Optimistic Concurrent Zero Knowledge

Alon Rosen[1] and abhi shelat[2]

[1] IDC Herzileya
alon.rosen@idc.ac.il
[2] U. of Virginia
shelat@cs.virginia.edu

Abstract. We design cryptographic protocols that recognize best case (optimistic) situations and exploit them. As a case study, we present a new concurrent zero-knowledge protocol that is expected to require only a small constant number of rounds in practice. To prove that our protocol is secure, we identify a weak property of concurrent schedules—called *footer-freeness*—that suffices for efficient simulation.

Keywords. concurrent zero-knowledge, rational, optimistic

1 Introduction

Cryptographic protocols anticipate *worst-case* behavior and therefore often contain complicated provisions that are meant solely to handle them. Such provisions can be expensive and counter-intuitive.

To circumvent these side-effects but still construct protocols that are secure against worst-case behavior, this paper proposes to use an *optimistic* technique for building protocols that is inspired by work on Byzantine agreement. The aim is to design protocols that can recognize the best cases and optimize for them, even in the midst of the protocol execution.

Optimism has been employed by researchers in distributed computing (e.g. the (Fast) Paxos algorithm [**?**]) and fair exchange [**?**]; the novelty of this work is to exploit *optimism* for the problem of concurrent zero-knowledge. Optimistic protocols make no attempt to improve worse-case performance. In fact doing so would require overcoming a lower bound argument in the case of zero-knowledge. Nonetheless, the optimistic cases that we exploit are common and meaningful to discuss.

1.1 Concurrent Zero-Knowledge

When many instances of a stand-alone zero-knowledge protocol are executed at the same time, the combination of all runs may leak information *about the theorem*. The standard methodology for arguing that a protocol transcript "does not leak information" is to exhibit a simulator algorithm that is able to produce transcripts that are indistinguishable from actual transcripts of protocol executions. Dwork, Naor and Sahai [**?**] observed that in a concurrent zero-knowledge setting, a malicious verifier who controls

the schedule of protocol messages can induce a schedule for which a "naive" simulation algorithm will require exponential time (and thus the execution may leak information). An example of such a scheduling of messages is given in 1.1. In the bottom (red) schedule, the verifier has "nested" many executions of the zero-knowledge protocol. This type of scheduling is a *concurrency attack* on the zero-knowledge property of the original protocol and it captures the fundamental problem with designing efficient concurrently secure zero-knowledge protocols.

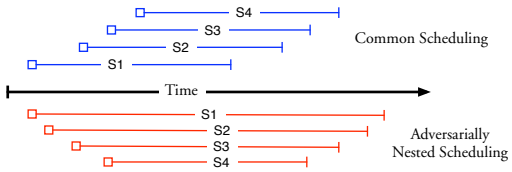

Fig. 1. Illustration of an "average-case" schedule, and an adversarial one.

To address the concurrency attack, Dwork, Naor and Sahai [**?**] proposed a timing model assumption and a protocol that limits the amount of nesting that can occur in an adversarial scheduling. Their protocol was an argument system; Goldreich [**?**] later showed that proof systems can also be constructed in such a model. Pass, Tseng, and Venkitasubramaniam [**?**] present an eye-for-an-eye solution in the timing model that reduces the overall delay of the protocol. Other protocols that handle concurrency attacks have been obtained by introducing different setup assumptions [**?**,**?**,**?**] such as a common reference string or a PKI.

Richardson and Kilian [**?**] constructed the first concurrent zero-knowledge argument system in the standard model without extra setup assumptions. Kilian and Petrank [**?**] introduced a simulation technique which led to simpler and cleaner analysis and fewer rounds. Finally, the work of Prabhakaran, Rosen and Sahai [**?**] (PRS) further simplified and improved the analysis of the Kilian and Petrank protocol to obtain a protocol with $\omega(\log n)$ rounds. This round complexity is close to optimal in the standard model because without any set-up assumptions, Canetti, Kilian, Petrank and Rosen [**?**,**?**] show that concurrent zero-knowledge argument systems for non-trivial languages using a "black-box" simulator require at least $\Omega(\log n/\log\log n)$ number of communication rounds. In order to show this lower bound, they rely on a framework proposed by Kilian, Petrank and Rackoff [**?**], with further improvements from Rosen [**?**], and present a specific malicious verifier and a particularly difficult schedule of messages. Recently, Pandey et al. [**?**] have proposed new precise concurrent zero-knowledge proofs with similar round complexity.

It has been a long-standing open problem to build communication-efficient concurrent zero-knowledge protocols. To circumvent the lower bound on the round complexity from [**?**,**?**], prior work (1) introduces additional trust assumptions [**?**,**?**,**?**,**?**], (2) relaxes the definition of security to allow quasi-polynomial time simulation [**?**,**?**,**?**], or (3) employs a more complicated and powerful non-black-box simulation technique [**?**] and

restricted the number of concurrent sessions. The latter technique also relies on complex tools and techniques that require NP-reductions.

Overview of slots Before detailing our approach, let us review the idea employed by the protocols from [**?**,**?**,**?**,**?**] to defend against concurrent attacks. In the first phase of the protocol, the verifier creates an irrelevant secret (such as a commitment to a string) and then (repeatedly) proves in zero-knowledge to the prover that it knows the secret. Each block of protocol messages during which the verifier proves knowledge of this secret is called a "slot." In the second phase, the Prover proves that it either knows the witness to the original theorem or that it knows the verifier's secret using a witness indistinguishable protocol. Prior work [**?**,**?**,**?**,**?**] proves that if the first phase has enough slots, then a simulation strategy can be devised such that for *any schedule of messages*, the simulator can successfully extract a witness from the verifier's proof and then use that witness in the second phase.

Optimistic Defense We propose an optimistic defense against concurrency attacks in which we relax the requirements from prior work and specifically [**?**] that (1) each protocol session involves an independent prover who does not know anything about the other protocol instances and (2) each protocol execution has exactly the same (fixed) number of rounds. Doing so allows us to build protocols that *optimistically avoid* the worst-case schedules used in the lower bounds.

When one server handles many concurrent requests, the server knows the exact schedule of messages. The work of Persiano and Visconti [**?**] also exploits this relaxation by using a Prover who counts the total number of bytes sent in all sessions[3].

We believe this to be a reasonable and practical relaxation. In many applications of zero-knowledge proofs, for example, the prover will be the same party (some server), and it will have the opportunity to share state between protocol sessions. In particular, servers on the internet routinely keep track of the various protocol session statistics such as the total number of protocol executions that run at a given time. Operating systems which make quality of service guarantees also inspect different protocol instances in order to throttle connections. While the original motivation of the concurrent session model in which the Prover instances run independently of one another was to simplify implementation of systems, there is no fundamental implementation reason that prevents sharing the global scheduling information among the Prover algorithms in different protocol sessions. (Of course, requiring the Verifiers to coordinate their sessions would be unrealistic, since the Verifiers may be separate parties.)

In our model, each session of the protocol may require a different number of communication rounds. This relaxation allows us to instruct the prover to handle schedules which are *easier to simulate* differently than schedules which are more difficult. In contrast, in typical cryptographic protocols, each execution of the protocol has a fixed number of messages and each successful invocation usually requires exactly the same number.

Our protocol can "short circuit" the normal protocol when it is clear that such a shortcut preserves the security properties. To the best of our knowledge, this idea has not been applied in the context of a security guarantee such as zero-knowledge.

[3] In contrast to this work, their solution uses non-blackbox simulation techniques.

1.2 Our Protocol

The idea behind our fast track protocol is to discourage the verifier V^* from nesting sessions within the slots of other sessions by penalizing a verifier whose slots have nested sessions. The penalty will be to gradually add more slots to the protocol until either enough slots with no (or few) nestings occur, or a pre-specified bound on the number of rounds is reached.

The basis for our protocol is the $c\mathcal{ZK}$ protocol by Prabhakharan, Rosen and Sahai [**?**] which uses statistically hiding commitments [**?**,**?**] and Blum's 3-message protocol for Hamiltonicity [**?**]. The only difference between our protocol and the PRS protocol is that it contains a special provision for exiting the "preamble" stage. Early exits are "approved" by the prover, provided that there is a slot in the current session that does not have any other session footers within it. Assuming that verifiers answer quickly, it is expected that the number of nested sessions within slots is generally small, optimistically resulting in an empty slot and thus in straightforward simulation.

Verifiers have incentive to answer fast since the longer they delay their answer, the more likely they are to have nested sessions (from some other verifier) within the slot that they are currently executing. Once the slot has a nested session within it, early exit is postponed to future rounds, and another slot is added to the protocol's execution. This process continues until $k = \omega(\log n)$ slots have been performed, in which case the Hamiltonicity proof takes place and the protocol terminates.

At the expense of a more involved analysis, one should be able to replace the PRS protocol with any other instantiation of a $c\mathcal{ZK}$ protocol that follows the RK "multi slot" paradigm, and obtain analogous results. One attractive instantiation would be the DDH-based $c\mathcal{ZK}$ protocols of Micciancio and Petrank [**?**]. These protocols admit fairly efficient implementations, and are thus a good match for our optimistic approach, whose primary objective is increased efficiency.

But worst-case schedules still require many rounds! Note that worse-case schedules still require the same number of rounds as PRS. Such an argument applies to any optimistic protocol, such as the Fast Paxos or Fair exchange protocols as well. The *worst-case* schedule, however, may be *rare* and *avoidable* by incentivized verifiers.

Comparison with Other Proposals In Appendix A, we compare our approach to other simple proposals and to the timing model proposed by [**?**].

2 Optimistic Rational Concurrency

Let $\langle P, V \rangle$ be an interactive proof (resp. argument) for a language L, and consider a single *concurrent adversary* (verifier) V^* that, given input $x \in L$, interacts an unbounded number of times with P (each with common input x) without any restrictions over the scheduling of its messages.

Formally, use the standard model for concurrency in the timing model put forth by Dwork, Naor, and Sahai [**?**]. The adversary V^* takes as input the prover's partial conversation transcript that includes the times on the provers local clock when each message was sent or received by the prover. The adversary's output is either a tuple $(recv, V, \alpha, t)$, indicating that P receives message α from V at local time t or

$(send, V, t)$, indicating that P must send the next message to V at time t on P's local clock. In both cases, the time t that adversary chooses must be greater than all the times given in the input transcript (i.e., the adversary cannot rewind P), the session with V must be well-formed, and α must be in the protocol's "message space" (i.e. standard well-formedness conditions apply). If these conditions are not met, the transcript is discarded.

The *transcript* of a concurrent interaction consists of the common input x, followed by the sequence of prover and verifier messages exchanged during the interaction. We denote by $\mathrm{view}^P_{V^*}(x)$ a random variable describing the content of the random tape of V^* and the conversation transcript between P and V^* as described above.

Definition 1 (Concurrent Zero-Knowledge). *Let $\langle P, V\rangle$ be an interactive proof system for a language L. We say that $\langle P, V\rangle$ is* concurrent zero-knowledge*, if for every probabilistic strict polynomial-time concurrent adversary V^* there exists a probabilistic polynomial-time algorithm S_{V^*} such that the ensembles $\{\mathrm{view}^P_{V^*}(x)\}_{x\in L}$ and $\{S_{V^*}(x)\}_{x\in L}$ are computationally indistinguishable.*

Discussion There may be other verifiers that are also interacting with P at the same time as V^*. In prior work, these sessions are ignored because either the monolithic adversary V^* can incorporate these sessions if they can be used to cheat, or because these extra sessions are completely independent of V^*'s view.

In our case, however, these extra sessions by honest verifiers are not completely independent of V^*'s view.[4] In the protocol we suggest, for example, a verifier will learn when one of its slot is not footer-free, and therefore it will learn the presence of another session. This is not necessarily the case with other concurrent ZK protocols such as PRS because the number of rounds in those protocols are not related to the schedule of messages. However, the aim for a zero-knowledge protocol in a networked setting is to ensure that no information about the witness w for instance x is leaked; we feel that it is reasonable for a protocol to leak network timing information because such information is typically leaked by the underlying network (or by timing or side channels).

To model this, we give V^* full control over the timing of *all network messages* including the Prover's messages and the timing of the messages from the honest verifier sessions that are not controlled by V^*. Although this is syntactically the same formal model with a single V^* as in prior work, there is a subtle difference. Our protocol and its simulator essentially guarantees that "a verifier V' who controls a subset of the sessions learns no more through interaction with P than a malicious verifier V^* who controls all network traffic, and such a verifier learns no more than the polynomial time simulator who does not have the witness."

Notation We use the symbols $(\mathrm{V0}), (\mathrm{P1}), (\mathrm{V1}), \ldots, (\mathrm{P}j), (\mathrm{V}j)$ to denote the messages in the *preamble*; these messages are completely independent of the common input and they serve to enable a successful simulation in the concurrent setting.

Every round (*slot*) in the preamble (i.e., every $(\mathrm{P}j), (\mathrm{V}j)$ pair) is viewed as a "rewinding opportunity." Successfully rewinding even one slot in the preamble is sufficient in order to cheat arbitrarily in the actual proof (messages $(\mathtt{p1}), (\mathtt{v1}), (\mathtt{v2})$) and thus complete the simulation.

[4] We thank the anonymous reviewer for pointing out this subtle distinction.

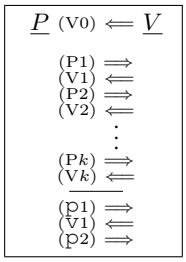

Fig. 2. A k-round preamble.

One problem faced by a $c\mathcal{ZK}$ simulator is that rewinding a specific session may result in loss of work done for other sessions, and therefore require the simulator to do the same amount of work again. This will happen whenever the rewound slot contains other sessions "nested" within it.

For example, if a slot of session B contains the $(\mathrm{V}0)$ message of session A within it, rewinding this slot will cause all simulation work done for session A to be lost. This is because the simulation of a session A hinges on the simulator "extracting" specific values that have been committed to by the verifier in message $(\mathrm{V}0)$ of this session. Rewinding past the $(\mathrm{V}0)$ message of A could alter the history of interaction up to this message and may result in a modification of its contents (rendering the extracted values irrelevant).

The simulator must invest work in session A whenever session A's preamble completes before the end of the slot of session B. In such a case, reaching the end of session A's preamble without having extracted the value committed to in message $(\mathrm{V}0)$ of session A may prevent the simulator to proceed beyond the end of this preamble (since the malicious verifier may refuse to continue if is not convinced in the validity of the statement being proved in session A). Failure to proceed beyond the end of the session A preamble translates directly to failure to rewind the session B slot within which this preamble is nested.

Definition 2 (Nested Footer). *Slot j of session B is said to have a* nested footer *of session A within it if session A's $(\mathrm{V}k)$ message occurs between messages $(\mathrm{P}j), (\mathrm{V}j)$ of session B. A slot is said to be* footer free *if it has no nested footer.*

Avoiding nested footers enables the completion of the slot between messages $(\mathrm{P}j)$ and $(\mathrm{V}j)$ of session B without having to first invest work in simulating session A (implying that there is no risk to lose and thus redo this work as a result of rewinding). This observation will be crucial to the analysis of the footer-free version of our protocol.

Two simulation strategies. Currently, there are two known approaches for concurrent simulation. The first simulation strategy *adaptively* looks for slots that do not have many sessions with nested headers within them, and this is where it focuses its attempts to rewind the interaction with the verifier [**?**]. The second simulation strategy is different in that it performs a sequence of rewinds *obliviously* of the actual scheduling of the messages [**?**,**?**].

The main advantage of the second approach over the first one is that it is known to guarantee correct "worst case" simulation using fewer slots ($\tilde{O}(\log n)$ vs. $O(n^{\epsilon})$

for every $\epsilon > 0$). However, being oblivious to the actual schedule, it does not seem suitable for taking advantage of lack of nested headers and/or footers within a slot. As we demonstrate in this paper, by adaptively identifying good places for multiple rewindings, the second approach can be tailored to work in our optimistic setting.

To the best of our knowledge, the idea of taking advantage of the lack of nested footers for the sake of improved concurrent simulation is new. As we argue in the paper, lack of nested footers within one slot is a fairly weak constraint on the schedule, and may be enforced using a variety of realistic mechanisms.

The protocol. Our protocol will have the prover monitor the scheduling of messages, and identify footer-free slots on the fly; once such a slot is identified, there is no need to keep adding slots to the execution of that specific session, so the protocol moves on to the execution of the actual constant-round $\mathcal{ZK}$ protocol.

Common Input: $x \in \{0,1\}^n$, security param n, max. # rounds param $k = \omega(\log n)$.
Prover's Input: a witness w such that $R_L(x, w) = 1$
Stage 1:
$P \to V$ (P0): Send first message of perfectly hiding commitment $\mathbf{Com}$.
$V \to P$ (V0): Using the commitment $\mathbf{Com}$, commit to random $\sigma, \{\sigma^0_{i,j}\}^k_{i,j=1}$, $\{\sigma^1_{i,j}\}^k_{i,j=1}$ such that $\sigma^0_{ij} \oplus \sigma^1_{ij} = \sigma$ for all i, j.
Slot j:
$P \to V$ (Pj)**:** Send a random challenge $r_i = r_{1,j}, \ldots, r_{k,j}$
$V \to P$ (Vj)**:** Upon receiving a message r_i, decommit to $\sigma^{r_{1,j}}_{1,j}, \ldots, \sigma^{r_{k,j}}_{k,j}$
$P \to V$**:** If any of the decommitments fails verification, abort.
If slot j is *footer free* **or** $j = k$ move to stage 2.
If slot j is **not** *footer free* **and** $j \leq k$ move to slot $j + 1$.
Stage 2: P and V engage in Blum's 3-round Hamiltonicity protocol using challenge σ:
1. $P \to V$ (p1): Use witness to produce first message of Ham protocol
2. $V \to P$ (v1): Decommit to σ and to $\{\sigma^{1-r_{i,j}}_{ij}\}^k_{i,j=1}$.
3. $P \to V$ (p2): If decommitments are valid and $\sigma^0_{i,j} \oplus \sigma^1_{i,j} = \sigma$ for all i, j, answer σ with third message of Ham protocol. Otherwise abort.

Fig. 3. Fast-track concurrency.

Completeness and soundness of Protocol 3 are inherited from the PRS protocol, and in particular follow from Proposition 4.3.2 in [**?**]. We now turn to demonstrating the $c\mathcal{ZK}$ property.

2.1 The Simulator

We exhibit the $c\mathcal{ZK}$ property using a black-box simulator S. Let V^* be a concurrent adversary verifier. S will rewind the interaction with V^* and examine its input/output behavior. The rewinding strategy of the simulator is specified by a SOLVE procedure whose goal is to supply the simulator with V^*'s "challenges" before reaching stage 2 in the protocol. This is done by rewinding the interaction with V^* while trying to achieve two "different" answers to some (Pj) message. *We refrain from specifying the*

way stage 2 messages are handled and focus only on stage 1 messages. For standard details on how to handle stage 2 messages see [**?**].

The timing of the rewinds performed by the SOLVE procedure depends on the number of stage 1 verifier messages received so far and on the size of the schedule. However, whenever it encounters a situation in which a slot of a given session is footer-free, the SOLVE procedure (adaptively) assumes that this is its only chance to solve that session and performs (an expected polynomial number of) extra rewinds in order to make sure that the slot is successfully rewound. The number of extra rewinds is not determined in advance, and is induced by the analysis of a constant round $\mathcal{ZK}$ protocol for $\mathcal{NP}$ by Rosen [**?**].

At a high level, the SOLVE procedure splits the first stage messages it is about to explore into two halves and invokes itself recursively twice for each half (completing the two runs of the first half before proceeding to the two runs of the second half). At the top level of the recursion, the messages that are about to be explored consist of the entire schedule, whereas at the bottom level the procedure explores only a single message (and as we said may do so multiple times, depending on whether the recursive call corresponds to a message-free slot). The solve procedure always outputs the sequence of "first explored" messages.

The input to the SOLVE procedure consists of a triplet $(\ell, \mathsf{hist}, \mathcal{T})$. The parameter ℓ corresponds to the total number of verifier messages, the string hist consists of the messages in the "first visited" history of interaction, and $\mathcal{T}$ is a table containing the contents of all the messages explored so far. The messages stored in $\mathcal{T}$ are used in order to determine σ according to answers $(\mathrm{V}j)$ to different $(\mathrm{P}j)$. They are kept relevant by constantly keeping track of the sessions that are rewound past their initial commitment. That is, whenever the SOLVE procedure rewinds past the $(\mathrm{V}0)$ message of a session, all messages belonging to this session are deleted from $\mathcal{T}$.

The analysis takes advantage of the fact that no rewound slot contains a footer, building on the assumption that footer-freeness is an event of non-negligible probability (as otherwise it is assumed not to have occurred to begin with). By repeatedly rewinding, the simulator is likely to run into a footer-free situation again, which means that it will not get stuck on that rewinding. This will enable it to successfully complete the rewinding attempt, and to solve the corresponding session (thus avoiding getting stuck on sessions that have strictly less than k slots).

2.2 Analysis of the Simulator

To show that the simulator S succeeds we will need to argue that: (1) S runs in polynomial time, (2) conditioned on the success of the SOLVE procedure, the output of S is indistinguishable from a concurrent interaction between P and V^*, and (3) for every session $i \in \{1, \ldots, m\}$, whenever session i reaches the second stage in the protocol, the simulator will have obtained the value of σ in this session if required (i.e. did not get *stuck*) with overwhelming probability. Once (3) is established, we may apply a union bound over the i's and conclude that SOLVE fails with only negligible probability. We focus on (1) and (3).

Lemma 1. *For every $m = \mathrm{poly}(n)$, S_m runs in (expected) polynomial-time in n.*

Procedure SOLVE(ℓ, hist, $\mathcal{T}$):
Bottom level ($\ell = 1$):

1. For each $s \in \{1, \ldots, m\}$, if the initial commitment, $(\mathrm{V0})$, of session s does not appear in hist, delete all session s messages from $\mathcal{T}$.
2. Run $\beta \leftarrow V^*(\mathsf{hist}, \mathrm{p})$. If β is of the form $(recv, V, \alpha, t)$, the continue to the next step. Else if it is $(send, V, t)$, then uniformly choose a first stage prover message p, append it to the transcript at time t, and repeat this step. If t or α are invalid, then halt the simulation and output the current transcript.
3. Let
   - $(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_t, \mathrm{v}_t) = (\mathsf{hist}, \mathrm{p}, \mathrm{v})$
   - i be the session number to which v corresponds,
4. If there exists a pair of indices (a, b) such that $a \in [t]$ and $b = t$ for which:
   - $\mathrm{v}_b \neq \mathtt{ABORT}$,
   - both v_b and p_a belong to session i.
   - the slot between messages p_a and v_b is footer-free.

   then pick one such (a, b) and rewind interaction to message p_a until
   - $\mathrm{v}_b \neq \mathtt{ABORT}$,
   - both v_b and p_a belong to session i.
   - the slot between messages p_a and v_b is footer-free.
5. Store the messages gathered in the rewindings along with p, v in $\mathcal{T}$
6. output $\mathcal{T}$, (p, v).

Recursive step ($\ell > 1$):

1. Set $\mathcal{T}_1$, $(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell/2}, \mathrm{v}_{\ell/2}) \leftarrow$SOLVE($\ell/2$, hist, $\mathcal{T}$).
2. Set $\mathcal{T}_2$, $(\tilde{\mathrm{p}}_1, \tilde{\mathrm{v}}_1, \ldots, \tilde{\mathrm{p}}_{\ell/2}, \tilde{\mathrm{v}}_{\ell/2}) \leftarrow$ SOLVE($\ell/2$, hist, $\mathcal{T}_1$).
3. Set $\mathcal{T}_3$, $(\mathrm{p}_{\ell/2+1}, \mathrm{v}_{\ell/2+1}, \ldots, \mathrm{p}_\ell, \mathrm{v}_\ell) \leftarrow$SOLVE($\ell/2$, $(\mathsf{hist}, \mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell/2}, \mathrm{v}_{\ell/2})$, $\mathcal{T}_2$).
4. Set $\mathcal{T}_4$, $(\tilde{\mathrm{p}}_{\ell/2+1}, \tilde{\mathrm{v}}_{\ell/2+1}, \ldots, \tilde{\mathrm{p}}_\ell, \tilde{\mathrm{v}}_\ell) \leftarrow$SOLVE($\ell/2$, $(\mathsf{hist}, \mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{\ell/2}, \mathrm{v}_{\ell/2})$, $\mathcal{T}_3$).
5. Output $\mathcal{T}_4$, $(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_\ell, \mathrm{v}_\ell)$.

Fig. 4. The SOLVE procedure.

Proof. We analyze the work invested at any given invocation of level $\ell = 1$. For any $G \in HC$, for any choice of hist, p, and of $a, b \in \{1, \ldots, t\}$ where $a \leq b$, let $\zeta_{a,b} = \zeta_{a,b}(G, \mathsf{hist}, \mathrm{p}, \mathrm{v})$ denote the probability that: (1) the verifier V^* does not send ABORT in message v_b, (2) both v_b and p_a belong to session i, (3) the slot between p_a and v_b is footer-free, (4) none of the v_j's correspond to message $(\mathrm{V0})$ of session i, and (5) none of the p_j's correspond to message $(\mathrm{p1})$ of session i. Let $\zeta'_{a,b}$ denote the probability that (1), (2) and (3) occur. The probabilities $\zeta_{a,b}$ and $\zeta'_{a,b}$ are taken over the random choices of the invocations of the SOLVE procedure. It can be seen that $\zeta'_{a,b} \geq \zeta_{a,b}$.

Using this notation, a pair (a, b) satisfying conditions (1)-(5) occurs with probability $\zeta_{a,b}$ and the SOLVE procedure is expected to repeat the loop in step 4 for at most $1/\zeta'_{a,b}$ times (since the condition in Step 4 is satisfied in each one of the rewinds with probability $\zeta'_{a,b}$, independently of other rewinds). For $i \in \{1, 2, 3, 4, 5, 6\}$, let $p_i(\cdot)$ be a polynomial bound on the work required in order to perform Step i in level $\ell = 1$ of the recursion (where in step 4, $p_4(\cdot) = p_{4,a,b}(\cdot)$ counts the number of steps required to perform one rewinding). By linearity of expectation, and because the total number t of

pairs of messages (and hence pairs $(a,b) \in R$) in the history of level $\ell = 1$ is at most $m \cdot (k+1)$ (recall that k is the maximal number of rounds in the protocol), the expected time required to execute level $\ell = 1$ of the recursion is upper bounded by:

$$\begin{aligned} & p_1(n) + p_2(n) + p_3(n) + \sum_{(a,b):a \leq b} \zeta_{a,b} \cdot \frac{1}{\zeta'_{a,b}} \cdot p_4(n) + p_5(n) + p_6(n) \\ & \leq p_1(n) + p_2(n) + p_3(n) + m \cdot (k+1) \cdot p_4(n) + p_5(n) + p_6(n) \\ & = \mathrm{poly}(n) \end{aligned}$$

Since each invocation of the SOLVE procedure with parameter $\ell > 1$ involves four recursive invocations of the SOLVE procedure with parameter $\ell/2$, we have that the expected work $W(\ell)$, that is invested by the SOLVE procedure in order to handle ℓ (first stage) verifier messages satisfies:

$$W(\ell) \leq \begin{cases} \mathrm{poly}(n) & \text{If } \ell = 1 \\ 4 \cdot W(\ell/2) & \text{If } \ell > 1 \end{cases} \tag{1}$$

Since the total number of first stage verifier messages in the m sessions of the concurrent schedule equals $m \cdot (k+1)$, the total expected running time of the simulation process (which consists of a single invocation of the SOLVE procedure with parameter $m \cdot (k+1)$) equals $W(m \cdot (k+1))$. By linearity of expectation we get that the expected value of $W(m \cdot (k+1))$ is upper bounded by:

$$4^{\log_2(m \cdot (k+1)) - \log_2 c} \cdot \mathrm{poly}(n) = \left(\frac{m \cdot (k+1)}{c} \right)^2 \cdot \mathrm{poly}(n) = \mathrm{poly}(n)$$

We now turn to show that for every $G \in HC$, the simulator's output distribution is computationally indistinguishable from V^*'s view of interactions with the honest prover P. Specifically,

Lemma 2. *The ensemble $\{S_m^{V^*}(G)\}_{G \in HC}$ is computationally indistinguishable from the ensemble $\{\mathrm{view}_{V^*}^P(G)\}_{G \in HC}$.*

Indistinguishability of the simulator's output from V^*'s view (of $m = \mathrm{poly}(n)$ concurrent interactions with P) is shown assuming that the simulator does not get "stuck" during its execution. Since the simulator S will get "stuck" only with negligible probability (see Lemma 3 below), indistinguishability will immediately follow.

The proof actually considers a "hybrid" simulator that on input $G = (V, E) \in HC$ obtains a directed Hamiltonian Cycle $C \subset E$ in G (as auxiliary input) and uses it in order to produce real prover messages whenever it reaches the second stage of the protocol. Specifically, whenever it reaches the second stage of session $s \in \{1, \ldots, m\}$, the hybrid simulator inspects the $\mathcal{T}$ table and checks whether it has managed to solve session s (thus being able to convince V^* in the section stage of session s). If it has not managed to solve session s, the hybrid simulator outputs $\perp$ and halts. Otherwise, the hybrid simulator follows the prescribed prover strategy and generates prover messages for the second stage of the session (by using the cycle C it possesses). The key for proving the above lies in the following two properties:

- First stage messages output by S are (almost) *identically* distributed to first stage messages sent by P. This property is proved based on the definition of the simulator's actions.
- Second stage messages output by S are *computationally indistinguishable* from second stage messages sent by P. This property is proved based on the special zero-knowledge property of Blum's Hamiltonicity protocol.

We now turn to argue that the hybrid simulator does not get stuck.

Lemma 3. *Let $\alpha : N \to N$ be any super-constant function, let $k(n) = \alpha(n) \cdot \log n$, and consider any instantiation of Protocol 3 with parameter $k = k(n)$. Then for any $i \in \{1, \ldots, m\}$ the probability of the hybrid simulator getting "stuck" on session i during the simulation is negligible.*

Proof. The SOLVE procedure is said to get *stuck* on session i if it reaches the second stage of session i and the following events occur: (1) the history of the interaction so far does not contain an ABORT message in session i, and (2) the table $\mathcal{T}$ does not contain two verifier messages $(\mathrm{V}j)$ and $(\mathrm{V}j)'$ that are replies to two *different* prover messages $(\mathrm{P}j)$ and $(\mathrm{P}j)'$. Note that if the history of the interaction does contain an ABORT message in session i then it is not necessary to obtain σ.

Consider any event in which the SOLVE procedure reaches the second stage of session i, and let hist denote the history of the interaction with which the second stage is reached. By definition of the solve procedure hist contains the messages first visited by the SOLVE procedure.

As before, we divide the analysis into two cases. In the first case, the number of slots in session i as they appear in hist is precisely k. The key for analyzing this case lies the fact that the SOLVE procedure as defined in this paper behaves identically to the SOLVE procedure described in [**?**], except that in the bottom levels of the recursion the former may potentially perform more rewindings than the latter (but *never less*). This means that whenever the PRS variant of the SOLVE procedure manages to obtain the relevant value of σ then so does our variant. By the [**?**] analysis, we know that as long as the number, k, of slots is super logarithmic, the PRS variant of the SOLVE procedure fails to obtain σ with negligible probability. Thus, the probability of getting stuck on session i in our case is negligible as well.

In the second case, the number of slots in session i as they appear in hist is strictly less than k. By definition of our protocol, this can happen only if there exists a slot in the history of the interaction that is footer-free.

Claim: *Suppose that the number of slots in session i is strictly less than k. Then, the schedule of messages as it appears in* hist *contains a slot in the history of the interaction that is footer free.*

Consider now any invocation of a bottom level of the recursion in which a footer free slot j of session i appears amongst messages $(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_t, \mathrm{v}_t) = (\mathsf{hist}, \mathrm{p}, \mathrm{v})$. Let $\mathrm{p}_a = (\mathrm{P}j), \mathrm{v}_b = (\mathrm{V}j)$ be those messages. By definition of the SOLVE procedure, the first messages generated in the visit will appear in hist. Let $\mathrm{p} = (\mathrm{P}j), \mathrm{v} = (\mathrm{V}j)$ be those messages. The simulator will get stuck if and only if: (1) hist does not contain an ABORT message in session i (and in particular if $\mathrm{v}_b \neq$ ABORT), and (2) the table $\mathcal{T}$

does not contain two verifier messages $\mathrm{v}_b = (\mathrm{V}j)$ and $\mathrm{v}_{b'} = (\mathrm{V}j)'$ that are replies to two *different* prover messages $\mathrm{p}_a = (\mathrm{P}j)$ and $\mathrm{p}_{a'} = (\mathrm{P}j)'$. Since p_a and v_b belong to the same session, then if condition (1) is satisfied we have that the following three rewinding conditions hold:

- $\mathrm{v}_b \neq \texttt{ABORT}$,
- both v_b and p_a belong to session i, and
- the slot between messages p_a and v_b is footer-free,

This in particular means that the SOLVE procedure will rewind the interaction in Step 4, sending random p's until it finds another pair $\mathrm{p}'_a = (\mathrm{P}j)', \mathrm{v}' = (\mathrm{V}j)'$ in session i so that $\mathrm{p}_a \neq \mathrm{p}'_a$.

We next show that the probability of getting stuck (over random choices of $\mathrm{p}'_a = r \in \{0,1\}^k$ in the visit to the bottom level of the recursion) is precisely $1/2^k$. Since k is super-logarithmic it will immediately follow that the probability that the simulator gets stuck is negligible.

The key observation for the analysis is that, in the event that the slot between messages p_a and v_b is footer free, it will ultimately be possible to successfully perform the rewinding and reach some $\mathrm{v}' = (\mathrm{V}j)'$ message, *without having to "re-solve" a different session that is nested within the j^{th} slot of session i*. In other words, conditioned on the event of slot j being "footer-free" again (and $(\mathrm{V}j)'$ not being equal to ABORT), the rewinding will go through smoothly (since the simulation cannot get stuck on another session during that specific rewinding attempt).

For any $G \in HC$, and for any choice of hist, let $\zeta_{i,a,b} = \zeta_{i,a,b}(G, \mathsf{hist})$ denote the probability that: (1) message v_b corresponds to a $(\mathrm{V}j)$ message that is not equal to ABORT, (2) both v_b and p_a belong to session i, and (3) the slot between messages p_a and v_b is footer-free. The probability $\zeta_{i,a,b}$ is taken over the random choices of p_a. Using this notation, the SOLVE procedure proceeds to Step 4 with probability $\zeta_{i,a,b}$ (note that the condition in Step 4 is satisfied in each one of the rewinds with probability $\zeta_{i,a,b}$, independently of other rewinds). We would like to bound the probability that S gets stuck (we denote the event of the simulation getting stuck by having S output $\perp$).

Let S^A denote the simulator's execution with black box access to a machine A, let $\widetilde{V}^* = \widetilde{V}^*(\mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{a-1}, \mathrm{v}_{a-1})$ denote the "residual" strategy of V^* when messages $\langle \mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_a \rangle$ are fixed (i.e., $\widetilde{V}^*(G, r) \stackrel{\text{def}}{=} V^*(G, r; \mathrm{p}_1, \mathrm{v}_1, \ldots, \mathrm{p}_{a-1}, \mathrm{v}_{a-1})$), let the phrase "$S$ rewinds in Step (4)" represent the event in which the three rewinding conditions from above hold, and let $\zeta_{i,a,b}$ be as above (in other words, the probability with which the "S rewinds in Step (4)" event holds). We then have:

$$
\begin{aligned}
&\Pr_r\left[S^{\widetilde{V}^*}(G, C) = \perp\right] \\
&= \Pr_r\left[S^{\widetilde{V}^*}(G, C) = \perp \;\middle|\; S \text{ rewinds in Step (4)}\right] \cdot \Pr_r\left[S \text{ rewinds in Step (4)}\right] \quad (2) \\
&= \Pr_r\left[S^{\widetilde{V}^*}(G, C) = \perp \;\middle|\; S \text{ rewinds in Step (4)}\right] \cdot \zeta_{i,a,b} \\
&= \Pr_r\left[\mathrm{p} = \mathrm{p}_t\right] \cdot \zeta_{i,a,b} \quad (3)
\end{aligned}
$$

Now, since p_a and p'_a are uniformly and independently chosen in $\{0,1\}^k$, and since the number of $r \in \{0,1\}^k$ for which $\widetilde{V}^*(G, r)$ is not equal to ABORT is precisely $2^k \cdot \zeta_{i,a,b}$,

then it holds that $\Pr[\mathrm{p}_a = \mathrm{p}'_a] = 1/(2^k \cdot \zeta_{i,a,b})$. Using Eq. 3 we infer that:

$$\Pr_r\left[S^{\widetilde{V}^*}(G,C) = \perp\right] \;=\; \frac{1}{2^k \cdot \zeta_{i,a,b}} \cdot \zeta_{i,a,b} \;=\; \frac{1}{2^k}$$

as required.

Empirical Study Here we provide some cursory evidence that the type of adversarial nesting which causes problems with concurrent simulation do not generally occur when verifiers are independently sending their protocol messages without delaying.

We performed a cursory empirical study of the webserver traffic at our University webserver. We analyzed roughly 122681 TCP sessions (syn-to-fin flows) served by our department webserver over a period of 16 hours; each session consisted of a SYN from a to our webserver, a SYN from the webserver to a, a FIN from a to the webserver, and a final FIN from the webserver to a such that the entire flow corresponded to a request and an error message response served by the webserver. We considered error messages because they are not input/output bound and therefore require roughly the same server processing time. The (4-flow) message pattern corresponds to a 1-slot preamble for our ZK protocol. From this experiment, we counted 26579 nested sessions. In other words, roughly 79% of the sessions were message-free, and would therefore only require 1 slot in our simplest optimistic protocol. (Of the remaining 21%, we cannot determine whether they would have required a second slot given the data set.) Moreover, this small data set reflected a high level of concurrency: there were 57161 instances when one session overlapped another session.

Acknowledgements We thank the anonymous reviewers and Vinod Vaikuntanathan for helpful comments concerning our definition of security, and in particular about the possibility of our protocols leaking information about the presence of other concurrent sessions (as discussed in the beginning of Section 2).

A Comparison with Other Approaches

Consider an alternative Prover strategy that we denote *reset-on-nesting*:

> For some fixed constant C, send a "reset" message to any protocol sessions that have more than C nested sessions that begin and end within a slot. When a slot is reset, the verifier starts the protocol from the beginning.

The security proof for schedules with an upper-bounded number of nestings is straightforward. Moreover, only one slot is needed in this "reset-on-nesting" strategy for the security proof. Unfortunately, the reset-on-nesting idea has two major problems. First is an issue of completeness: it is possible for an honest Prover, and an honest *but very slow* Verifier to repeatedly fail in successfully completing a protocol.

Definition 3 (Completeness). *A concurrent protocol $\Pi = (P_1, \ldots, P_n)$ is* complete*, when for any schedule of concurrently executing sessions, and for every execution between honest parties $P_1, \ldots, P_n$, every P_i eventually* HALT*s and outputs $(1, z)$ (to indicate success).*

A second more troubling problem is one of *intentional starvation*: a malicious Prover may indefinitely postpone a proof by claiming the session has become too nested. An honest verifier has no way to audit the schedule of messages received by the Prover, and thus no recourse but to restart the protocol (which may fail again for the same reason). Even with auditing, the malicious prover may create a fictitious verifier instance and intentionally schedule this verifier so as to create nested sessions in the the honest verifier's slots. Thus, even an "honestly recorded" transcript of all of the Prover's messages could be justifiably used to starve the honest verifier.

Accountable Aborting versus fail To be sure, a malicious prover may ABORT a protocol for many reasons; but this event is fundamentally different than the postponement attack discussed above: An ABORT is an admission of guilt by the malicious prover; a postponement attack is an accusation by the Prover of malice on the part of the Verifier!

Borrowing terminology from the distributed algorithms community, we state the concept of starvation-free protocols below. As mentioned, the solution in this paper is a starvation-free protocol, while the reset-on-nesting protocol is not.

Definition 4 (Starvation-Free Protocol). *A starvation-free concurrent protocol $\Pi = (P_1, \ldots, P_n)$ is one that guarantees that for any adversary P_i^*, and for any schedule of messages of concurrently executing sessions, every honest party $P_j, j \neq i$ interacting with P_i^* eventually* HALT*s with output $(1, z)$ or* ABORT*s with output $(0, z)$.*

(Note, that the z is arbitrary protocol-specific output, i.e., it could be $f(x, y)$ in the case of two-party secure function evaluation.)

A reset-on-nesting protocol cannot be both complete and starvation-free. Either the protocol requires the Verifier to tolerate an infinite number of resets (in order to satisfy completeness)—in which case it is not starvation free—or it requires the Verifier to upper-bound the number of messages it tolerates before ABORT and output 0 (in order to satisfy starvation free-ness)—in which case it is not complete.

For this reason, we prefer our optimistic model to the reset-on-nesting protocol.

A.1 Comparison with the Timing Model

The timing model adds a notion of time on the standard communication model by (a) giving each party a *local* clock, (b) having all parties share a global bound $\rho \geq 1$ on the relative rates of the different clocks (i.e., clock drift), and (c) having all parties share a global bound Δ on the message-delivery time (which includes the time for local computation to receive and prepare messages). Protocols in the timing model can TIMEOUT messages that have not arrived in time Δ, and DELAY outgoing messages by a delay period that is also at least as big as Δ.

Prior work [**?**,**?**,**?**] in this model employ the TIMEOUT and DELAY operations. Protocols in this model have two disadvantages: first, every protocol execution is forced to run for worst-case time $c \cdot \Delta$ even if the parties involved can communicate quickly. Transmission delays to some parts of the internet can be measured in fractions of a day, and so for completeness, Δ would have to be reasonably large. Whereas our protocol allows fast participants to complete interactions "as fast as the network allows," the timing protocols of [**?**,**?**] require all sessions to run in time related to *worst-case* network

delays. Conceptually, our protocol handles more diverse schedules, whereas the timing model protocols use timing to ensure "roughly parallel" composition.

The work of [**?**] reduces the required delay so some small constant $c < 1$. This protocol is major practical improvement to the timing model; however, it too must delay the verifier by some multiplicative penalty of the time it takes for the verifier to respond, and it requires 3 slots. For example, every session must run at least twice as slow (their penalty function is a parameter and can be $\omega(1)$ in some cases also) "as the network allows" and each verifier must still complete multiple slots (whereas in optimistic cases, only 1 slot is required).

New problems of Accountability Unfortunately, any setting of Δ introduces the second more subtle problem with TIMEOUT: much like intentional starvation discussed above, a malicious Prover can send a TIMEOUT to a Verifier to avoid having to abort a session that it cannot complete. The verifier has no way to "contest" this timeout. As we will argue, such a use of TIMEOUTs introduces a new way for a malicious prover to cheat that is not possible in the standard model.

The basis for this problem is that clocks in the timing model must be *local* and unauthenticatable for rewinding of the verifier (or prover in the case of a proof of knowledge) to be possible. If a local clock can be authenticated, then a malicious verifier V^* algorithm could refuse to answer any message that is too old according to its clock, and this would eliminate the possibility of rewinding altogether. As a result, the local timestamp can be forged by any party, and it is not for a third party to verify such a timestamp.

This leads to the problem that transcripts that arise from the following two cases are indistinguishable and *accountability for aborting* is lost:

1. A malicious Prover algorithm receives a message from an honest Verifier, waits for time $\Delta + \epsilon$, and then sends TIMEOUT.
2. A malicious Verifier algorithm delays sending a message for time $\Delta + \epsilon$, and then sends its message. The honest Prover, consequently sends a TIMEOUT messsage.

Let us compare this situation to the standard model in which—say—messages can be authenticated. (Notice that messages *can* be authenticated and still allow rewinding.) Of course, a malicious prover can always abort a protocol by either sending an incorrect message or refusing to send any message. Both cases, however, are fundamentally different then sending when a malicious prover can send a TIMEOUT.

When the Prover sends a bogus message, the verifier has proof that the Prover *cannot* supply a proof of the statement, and the prover is therefore accountable for the abort. In fact, the second case is the same. As described by Canetti [**?**], "not sending a message" in the standard model is handled by an explicit halt which is proof that the Prover has failed.

> Protocol participants are modeled as strict polynomial-time interactive Turing machines; one machine sends a message to another by writing on the recipient's "communication tape." Message delivery is not guaranteed. However, when one machine executes a HALT operation, the other party in a protocol execution is informed of the HALT via this communication tape. This modeling guarantees that one party is not inadvertently left waiting for a message

that will never arrive.[5] It is important to note that the standard model does not have any notion of "time" except for steps of computation. Without time, it is not possible to model "timeouts," and so a refusal to send a message must be modeled in this way.

Thus, the standard model makes it possible to determine which of the two parties cheated in an interaction. In contrast, the timing model with TIMEOUTS allows a malicious Prover to be unaccountable for its cheating.

Comparison with Responsive Round Complexity Cohen, Kilian and Petrank propose the notion of responsive round complexity [**?**]. A protocol is said to have *responsive round complexity* m with party A if it can guarantee that if A responds to every message of the protocol in at most time t, then the overall communication delay of the protocol execution is $m \cdot t$. The idea behind the protocol in that paper is the following:

> *Every verifier is assigned a time bin T. If a verifier V delays a message by time $t < T$, then the prover delays the response to V by time $2T$. If a verifier delays a message by time $t > T$, then the verifier is moved into time bin $2T$, and the verifier must restart the protocol from the beginning.*

Slow verifiers are penalized. In particular, this Prover strategy clumps verifiers into "time buckets" such that all verifiers in the same bucket act in a roughly parallel manner. The analysis then used in the timing model work can be applied.

Overall, the goal of this notion is to assure that a party that always responds quickly has a stronger guarantee on the communication time of the overall protocol. Similarly, our work also attempts to improve the communication time for Verifiers that respond quickly. However, the protocols in [**?**] still have at least $\omega(\log n)$ slots in the best case when the verifiers respond quickly. (In other words, their protocol guarantees response round complexity of $O(\log n)$ whereas our protocol can use only 1 slot when the Verifier responds quickly.)

Buffering Sessions Another idea is for the Prover to buffer sessions so that each one starts only after the previous session finishes. Buffering sessions, i.e. serializing them, eliminates the benefits of having multiple sessions run safely at the same time.

Denial of Service A malicious verifier can "force" the protocol to require just as many rounds as the current best fixed-round $c\mathcal{ZK}$ protocol. This is not a denial-of-service attack because the malicious verifier can only force the same round complexity that the best current protocols achieve—thus, the optimistic approach is never worse than PRS. Moreover, in every additional round forced by a bad schedule, V^* is required to communicate and compute more than the Prover.

Handling Server Farms Our optimistic approach requires the Prover to know the global schedule of Verifier messages. Very large systems, however, are usually built on *clusters* of servers instead of a single machine. Our optimistic approach can be made

[5] In particular, this mechanism is how a malicious party that does not send a message is modeled—since the party must be strict polynomial-time, if it refuses to send a message, we assume it runs a computation, eventually HALTS and then the recipient learns that the other party has aborted.

to work on clusters using a consensus protocol to share schedules among the servers. Since all servers belong to the same entity and are connected through internal fast links, the consensus protocol would work "in the best case" (as opposed to the Byzantine case) for most sessions. In other words, our protocol is viable even after counting the overhead to make all prover machines agree on a schedule of verifier requests. To be sure, many very large systems in existence today require even more complicated consensus on the order of requests that they serve. For example, consider distributed database systems (sometimes distributed over tens of thousands of machines), social network sites, and some distributed file systems that are implemented across thousands of machines.