rangle$ (which is a proof of membership) to obtain $\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle$ (which is a proof of knowledge). We refer to [BL02] for details.

STEP II. Constructing $\Pi_{\mathsf{zk}}^{\mathcal{C}}$:

Let $\langle P, V\rangle$ denote the classical Blum's protocol for Graph Hamiltonicity. We describe the steps involved in obtaining $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ from $\langle P, V\rangle$.

Concurrently Extractable Commitments. Our first step is to construct a concurrently extractable commitment scheme Π_{com}. Specifically, we require that it is possible to extract the committed value in every session in a "straight-line" manner, even when Π_{com} is executed within Π. We construct such a scheme Π_{com} by plugging in the ZK protocol $\Pi_{\mathsf{zk}}^{\mathcal{S},(2)}$ in the construction of [BL02].

From $\langle P, V\rangle$ and Π_{com} to $\Pi_{\mathsf{zk}}^{\mathcal{C}}$. Finally, we apply the transformation of [BL02] using Π_{com} to transform $\langle P, V\rangle$ (which is a proof of membership) into $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ (which is a proof of knowledge). $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ will have the same extraction properties as Π_{com}.

Non-malleability of $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ w.r.t. $\Pi_{\mathsf{zk}}^{\mathcal{S}}$. We require that the soundness of $\langle P_\ell^{\mathcal{C}}, V_\ell^{\mathcal{C}}\rangle$ holds even when an adversarial prover is receiving a simulated proof via $\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle$. We reduce this property to the stand-alone soundness of $\langle P_\ell^{\mathcal{C}}, V_\ell^{\mathcal{C}}\rangle$ by leveraging the strong simulation property of $\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle$. Very briefly, suppose for contradiction that $\langle P_\ell^{\mathcal{C}}, V_\ell^{\mathcal{C}}\rangle$ is not sound in the aforementioned sense. Then, we can forward the adversarial proof to an external verifier. Note that due to the strong simulation property of $\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle$, it is still possible to simulate the proof received by the adversary, and therefore we will be able to break the stand-alone soundness of $\langle P_\ell^{\mathcal{C}}, V_\ell^{\mathcal{C}}\rangle$, which is a contradiction. We refer the reader to [Lin04, PR03, Pas04] for more details.

STEP III. Final Compilation:

Finally, having constructed the special-purpose ZK protocols $\Pi_{\mathsf{zk}}^{\mathcal{S}}$ and $\Pi_{\mathsf{zk}}^{\mathcal{C}}$, protocol $\Pi = (\mathsf{Server}, \{\langle \mathcal{S}_\ell, \mathcal{C}_\ell\rangle\}, \mathcal{T})$ is obtained by compiling Π_{tpc} with $\Pi_{\mathsf{zk}}^{\mathcal{S}}$ and $\Pi_{\mathsf{zk}}^{\mathcal{C}}$ in the same manner as [Lin04, PR03]. We omit the details here, but remark that since each $\langle \mathcal{S}_\ell, \mathcal{C}_\ell\rangle$ will consist of multiple instances of $\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle$, the load parameter for $\langle \mathcal{S}_\ell, \mathcal{C}_\ell\rangle$ will actually be a *vector* of load parameters, consisting of load parameters of all the instances of $\langle P_\ell^{\mathcal{S}}, V_\ell^{\mathcal{S}}\rangle$ used in $\langle \mathcal{S}_\ell, \mathcal{C}_\ell\rangle$.