Public-Coin Concurrent Zero-Knowledge in the Global Hash Model*

Ran Canetti[1,2], Huijia Lin[1,3], and Omer Paneth[1]

[1] Boston University,
[2] Tel Aviv University
[3] MIT

Abstract. *Public-coin zero-knowledge* and *concurrent zero-knowledge (*cZK*)* are two classes of zero knowledge protocols that guarantee some additional desirable properties. Still, to this date no protocol is known that is both public-coin and cZK for a language outside BPP. Furthermore, it is known that no such protocol can be black-box ZK [Pass et.al, Crypto 09].

We present a public-coin concurrent ZK protocol for any NP language. The protocol assumes that all verifiers have access to a globally specified function, drawn from a collision resistant hash function family. (This model, which we call the Global Hash Function, or GHF model, can be seen as a restricted case of the non-programmable reference string model.) We also show that the impossibility of black-box public-coin cZK extends also to the GHF model.

Our protocol assumes CRH functions against quasi-polynomial adversaries and takes $O(\log^{1+\epsilon} n)$ rounds for any $\epsilon > 0$, where n is the security parameter. Our techniques combine those for (non-public-coin) black-box cZK with Barak's non-black-box technique for public-coin constant-round ZK. As a corollary we obtain the first simultaneously resettable zero-knowledge protocol with $O(\log^{1+\epsilon} n)$ rounds, in the GHF model.

1 Introduction

Zero-knowledge (ZK) proofs and arguments are protocols that enable a prover to convince a verifier in the verity of a statement without revealing any information other than the fact that the statement is true. This is captured by requiring that for any efficient adversarial verifier there exists an efficient simulator that, knowing only whether the statement is correct, essentially recreates the adversary's view of the entire execution. ZK protocols are a fundamental building block in cryptographic protocols and applications; furthermore, the techniques used to construct ZK protocols often evolve and percolate to protocols for other cryptographic tasks.

* Supported by an ISF grant, NSF grant 1218461, the Check Point Institute for Information Security and the Center for Reliable Information Systems and Cyber-Security.

The first ZK protocols by [16, 15] and others have a very simple form, where the verifier's messages consist only of random strings with no additional structure. In the end of the protocol the verifier evaluates a deterministic predicate of the communication. The simplicity of this *public-coin*, or *Arthur-Merlin* structure is indeed attractive in of itself; in addition it has been shown over the years to have many other advantages, such as public verifiability, amenability to delegation, and better resilience to leakage [12, 4]. (In fact, we make use of some of these advantages in this work.)

However, it also soon became clear that obtaining stronger efficiency and security properties for ZK protocol while preserving the simple public-coin structure is challenging. One such parameter is the number of rounds: The basic protocols of [16, 15] take take super-logarithmic number of rounds — essentially, via sequential repetition of a basic building block that gives soundness error of one half. The first protocols that obtain a constant number of rounds have the verifier *commit* to its randomness ahead of time, thus losing the PC property [13]. Furthermore, [14] show that no constant rounds public-coin ZK protocol with negligible error probability can be proven secure via black-box simulation. A protocol public-coin ZK protocol with constant number of rounds came only years later and uses a completely new proof technique, which indeed involved non-black-box simulation [1].

Another security property that appears to stand at odds with public-coin ZK is parallel and concurrent ZK (cZK). Here we want the protocol to remain ZK even when the prover participates in many independent sessions for proving the same statement, and these sessions are scheduled in an adversarially controlled concurrent way. Also here known protocols are not public-coin ZK [23, 17, 22], and for a similar reason: an essential ingredient in these protocols is having the verifier commit to its randomness ahead of time. Furthermore, also here we know that no PC protocol can be proven to be concurrent (or even parallel) ZK via black-box simulation [21]. However, here we do not currently know of any way to get around this black-box impossibility result. In particular, the technique of [1] fails, at least in of itself. We are thus left with the question:

Do there exist public-coin concurrent zero-knowledge protocols?

A first indication that the answer might be positive was given by Pass, Rosen and Tseng [20], who construct a public-coin *parallel* ZK protocol. That is, their protocol (which is a relatively simple adapration of the [1] protocol) remains ZK even under parallel composition. However, their security analysis falls apart in the general concurrent setting.

We provide a positive answer to this question in the general concurrent setting, albeit with a caveat: We consider a setting where all verifiers have access to a single hash function h. In that setting, we design a public-coin protocol and show that this protocol is cZK, unless it is possible to efficiently find collisions in h. That is, we show how to efficiently construct a simulator, given an adversary, and then provide an *explicit efficient reduction* that turns an adversary that breaks the cZK property of the protocol w.r.t. the constructed simulator into an algorithm that finds collisions in h. We call this model the global hash

function (GHF) model. See further discussion on the GHF model at the end of the Introduction. That is, we show:

Theorem (informal): Assuming existence of collision resistant hash function families against quasipolynomial adversaries, there exist public-coin cZK protocols in the GHF model. In contrast, there exist no black-box public-coin cZK protocols in the GHF model.

Round complexity. We present two public-coin cZK protocols. The first one has a polynomial number of rounds. The second one, which is considerably more involved, takes only $O(\log^{1+\epsilon} n)$ rounds for any $\epsilon > 0$, where n is the security parameter. This almost matches the best known round complexity for cZK, regardless of the public-coin property [22]. Recall that for black-box simulation this is the best possible [7]

Simultaneously resettable ZK in logarithmic rounds. A question that is very related to public-coin cZK is the question of *simultaneously resettable* ZK. Such ZK protocols remain secure even if a cheating party (playing the role of either the prover or the verifier) has the ability to repeatedly reset the honest party to its initial state and random tape, and interact with it several times. The only known simultaneously resettable ZK protocol [10] in the plain model has polynomial number of rounds. (In the bare public key model, a protocol with constant number of rounds is known [9].)

As a corollary of our main result we get a new simultaneously resettable ZK protocol in the GHF model with only logarithmic number of rounds. The simultaneously resettable ZK protocol is obtained from our public-coin cZK protocol by applying two generic transformations: first we apply the transformation of [21] to go from a public-coin cZK to a resettably-sound cZK protocol that is also sound against resetting provers. Then we can apply the transformation of [10] to get simultaneously resettable ZK. Both transformations do not increase the round complexity of the protocol.

Our techniques. In a nutshell, our protocols use the multiple-opportunity-slots simulation technique of the cZK protocols of [23, 17, 22] (which are inherently not public-coin) to make the public-coin protocol of [1] fully concurrent. In particular, in the context of the non-black-box simulator of [1], we generalize the concept of rewinding to re-running of certain portions of the simulation of the adversary's code.

The global hash function is used in the universal argument (UA) portion of the protocol of [1], allowing all instances of the UA in all concurrent sessions to the use the same hash function. This allows our simulator to amortize the work spent on preparing the universal arguments across multiple concurrent sessions.

On the global hash function model. We design and analyze our protocols in the global hash function (GHF) model, where all parties have access to a public hash function h, and the security of the protocol is argued by way of an

efficient and explicit reduction from an adversary that breaks the security of the protocol to an adversary that finds collisions in h. Results in this model can be interpreted in several alternative ways. One interpretation, in the spirit of [24], is that the protocol indeed uses a single and fixed hash function h (say, SHA2) and the security (in our case, cZK) property "in practice" is based on the inability of Mankind to find explicit collisions in h — although such collisions exist in principle and can be found "in polynomial time". Note that this interpretation makes sense both when security is formalized in an asymptotic way *and* in terms of concrete, non-asymptotic security guarantees.

Another interpretation of results in this model is that they guarantee security against uniform-complexity polytime adversaries, as long as the (single) global hash function used by the protocol is collision resistant in an asymptotic way against such adversaries. We note however that this interpretation is relatively weak. In particular, it is not clear how to translate it into concrete, non-asymptotic security guarantees.

Yet another interpretation of results in this model is that they guarantee security in the "global reference string model", where the reference string is randomly chosen and consists of the description of a hash function h drawn from a collision resistant hash function family. Here the zero-knowledge simulator has to work with a given h rather than making up its own one. In fact, the simulation should succeed even when the function h is chosen adversarially.

The GHF model for zero knowledge protocols should be contrasted with the common reference string (CRS) model used elsewhere in cryptography (e.g. for non-interactive and universally composable zero knowledge [5, 6]). Indeed, the models are quite different: In the CRS model the public reference string is chosen as part of the protocol execution, and a distinguisher between a real execution and an ideal has no a-priori information on that string. In particular, the CRS model provides no guarantees whatsoever when the reference string is chosen adversarially, or even when the adversary is allowed to see trapdoor information related to the reference string.

Furthermore, the impossibility of public-coin black-box cZK protocols extends to the GHF model, whereas in the CRS model such protocols are known to exist (in fact, any NIZK protocol is such).

Organization. This extended abstract contains only high level descriptions of our results as well as our protocols and it's proof. Detailed definitions, constructions and analysis are given in the full version of this paper [8].

2 Overview of Our Public-Coin cZK Protocol

In the black-box simulation world, there has been a rich set of constructions [23, 17, 22] of fully concurrent ZK protocols; however, these constructions are not public-coin. In fact, as shown in [21], this is inherent: only languages in BPP have public-coin black-box parallel ZK protocols (that is, protocols that remain ZK under parallel composition). In contrast, in the non-black-box simulation

world, known constructions [1, 20] are indeed public-coin; however, they are only ZK under composition with restricted concurrency (e.g., bounded concurrent composition, and parallel composition). Our construction can be viewed as "upgrading" the existing non-black-box simulation techniques to be fully concurrent, using the recursive the rewinding strategies from black-box cZK *while remaining public-coin.* We first a give quick overview of the current techniques and their limitations. Next, we present high-level ideas behind our construction.

2.1 Current Techniques

Public-coin ZK Protocols

Barak's Protocol. We briefly recall the idea behind Barak's protocol. Roughly speaking, for language L and common input $x \in \{0,1\}^n$, the prover P and verifier V proceed in three stages.

- *Stage 1:* V starts by sending P a function h chosen randomly from a family of collision-resistant hash functions.
- *Stage 2:* P sends a commitment $c \in \{0,1\}^n$ to 0; V follows by sending a uniformly random "challenge" $r \leftarrow \{0,1\}^n$; we informally refer to the pair of messages (c, r) as a *slot*, for reasons that will become clear later.
- *Stage 3:* P proves that either $x \in L$ or c is a commitment to a hash of a program Π such that $\Pi(c) = r$.

The proof of Stage 3 proceeds via a public-coin witness indistinguishable universal argument (UA) [2]. This is the crux of the protocol, and where all the Difficulties lie. A UA system has the crucial property that the verification time and communication complexity are independent of the length of the witness. Still, the prover's complexity grows with the length of the witness.

Soundness follows from the fact that even if a malicious prover P tries to commit to some program Π (instead of committing to 0), with high probability, the V's challenge r will be different from $\Pi(c)$. To prove ZK, consider the non-black-box simulator that sets c to be a commitment to the hash of the code of the malicious verifier V^*; note that by definition it holds that $\Pi(c) = V^*(c) = r$, and the simulator can use Π as a "fake" witness in the final proof.

Bounded concurrency. Barak's protocol can be extended to a bounded concurrent ZK protocol by slightly changing the UA statement proven in State 3, and allowing Π to receive, other than c, some additional auxiliary input. Soundness holds as long as the length of the auxiliary input is significantly shorter than $|r|$. Now, the simulator can complete the UA by proving that V^* on input c, *and having received all messages from other sessions before generating its second message*, outputs r. As long as the total number of concurrent sessions is bounded, r can be chosen to be longer than the total length of messages V^* might receive inside any slot. Therefore, the simulation goes through. However, this approach is inherently limited to the bounded concurrency setting. In the

unbounded concurrent setting, there is no a priori bound on the length of the messages that V^* receives. However, the protocol cannot allow the committed program Π to receive an arbitrarily long input, as otherwise soundness falls apart.

COMMITTING TO THE SIMULATOR'S CODE. One potential approach to circumvent the above limitation is having the simulator $\mathcal{S}$ commit to the code of *itself* (i.e., $\mathcal{S}$) instead of commiting to the code of V^*. The intuition behind this idea is that, although in the unbounded concurrent setting the length of the messages that V^* receives is unbounded, these messages are generated by the simulator $\mathcal{S}$, and thus can be shortly represented by the code of $\mathcal{S}$. Therefore, if the simulator $\mathcal{S}$ commits to a machine Π that emulates its own execution until the message r is simulated, it can again prove in the UA argument that $\Pi() = r$, since all the messages V^* receives will be generated by Π in emulation of $\mathcal{S}$. (Note that here we treat the simulator code as already including the code of V^* in some form.) Indeed, this idea is the main enabler in the public-coin parallel ZK protocol of Pass, Rosen and Tseng [20].

However, when moving to the concurrent setting, this technique runs into the problem that the running time of $\mathcal{S}$ grows exponentially with the number of "nested concurrent sessions". This problem is similar to the problem encountered by black-box simulation in the general, non-public-coin settings. In particular, this blow-up in simulation running time is demonstrated by the example of Dwork et. al [11]. To see the problem, consider a concurrent verifier $\mathcal{V}^*$ that starts two nested sessions, where session 1 is completely "enclosed" in the slot of session 2. In session 1, the simulator commits to a program Π_1 that emulates $\mathcal{S}$ until the challenge message in the first session r_1 is sent. $\mathcal{S}$ then completes the simulation of session 1 by proving that Π_1 outputs r_1. Similarly, in session 2, the simulator commits to a program Π_2 that emulates $\mathcal{S}$ until it simulates r_2. If Π_1 takes T steps to output r_1, then it takes $\mathcal{S}$ at least another T steps to give a UA argument that this is true. Therefore, in the second session, Π_2 takes at least $2T$ steps to output r_2 (since Π is emulating $\mathcal{S}$, it needs to simulate the entire first session including its UA proof before V^* outputs r_2) and the time for giving the UA argument in session 2 is at least $2T$. Overall, the simulation time is at least $4T$. As in the case of [11], it is not hard to see that with d levels of nesting (i.e., d sessions, with session i entirely enclosed in the slot of session $i+1$), the simulation time grows to at least 2^dT. (In fact, the situation is even worse since the prover complexity in the best UAs is at least $O(T \log T)$.)

We remark that the idea described above, as well as the problem of exponential time simulation, were already described by Deng,Goyal and Sahai [10] in the context of simultaneously resettable ZK. In their protocol, the simulator commits to the code of the adversarial verifier together with some parts of the code and state of the simulator. The exponential time simulation problem is resolved using a combination of new black-box and non-black-box simulation techniques. However, the resulting concurrent ZK protocol is not public-coin.

cZK Protocols The design of all existing cZK protocols follow the Feige-Lapidot-Shamir (FLS) paradigm: at the beginning of the protocol, the verifier sets up a "trapdoor" (e.g., by sending a commitment to a secret random value), followed by many invocations of a sub-protocol that hides the trapdoor, but allows a simulator to extract the trapdoor by rewinding some messages in the sub-protocol, referred to as a *slot*. Then, the prover proves, using a witness-indistinguishable proof, that either the statement is true or it knows the trapdoor. Roughly speaking, the protocol is ZK, since the simulator can extract the trapdoor via rewinding of any slot in the session and use it as a "fake" witness to "cheat" in simulation. The simulator will use a *rewinding strategy* to decide which slots to rewind in order to guarantee successful extraction of a trapdoor for each session in the concurrent setting.

The "recursive rewinding" problem. A good rewinding strategy of a cZK protocol needs to also guarantee that the time spent on rewinding is bounded. As observed already by [11], this turns out to be non-trivial and encounters a similar difficulty as the exponential-time simulation problem in the context of non-black-box simulation. To demonstrate the difficulty, consider a simplified protocol that has the structure describe above, but contains only one slot for rewinding, and a cheating verifier V^* that starts two nested sessions, where the first session is entirely enclosed in the slot of the second session. To simulate the second session, the simulator needs to rewind the slot in this session to extract a trapdoor; however, before V^* completes this slot, the simulator needs to first simulate messages in the first session for V^*, which requires it to recursively rewind the slot in the first session. This quickly leads to an exponential number of rewindings and the simulation time explodes.

Known black-box cZK protocols resolve this problem by having many sequential slots in the protocol, so that there are many extraction opportunities for the simulator. It is shown that when the number of slots is large enough, there are *recursive rewinding strategies* [23, 17, 22] that, by carefully choosing which parts of the execution to rewind, guarantee that the depth of nesting (i.e., the depth of recursive rewinding) is bounded and thus the simulation time is bounded. Below we recall the KP-PRS rewinding strategy, which will be useful for our construction later.

The KP-PRS rewiding strategy. The simulator of [17, 22] simulates the view of the cheating verifier in a "main thread", using the trapdoors extracted via many recursive rewindings also called "lookahead threads". The KP-PRS rewinding strategy tells the simulator which parts of the execution to rewind based on the transcript simulated so far. The simulation strategy is recursive since rewindings are also used during the simulation of lookahead threads.

In KP-PRS, the rewinding strategy decides when to rewind the verifier *obliviously* of the content of the simulated messages, depending only on the number of simulated messages. More specifically, it divides messages in the main thread (resp. lookahead threads) into blocks of 2^i messages. Then, at the end of each block, it recursively rewinds the verifier from the beginning of the block once;

by rewinding an entire block, the simulator rewinds all the slots contained in that block "in one shot". It has been shown in [18] that the KP-PRS rewinding strategy can be generalized to consider blocks of length b^i for $b > 2$. Intuitively, the KP-PRS rewinding strategy is efficient since rewindings are performed only at selected points (i.e., the end of blocks) and the depth of nesting is bounded by $O(\log_b n)$. Furthermore, as long as the number of slots is $\omega(b \log_b n)$, it is guaranteed that at the end of every session, a trapdoor would be extracted successfully.

2.2 Our Approach

At a very high-level, the recursive rewinding problem in the context of black-box simulation and the exponential time simulation problem in the context of non-black-box simulation are similar: both are caused by the recursive execution of the simulator's code. In the context of black-box simulation the problem can be solved by providing more slots. We show how to solve the exponential time simulation problem in the context of public-coin non-black-box simulation. Towards this, we introduce a non-black-box analog of "rewinding slots" and use these slots to manage the complexity of the simulation. To illustrate the idea, consider the following overly simplified protocol (P_0, V_0) which is a k-slot variant of Barak's protocol. Our solution will require to replace the UA in State 3 of the protocol with a new type of interactive argument we call a "*special proof*". The properties of the special proof and its construction is the focus of the rest of this section.

An Overly Simplified Protocol (P_0, V_0)**:**

- *Stage 1 (Hash Function Selection):* V sends P a randomly chose collision-resistant hash function $h \leftarrow \mathcal{H}$.
- *Stage 2 (k Slots):* This stage contains k sequential slots, where in the i^{th} slot the prover sends a commitment c_i and the verifier replies with a challenge r_i.
- *Stage 3 (Proof Stage):* The prover proves using a *special proof* that either $x \in \mathcal{L}$, or there is a slot i, in which c_i is a commitment to a hash of a program Π that outputs r_i.

The idea of committing to the simulator's code can be adapted to work with this protocol as follows: on the main thread, the simulator simulates the view of V^* in a straight line. In every slot, the simulator commits to a program Π that mimics the simulation of the main thread. When the simulation of a session reaches Stage 3, the simulator proves that there is a slot i with transcript (c_i, r_i) such that c_i is a commitment to a program that "predicts" the challenge r_i.

With many slots, the simulator now gains the freedom to choose which slot to use as a witness for the special proof in each session. Similarly to the case of black-box simulation, the simulator will use a *proving strategy* to choose which slot to use in the proof of every session. The simulation might still recursively prove statements about its own computation, however, the proving strategy will control the recursion depth and thus also the complexity of the simulation. To do

that, the proving strategy will reuse ideas from the black-box recursive rewinding strategies.

We start by spelling out the analogy between our situation and the case of black-box rewinding. At every slot, the simulator will commit to a program Π that mimics the execution of the main thread from the point in the simulation where the slot starts to the point where the slot ends. Now, the execution of Π can be thought of as analogous to the rewinding of the simulation in the slot. However, the non-black-box simulator does not directly execute Π. Instead, it generates a special proof about the execution of Π. Thus, the running time that is spent on the rewinding by the black-box simulator is spent by the non-black-box simulator on constructing a proof about the execution of Π. Similarly, constructing a UA proof for a program that recursively constructs proofs for other programs is analogous to recursive rewindings. Following this observation we will design a proving strategy based on the KP-PRS rewinding strategy.

A KP-PRS-style proving strategy. Roughly speaking, the simulator divides the messages in the main thread into blocks of length b^i (where b is a parameter of the simulation); at the end of every block, the simulator constructs special proofs for slots contained in the block, this corresponds to rewinding the block[4]. After constructing the special proof at the end of the block, the simulator can use it to simulate the proof stage (Stage 3) of the corresponding session.

To turn the overly simplified version above into a working protocol we need to overcome a number of obstacles, mostly related to the special proof in use. Below, we proceed in two steps: first we construct a relatively simple public-coin cZK protocol with $O(n^\epsilon)$ rounds (for any constant ϵ), and then improve the round complexity to $O(\log^{1+\epsilon} n)$ to obtain our final protocol.

2.3 An $O(n^\epsilon)$-Round Protocol

To realize the proposed KP-PRS-style proving strategy, we need to construct a "special proof" as described above. In this section, we describe the challenges in constructing such a proof and how to resolve them to get a $O(n^\epsilon)$-round public-coin cZK protocol.

Using UA as a special proof. The KP-PRS rewinding strategy crucially relies on the fact that rewindings are only performed at the end of blocks to show that the depth of nesting and the simulation time are bounded. Similarly, we will require that the time spent by the simulator on constructing special proofs will be spent only in the end of blocks. This rules out using standard UA as special proofs, since constructing a UA requires the simulator to interact with the verifier and get its random challenges. However, the concurrent verifier might schedule the UA that corresponds to a session within a block only long after the end of the block.

[4] In fact, the simulator only constructs proofs for sessions that haven't been "solved", that is, sessions for which no previous proof was constructed.

ONLINE/OFFLINE UA. To resolve this problem, we observe that the construction of UA in [2] can be separated into an *expensive offline stage* and an *efficient online stage* as follows. Let x be a statement that can be proven by a UA where the prover runs in time t. The first verifier message specifies a hash function h and is independent of the statement x. After the first message is sent, the prover's work can be separated into an expensive offline stage that runs in time at most t, and an efficient online stage that runs in a fixed polynomial time in $|x|$. More precisely, we separate the prover in the construction of [2] into an "offline prover" $\mathcal{P}_{\mathsf{UA-OFF}}$ and an "online prover" $\mathcal{P}_{\mathsf{UA}}$ as follows: in the offline stage the verifier specifies a hash function h. Then, the offline prover $\mathcal{P}_{\mathsf{UA-OFF}}$ on input x, witness w and the hash h, constructs a PCP proof σ and a Merkle hash tree HT of σ using h. Finally, $\mathcal{P}_{\mathsf{UA-OFF}}$ outputs the string $\pi = \sigma \| HT$ which we refer to as the offline UA proof. In the online stage, the online prover $\mathcal{P}_{\mathsf{UA}}$ is given x and *oracle access* to π. $\mathcal{P}_{\mathsf{UA}}$ first sends the root of the hash tree to "commit" to the PCP proof. Then, the verifier sends its PCP queries and $\mathcal{P}_{\mathsf{UA}}$ produces answers by querying π. $\mathcal{P}_{\mathsf{UA}}$ obtains the relevant bits of the PCP proof from σ and the corresponding authentication paths from HT. See Protocol 1 for a description of the offline and online stages of the UA.

Pubic Coin Online/Offline UA Argument System

Building Blocks: A family of collision-resistant hash functions $\mathcal{H}$. A PCP proof system $(\mathcal{P}_{\mathsf{PCP}}, \mathcal{V}_{\mathsf{PCP}})$ with properties as defined in [2]).
Inputs: Common input $x \in \mathcal{L}$, and auxiliary input $w \in \mathcal{R}_{\mathcal{L}}(x)$ to $\mathcal{P}_{\mathsf{UA-OFF}}$.

OFFLINE STAGE

$1^{\mathbf{st}}$ **Message:** *The verifier $\mathcal{V}_{\mathsf{UA}}$ sends a random hash function $h \leftarrow \mathcal{H}$.*

The offline prover $\mathcal{P}_{\mathsf{UA-OFF}}$ runs $\mathcal{P}_{\mathsf{PCP}}$ on input (x, w) to construct a PCP proof σ, and computes the Merkle hash tree HT of σ using h; let δ be the root of HT. We call $\pi = \sigma \| HT$ an offline UA proof string .

ONLINE STAGE*: The online prover $\mathcal{P}_{\mathsf{UA}}$ with* oracle access *to π and input x, interacts with $\mathcal{V}_{\mathsf{UA}}$ as follows:*

$2^{\mathbf{nd}}$ **Message:** The prover ${\mathcal{P}_{\mathsf{UA}}}^{\pi}$ sends δ.
$3^{\mathbf{rd}}$ **Message:** The verifier $\mathcal{V}_{\mathsf{UA}}$ sends a sufficiently long random string r.
$4^{\mathbf{th}}$ **Message:** The prover ${\mathcal{P}_{\mathsf{UA}}}^{\pi}$ runs $\mathcal{V}_{\mathsf{PCP}}$ on input (x, r) to generate a set of queries Q; for each query $q \in Q$, it sends σ_q and an authentication path in the Merkle hash tree HT that leads to σ_q.
$\mathcal{V}_{\mathsf{UA}}$**'s decision:** $\mathcal{V}_{\mathsf{UA}}$ accepts if all the authentication paths verify, and $\mathcal{V}_{\mathsf{PCP}}$ on input $(x, r, \{\sigma_q\}_{q \in Q})$ accepts.

Fig. 1. The UA construction of [2] as an online/offline UA

Using the online/offline UA in our protocol. In our cZK protocol, the verifier specifies a hash function in Stage 1, which will be used as the offline verifier's message in the UA. Now the simulator can apply the KP-PRS-style proving strategy: at the end of a block, the simulator constructs an offline UA proof for each slot contained in that block. When a session enters the proof stage (Stage 3), the simulator uses a previously constructed offline proofs to generate messages in the online stage of the UA arguments.

However, the proof given in Stage 3 of the protocol cannot simply be the online stage of the UA. To see way, recall that following the FLS paradigm, the proof stage should consist of a witness-indistinguishable proof that $x \in \mathcal{L}$ or that the prover obtained a trapdoor for one of the slots in the session. The problem is that the proving the above statement (or even stating it) requires knowing the messages sent in all the slots of the session. However, it might be that at the end of the block, when to simulator needs to construct a proof, some of the slots of the session were not simulated yet. To fix the problem we use the online/offline UA to construct a *"special-purpose" witness-indistinguishable* UA similar to the one constructed in [19, 2]. Recall that at the end of the block the simulator constructs an (expensive) offline UA proof for some slot. We change the proof stage of the protocol as follows: the prover first provides an online proof that it has a trapdoor for some slot in the session (note that this statement involves only a single slot). To keep the proof witness-indistinguishable, the proof must not reveal which slot is used. Therefore, the online stage of this UA is executed in the following "oblivious" manner: the prover commits to the statement it proves as well as to all of its online UA messages instead of sending them in the clear, while the verifier simply sends random coins (here we use the fact that the online UA is public-coin). The honest prover (that does not have any trapdoor) will just commit to the all-zero string in every round. We refer to this as an *oblivious* UA execution. Then, the prover will provide a standard witness-indistinguishable proof of knowledge (for NP) to prove that $x \in \mathcal{L}$ or that the committed online UA messages form an accepting proof transcript for the statement defined by one of the slots.

The problem of exponential size state. By separating the work of the UA prover into offline and online stages, the simulator has the freedom to construct a proof for a slot at any time and thus the KP-PRS-style proving strategy can be applied. However, here we encounter yet another difference between black-box and non-black-box simulation. In the former, after rewinding a slot successfully, the simulator extracts a *short* trapdoor of a fixed polynomial length, and thus can afford to remember all the trapdoors extracted so far and use them to complete the simulation of corresponding sessions in both the main and lookahead threads. In contrast, the non-black-box simulator does not obtain a short trapdoor; instead, it obtains *long* offline UA proofs (the length of which is not a priory bounded by some polynomial), of length proportional to the running time of the simulator when simulating the execution in a slot. Still, the simulator needs to remember all previously constructed offline proofs in order to simulate the online stage of the corresponding UAs in the main thread. This means that in each

slot, when the simulator commits to its own code and state, it commits also to a record of all the offline proofs constructed so far. Thus, the offline proof arguing about the execution of the slot will be at least as long as all previously generated offline proofs. Again we encounter the problem of exponential blowup in the size of the proof. This time, however, it is due to the size of the state kept by the simulator rather than due to the computation time.[5] To resolve this problem, we first observe that though an offline proof can be arbitrarily long, only a few (fixed polynomial number of) bits of this proof are accessed when simulating the online stage of the UA. If the simulator knew which bits in an offline proof would be accessed later, it could have committed to a program Π containing only these bits instead of the whole offline proof. Then the space complexity of Π would have been bounded by a fixed polynomial (depending only on the size of the cheating verifier), and the size of the offline proofs would not have grown exponentially. However, this wishful thinking seems doomed, since at the time when the simulator needs to commit to Π (i.e., when a slot opens), it does not know which bits of the proof would be accessed, since these bits depend on the verifier's queries sent in the proof stage.

A potential alternative strategy is the following: when a slot opens, the simulator simply commits to a program Π that does not contain any information about previously constructed offline proofs. Only later, when the simulator needs to prove that Π predicts the verifier's random challenge, it does so by providing the appropriate bits of the proof as an auxiliary input to Π. The simulator can do so because at the time of constructing an offline proof about Π, the slot in which Π is committed to is already completely simulated and the simulator knows which bits of previous offline proofs are accessed during the simulation of that slot. However, this strategy fails again. This is because the number of bits accessed in a slot can be an arbitrary polynomial that depends on the number of concurrent sessions started by the cheating verifier. However, for soundness to hold, it is crucial that the committed program only receives auxiliary input much shorter than the length of the verifier's random challenge, which is bounded by an a priori fixed polynomial.

A "HASH-INVERTING" ORACLE. We finally resolve this problem by combining the ideas behind the above two failed approaches: when a slot opens, the simulator commits to a program Π containing *a root of a Merkle hash* of each offline proof; later, the simulator proves that Π, when given appropriate proof bits *that are consistent with the roots*, predicts the verifier's challenge in the slot. For soundness to hold, the proof bits must be given to the program via a carefully defined interface. The interface we describe next is inspired by the non-black-box simulation technique of [10]. The program Π will be given access to a "hash-inversion" oracle that can "invert" the hash tree. That is, when the program Π wants to access the bit j of the offline proof P, it will query the oracle with the root δ_P of the hash tree of P and the index j. The oracle will answer with the

[5] In fact, the time complexity for constructing a UA offline proof and the length of the proof is at least quasi-linear in the space complexity of the computation.

bit $P[j]$ together with the authentication path certifying that $P[j]$ is consistent with δ_P. The oracle will only respond to the query if the value of the root δ_P is contained in the initial state of Π committed in the beginning of the slot.

Giving the committed program Π access to such an oracle is different than just giving it the proof bit as auxiliary input. Even though the number of answer bits Π can obtain from its oracle is not bounded by any fixed polynomial, soundness still holds. The intuition is that all the oracle's answers are "computationally determined" by the starting state of Π. A bit more formally, we prove that no computationally bounded algorithm can produce two valid oracles that answer differently to one of Π's queries. This guarantees that the information that Π learns from its oracle is independent of the verifier's challenge r that is chosen after Π's code and the hash tree roots are committed to.

We modify the protocol correspondingly: in the proof stage (Stage 3), the prover proves that either the statement is true, or that, in one of the slots, it has committed to a program Π that predicts the verifier's challenge, given access to a valid hash-inverting oracle as described. When the simulation reaches the end of a block where the simulator needs to construct a proof for the computation of a committed program of some slot, the simulator has all the information about what proof bits were accessed during the simulation of the slot. Therefore, the simulator can construct the appropriate oracle that Π expects to access. The main difference between the oracle described above and the oracle used in [10] is that in [10], the oracle's answers are information-theoretically determined by the queries, whereas here, answers of the hash-inverting oracle are only computationally determined. However, as we show, in our settings this suffices for achieving soundness.

The global hash model. In the description of the modified protocol above, the committed program is given access to a "hash-inverting" oracle. However, we did not specify how to choose the hash function inverted by the oracle. For soundness to hold, the hash function must not be specified by the prover, as otherwise, a cheating prover may specify a hash function with respect to which the hash tree roots are not binding. However, letting the verifier choose the hash function results in a problem with concurrent simulation: let h_i be the hash function specified by the cheating verifier in the i^{th} session. Now, when the simulation commits to a program Π in a slot of the i^{th} session, Π must contain the roots of Merkle hash trees using hash h_i for all previously constructed offline proofs. Otherwise, Π will not be able to query its oracle for bits of these proofs. It follows that whenever the cheating verifier starts a new session and sends a new hash function h_i, the simulator must recompute the hash tree on all previously constructed offline proofs using h_i. This operation may be as expensive as constructing all these offline proofs from scratch. Since we cannot guarantee that this expensive hash computations are performed only at the end block, we can no longer bound the running time of the simulation.

We resolve this problem by considering a global hash function h shared by all protocol executions. In this case, the simulator can construct Merkle hash trees of every offline proof using the same shared function h, and use the same hash

tree roots in commitments given in all sessions. Now there is never a need to recompute a hash tree on a previously constructed proof and simulation running time is bounded. As explained above, soundness holds only if a cheating prover is unable to find collisions in h. Therefore we can prove the security of our protocol in the global hash model where the prover and all concurrent verifiers are given a single hash function that is assumed to be collision-resistent. The meaningfulness of this model is discussed in the introduction.

Tackling the number of rewindings per block. To complete the description of the "special proofs" we need to address one more problem: unlike the KP-PRS rewinding strategy where the black-box simulator can rewind all slots contained in a block all at once, our non-black-box simulator creates a separate offline UA proof for each slot contained in the block. The result is that the time spent by the simulator on constructing proofs at the end of the block grows with the number of slots contained in the block. One consequence of this approach is that the running time of the simulation grows much faster as a function of the recursion depth. Unlike the case in [22] where the simulation can accommodate a logarithmic level of nesting, we can only tolerate a constant level of nesting. This can be ensured at the price of increasing the round-complexity of the protocol: if the simulator uses blocks of size b^i for a $b = n^\epsilon$ (ϵ is a constant), the level of nesting $O(\log_b n)$ becomes constant. However, to guarantee successful extraction of the trapdoor, the protocol must use $\omega(b \log_b n) = \omega(n^\epsilon)$ slots.

the simulator's randomness. So far we described how to construct "special proofs" that will allow realizing the KP-PRS-style proving strategy. Our starting point was the analogy between a simulator that commits to it own code and a rewinding black-box simulator. However, before we can implement this high-level idea, we need to introduce a final modification to the protocol that will enable the simulator to commit to its own code. The difficulty has to do with the way that the simulator generates its randomness. As described above, the simulator needs to use randomness to simulate the prover's messages. In particular, in every slot the simulator uses randomness to commit to a program Π that emulates the execution of the simulator itself, and in every session the simulator uses randomness to generate messages in the special proofs. Since the program Π must precisely emulate the simulation, it must use the same randomness as the simulator. This could be done, for example, by using a PRF: the simulator will choose a PRF seed s and use it to generate all the randomness needed. The committed program Π will use the same seed s to generate identical randomness. The problem is that the simulator commits to (a hash of) the code of Π that contains the seed s using randomness generated from s. Since the committed program is correlated with the randomness of the commitment, we cannot rely on the hiding property of the commitment.

This problem can be circumvented, as pointed out in [20], by committing to a program Π that does not contain s and instead receives s as a (short) auxiliary input. This allows us to use the hiding property of the commitment. However, we still encounter a similar problem when generating special proofs. In

the special proof, the simulator proves that Π on input s (and given access to some oracle) predicts the verifier's random challenge. Thus, the witness of the special proof includes s and it is therefore correlated with the randomness used to generate the special proof. When this is the case we cannot rely on the witness-indistinguishability property of the special proof. We finally resolve this problem by letting the simulator use a list of PRF seeds $s_0, \ldots, s_m$, all generated from the last seed s_m in a "reverse chain" fashion, that is, $s_i = \mathsf{PRF}_{s_{i+1}}(\text{“}\mathsf{NEXT}\text{”})$ (where “NEXT” is an arbitrary fixed value in the domain of the PRF). In simulation, the simulator orders all the special proofs simulated in the concurrent execution according to the order in which their first message is sent. The simulator starts by using the first seed s_0, and when the i^{th} special proof starts, it switches to using seed s_i. Therefore, all the randomness used in the simulation before the i^{th} special proof starts can be recovered using s_{i-1}. Let Π be the program used as a witness in the i^{th} special proof. Since Π only emulates the main simulation until a point prior to the beginning of the i^{th} special proof, Π only needs to receive the seed s_{i-1} in order to run correctly. Now, both the witness s_{i-1} for the i^{th} special proof and the randomness used to generate the i^{th} special proof are generated using PRF from seed s_i. In this setting, we can prove that the special proofs are witness-indistinguishable based on the properties of the PRF.

Putting all the elements together. We obtain a $O(n^\epsilon)$-round public-coin cZK protocol (P_1, V_1) as informally described below. As this protocol only serves as an intermediate step towards our final protocol, we omit the formal description.

An $O(n^\epsilon)$-round public-coin cZK protocol (P_1, V_1):

- *Stage 1 (Global Hash):* P and V obtain the global hash function h.
- *Stage 2 (k Slots):* P and V run k slots. In the i^{th} slot the prover sends a commitment c_i and the verifier responds with a random challenge r_i.
- *Stage 3 (Proof Stage):* the prover proves using a "special purpose" witness-indistinguishable UA that either $x \in \mathcal{L}$, or there is a slot i, in which c_i is a commitment to a hash of a program Π containing a set of hash tree roots, such that, Π on a short input s, and with access to some valid hash-inverting oracle, outputs r_i.

2.4 Improving the Round Complexity

In this section, we describe at a high-level how to improve the round complexity of the protocol (P_1, V_1) to obtain our final protocol with $O(\log^{1+\epsilon} n)$ rounds, for any constant ϵ. Towards this, recall that as discussed in Section 2.3, the reason that we set the number of slots in $(\mathcal{P}, \mathcal{V})$ to n^ϵ is to guarantee a constant nesting depth. This is required, since the simulation running time increases too fast as a function of the nesting depth. Thus, the key to improving the round complexity is to better control the growth of the simulation running time as a function of the nesting depth.

Let us first review the contributions to the simulation running time in the protocol (P_1, V_1). Recall that at the end of a block, the simulator needs to

generate an offline UA proof for each slot that it contains. The time spent by the simulation on constructing proofs at the end of the block can therefore be attributed to two factors: first, if the simulation of a slot takes time T, the time for constructing an offline UA proof about that slot is $\text{poly}(T)$, where the specific polynomial depends on the underlying UA system. Second, since the simulator may potentially construct an offline UA proof for each slot contained in it, the number of offline proofs that needs to be constructed can be m, the number of concurrent sessions started by the verifier. Overall, the time spent at the end of a block can be $m \cdot \text{poly}(T)$. This implies a polynomial factor increase in the simulation running time for every level of nesting. To decrease the simulation time, we address both factors mentioned above:

- To improve the time complexity for constructing a single offline proof, we make use of a UA system where the offline prover's time complexity is quasi-linear; we can get such system by instantiating the construction of [2] with an underlying PCP system that has quasi-linear prover complexity [3].
- To decrease the number of proofs constructed at the end of each block, we modify the protocol and the simulation strategy so that essentially, only one offline proof needs to be constructed at the end of each block. This is harder to achieve and we describe the ideas in more details below.

As a result of the above modifications, the time spent by the simulation at the end of a block improves to $\tilde{O}(T)$, allowing the nesting depth to grow up to $O(\frac{\log n}{\log \log n})$ and leading to a protocol with $\log^{1+\epsilon} n$ slots, and $O(\log^{1+\epsilon} n)$ rounds.

The main idea behind achieving the second improvement described above is that instead of constructing an offline proof about the simulation of each slot, the simulator constructs a single "block-proof" arguing about the simulation of the *whole block* and then reuses the block-proof for the slots contained in the block. In the block-proof, the simulator proves that the committed program Π that mimics the execution of the main thread in the block outputs a transcript τ of the block (which includes the verifier's challenge messages of all the slots contained in this block). In order to use a block-proof to argue about one slot (c, r) contained in the block, the simulator creates, in addition to the block-proof, a second offline UA proof that r is contained in τ—we call this a "session-proof" (note that a UA is used since the transcript τ may be long). Informally speaking, putting the block-proof and the session-proof together, the simulator can now "cheat" by proving that c is a commitment to the hash of a program that outputs *a transcript containing the random challenge*. Intuitively, soundness still holds, as it is hard for a cheating prover to find a program that outputs any transcript of polynomial length that will contain the random challenge. To implement this idea, we need to make a few changes to the protocol as highlighted below.

- Let (c_i, r_i) be the i^{th} slot in a session j, and let B be the block of minimal size that contains (c_i, r_i). As described above, at the end of block B, the simulator constructs a "block-proof" about the execution of Π that mimics the execution of the main thread in this block. For the simulator to be able to reuse this block-proof in the i^{th} slot of session j, c_i must be a commitment to

Π. However, at the time the commitment c_i is generated, the simulator does not know when the message r_i will be scheduled and which block will be the minimal block containing this slot. Thus, the simulator does not know which program to commit to. To resolve this problem, we modify a slot to consist of n commitments $c_{i,1}, \ldots, c_{i,n}$ from the prover (and still one random challenge r_i from the verifier). Now, when a slot opens, the simulator can commit to all the programs that emulate the execution of the simulation in all the blocks that are currently open (that is, all blocks that may potentially contain this slot), and later generate a proof with respect to one of these commitments.

– We modify the special proof to consist of both the session-proof and the corresponding block-proof. The special proof will consist of two separate oblivious executions of the online stage of the UA standing for both proofs. Then, a witness-indistinguishable proof is given that either the statement is true or that the transcripts of UA hidden in the two oblivious executions are both accepting, and together form a trapdoor for one slot in the session.

Finally, we remark that now indeed only one block-proof is created after each block. However, it seems that we haven't gained anything as the simulator still needs to create a session-proof for each slot contained in it. However, since the length of τ is bounded by the running time of $\mathcal{V}^*$, the time complexity for constructing the session-proof is always bounded by a fixed polynomial in the running time of $\mathcal{V}^*$. Therefore, only the time complexity for constructing the block-proof grows with the nesting depth. This suffices for the purpose of controlling the simulation time from growing too fast.

3 The Final Protocol

In this section we give an informal description of our public-coin concurrent ZK protocol in the global hash model (Protocol 3). The number of rounds of the protocol depends on the parameter k. Next, we describe the notations used to describe Protocol 3.

Primitives. Protocol 3 makes use of a statistically binding commitment Com (described for simplicity as a non-interactive commitment), a witness indistinguishable argument of knowledge (WIAOK), and a hash function h sampled randomly from a family of collision-resistant hash functions and given to both parties as a common input. In the description of the language Λ_1 below we abuse notation and use h as a hash tree rather then a simple hash function. That is, $\delta = h(P)$ represents a root of a Merkle hash tree applied to a long string P. For an index j, we can compute the authentication path from δ to $P[j]$, certifying that the value of δ is consistent with $P[j]$.

Oblivious UA. Our protocol uses the online/offline public-coin UA protocol as given by Protocol 1, where the verifier's hash function sent in the offline stage is replaced by the global hash h. In Protocol 3 the online part of the online/offline

UA is executed twice in an oblivious way. That is, the online UA prover commits to the statement it wants to prove, and to all UA messages. The online UA verifier is only given the length of the proven statement (recall that the verifier messages in online UA are simply its random coins and therefore it can compute these messages without knowing the statement). For example, when proving any statement of the form $(h, h_{\Pi}, h_{\tau}, c, r) \in \Lambda_2$ (the language Λ_2 is described below) using an oblivious UA, the verifier is only given the length of the canonical statement $|\text{“}(h, 0^n, 0^n, c_{1,1}, r_1) \in \Lambda_2\text{”}|$. After the two oblivious executions of the online UA are completed, a WIAOK is used to prove that the transcript of each oblivious UA execution is consistent with valid online UA proof for the appropriate statements. That is, there exist openings for all the commitments sent by the prover in the oblivious UA executions into messages, such that these messages (together with the random messages sent by the verifier) form an accepting online UA proof for the statement of interest.

Block-proofs and session-proofs. We refer to the first oblivious UA as a block-proof and the second as session-proof. Block-proofs are proofs of membership in the language Λ_1 defined as follows: $(h, h_{\Pi}, h_{\tau}) \in \Lambda_1$ if (h_{Π}, h_{τ}) are hashes of a program Π and a transcript τ respectively, such that program Π, given access to some valid oracle, and some short auxiliary input, produces the transcript τ. Session-proofs are proofs of membership in the language Λ_2 defined as follows: $(h, h_{\Pi}, h_{\tau}, c, r) \in \Lambda_2$ if c is a commitment to the hash h_{Π} and h_{τ} is the hash of a transcript τ that contains the message r. More formally, for the super-polynomial function $T(n) = n^{\log \log n}$ the languages Λ_1, Λ_2 are defined in Figure 2:

Block-proof: $(h, h_{\Pi}, h_{\tau}) \in \Lambda_1$ iff there exist:

- Π - description of a program such that $|\Pi| < T(n)$.
- τ - a transcript such that $|\tau| < T(n)$.
- O - description of an oracle such that $|O| < T(n)$.
- β - Auxiliary input Π such that $|\beta| < |r| - n$.

And the following conditions hold:

- $h_{\Pi} = h(\Pi)$, $h_{\tau} = h(\tau)$ and $\Pi^{O}(\beta)$ generates τ within $T(n)$ steps.
- O contains an answer to every query Π makes and only contains query-answer pairs of the form $((j, q), (b, a))$ such that:
  - The description of Π contains the variable δ_j.
  - There is a string P_j such that $\delta_j = h(P_j)$, $b = P_j[q]$ and a is the corresponding authentication path.

Session-proof: $(h, h_{\Pi}, h_{\tau}, c, r) \in \Lambda_2$ iff there exists randomness ρ for Com and a transcript $\tau, |\tau| < T(n)$ such that $c = \mathsf{Com}(h_{\Pi}; \rho)$,$h_{\tau} = h(\tau)$, and τ contains r.

Fig. 2. Block-proof and session-proof

Common Input: $x \in \mathcal{L}$.
Auxiliary Input to $\mathcal{P}$: $w \in \mathcal{R}_{\mathcal{L}}(x)$.
Common Reference String: A hash function h.

1. $\mathcal{P}$ and $\mathcal{V}$ repeat the following for every $i \in [k]$:
   (a) $\mathcal{P}$ computes $c_{i,j} \leftarrow \mathsf{Com}(h(0^n), U_n)$ for $j \in [n]$ and sends $\{c_{i,j}\}_{j\in[n]}$ to $\mathcal{V}$.
   (b) $\mathcal{V}$ samples a random string $r_i \leftarrow U_{2n}$ and sends r_i to $\mathcal{P}$.
2. $\mathcal{P}$ and $\mathcal{V}$ run an oblivious UA for a statement of the same length as "$(h, 0^n, 0^n) \in \Lambda_1$". All commitments sent by $\mathcal{P}$ are to the all-zero string.
3. $\mathcal{P}$ and $\mathcal{V}$ run an oblivious UA for a statement of the same length as "$(h, 0^n, 0^n, c_{1,1}, r_1) \in \Lambda_2$". All commitments sent by $\mathcal{P}$ are to the all zero string.
4. $\mathcal{P}$ proves to $\mathcal{V}$ using a public-coin WIAOK that either $x \in \mathcal{L}$ or there exists $i \in [k]$, $j \in [n]$ and a hash values h_Π, h_τ such that both of the following hold:
   (a) The transcript of the first oblivious UA is consistent with an accepting proof for the statement: "$(h, h_\Pi, h_\tau) \in \Lambda_1$".
   (b) The transcript of the second oblivious UA is consistent with an accepting proof for the statement: "$(h, h_\Pi, h_\tau, c_{i,j}, r_i) \in \Lambda_2$".

Fig. 3. Pubic Coin cZK Protocol (Protocol 3)

References

[1] Barak, B.: How to go beyond the black-box simulation barrier. In: FOCS (2001)
[2] Barak, B., Goldreich, O.: Universal arguments and their applications. SIAM J. Comput. (2008)
[3] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete-efficiency threshold of probabilistically-checkable proofs. Electronic Colloquium on Computational Complexity (ECCC) (2012)
[4] Bitansky, N., Canetti, R., Halevi, S.: Leakage-tolerant interactive protocols. In: TCC (2012)
[5] Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications (extended abstract). In: STOC (1988)
[6] Canetti, R., Fischlin, M.: Universally composable commitments. IACR Cryptology ePrint Archive (2001)
[7] Canetti, R., Kilian, J., Petrank, E., Rosen, A.: Black-box concurrent zero-knowledge requires (almost) logarithmically many rounds. SIAM J. Comput. (2002)
[8] Canetti, R., Lin, H., Paneth, O.: Public-coins concurrent zero-knowledge in the global hash model. IACR Cryptology ePrint Archive (2013)
[9] Deng, Y., Feng, D., Goyal, V., Lin, D., Sahai, A., Yung, M.: Resettable cryptography in constant rounds - the case of zero knowledge. In: ASIACRYPT (2011)
[10] Deng, Y., Goyal, V., Sahai, A.: Resolving the simultaneous resettability conjecture and a new non-black-box simulation strategy. In: FOCS (2009)

[11] Dwork, C., Naor, M., Sahai, A.: Concurrent zero-knowledge. In: STOC (1998)
[12] Garg, S., Jain, A., Sahai, A.: Leakage-resilient zero knowledge. In: CRYPTO (2011)
[13] Goldreich, O., Kahan, A.: How to construct constant-round zero-knowledge proof systems for NP. Journal of Cryptology (1996)
[14] Goldreich, O., Krawczyk, H.: On the composition of zero-knowledge proof systems. SIAM J. Comput. (1996)
[15] Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity for all languages in np have zero-knowledge proof systems. J. ACM (1991)
[16] Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: STOC (1985)
[17] Kilian, J., Petrank, E.: Concurrent and resettable zero-knowledge in poly-loalgorithm rounds. In: STOC (2001)
[18] Pandey, O., Pass, R., Sahai, A., Tseng, W.L.D., Venkitasubramaniam, M.: Precise concurrent zero knowledge. In: EUROCRYPT (2008)
[19] Pass, R., Rosen, A.: Concurrent non-malleable commitments. In: FOCS (2005)
[20] Pass, R., Rosen, A., Tseng, W.: Public-coin parallel zero-knowledge for np. Journal of Cryptology (2011)
[21] Pass, R., Tseng, W.L.D., Wikström, D.: On the composition of public-coin zero-knowledge protocols. In: CRYPTO (2009)
[22] Prabhakaran, M., Rosen, A., Sahai, A.: Concurrent zero knowledge with logarithmic round-complexity. In: FOCS (2002)
[23] Richardson, R., Kilian, J.: On the concurrent composition of zero-knowledge proofs. In: EUROCRYPT (1999)
[24] Rogaway, P.: Formalizing human ignorance. In: VIETCRYPT (2006)