Concurrent Zero-Knowledge With Timing, Revisited*

Oded Goldreich
Department of Computer Science
Weizmann Institute of Science
Rehovot, Israel.
`oded.goldreich@weizmann.ac.il`

November 2001. Revised: May 25, 2005

Abstract

Following Dwork, Naor, and Sahai (*30th STOC*, 1998), we consider concurrent executions of protocols in a semi-synchronized network. Specifically, we assume that each party holds a local clock such that bounds on the relative rates of these clocks as well as on the message-delivery time are a-priori known, and consider protocols that employ time-driven operations (i.e., `time-out` in-coming messages and `delay` out-going messages).

We show that the constant-round zero-knowledge proof for $\mathcal{NP}$ of Goldreich and Kahan (*Jour. of Crypto.*, 1996) preserves its security when polynomially-many independent copies are executed *concurrently under the above timing model.*

We stress that our main result refers to zero-knowledge of *interactive proofs*, whereas the results of Dwork *et. al.* are either for zero-knowledge *arguments* or for a *weak notion* of zero-knowledge (called ϵ-knowledge) proofs.

Our analysis identifies two extreme schedulings of concurrent executions under the above timing model: the first is the case of *parallel execution* of polynomially-many copies, and the second is of concurrent execution of polynomially-many copies such that only a small (i.e., constant) number of copies are simultaneously active at any time (i.e., *bounded simultaneity*). Dealing with each of these extreme cases is of independent interest, and the general result (regarding concurrent executions under the timing model) is obtained by combining the two treatments.

Keywords: Zero-Knowledge, parallel composition, concurrent composition, timing assumptions, proofs versus arguments, black-box simulation, expected probabilistic polynomial-time.

*Preliminary version has appeared in the proceedings of the 34th *ACM Symposium on the Theory of Computing*, 2002. The current revision was prepared towards publication in a forthcoming book in memory of Shimon Even (1935–2004). I find it especially fitting that my wish to pay tribute to his memory has caused me to fulfill my duty (neglected for a couple of years) to produce a final version of the current work.

Contents

1 Introduction

Zero-Knowledge proofs, introduced by Goldwasser, Micali and Rackoff [22, 23], are fascinating and extremely useful constructs. Their fascinating nature is due to their seemingly contradictory definition: they are both convincing and yet yield nothing beyond the validity of the assertion being proven. Their applicability in the domain of cryptography is vast: they are typically used to force malicious parties to behave according to a predetermined protocol (which requires parties to provide proofs of the correctness of their secret-based actions without revealing these secrets). Such applications are based on the fact, proven by Goldreich, Micali and Wigderson [19], that any language in $\mathcal{NP}$ has a zero-knowledge proof system, provided that commitment schemes exist.[1] The related notion of a zero-knowledge *argument* was suggested (and implemented) by Brassard, Chaum and Crépeau [7], where the difference between proofs and arguments is that in the latter the soundness condition refers only to computationally-bounded cheating provers.

In this work we consider the preservation of zero-knowledge under restricted types of concurrent composition. Specifically, we consider multiple executions of a protocol under a naturally limited model of asynchronous computation (which covers synchronous computation as an important special case). We start by recalling the basic notion of zero-knowledge and providing a wider perspective on the question of its preservation under various forms of composition.

1.1 Zero-knowledge protocols

Recall that an interactive proof system for a language L is a (randomized) protocol for two parties, called verifier and prover, allowing the prover to convince the verifier to accept any common input in L, while guaranteeing that no prover strategy may fool the verifier to accept inputs not in L, except than with negligible probability. The first property is called completeness, and the second is called soundness. The prescribed verifier strategy is always required to be probabilistic polynomial-time. Furthermore, like in other application-oriented works, we focus on prescribed prover strategies that can be implemented in probabilistic polynomial-time given adequate auxiliary input (e.g., an NP-witness in case of NP-languages). Recall that the latter refers to the prover prescribed for the completeness condition, whereas (unlike in argument systems [7]) soundness must hold no matter how powerful the cheating prover is.

Zero-knowledge is a property of some prover-strategies. Loosely speaking, these strategies yield nothing to the verifier, beyond the fact that the input is in the prescribed language L. The fact that "nothing is gained by the interaction" is captured by stating that whatever the verifier can efficiently compute after interacting with the (zero-knowledge) prover on a specific common input, can be efficiently computed from the assertion itself, without interacting with anyone. Thus, the formulation of the zero-knowledge condition considers two ensembles of probability distributions, each ensemble associates a probability distribution to each input in L: The first ensemble represents the output distribution of the verifier after interacting with the specified prover strategy P, where the verifier is using an arbitrary efficient (i.e., probabilistic polynomial-time) strategy, not necessarily the one specified by the protocol. The second ensemble represents the output distribution of some probabilistic polynomial-time algorithm (which does not interact with anyone). The basic paradigm of zero-knowledge asserts that for every ensemble of the first type there exist a "similar" ensemble of the second type. The specific variants differ by the interpretation given to the notion of 'similarity', and in this work (as in most of the literature) we focus on the most liberal interpretation. Under this (liberal) interpretation, similarity means computational indistinguishability

[1] Or, equivalently [27, 24], that one-way functions exist.

(i.e., failure of any efficient procedure to tell the two ensembles apart). The ensembles $\{X_\alpha\}$ and $\{Y_\alpha\}$ are said to be computationally indistinguishable if, for every efficient procedure D (and every α), it holds that

$$|\Pr[D(\alpha, X_\alpha)=1] - \Pr[D(\alpha, Y_\alpha)=1]| \;<\; \mu(|\alpha|)$$

where μ is a *negligible function* (i.e., a function vanishing faster than the reciprocal of any positive polynomial).

1.2 Composition of zero-knowledge protocols

A fundamental question regarding zero-knowledge proofs (and arguments) is whether the zero-knowledge condition is preserved under a variety of composition operations. Three types of composition operations were considered in the literature, and we briefly review these operations and what is known about the preservation of the zero-knowledge condition under each of them.

1.2.1 Sequential composition

Here the protocol is invoked (polynomially) many times, where each invocation follows the termination of the previous one. At the very least, security (e.g., zero-knowledge) should be preserved under sequential composition, otherwise the applicability of the protocol is severely limited (because one cannot safely use it more than once).

Although the basic definition of zero-knowledge (as in the preliminary version of Goldwasser *et. al.* [22]) is not closed under sequential composition (cf. [18]), a minor augmentation of it (by auxiliary inputs) is closed under sequential composition (cf. [20]). Indeed, this augmentation was adopted in all subsequent works (as well as in the final version of Goldwasser *et. al.* [23]).

1.2.2 Parallel composition

Here (polynomially) many instances of the protocol are invoked at the same time and proceed at the same pace. That is, we assume a synchronous model of communication, and consider (polynomially) many executions that are totally synchronized such that the ith round message in all instances is sent exactly (or approximately) at the same time.

Goldreich and Krawczyk presented a simple protocol that is zero-knowledge (in a strong sense), but is not closed under parallel composition (even in a very weak sense) [18]. At the time, their result was interpreted mainly in the context of *round-efficient error reduction*; that is, the construction of full-fledge zero-knowledge proofs (of negligible soundness error) by composing (in parallel) a basic zero-knowledge protocol of high (but bounded away from 1) soundness error. Since alternative ways of constructing constant-round zero-knowledge proofs (and arguments) were found (cf. [17, 15, 8]), interest in parallel composition (of zero-knowledge protocols) has died. In retrospect, as we argue in §1.4.3, this was a conceptual mistake.

We comment that parallel composition is problematic also in the context of reducing the soundness error of arguments (cf. [3]), but our focus here is on the zero-knowledge aspect of protocols regardless if they are proofs, arguments or neither.

1.2.3 Concurrent composition

This notion of concurrent composition generalizes both the notions of sequential composition and parallel composition. Here (polynomially) many instances of the protocol are invoked at arbitrary times and proceed at arbitrary pace. That is, we assume an asynchronous (rather than synchronous) model of communication.

In the 1990's, when extensive two-party (and multi-party) computations became a reality (rather than a vision), it became clear that it is (at least) desirable that cryptographic protocols maintain their security under concurrent composition (cf. [12]). In the context of zero-knowledge, concurrent composition was first considered by Dwork, Naor, and Sahai [13]. Their actual suggestions refer to a model of naturally-limited asynchronicity (which certainly covers the case of parallel composition). Essentially, they assumed that each party holds a local clock such that the relative clock rates as well as the message-delivery time are bounded by a-priori known constants, and considered protocols that employ time-driven operations (i.e., `time-out` in-coming messages and `delay` outgoing messages). This timing model is the main focus of the current paper (and we shortly discuss the pure asynchronous model in §1.4.3). The previously known main results *for the timing model* are (cf. [13]):

- Assuming the existence of one-way functions, every language in $\mathcal{NP}$ has a constant-round concurrent zero-knowledge *argument*.
- Assuming the existence of two-round perfectly-hiding commitment schemes (which in turn imply one-way functions), every language in $\mathcal{NP}$ has a constant-round concurrent ϵ-knowledge proof, where ϵ-knowledge means that (for every noticeable function $\epsilon:\mathsf{N}\to(0,1]$) a simulator working in time poly$(n/\epsilon(n))$ can produce output that is ϵ-indistinguishable from the one of a real interaction. (For further discussion of ϵ-knowledge, see Section 1.6.)

Thus, no *constant-round proofs* for $\mathcal{NP}$ were previously known to be concurrent zero-knowledge (under the timing model). We comment that proof with *non-constant* number of rounds were known to be concurrent zero-knowledge (even in the pure asynchronous model; cf. §1.4.3).

1.3 Our results

Our main result closes the gap mentioned above, by showing that a (known) constant-round zero-knowledge proof for $\mathcal{NP}$ is actually *concurrent zero-knowledge under the timing model.* That is, we prove:

Theorem 1.1 *The* (five-round) *zero-knowledge proof system for* $\mathcal{NP}$ *of Goldreich and Kahan [17] is* concurrent zero-knowledge under the timing model.

Thus, the zero-knowledge property of the proof system (of [17]) is preserved under any concurrent composition that satisfies the timing model. In particular, *the zero-knowledge property is preserved under parallel composition*, a result we consider of independent interest.

Recall that the proof system of [17] relies on the existence of two-round perfectly-hiding commitment schemes (which is implied by the existence of claw-free pairs of functions and implies the existence of one-way functions). Thus, we get:

Theorem 1.2 *Assuming the existence of two-round perfectly-hiding commitment schemes, there exists a* (constant-round) *proof system for* $\mathcal{NP}$ *that is* concurrent zero-knowledge under the timing model.

Using the same techniques, we can show that several other known (constant-round) zero-knowledge protocols remain secure under the concurrent timing-model. Examples include the (constant-round) zero-knowledge *arguments* of Feige and Shamir [15] and of Bellare, Jakobsson and Yung [4]. The latter protocol (referred to as the BJY-protocol) is of special interest because it is a four-round argument for $\mathcal{NP}$ that relies only on the existence of one-way functions. The above protocols are simpler (and use fewer rounds) than the argument systems previously shown (in [13]) to be concurrent zero-knowledge (under the timing-model), alas their security (under this model) is established by a more complex simulator. (See further details in Section 6.1.)

1.4 Discussion of some issues

We clarify some issues that underly our study. Some of these issues were mentioned explicitly above.

1.4.1 The meaning of composition

We stress that when we talk of composition of protocols (or proof systems) we mean that the honest users are supposed to follow the prescribed program (specified in the protocol description) that refers to a single execution. That is, the actions of honest parties in each execution are independent of the messages they received in other executions. The adversary, however, may coordinate the actions it takes in the various executions, and in particular its actions in one execution may depend also on messages it received in other executions.

Let us motivate the asymmetry between the independence of executions assumed of honest parties but not of the adversary. Coordinating actions in different executions is typically difficult but not impossible. Thus, it is desirable to use composition (as defined above) rather than to use protocols that include inter-execution coordination-actions, which require users to keep track of all executions that they perform. Actually, trying to coordinate honest executions is even more problematic, because one may need to coordinate executions of different honest parties (e.g., all employees of a big cooperation or an agency under attack), which in many cases is highly unrealistic. On the other hand, the adversary attacking the system may be willing to go into the extra trouble of coordinating its attack on the various executions of the protocol.

1.4.2 Important zero-knowledge technicalities

We shortly discuss seemingly technical but actually fundamental variants on the basic definition of zero-knowledge. In particular, these variants play an important role in our work.

Auxiliary inputs and non-uniformity: As mentioned above, almost all work on zero-knowledge actually refer to zero-knowledge with respect to (non-uniform) auxiliary inputs. This work is no exception, but (as in most other work) the reference to auxiliary inputs is typically omitted. We comment that zero-knowledge with respect to auxiliary inputs "comes for free" whenever zero-knowledge is demonstrated (like in this work) via a black-box simulator (see below). The only thing to bear in mind is that allowing the adversary (non-uniform) auxiliary inputs means that the computational assumptions that are used need to be non-uniform ones. For example, when we talk of computational-binding (resp., computational-hiding) commitment schemes we mean that the binding (resp., hiding) property holds with respect to any family of polynomial-size circuits (rather than with respect to any probabilistic polynomial-time algorithm).

Black-box simulation: The definition of zero-knowledge (only) requires that the interaction of the prover with any cheating (probabilistic polynomial-time) verifier be simulateable by an ordinary probabilistic polynomial-time machine (which interacts with no one). A black-box simulator is one that can simulate the interaction of the prover with any such verifier when given oracle access to the strategy of that verifier. All previous zero-knowledge arguments (or proofs), with the exception of the recent (constant-round) zero-knowledge argument of Barak [1], are established using a black-box simulator, and our work is no exception (i.e., we also use a black-box simulator). Indeed, Barak demonstrated that (contrary to previous beliefs) non-black-box simulators may exist in cases where black-box ones do not exist [1]. However, black-box simulators, whenever they exist, are preferable to non-black-box ones, because the former offers greater security: Firstly, as mentioned above, black-box simulators imply zero-knowledge with respect to auxiliary inputs.[2] Secondly, black-box simulators imply polynomial bounds on the knowledge tightness, where *knowledge tightness* is the (inverse) ratio of the running-time of any cheating verifier and the running-time of the corresponding simulation [16, Sec. 4.4.4.2].[3]

Expected polynomial-time simulators: With the exception of the recent (constant-round) zero-knowledge argument of Barak [1], all previous *constant-round* arguments (or proofs) utilize an *expected* polynomial-time simulator (rather than a *strict* polynomial-time simulator). (Indeed our work inherits this "feature" of [17].) As recently shown by Barak and Lindell [2], this is no coincidence, because all the above (with the exception of [1]) utilize black-box simulators, whereas no *strict* polynomial-time black-box simulator can demonstrate the zero-knowledge property of a constant-round argument system for a language out of $\mathcal{BPP}$.

1.4.3 Types of concurrent composition

We shortly discuss various types of asynchronous concurrent composition, starting with the pure asynchronous model and ending with the synchronous (or parallel) model.

Perspective: the pure asynchronous model. Regarding the pure asynchronous model, the current state of the art is as follows:

- Black-box simulator cannot demonstrated the concurrent zero-knowledge property of non-trivial proofs (or arguments) having significantly less than logarithmically many rounds (cf. Canetti *et. al.* [10]). By *non-trivial* proof systems we mean ones for languages outside $\mathcal{BPP}$, whereas by *significantly less than logarithmic* we mean any function $f : \mathsf{N} \to \mathsf{N}$ satisfying $f(n) = o(\log n/\log\log n)$.
- Every language in $\mathcal{NP}$ has a concurrent zero-knowledge proof with almost-logarithmically many rounds, and this can be demonstrated using a black-box simulator (cf. [28], building upon [25], which in turn builds upon [29]).
- Recently, Barak [1] demonstrated that the "black-box simulation barrier" can be bypassed. With respect to concurrent zero-knowledge he only obtains partial results: constant-round zero-knowledge arguments (rather than proofs) for $\mathcal{NP}$ that maintain security as long as an

[2] In contrast, whether or not a non-black-box simulator implies zero-knowledge *with respect to auxiliary inputs*, depends on the specific simulator: In fact, in [1], Barak first presents (as a warm-up) a protocol with a non-black-box simulator that cannot handle auxiliary inputs, and next uses a more sophisticated construction to handle auxiliary inputs.

[3] That is, a protocol is said to have knowledge tightness $k:\mathsf{N}\to\mathsf{R}$ if for some polynomial p and every probabilistic polynomial-time verifier V^* the interaction of V^* with the prover can be simulated within time $k(n)\cdot T_{V^*}(n)+p(n)$, where T_{V^*} denotes the time complexity of V^*. In fact, the running-time of the simulator constructed by Barak [1] is polynomial in T_{V^*}, and so the knowledge tightness of his protocol is not bounded by any polynomial.

a-priori bounded (polynomial) number of executions take place concurrently. (The length of the messages in his protocol grows linearly with this a-priori bound.)[4]

Thus, it is currently unknown whether constant-round arguments for $\mathcal{NP}$ may be concurrent zero-knowledge (in the pure asynchronous model).

On the timing model: The timing model consists of the *assumption* that talking about the actual timing of events is meaningful (at least in a weak sense) and of the *introduction of time-driven operations*. The *timing assumptions* amount to postulating that each party holds a local clock and knows a global bound, denoted $\rho \geq 1$, on the relative rates of the local clocks.[5] Furthermore, it is postulated that the parties know a (pessimistic) bound, denoted Δ, on the message-delivery time (which also includes the local computation and handling times). In our opinion, these timing assumptions are most reasonable, and are unlikely to restrict the scope of applications for which concurrent zero-knowledge is relevant. We are more concerned about the effect of the *time-driven operations* introduced in the timing model. Recall that these operations are the `time-out` of incoming messages and the `delay` of out-going messages. Furthermore, typically (and in fact also in our work), the delay period is at least as long as the time-out period,[6] which in turn is at least Δ (i.e., the time-out period must be at least as long as the pessimistic bound on message-delivery time so not to disrupt the proper operation of the protocol). This means that the use of these time-driven operations yields slowing down the execution of the protocol (i.e., running it at the rate of the pessimistic message-delivery time rather than at the rate of the actual message-delivery time, which is typically much faster). Still, in absence of more appealing alternatives (i.e., a constant-round concurrent zero-knowledge protocol for the pure asynchronous model), the use of the timing model may be considered reasonable. (We comment that other alternatives to the timing-model include various set-up assumptions; cf. [9, 11].)

On parallel composition: Given our opinion about the timing model, it is not surprising that we consider the problem of parallel composition almost as important as the problem of concurrent composition in the timing model. Firstly, it is quite reasonable to assume that the parties' local clocks have approximately the same rate, and that clock drifting is corrected by occasional clock synchronization. Thus, it is reasonable to assume that the parties have approximately-good estimates of some global time. Furthermore, the global time may be partitioned into phases, each consisting of a constant (e.g., 5) number of rounds, so that each party wishing to execute the protocol just delays its invocation to the beginning of the next phase. Thus, concurrent execution of

[4]We are quite sure that Barak's arguments remain zero-knowledge under concurrent executions that satisfy the *timing model*. But since these are arguments (rather than proofs) such a result will not improve upon the previously known result of [13] (which uses black-box simulations).

[5]Defining the rate of a clock is less straightforward than one may think. Firstly, clocks (or rather their reading) are typically discrete, and thus their relative rate is a ratio between pairs of reading (i.e., initial reading and final reading). Thus, rate must be computed with respect to sufficiently long time intervals. In particular, these intervals should be long enough such that the effect of a single change in the clock reading (i.e., a single "clock tick") can be neglected. Secondly, the clock rate may change with time, and so the aforementioned time intervals should not be too long. In the context of the current work, it is reasonable to measure the clock rate with respect to time intervals of length Δ. Thus, when we say that the relative rate of two clocks is ρ we mean that a time period of Δ units on one clock is measured as at least Δ/ρ (and at most $\rho\Delta$) units on the other clock.

[6]Following the conference presentation of this work, Barak and Micciancio raised the possibility of using a delay period that is smaller and yet linearly related to the time-out period. It seems plausible that, following their approach, security will deteriorate exponentially with the constant of the said proportion. We stress that so far their approach was not proved to work, but it does indicate that the common practice (of using a delay period that is at least as long as the time-out period) may not be inherent to the model.

(constant-round) protocols in this setting amounts to a sequence of (time-disjoint) almost-parallel executions of the protocol. Consequently, proving that the protocol is parallel zero-knowledge suffices for concurrent composition in this setting.

1.5 Techniques

To discuss our techniques, let us fix a timing assumption (i.e., an a-priori bound ρ on the local clock rates and a bound Δ on the message-delivery time) and consider a c-round protocol that utilizes appropriately selected `time-out` and `delay` mechanisms (which depend on the above bounds; e.g., timing-out in-coming messages after Δ time units). The reader may think of the bound on the relative rates of local clock as being close to 1 (or even just 1; i.e., equal rates), and of c as being a constant (in fact, we will use $c = 5$). Furthermore, suppose that *all prover's actions* in the protocol are time-driven (by the `time-out` and `delay` mechanisms).

A key observation underlying our work is that a concurrent scheduling (of such protocol instances) under the timing model can be *decomposed* into a sequence of parallel executions, called blocks, such that the number of simultaneously active blocks is bounded by $O(c)$. That is, each block consists of protocol instances that are executed almost in parallel, and the number of blocks that are (pairwise) active at the same time is $O(c)$, where two blocks are said to be active at the same time if for some time t each block has a protocol instance that is active at time t. The constant in the O-notation depends on the a-priori known bound on the relative clock rates. This decomposition applies whenever the timing model is used (and is not restricted to the context of zero-knowledge), and it may be useful towards the analysis of the concurrent execution of any set of protocols under the timing model.

Let us clarify the above observation by providing a proof for a special (simple) case. Our first simplifying assumption is that the clock rates are all equal. We further assume that the prover utilizes equal delays between its messages, and that these delays are four times the length of the time-out period, which is defined as our basic time unit. Considering an arbitrary scheduling of protocol instances, under the aforementioned timing model, we place a protocol instance in the i^{th} block if it is invoked during the i^{th} time-interval (i.e., the time interval $(i-1, i]$). Then, each block consists of an almost-parallel execution of instances of the protocol (i.e., the $(j+1)$-st message in any instance of block i is supposed to be sent at time $t + 4j > i - 1 + 4j$ and is timed-out at time $t + 4j + 1 < i + 4j + 1$, where $t \in (i-1, i]$ is the invocation time of this instance). Clearly, the i^{th} and j^{th} blocks are simultaneously active (at some time) only if $|i - j| < 4c$, where c is the number of rounds in the protocol. Thus, at most $8c + 1$ blocks are simultaneously active.

In view of the above, it is quite natural to conjecture that in order to analyze the concurrent composition of protocols under the timing model it suffices to deal with two extreme schedulings: the parallel scheduling and the bounded-simultaneity scheduling. Indeed, this conjecture is essentially correct in the cases considered in this work (i.e., for certain zero-knowledge proofs).

Handling parallel composition. At first glance, one may be tempted to say that the techniques used for proving that the Goldreich–Kahan (GK) protocol is zero-knowledge (cf. [17] and Section 2.3) extend to showing that it remains zero-knowledge under parallel composition. This would have been true if we were handling *coordinated* parallel executions of the GK-protocol (where the prover would abort all copies if the verifier decommits improperly in any of them). However, this is not what we are handling here (i.e., parallel composition refers to uncoordinated parallel execution of many copies of the protocol). Consequently, a couple of new techniques are introduced in order to deal with the parallel composition of the GK-protocol. We consider these simulation techniques to be of independent interest.

Handling bounded-simultaneity concurrent composition. Experts in the area may not find it surprising that the GK-protocol remains zero-knowledge under bounded-simultaneity concurrent composition. In fact, previous works (e.g., [13]) suggest that the difficulty in simulating concurrent executions of the GK-protocol arises from the case in which a large number of instances is executed in a "nested" (and in particular simultaneous) manner.[7] Furthermore, the work of Richardson and Kilian [29] suggests that certain (related) protocols may be zero-knowledge under bounded-simultaneity concurrent composition. Still, to the best of our knowledge, such a technically-appealing result has not been proven before. We prove the result by using a rather straightforward approach, which nevertheless requires careful implementation. We stress that not every zero-knowledge protocol remains zero-knowledge under bounded-simultaneity concurrent composition (e.g., Goldreich and Krawczyk presented a simple (constant-round) protocol that is zero-knowledge, but parallel execution of two instances of it is not zero-knowledge [18]).

Handling the general case. Combining the techniques employed in handling the two extreme cases, we show that (augmented with suitable timing mechanisms) the GK-protocol is concurrent zero-knowledge under the timing model. This is shown by using the abovementioned decomposition, and applying the bounded-simultaneity simulator to the blocks while incorporating the parallel-composition simulator inside of it (i.e., to the individual blocks). Note that, by definition, the bounded-simultaneity simulator handles the special case in which each block contains a single copy, and does so by employing the single-copy simulator. Capitalizing on the high-level similarity of the parallel-composition simulator and the single-copy simulator, we just need to extend the bounded-simultaneity simulator by incorporating the former simulator in it. (Our presentation of the bounded-simultaneity simulator uses terminology that makes this extension quite easy.)

1.6 Zero-knowledge versus ϵ-knowledge

Recall that ϵ-knowledge means that for every noticeable function (i.e., a reciprocal of some positive polynomial) $\epsilon : \mathsf{N} \to (0,1]$ there exists a simulator working in time $\mathrm{poly}(n/\epsilon(n))$ that produces output that is ϵ-indistinguishable from the one of a real interaction, where n denotes the length of the input and the ensembles $\{X_\alpha\}$ and $\{Y_\alpha\}$ are said to be ϵ-indistinguishable if for every efficient procedure (e.g., a polynomial-time algorithm) D, it holds that

$$|\Pr[D(\alpha, X_\alpha)=1] - \Pr[D(\alpha, Y_\alpha)=1]| \;<\; \epsilon(|\alpha|) + \mu(|\alpha|)$$

where μ is a negligible function. (Indeed, the standard notion of computational indistinguishability [21, 31] is a special case obtained by setting $\epsilon \equiv 0$.)

Indeed, as mentioned in [13], ϵ-knowledge does provide some level of security. However, this level of security is lower than the one offered by the standard notion of zero-knowledge, and more so when compared to simulators with bounded knowledge tightness (as discussed above; cf. [16, Sec. 4.4.4.2]). Furthermore, unlike zero-knowledge, the notion of ϵ-knowledge is not closed under sequential composition (i.e., t sequential executions of a ϵ-knowledge protocol yield a $t \cdot \epsilon$-knowledge (rather than ϵ-knowledge) protocol).

[7] In fact, even if each level of nesting only multiplies the simulation time by a factor of 2, we get an exponential blow-up.

Expected polynomial-time simulators versus ϵ-knowledge. The above discussion applies also to the comparison of ϵ-knowledge and zero-knowledge via expected polynomial-time simulators (rather than via strict polynomial-time simulators). Furthermore, simulation by an expected polynomial-time simulator implies ϵ-knowledge simulator (running in strict time inversely proportional to ϵ).[8] The converse does not hold (e.g., consider a prover that, for $i = 1, 2...$, with probability 2^{-i} sends the result of a BPTime(2^{2i})-complete computation).

We comment that even a stronger notion of ϵ-knowledge, by which the simulator's running-time is linear (rather than polynomial) in $1/\epsilon$ does not seem to imply zero-knowledge (via an expected polynomial-time simulator). Note that the naive attempt (of converting the former simulator into one that establishes zero-knowledge) fails.[9]

1.7 Relation to Shimon Even (a personal comment)

This work grew out of my sudden realization that the question of parallel composition of zero-knowledge protocols has not received the attention that it deserves. Specifically, when asked for a protocol that preserves zero-knowledge under parallel composition, one would have referred to the preservation of zero-knowledge under concurrent composition (possibly in the timing model). Thus, a potentially easier problem was reduced to a harder problem, which is not the 'right' way to go. Things were even worst because, as argued in §1.4.3, the preservation of zero-knowledge under parallel composition is a natural and important problem.

Readers that were fortunate to know Shimon well will immediately associate the mood of the previous paragraph with him. Indeed, the moment I reached the conclusion stated above, I got reminded of Shimon.

I then asked myself whether I already know of a simple protocol that preserve zero-knowledge under parallel composition, and my immediate conjecture was that this should be true of the GK-protocol. Once I proved this conjecture, which turned out to be harder to establish than I've originally thought, I asked myself whether this argument can be extended further (i.e., to concurrent composition under the timing model). Thus, I have established results similar to those known before, using a different approach that goes from a natural special case to the general case. This entire development reminds me again of Shimon.

Finally, I wish to recall another connection to Shimon. In 1978, as an undergraduate, I attended his course *Graph Algorithms*. At some point, one student was annoyed at Shimon's "untraditional" way of analyzing algorithms and asked whether Shimon's arguments constituted a proof and if so what is a proof. Shimon answer was immediate, short and clear: *A proof is whatever convinces me.* A few years later, when first seeing the definition of interactive proofs, I was reminded of Shimon's answer. I think that interactive proofs are a perfect formalization of Shimon's intuition: interactive proofs are indeed convincing, and essentially any convincing argument is actually an interactive proof.

1.8 Organization

In Section 2, we recall some basic notions as well as review the *GK-protocol* (i.e., the five-round zero-knowledge proof system of Goldreich and Kahan [17]). In Section 3 we prove that the GK-protocol remains zero-knowledge under parallel composition. In Section 4 we prove that the GK-

[8]This can be seen by truncating all runs of the original simulator that exceed its expected running-time by a factor of $1/\epsilon$ (or so).

[9]That is, selecting i with probability 2^{-i} and invoking the former simulator with $\epsilon = 2^{-i}$ does yield an expected polynomial-time simulator, but its output may not be computationally indistinguishable from the real interaction.

protocol remains zero-knowledge under bounded-simultaneity concurrent composition. The latter two sections can be read independently of one another, and are believed to be of independent interest.

In Section 5, we augment the GK-protocol with adequate `time-out` and `delay` mechanisms, and prove that the resulting protocol is concurrent zero-knowledge under the timing model. This is done by extending the simulator presented in Section 4, where the extension relies on the ideas underlying the simulator presented in Section 3. We conclude (cf. Section 6) by applying our techniques to the zero-knowledge argument system of Bellare, Jakobsson and Yung [4] and by presenting a class of protocols to which our techniques can be applied.

2 Background

Zero-knowledge is a property of some prover-strategies. Loosely speaking, it means that anything that is feasibly computable by (possibly improperly) interacting with the prover, can be feasibly computable without interacting with the prover. That is, the most basic definition of zero-knowledge (of a prover P w.r.t a language L) requires that, for every probabilistic polynomial-time verifier strategy V^*, there exists a probabilistic polynomial-time simulator M^* such that the following two probability ensembles are computationally indistinguishable:

1. $\{\langle P, V^* \rangle(x)\}_{x \in L} \stackrel{\rm def}{=}$ the output of V^* when interacting with P on common input $x \in L$; and
2. $\{M^*(x)\}_{x \in L} \stackrel{\rm def}{=}$ the output of M^* on input $x \in L$.

(The formulation can be easily extended to allow for auxiliary inputs to V^*; cf. Definition 2.1.) Recall that the ensembles $\{X_\alpha\}_{\alpha \in S}$ and $\{Y_\alpha\}_{\alpha \in S}$ are said to be computationally indistinguishable if, for every efficient procedure D, it holds that

$$|\Pr[D(\alpha, X_\alpha)=1] - \Pr[D(\alpha, Y_\alpha)=1]| \ < \ \mu(|\alpha|)$$

where μ is a negligible function. Recall that $\mu : \mathsf{N} \to [0,1]$ is called negligible if it vanishes faster than the reciprocal of any positive polynomial (i.e., for every positive polynomial p and all sufficiently large n, it holds that $\mu(n) < 1/p(n)$). We say that an event occurs with overwhelmingly high probability if it occurs with probability that is negligibly close to 1 (i.e., the event occurs with probability $1 - \mu$, where μ is a negligible function). Indeed, our entire treatment will refer to executions that are parameterized by some parameter, denoted n, which is polynomially related to the length of some relevant input.

2.1 Expected polynomial-time simulation and black-box simulation

As discussed in the introduction, we use two variants of the above definition (or definitional schema): One one hand, we allow the simulator to run in *expected* probabilistic polynomial-time (rather than *strict* probabilistic polynomial-time). On the other hand, we require the simulator to be implementable by a universal machine that gets oracle access to the (verifier) strategy V^*. See [16, Sec. 4.3.1.6] (resp., [16, Sec. 4.5.4.2] and [1]) for further discussion of the first (resp., second) issue.

Definition 2.1 (black-box zero-knowledge):
Next message function: *Let B be an interactive Turing machine, and x, z, r be strings representing a common-input, auxiliary-input, and random-input, respectively. Consider the function $B_{x,z,r}(\cdot)$ describing the messages sent by machine B such that $B_{x,z,r}(\overline{m})$ denotes the message sent by B on common-input x, auxiliary-input z, random-input r, and sequence of incoming messages $\overline{m}$. For simplicity, we assume that the output of B appears as its last message.*

Black-box simulator: *We say that an* expected *probabilistic polynomial-time oracle machine* M *is a* black-box simulator for the prover P and the language L *if for every polynomial-time interactive machine* B*, every probabilistic polynomial-time oracle machine* D*, every positive polynomial* $p(\cdot)$*, all sufficiently large* $x \in L$*, and every* $z, r \in \{0,1\}^*$*:*

$$\left|\Pr\left[D^{B_{x,z,r}}(\langle P, B_r(z)\rangle(x))=1\right] - \Pr\left[D^{B_{x,z,r}}(M^{B_{x,z,r}}(x))=1\right]\right| < \frac{1}{p(|x|)}$$

where $B_{x,z,r}$ *is the next-message function define above, and* $B_r(z)$ *denotes the interaction of machine* B *with auxiliary-input* z *and random-input* r*. That is,* $\langle P, B_r(z)\rangle(x)$ *denotes the output of* B*, having auxiliary-input* z *and random-input* r*, when interacting with* P *on common input* x*.*

We say that P *is* black-box zero knowledge *if it has a black-box simulator.*

Note that an auxiliary-input for the verifier is explicitly incorporated in Definition 2.1, whereas an auxiliary input for the prover is only implicit in it. Specifically, P may be a probabilistic polynomial-time that is given an adequate additional information regarding the common input x as an auxiliary input (e.g., an NP-witness that $x \in L$, in case L is in $\mathcal{NP}$).

2.2 Parallel and concurrent zero-knowledge and the timing model

The definition of parallel and concurrent zero-knowledge are derived from Definition 2.1 by considering appropriate adversaries (i.e., adversarial verifiers) that invoke multiple copies of the (basic) protocol. For simplicity, we will assume throughout this work that all copies are invoked on the same (common) input. Each execution of such an individual copy is called a session. In case of parallel zero-knowledge, we consider adversaries that simultaneously invoke a polynomial number of copies (or sessions) of the protocol, and interact with this multitude of copies in a synchronized way (i.e., send their i^{th} message in all copies at the same time). In case of concurrent zero-knowledge, we consider adversaries that invoke a polynomial number of copies (or sessions), and interleave their interaction with this multitude of copies in an arbitrary way. In case of concurrent zero-knowledge under the timing model, the interleaving of executions by the adversary must satisfying the timing model. (Without loss of generality, we may assume that the adversary never violates the time-out condition; it may instead send an illegal message at the latest possible adequate time.)[10]

An important technicality. As discussed by Canetti *et. al.* [10], Definition 2.1 is too restrictive for serving as a basis for a definition of (unbounded) composition, where the adversary B may invoke a (polynomial) number of sessions with P but this polynomial is not a-priori known. The problem is that the universal (black-box) simulator may invoke (the next message function associated with) B only for an (expected) polynomial number of times, whereas B may describe a strategy that initiates a larger number of sessions with P. One solution is to consider for each polynomial a different universal simulator that can handle all adversaries that invoke at most a number of sessions (with P) that is bounded by that polynomial. For simplicity, we adopt this solution here.

[10] Furthermore, without loss of generality, we may assume that all the adversary's messages are delivered at the latest possible adequate time. The latter assumption is justified by noting that the prescribed prover strategy may be modified such that a time-out condition applied to the verifier's message is always followed by at least a similar delay of the next prover message. (Indeed, this may slow down some executions in which the verifier is honest, but never slows them down by more than can be caused by a cheating verifier.) We comment that this modification is unnecessary in our protocol (of Section 5), since it already satisfies the above convention.

2.3 The Goldreich–Kahan (GK) Protocol

Loosely speaking, the Goldreich–Kahan (GK) proof system for Graph 3-Colorability (G3C) proceeds in four steps:

1. The verifier commits to a challenge (i.e., sequence of edges in the input graph).
2. The prover commits to a sequence of values (i.e., the colors of each vertex under several random relabelings of a fixed 3-coloring of the graph). This sequence is partitioned into subsequences, each corresponding to a different random relabeling of the coloring of the graph.
3. The verifier decommits (to the edge-sequence).
4. If the verifier has properly decommits then the prover decommits to a subset of the values as indicated by the decommitted challenge. Otherwise the prover sends nothing.

   Specifically, the challenge is a sequence of edges, each associated with an independently selected 3-coloring of the graph, and the prover responses to the $i^{\rm th}$ edge by decommitting to the values in the $i^{\rm th}$ committed coloring that correspond to the end-points of the $i^{\rm th}$ edge.

A detailed description of the above protocol is provided in Construction 2.2 (below). We note that many of the specific details are not important to our analysis, and are provided merely for sake of clarity. We highlight a couple of points that are relevant to the analysis: Firstly, the prover's commitment is via a commitment scheme that is (perfectly-binding but only) computationally-hiding, and so commitments to different values are (only) computationally-indistinguishable (which considerably complicates the analysis; cf. [17]). Secondly, the verifier's commitment is via a commitment scheme that is (perfectly-hiding but only) computationally-binding, and so it is (only) infeasible for it to properly decommits in two different way (which slightly complicates the analysis).

Implementation Details: The Goldreich–Kahan protocol [17] utilizes two "dual" commitment scheme (see terminology in [16, Sec. 4.8.2]). The first commitment scheme, denoted C, is used by the prover and has a perfect-binding property. For simplicity, we assume that this scheme is non-interactive, and denote by $C(v)$ a random variable representing the output of C on input v (i.e., a commitment to value v).[11] The second commitment scheme, denoted $\mathcal{C}$, is used by the verifier and has a perfect-hiding property. Such a scheme must be interactive, and we assume that it consists of the receiver sending a random index, denoted α, and the committer responds by applying the randomized process $\mathcal{C}_\alpha$ to the value it wishes to commit to (i.e., $\mathcal{C}_\alpha(v) = \mathcal{C}(\alpha, v)$ represents a commitment to v relative to the receiver's message α). Consequently, Step 1 in the high-level description is implemented by Steps P0 and V1 below.

Construction 2.2 (The GK zero-knowledge proof for G3C):

Common Input: *A simple* (3-colorable) *graph* $G=(V,E)$.

Let $n \stackrel{\rm def}{=} |V|$, $V = \{1, ..., n\}$, *and* $t \stackrel{\rm def}{=} 2n \cdot |E|$.

Auxiliary Input to the Prover: *A 3-coloring of* G*, denoted* ψ*.*

Prover's preliminary step (P0): *The prover invokes the commit phase of the perfectly-hiding commitment scheme, which results in sending to the verifier a message* α*.*

[11] Non-interactive perfectly-binding commitment schemes can be constructed using any one-way *permutation*. In case one wishes to rely here only on the existence of one-way *functions*, one may need to use Naor's two-round perfectly-binding commitment scheme [27]. This calls for a minor modification of the description below.

Verifier's commitment to a challenge (V1): *The verifier uniformly and independently selects a sequence of t edges, $\overline{e} \stackrel{\text{def}}{=} ((u_1, v_1), ..., (u_t, v_t)) \in E^t$, and sends to the prover a random commitment to these edges. Namely, the verifier uniformly selects $s \in \{0,1\}^{\text{poly}(n)}$, and sends $c \stackrel{\text{def}}{=} \mathcal{C}_\alpha(\overline{e}, s)$ to the prover.*

Motivating Remark: *At this point the verifier is effectively committed to a sequence of t edges.* (This commitment is of perfect secrecy.)

Prover's commitment step (P1): *The prover uniformly and independently selects a sequence of t random relabeling of the 3-coloring ψ, and sends the verifier commitments to the color of each vertex under each of these colorings. That is, the prover uniformly and independently selects t permutations, $\pi_1, ..., \pi_t$, over $\{1,2,3\}$, and sets $\phi_j(v) \stackrel{\text{def}}{=} \pi_j(\psi(v))$, for each $v \in V$ and $1 \leq j \leq t$. It uses the perfectly-binding commitment scheme to commit itself to the colors of each of the vertices according to each 3-coloring. Namely, the prover uniformly and independently selects $r_{1,1}, ..., r_{n,t} \in \{0,1\}^n$, computes $c_{i,j} = C(\phi_j(i), r_{i,j})$, for each $i \in V$ and $1 \leq j \leq t$, and sends $c_{1,1}, ..., c_{n,t}$ to the verifier.*

Verifier's decommitment step (V2): *The verifier decommits the sequence $\overline{e} = ((u_1, v_1), ..., (u_t, v_t))$ to the prover. Namely, the verifier send $(s, \overline{e})$ to the prover.*

Motivating Remark: *At this point the entire commitment of the verifier is revealed. The verifier now expects to receive, for each j, the colors assigned by the j^{th} coloring to vertices u_j and v_j* (i.e., the endpoints of the j^{th} edge in $\overline{e}$).

Prover's partial decommitment step (P2): *The prover checks that the message just received from the verifier is indeed a valid revealing of the commitment c made by the verifier at Step* (V1) *(i.e., it checks that $c = \mathcal{C}_\alpha(\overline{e}, s)$ indeed holds). Otherwise the prover halts immediately. Let us denote the sequence of t edges, just revealed, by $(u_1, v_1), ..., (u_t, v_t)$. The prover reveals* (to the verifier), *for each j, the j^{th} coloring of vertices u_j and v_j, along with appropriate decommitment information. Namely, the prover sends to the verifier the sequence of four-tuples*

$$(r_{u_1,1}, \phi_1(u_1), r_{v_1,1}, \phi_1(v_1)), ..., (r_{u_t,t}, \phi_t(u_t), r_{v_t,t}, \phi_t(v_t))$$

Verifier's local decision step (V3): *The verifier checks whether, for each j, the values in the j^{th} four-tuple constitute a correct revealing of the commitments $c_{u_j,j}$ and $c_{v_j,j}$, and whether the corresponding values are different. Namely, upon receiving $(r_1, \sigma_1, r_1', \tau_1)$ through $(r_t, \sigma_t, r_t', \tau_t)$, the verifier checks whether for each j, it holds that $c_{u_j,j} = C(\sigma_j, r_j)$, $c_{v_j,j} = C(\tau_j, r_j')$, and $\sigma_j \neq \tau_j$* (and both are in $\{1,2,3\}$). *If all conditions hold then the verifier accepts. Otherwise it rejects.*

Goldreich and Kahan proved that Construction 2.2 constitute a (constant-round) zero-knowledge interactive proof for Graph 3-Colorability [17]. (We briefly review their simulator below.) Our first goal, undertaken in Section 3, is to show that the zero-knowledge property (of Construction 2.2) is preserved under parallel composition. We later extend the result to yield concurrent zero-knowledge under the timing-model.

High level description of the simulator used in [17]. The simulator (using oracle access to the verifier's strategy) proceeds in three main steps:

The Scan Step: The simulator emulates Steps (P0)–(V2), by using commitments to dummy values in Step (P1), and obtains the verifier's decommitment for Step (V2), which may be either proper or not. In case of improper decommitment the simulator *outputs the partial transcript just generated and* `halts`. Otherwise, it records the sequence $(u_1, v_1), ..., (u_t, v_t)$, just revealed, and proceeds as follows.

The Approximation Step: For technical reasons (discussed below), the simulator next approximates the probability that the first scan ended with a proper decommitment. (This is done by repeated trials, each as in the first scan, until some polynomial number of proper decommitments is found.)

The Generation Step: Using the (proper) decommitment information (i.e., the edge sequence $(u_1, v_1), ..., (u_t, v_t)$), obtained in the first scan, the simulator repeatedly tries to generate a full transcript by emulating Steps (P1)–(V2), using commitments to "pseudo-colorings" that do not "violate the coloring conditions imposed by the decommitted edges". That is, in each trial, the simulator sets $c_{i,j}$ to be a commitment to a dummy value if $i \notin \{u_j, v_j\}$, and sets $c_{u_j,j}$ and $c_{v_j,j}$ to be commitments to two different random values in $\{1, 2, 3\}$. The number of trials is inversely proportional to the probability estimated in the approximation step.

This completes the (high level) description of the simulator used in [17]. We conclude this section with a discussion of the purpose of the Approximation Step.

The purpose of the Approximation Step. The foregoing simulation procedure is a variant of the more natural (and in fact naive) procedure in which the Approximation Step is omitted and the Generation Step is repeated (indefinitely) untill a full transcript is generated. The problem with the naive variant is that the probability (denoted p) of proper verifier decommitment during the Scan Step is *not* identical to the probability (denoted p') of a proper verifier decommitment during the Generation Step. The difference is due to the fact that in the Scan Step we feed the verifier with commitments to dummy values, whereas in the Generation Step we feed the verifier with commitments to "pseudo-colorings". Indeed, the hiding property of commitment schemes guarantees that $|p - p'|$ is negligible (in n), but this does not mean that p/p' is upper-bounded by a polynomial in n (e.g., $p = 2^{-n/3}$ and $p' = 2^{-n/2}$). Thus, the expected running-time of the naive simulation procedure (i.e., $(p/p') \cdot \text{poly}(n)$) is *not* necessarily polynomial. This problem is resolved by the actual simulation procedure of [17] outlined above, whose running time is $p \cdot \frac{\text{poly}(n)}{\tilde{p}}$, where $\tilde{p} = \Theta(p)$ is the approximation of p (obtained in the Approximation Step, and $\tilde{p} = \Theta(p)$ holds with probability $1 - 2^{-\text{poly}(n)}$).

An alternative approach. An alternative way of coping with the aforementioned problem is to use a different protocol that allows for the Scan Step to use the same distribution as in the Generation Step. This approach was recently pursued by Rosen [30], who suggested an alternative constant-round zero-knowledge proof for $\mathcal{NP}$ (by adapting the protocol of [29]). Rosen's protocol could be applied in the context of the current paper and yield a noticeable simplification of the proof of our main results (of Sections 3–5), but this will not allow to obtain the secondary results presented in Section 6 (which refer to protocols that do not satisfy the stronger property stated above). Furthermore, using Rosen's protocol avoids a natural problem that we would like to treat in the current paper, because this problem is likely to arise in future work (where, like in Section 6, it may not be avoided).

3 Simulator for the Parallel Case

Recall that the GK-protocol proceeds in four (abstract) steps:

1. The verifier commits to a challenge.

   (The actual implementation is by two rounds/messages.)
2. The prover commits to a sequence of values.

   (The challenge specifies a subset of the locations in the latter sequence.)
3. The verifier decommits to its challenge (either properly or not).
4. Pending on the verifier's proper decommitment, the prover decommits to the corresponding values.

The basic approach towards simulating this protocol (without being able to answer a random challenge) is to first run the first three steps with prover-commitments to arbitrary (dummy) values, obtaining the challenge, and then rewind to Step 2 and make a prover-commitment that passes this specific challenge (alas no other challenge). In case the verifier always decommits properly, this allows to easily simulate a full run of the protocol. In case the verifier always decommits improperly, things are even easier because in this case we only need to simulate Steps 1–3. The general case is when the verifier decommits with some probability. Intuitively, this can be handled by outputting the initial transcript of Steps 1–3 in case it contains an improper decommitment, and repeatedly trying to produce a full passing transcript (as in the first case) otherwise. Difficulties arise in case the probability of proper verifier decommitment is small but not negligible and furthermore when it depends (in a negligible way) on whether the prover commits to dummy or to "passing" values. Indeed, the focus of [17] is on resolving this problem (and their basic approach is to approximate the probability of proper decommitment in case of dummy values, and keep trying to produce a full passing transcript for at most a number of times that is inversely proportional to the latter probability).

The problem we face here is more difficult: several (say n) copies of the protocol are executed in parallel and the verifier may properly decommit in some of them but not in others. Furthermore, the verifier decision regarding in which copies it properly decommits may depend on the prover's messages in all copies. That is, in the general case, each (parallel) execution of Steps 1-3 may yield a different configuration (out of 2^n possible ones) of proper/improper decommitment in the n copies. Still, we need to simulate a transcript of all steps in copies in which the verifier commits properly. Thus, the naive generalization of the case $n = 1$ (which consists of insisting on generating the same configuration as in the initial run) will not work.[12] Instead, referring to the n probabilities that correspond to proper decommitment in each of the n copies, we add additional rewindings in which we try to obtain a proper decommit from all copies that have at least as high a probability as the copies that actually performed proper-decommitment in the initial simulated run. That is, letting p denote the minimum probability of proper-decommitment taken only over the copies that have proper-decommitted in the initial run, we try to obtain the challenges of all copies having proper-decommitment probability at least p. Once these challenges are obtained, we try to generate a parallel run in which *only* copies having at least as high a probability (but not necessarily all of them!) properly decommit. Furthermore, in order not to skew the distribution (towards high proper-decommitment probabilities), we insist on having at least one copy with a corresponding probability as low as some copy in the initial run. That is, we try to generate a parallel run in which only copies having proper-decommitment probability at least p perform proper-decommit,

[12] We refer to a procedure that obtains some challenges via an initial "dummy" execution of Steps 1-3, and next tries to produce an adequate simulation by repeatedly rewinding Steps 2-4 until one obtains again the same configuration. This may fail because all 2^n configurations may be equally likely, in which case the simulation is expected to make 2^n trials.

while insisting that at least one copy having proper-decommitment probability approximately p performs proper-decommit.

One obvious problem with the above description is that we do not know the relevant proper-decommitment probabilities. Indeed, we may obtain good (multiplicative) approximation of them, but using these approximations in a straightforward manner will not do (because such approximations do not allow to rank the actual probabilities).[13] Instead, we cluster the n copies according to the probability that each of them properly decommits, and try to obtain a proper decommit from all the copies that are in the same (or heavier) cluster as the copies that properly decommit in the initial simulated run. Once this is obtained, we try to generate a parallel run in which only copies that belong to the above (or heavier) cluster (but not necessarily all of them) properly decommit. As one may expect, clusters are defined according to threshold probabilities, but picking these thresholds naively (e.g., as fixed quantities) is going to fail. Below, we will pick these thresholds at random from fixed intervals.

3.1 A high level description

Recall that our aim is to analyze the parallel execution of the GK-protocol. Specifically, we will consider n copies of the protocol, being executed in parallel under the coordinated attack of an adversary (called a verifier) that plays the role of the verifier in all copies. The parameter n is polynomially related to the length of the input to each of these copies, and thus we deal with the general case of parallel composition (of the GK-protocol). When we say that some quantities are negligible or overwhelmingly high we refer to these quantities as a function of the parameter n.

The following basic notions are central to our analysis (of the parallel execution of the GK-protocol): An execution of a copy (of the GK-protocol) is said to properly decommit if the verifier message in Step 3 is a valid decommitment to its (i.e., the verifier's) commitment in Step 1. In the first part of the simulation, we use prover's commitments to arbitrary values, which are referred to as commitment to dummy values. Later (in the simulation) we use commitments to values that will pass for a certain challenge (which is understood from the context). These are called commitment to passing values.[14] In addition, we also refer to the following more complex notions and notations:

- Let p_i denote the probability that the verifier properly decommits in the i^{th} copy of the parallel run, when Step 2 is played with commitment to dummy values.

  (When using other commitments (e.g., passing commitments) the probability of proper decommitment may be any p'_i such that $|p'_i - p_i|$ is negligible.)

- We will use a sequence of thresholds, denoted $t_1, ..., t_n$, that will be determined (probabilistically) on the fly such that with overwhelmingly high probability it holds that
  1. $t_j \in (2^{-(j+1)}, 2^{-j})$,
  2. no p_i lies in the interval $[t_j \pm (1/9n) \cdot 2^{-j}]$.

[13] Consider, for example, the case that each of the copies properly decommits with probability $(1/2) \pm \epsilon(n)$ for some negligible function ϵ or even for $\epsilon(n) = 1/t(n)$, where $t(n)$ is the running time of our approximation procedure.

[14] Recall that in the actual implementation (of the GK-protocol), challenges correspond to sequences of t edges (over the vertex-set $\{1, 2..., n\}$), and the prover commits to a sequence of $t \cdot n$ values in $\{1, 2, 3\}$ (i.e., a block of n values per each of the t edges). For a given edge sequence (i.e., a challenge), a passing sequence of values is one in which (for every i) the values assigned to the i^{th} block are such that the endpoints of the i^{th} edge (in the challenge) are assigned a (random) pair of distinct elements.

Such t_j's exist and t_j can be found when given approximations of all p_i's up-to $(1/9n) \cdot 2^{-j}$ (or so). We also define $t_0 \stackrel{\rm def}{=} 1$, and so $p_i \leq t_0$ for all i. We assume, without loss of generality, that for every i it holds that $p_i > 2^{-n}$, and so each p_i lies in one of the intervals $(t_j, t_{j-1}]$.

- For such t_j's, define $T_j = \{i : p_i > t_j\}$. (Indeed, $T_0 = \emptyset$, $T_{j-1} \subseteq T_j$ for all j, and $T_n = \{1, ..., n\}$.)

  Membership in T_j can be determined (probabilistically with negligible error probability) in time $\mathrm{poly}(n) \cdot 2^j$, since t_j was selected to be sufficiently far-away from all the p_i's (i.e., $|t_j - p_i| = \Omega(2^{-j}/n)$).

- Referring to a specific run of the parallel execution, we denote by E_j the event that the verifier properly decommits to some copy in $T_j \setminus T_{j-1}$ but to no copy outside T_j. That is, we consider the set of copies in which the verifier properly decommits in the specific run (of the parallel execution), and say that E_j holds if j is the minimum integer such that the said set contains an element of T_j. (Equivalently, j is the minimum integer such that the said set contains an element of $T_j \setminus T_{j-1}$.)

  Let $q_j = \Pr[E_j]$, when E_j refers to a random run with dummy values. Note that $q_j \leq n \cdot t_{j-1}$ (because E_j mandates that the verifier properly decommits to some copy in $T_j \setminus T_{j-1}$, which implies one of $|T_j \setminus T_{j-1}| \leq n$ events, each occuring with probability at most t_{j-1}). However, q_j may be much smaller than $t_j < t_{j-1}$, because the event E_j refers to n possibly dependent events (occurring in n sessions).

  Since $\{1, ..., n\} = T_n \supseteq T_{n-1} \cdots \supseteq \cdots T_1 \supseteq T_0 = \emptyset$, whenever the verifier properly decommits in some copy, one of the events E_j (for $j \geq 1$) must hold. Otherwise (i.e., whenever the verifier decommits improperly in all copies), we say that event E_0 holds.

We now turn to the simulator, which generalizes the one in [17]. All approximations referred to below are quite good w.v.h.p. (i.e., with $1 - 2^{-n}$ each approximation is within a factor of $(1 + (1/\mathrm{poly}(n))$ of the corresponding value). Loosely speaking, after fixing the verifier's coins (at random), the simulator proceeds as follows (while using the residual verifier strategy as a black-box):

Step S0: Obtain the verifier's commitments (of Step 1) in the n parallel copies.

Step S1: The purpose of this step is to generate an index $j \in \{0, 1, ..., n\}$ with distribution corresponding the probability that event E_j holds for a random parallel execution of the protocol, as well as to determine the sets T_j and T_{j-1} (as defined above, based on adequate thresholds t_j and t_{j-1}, which will be selected too). This has to be done in expected polynomial time. Recalling that event E_j occurs with probability $O(n/2^j)$, when we select a specific j, we may use $\mathrm{poly}(n) \cdot 2^j$ steps.

We stress that we only determine the sets T_j and T_{j-1}, for the specific j that is selected, rather than determine all sets (i.e., $T_1, ..., T_n$). The sets T_j and T_{j-1} will allow us to determine (in subsequent steps) whether or not event E_j holds for other random parallel executions of the protocol.

The selection of j as well as the determination (or construction) of the sets T_j and T_{j-1} is achieved as follows:

1. First we simulate Steps 2–3 of the (parallel execution of the) protocol, while using (in Step 2) commitments to dummy values. Based on the verifier's decommitments in Step 3 (of the parallel execution), we determine the set $I \subseteq [n]$ of copies in which the verifier has properly decommitted.

2. Next, we determine an appropriate sequence $t_1, ..., t_j$ of thresholds such that event E_j holds for the simulated run. Specifically, we determine the t_j's on the fly, starting with t_1, until we see that E_j holds. Thus, we stop without determining $t_{j+1}, ..., t_n$.
3. Finally, using t_{j-1} and t_j, we determine for each $i \in \{1, 2, ..., n\}$ whether or not $p_i > t_j$ (i.e., $i \in T_j$) and whether or not $p_i > t_{j-1}$ (i.e., $i \in T_{j-1}$). (Actually, this is done for every $i \in I$ during the prior sub-step, since the values of $I \cap T_j$ and $I \cap T_{j-1}$ are used to determine that E_j holds.)

Indeed, the above description (especially of the second sub-step) does not specify how the corresponding actions are performed (let alone within time $\text{poly}(n) \cdot 2^j$). We defer these crucial details to Section 3.2, where we show how to actually implement the current step within time $\text{poly}(n) \cdot 2^j$.

Step S2: For each copy $i \in T_j$, we wish to obtain the challenge committed to in Step 1, while working within time $\text{poly}(n) \cdot 2^j$. This is done by rewinding and re-simulating Steps 2–3 for at most $\text{poly}(n) \cdot 2^j$ times, while again using (in Step 2) commitments to dummy values.

Step S3: For technical reasons[15], analogously to [17], we next obtain a good (i.e., constant factor) approximation of $q_j = \Pr[E_j]$. This approximation, denoted $\tilde{q}_j$, will be obtained within expected time $\text{poly}(n)/q_j$ by repeated rewinding and re-simulating Steps 2–3. (Specifically, we continue till we see some fixed polynomial number (say n^5) of runs in which event E_j holds.)

Step S4: We now try to generate a simulation of Steps 2–3 in which event E_j occurs. However, unlike in previous simulations, here we use (in Step 2) commitments to values that pass the challenges that we have obtained. This will allow us to simulate also Step 4, and complete the entire simulation.

Specifically, we make at most $\text{poly}(n)/\tilde{q}_j$ trials to rewind and re-simulate Steps 2–3, while using (in Step 2 of each copy in T_j) commitments to values that pass the corresponding challenge (which we obtained in Step S2). If the verifier answers (for Step 3) fit event E_j then we proceed to simulate Step 4 in the obvious manner. Otherwise, we rewind and try again (but never try more than $\text{poly}(n)/\tilde{q}_j$ times).

A more detailed description of the above steps is provided in Sections 3.2 and 3.3. A detailed analysis of the simulator is provided in Section 3.4, relying on the following observations:

1. Pending on the ability to properly implement Step S1, the (overall) expected running time of the simulation is some fixed polynomial, because each attempt (in Steps S2, S3, and S4) is repeated for a number of times that is inversely proportional to the probability of entering this repeated-attempts step. Specifically, each of these steps is repeated at most $(\text{poly}(n)/\tilde{q}_j) \approx (\text{poly}(n)/q_j)$ times (use $q_j = O(n \cdot 2^{-j})$ for Step S2), whereas j is selected with probability q_j.

2. The computational-binding property of $\mathcal{C}$ implies that we rarely get into trouble in Step S4; that is, only with negligible probability will it happen that in Step S4 the verifier properly decommits to a value different from the one to which it has properly decommitted in Step S2.

[15] We refer the reader to the end of Section 2.3 for a discussion of the purpose of the approximation step. Note that this step could have been eliminated if we had follows Rosen's alternative approach (also discussed at the end of Section 2.3).

3. Since the probabilities of verifier's proper-decommitment (in Step 3) are almost unaffected by the prover's commitments (of Step 2) and since passing commitments look like commitments to truly valid values, the simulated interaction is computationally indistinguishable (cf. [21, 31]) from the real one.

3.2 Setting the thresholds and implementing Step S1

One naive approach is to try to use fixed thresholds such as $t_j = 2^{-j}$. However, this may not allow to determine (for a given i), with high probability and within time $\mathrm{poly}(n) \cdot 2^j$, whether or not p_i is smaller than t_j. (The reason being that p_i may be very close to 2^{-j}; e.g., $|p_i - 2^{-j}| = 2^{-2n}$.)

Instead, the t_j's will be selected in a more sophisticated way such that they are approximately as above (i.e., $t_j \approx 2^{-j}$) but also far enough (i.e., at distance at least $2^{-j}/9n$) from each p_i. This will allow us to determine, with high probability and within time $\mathrm{poly}(n) \cdot 2^j$, whether or not p_i is smaller than t_j. The question is how to set the t_j's such that they are appropriately far from all p_i's. Since the p_i's are unknown probabilities (which we can only approximate), it seems infeasible to come-up with a deterministic setting of the t_j's. Indeed, we will settle for a probabilistic setting of the t_j's (provided that this setting is independent of other events).

Recall that Step S1 calls for the setting of $t_1, ..., t_j$ such that event E_j holds (for a random run), where whether or not event E_j holds depends on t_j and t_{j-1}. Furthermore, it is important that the setting of t_{j-1} in case event E_j holds be the same as the setting of t_{j-1} in case event E_{j-1} holds. Moreover, recalling that the setting of t_j must be performed in time $\mathrm{poly}(n) \cdot 2^j$, we cannot afford to set all t_k's whenever we set a specific t_j. Still, we provide below an adequate threshold-setting process. We start with the following key procedure, which selects $t_j \approx 2^{-j}$ such that with overwhelmingly high probability $|p_i - t_j| > 2^{-j}/9n$ for every i. We stress that the following procedure (has to run and indeed) runs in time $\mathrm{poly}(n) \cdot 2^j$, which requires a slightly non-straightforward implementation.[16]

Procedure $T(j,n)$, returns $t_j \in [(3/4) \pm (1/8)] \cdot 2^{-j} \subset (2^{-(j+1)}, 2^{-j})$: The procedure first approximates all p_i's sufficiently well, and then sets t_j in the desired interval such that t_j is sufficiently far from all the approximated values of the p_i's. A specific implementation follows.

1. For $i = 1, ..., n$, the procedure approximates p_i sufficiently well (in the following sense, which is motivated in Footnote 16). Specifically, with overwhelmingly high probability, the approximated value, denoted a_i, should satisfy:

   (a) If $p_i < 2^{-j-2}$ then $a_i < 2^{-j-1}$.

   (b) If $p_i > 2^{-j+1}$ then $a_i > 2^{-j}$.

   (c) If $2^{-j-2} \le p_i \le 2^{-j+1}$ then $|a_i - p_i| < (1/19n) \cdot 2^{-j}$.

   Each approximation is produced in time $\mathrm{poly}(n) \cdot 2^j$ as follows. First, we decide whether or not $p_i \ge 2^{-j-2}$. Actually, we distinguish with overwhelmingly high probability, between the case $p_i \ge 2^{-j-2}$ and (say) the case $p_i < 2^{-j-3}$, where in the intermediate range any decision is admissible. Likewise, we decide whether or not $p_i \le 2^{-j+1}$ (i.e., distinguish between the case $p_i \le 2^{-j+1}$ and the case $p_i > 2^{-j+2}$). These decisions can be made using $\mathrm{poly}(n) \cdot 2^j$ trials.

[16]The straightforward approach is to approximate each p_i up to an additive deviation of $\Theta(2^{-j}/n)$. The problem is that, in general, this requires $\Omega((2^{-j}/n)^{-2})$ samples. However, for $p_i \approx 2^{-j}$, such an additive approximation translates to a multiplicative approximation of $1 \pm \Theta(1/n)$, which can be obtained based on a sample of size $\mathrm{poly}(n)/p_i = \mathrm{poly}(n) \cdot 2^j$. We note that, for $p_i \notin [2^{-j-2}, 2^{-j+1}]$, a more crude approximation suffices, and can be obtained using a sample of size $\mathrm{poly}(n) \cdot 2^j$.

In case we decided that $p_i \in [2^{-j-2}, 2^{-j+1}]$, we approximate p_i up-to an additive deviation of $(1/19n) \cdot 2^{-j}$, which can be implemented using $\text{poly}(n) \cdot 2^j$ trials (because it calls for an approximation to within a factor of $1 \pm \Theta(1/n)$). Otherwise, we output the threshold value (i.e., $a_i = 2^{-j-2}$ if we decided that $p_i < 2^{-j-2}$ and $a_i = 2^{-j+1}$ if we decided that $p_i > 2^{-j+1}$).

Note that if $p_i < 2^{-j-2}$ (and even if $p_i < 2^{-j-3}$) then both $a_i = 2^{-j-2}$ and $a_i = p_i \pm 2^{-j}/19n$ satisfy $a_i < 2^{-j-1}$. Similarly, if $p_i > 2^{-j+1}$ then both $a_i = 2^{-j+1}$ and $a_i = p_i \pm 2^{-j}/19n$ satisfy $a_i > 2^{-j}$. Finally, if $2^{-j-2} \leq p_i \leq 2^{-j+1}$ then we decided that $p_i \in [2^{-j-2}, 2^{-j+1}]$ and produced $a_i = p_i \pm 2^{-j}/19n$ as required.

2. Starting from a set of evenly spaced points in the desired interval (i.e., $\{(5/8), (5/8) + (1/4n), ..., (5/8) + (n/4n)\}$), we discard all points that are close to one of the a_i's obtained in Step 1. Specifically, the procedure determines

$$K \leftarrow \left\{ k \in \{0,1,...,n\} : (\forall i)\ a_i \notin \left(\frac{5}{8} + \frac{k}{4n} \pm \frac{1}{8n}\right) \cdot 2^{-j} \right\} . \tag{1}$$

That is, a_i rules out the value k if $a_i \in (5n + 2k \pm 1) \cdot 2^{-j}/8n$. Note that K is not empty, because each a_i can rule out at most one element of K (whereas $|\{0,1,...,n\}| = n+1$ and they are only n values of i).

Select an arbitrary (say at random or the first) $k \in K$. Output $t_j = ((5/8) + (k/4n)) \cdot 2^{-j}$.

By construction, $|t_j - a_i| \geq (1/8n) \cdot 2^{-j}$, for all i's. If $p_i \in [2^{-j-2}, 2^{-j+1}]$ then $|a_i - p_i| \leq (1/19n) \cdot 2^{-j}$ (with overwhelming probability), and it follows that p_i does not fall in the interval $t_j \pm (1/9n) \cdot 2^{-j}$ (because $|p_i - t_j| \geq |a_i - t_j| - |a_i - p_i| \geq ((1/8n) - (1/19n)) \cdot 2^{-j} > (1/9n) \cdot 2^{-j}$). Otherwise (i.e., if either $p_i < 2^{-j-2}$ or $p_i > 2^{-j+1}$), p_i does not fall in the interval $t_j \pm (1/9n) \cdot 2^{-j} \subset (2^{-j-2}, 2^{-j+1})$ (simply by the case hypothesis). We conclude that, with overwhelming probability, no p_i falls in the interval $t_j \pm (1/9n) \cdot 2^{-j}$.

Implementation of Step S1: Recall that the purpose of Step S1 is to generate an index $j \in \{0,1,...,n\}$ with distribution corresponding the probability that event E_j holds (for a random parallel run of the protocol), as well to determine the thresholds $t_1, ..., t_j$, and using these to determine for every $i = 1, ..., n$, whether or not $i \in T_j$ and whether or not $i \in T_{j-1}$. We thus start by generating a random run, and next determine all necessary objects with respect to it.

1. *Generating a* reference run: Simulate Steps 2–3 of the (parallel execution of the) protocol, while using (in Step 2) commitments to dummy values. Based on the verifier's decommitments in Step 3 (of the parallel execution), determine the set $I \subseteq [n]$ of copies in which the verifier has properly decommitted.

2. *Determining the event E_j occuring in the reference run, as well as the sets T_j and T_{j-1}:*
   Case of empty I: Set $j = 0$ and $T_j = T_{j-1} = \emptyset$.
   Case of non-empty I: Set $t_0 = 1$ and $T_0 = \emptyset$. For $j = 1, ..., n$ do
   (a) $t_j \leftarrow T(j, n)$. (We stress that the value of t_j is set obliviously of I.)
   (b) Determine the set T_j by determining, for each i, whether or not $p_i > t_j$. We use approximations to each p_i (as computed in procedure $T(j,n)$), and rely on $|p_i - t_j| > (1/9n) \cdot 2^{-j}$. Recall that for each i, we obtain an approximation a_i such that $|a_i - p_i| < (1/9n) \cdot 2^{-j}$ if $2^{-j-2} \leq p_i \leq 2^{-j+1}$ and $a_i < 2^{-j-1} \leq t_j$ (resp., $a_i > 2^{-j} \geq t_j$) if $p_i < 2^{-j-2} < t_j$ (resp., if $p_i > 2^{-j+1} > t_j$). Thus, we may decide that $p_i > t_j$ if and only if $a_i > t_j$.

(c) Decide whether or not event E_j holds for the reference run, by using T_{j-1} (of the previous iteration) and T_j (just computed). Recall that event E_j holds (for the reference run) if and only if both $I \subseteq T_j$ and $I \not\subseteq T_{j-1}$ hold.

(d) If event E_j holds then exit the loop with the current value of j as well as with the values of T_j and T_{j-1}. Otherwise, proceed to the next iteration.

Since we have assumed that $(\forall i)\ p_i > 2^{-n}$, some event E_j must hold.[17]

A key point in the analysis is that the values of the T_k's, as determined by Step S1 (i.e., $T_0, ..., T_j$), are independent of the value of j. Of course, which of the T_k's were determined does depend on the value of j. Thus, we may think of Step S1 as of an efficient implementation of the mental experiment in which all T_k's are determined, next j is determined accordingly (analogously to the above), and finally one outputs T_j and T_{j-1} for subsequent use.

3.3 A detailed description of the simulator

For sake of clarity we present a detailed description of the simulator, before turning to its analysis. Recall that our aim is to simulate a parallel execution of n copies of the GK-protocol. We start by selecting and fixing the verifier's coins at random. With respect to these fixed coins, we simulate the interaction of the *residual deterministic* verifier (with copies of the predetermined prover) as follows:

Step S0: We simulate the parallel execution of Step 1 (i.e., Steps P0 and V1 of Construction 2.2) as follows. First, acting as the real prover in Step P0, we randomly generate messages $\alpha^1, ..., \alpha^n$ (one per each copy). Invoking the verifier (as per Step V1), while feeding it with $\alpha^1, ..., \alpha^n$, we obtain its n commitments, $c^1, ..., c^n$, for the n copies.

Step S1: As explained in Section 3.2, we determine (*for a random reference run*)[18] the index j for which E_j holds, as well as the sets T_j and T_{j-1}. Recall that this (and specifically procedure $T(\cdot,\cdot)$) involves $\mathrm{poly}(n) \cdot 2^j$ rewindings and re-simulations of Steps 2–3, while using commitments to dummy values. Each rewinding is performed as in Step S2 below.

In case $j = 0$, we may skip all subsequent steps, and just output the reference run produced in the current step.

Step S2: For each copy $i \in T_j$, we wish to obtain the challenge (edge-sequence) committed to in Step 1, while working within time $\mathrm{poly}(n) \cdot 2^j$. This is done by rewinding and re-simulating Steps 2–3 (i.e., Steps P1 and V2 of Construction 2.2) for $\mathrm{poly}(n) \cdot 2^j$ times, while using commitments to dummy values. (Actually, we may as well do the same for all i's (regardless whether $i \in T_j$ or not), but we are guaranteed to succeed only for i's in T_j. Furthermore, we may work on all i's concurrently.)

Specifically, each rewinding attempt proceeds as follows:

[17] Removing this assumption enables the situation that no event E_j occurs. This may happen only if $p_i \leq t_n < 2^{-n}$, for every $i \in I$. But the probability that the reference run corresponds to such a set I is at most $\sum_{i:p_i<2^{-n}} p_i < n \cdot 2^{-n}$, and we may ignore this rare event. Alternatively, we may modify the verifier such that $p_i > 2^{-n}$ holds for all i, by making it properly decommit to all copies with probability 2^{-n+1}, and note that the execution of the modified verifier is indistinguishable from the execution of the original verifier.

[18] Here and in the sequel, when referring to runs and steps of the protocol, we actually means steps in the parallel execution of the protocol.

1. Generate n sequences of random (prover) commitments to the dummy value 0. That is, for every (copy) $i = 1, ..., n$, select uniformly $r^i_{1,1}, ..., r^i_{n,t} \in \{0,1\}^n$, and compute $\overline{c}^i \stackrel{\text{def}}{=} (c^i_{1,1}, ..., c^i_{n,t})$, where $c^i_{k,\ell} = C(0, r^i_{k,\ell})$.
2. Feeding the verifier with (the n prover commitments) $\overline{c}^1, ..., \overline{c}^n$, obtain the verifier's n (Step 3) responses, denoted $(s^1, \overline{e}^1), ..., (s^n, \overline{e}^n)$.
3. For every properly decommitted copy (i.e., i such that $c^i = \mathcal{C}_{\alpha_i}(s^i, \overline{e}^i)$), store the corresponding challenge (i.e., the edge sequence $\overline{e}^i$).

(Note that it is unlikely that we will obtain two conflicting proper decommitments to the same verifier commitment c^i.)

Step S3: For technical reasons, analogously to [17], we next obtain a good approximation of $q_j = \Pr[E_j]$. This approximation, denoted $\tilde{q}_j$, will be obtained within expected time $\text{poly}(n)/\tilde{q}_j$ by repeated rewinding and re-simulating Steps 2–3 (i.e., Steps P1 and V2 of Construction 2.2). Specifically, we repeat the following steps until we obtain n^5 runs in which event E_j holds.

1. Perform Items 1 and 2 as in Step S2. Let I' denote the set of copies in which the verifier has properly decommitted.
2. If I' fits event E_j (i.e., $I' \subseteq T_j$ and $I' \not\subseteq T_{j-1}$) then increment the "success counter" by one unit. (We proceed to the next iteration only if the "success counter" is still smaller than n^5.)

Suppose we have obtained n^5 successes while making τ trials. Then we set $\tilde{q}_j = n^5/\tau$.

Step S4: We now try to generate a simulation of Steps 2–3 of the protocol (i.e., Steps P1 and V2 of Construction 2.2) in which event E_j occurs. However, unlike in previous simulations, here we use (in Step 2) prover-commitments to values that pass the challenges that we have obtained. This will allow us to simulate also Step 4, and complete the entire simulation. Specifically, we make at most $\text{poly}(n)/\tilde{q}_j$ trials to rewind and re-simulate Steps 2–3, while using (in Step 2 of each copy in T_j) commitments to values that pass the corresponding challenge (which we obtained in Step S2). Each attempt proceeds as follows:

1. Generate n sequences of random commitments to passing values (for copies in T_j, and dummy values otherwise). Specifically, suppose that $i \in T_j$ (or more generally that we have obtained (in Step S2) a proper decommitment to c^i), and denote by $((u^i_1, v^i_1), ..., (u^i_t, v^i_t))$ the value of the decommitted challenge (edge sequence $\overline{e}^i$). Then, for every $\ell = 1, ..., t$, select uniformly $r^i_{1,\ell}, ..., r^i_{n,\ell} \in \{0,1\}^n$ and $a^i_\ell \neq b^i_\ell \in \{1,2,3\}$, and compute $c^i_{u^i_\ell,\ell} = C(a^i_\ell, r^i_{u^i_\ell,\ell})$, $c^i_{v^i_\ell,\ell} = C(b^i_\ell, r^i_{v^i_\ell,\ell})$, and $c^i_{k,\ell} = C(0, r^i_{k,\ell})$ for $k \notin \{u^i_\ell, v^i_\ell\}$. Let $\overline{c}^i \stackrel{\text{def}}{=} (c^i_{1,1}, ..., c^i_{n,t})$. For $i \notin T_j$ (or for i's for which we failed in Step S2), we produce $\overline{c}^i \stackrel{\text{def}}{=} (c^i_{1,1}, ..., c^i_{n,t})$ as in (Item 1 of) Step S2.
2. Feeding the verifier with (the prover's commitments) $\overline{c}^1, ..., \overline{c}^n$, obtain the verifier's n (Step 3) responses, denoted $(s^1, \overline{e}^1), ..., (s^n, \overline{e}^n)$. Let $I' = \{i : \mathcal{C}_{\alpha_i}(s^i, \overline{e}^i) = c^i\}$ denote the set of copies that have properly decommitted (in the current attempt). If I' does not fit event E_j (i.e., $I' \not\subseteq T_j$ or $I' \subseteq T_{j-1}$) then we abort this attempt. That is, we proceed only if I' fits event E_j.
3. For every properly decommitted copy (i.e., $i \in I'$), we provide a proper decommitment (as in Step 4 of the protocol). This complete a full simulation of such a copy, whereas improperly committed copies are simulated by their transcript so far.

Specifically, ignoring the rare case of conflicting proper decommitments, a proper decommitment to copy $i \in I' \subseteq T_j$ must use the same challenge (edge sequence) as (found in Step S2 and) used in Item 1 (of the current attempt). Then, for every $i \in I'$ and $\ell = 1, ..., t$, we merely provide the 4-tuple $(r^i_{u^i_\ell,\ell}, a^i_\ell, r^i_{v^i_\ell,\ell}, b^i_\ell)$, where $((u^i_1, v^i_1), ..., (u^i_t, v^i_t))$ is the corresponding challenge. Indeed, this answer (like the prover's answer in Step 4 of the protocol) passes the verifier's check (since $a^i_\ell \neq b^i_\ell \in \{1,2,3\}$, $c^i_{u^i_\ell,\ell} = C(a^i_\ell, r^i_{u^i_\ell,\ell})$, and $c^i_{v^i_\ell,\ell} = C(b^i_\ell, r^i_{v^i_\ell,\ell})$).

In the rare case in which a conflicting proper decommitment is received, we proceed just as in case event E_j does not occur. If all $\text{poly}(n)/\tilde{q}_j$ trials fail then we output a special failure symbol.

For technical reasons, we modify the above simulation procedure by never allowing it to run more than 2^n steps. (This is easily done by introducing an appropriate step-count (which is implemented in linear or almost-linear time and so does not affect our running-time analysis).)

3.4 A detailed analysis of the simulator

Lemma 3.1 (Simulator's running-time): *The simulator runs in expected polynomial-time.*

Proof: The key observation is that each repeated attempt to produce something is repeated for a number of times that is inversely proportional to the probability that we try this attempt at all. This reasoning is applied with respect to each of the main steps (i.e., Steps S1, S2, S3 and S4). Specifically:

- For Step S1: Recall that event E_j occurs in the reference run (generated at the onset of Step S1), with probability q_j. Letting $Q \stackrel{\text{def}}{=} T_j \setminus T_{j-1}$, we have $q_j \leq |Q| \cdot \max_{i \in Q}\{p_i\} \leq n \cdot t_{j-1} < n \cdot 2^{-(j-1)}$. Also, with probability at least $1 - 2^{-n}$, Step S1 correctly determines j. Pending on the latter (overwhelmingly high probability) event, the expected number of steps conducted in Step S1 is

$$\sum_{j=0}^{n} q_j \cdot (\text{poly}(n) \cdot 2^j) \; < \; \sum_{j=0}^{n} (n \cdot 2^{-(j-1)}) \cdot (\text{poly}(n) \cdot 2^j) \; = \; \text{poly}(n) \tag{2}$$

  Relaying on the fact that the simulator never runs for more than 2^n steps, we cover also the highly unlikely case (in which Step S1 determines a wrong j).

  The same reasoning applies to Step S2. That is, again assuming that Step S1 correctly determines j, the expected number of steps made in Step S2 is as in Eq. (2).

- For Step S3: Assuming that $\tilde{q}_j = \Theta(q_j)$, the expected number of steps made in Step S3 is $\sum_{j=0}^{n} q_j \cdot (\text{poly}(n)/\tilde{q}_j) = \text{poly}(n)$. The above assumption holds with probability at least $1 - 2^{-n}$, and otherwise we relay on the fact that the simulator never runs for more than 2^n steps. The same reasoning applies to Step S4.

Thus, the overall expected running-time is polynomial (and this is proven without relying on any security properties of the commitment schemes). ∎

Lemma 3.2 (Simulator's output distribution): *Assuming that the verifier's commitment scheme* (i.e., $\mathcal{C}$) *is computationally-binding and that the prover's commitment scheme* (i.e., C) *is computationally-hiding, the output of the simulator is computationally indistinguishable from the real parallel interaction.*

Recall that the assumption that $\mathcal{C}$ is perfectly-hiding and C is perfectly-binding is used in establishing (cf. [17, Sec. 4]) the soundness of the GK-protocol (as a proof system).

Proof: For sake of clarity of the analysis, one may consider an imaginary simulator that goes on to determine all t_j's (rather than determining only part of them as in Item 2 of Step S1). We may assume that all approximations made by the simulator are sufficiently good; that is, in Step S1 the simulator correctly determines j as well as T_j and T_{j-1}, and in Step S3 it obtains $\tilde{q}_j = \Theta(q_j)$. (Indeed, the assumption holds with probability at least $1 - 2^{-n}$.)

Next, we consider three unlikely events in the simulation:

1. In Step S2, the simulator fails to obtain a proper decommitment of some $i \in T_j$. This may happen only with exponentially vanishing probability, because we keep trying for $\text{poly}(n) \cdot 2^j$ times and each time a proper decommitment (for i) occurs with probability $p_i > t_j \geq 2^{-(j+1)}$.

2. In Step S4, the simulator fails to generate a simulation in which event E_j holds. We will show that this failure may happen only with negligible probability. Note that in order for this failure to occur, it must be that event E_j occurs in Step S1 but does not occur in the $\text{poly}(n)/\tilde{q}_j = O(\text{poly}(n)/q_j)$ trials conducted in Step S4, although event E_j may occur in each such trial with probability q'_j that is negligibly close to q_j. (Recall that q_j refers to the probability that event E_j occurs for a dummy commitment, whereas q'_j refers to its probability for a "passing" commitment, and $|q_j - q'_j|$ is negligible because C is computationally-hiding (cf. [17, Clm. 3]).) Thus, the probability of this failure is upper-bounded by
$$\sum_{j=0}^{n} q_j \cdot (1 - q'_j)^{\text{poly}(n)/q_j} \tag{3}$$
Letting $\Delta_j \stackrel{\text{def}}{=} |q_j - q'_j|$, we consider two cases (cf. [17, Clm. 2]): in case $\Delta_j \leq q_j/2$, the corresponding term is exponentially vanishing (because $q'_j \geq q_j/2$ and $(1 - (q_j/2))^{2n/q_j} < \exp(-n)$), whereas in case $\Delta_j \geq q_j/2$ we simply bound the corresponding term by $q_j \leq 2\Delta_j$. Thus, in both cases, we obtain that each term in Eq. (3) is negligible (because it is upper-bounded by $\max(2\Delta_j, \exp(-n))$). Noting that Eq. (3) refers to the sum of $n + 1$ such terms, the claim follows.

3. In Step S4, the simulator obtains a proper decommitment to some copy such that the decommitted value is different from the one obtained for the same copy in Step S2. (In such a case, the simulator's may end-up outputting a failure symbol.) However, the hypothesis that $\mathcal{C}$ is computationally-binding implies that this bad event occurs only with negligible probability.

We conclude that, except with negligible probability, the simulator produces an output that looks syntactically fine. Finally, the hypothesis that C is computationally-hiding is used to demonstrate that the simulator's output is computationally indistinguishable from a random transcript of the real interaction. The details are analogous to the proof of [17, Clm. 4]: First we prove that the probabilities of each E_j event is about the same (i.e., differ by a negligible amount) in the simulation's output and in the real interaction. Next we focus on each "likely" E_j event, and prove that the conditional spaces for it are indistinguishable. We capitalize on the fact that a non-negligible difference in the unconditional space must translate to a non-negligible difference on some likely E_j, and that for likely E_j the simulation runs in strict polynomial-time.[19] ■

Combining Lemmas 3.1 and 3.2, we obtain

Theorem 3.3 *The* (constant-round) *GK-protocol is zero-knowledge under parallel composition.*

Recall that the GK-protocol is a proof system for $\mathcal{NP}$ (with exponentially vanishing soundness error) [17].

3.5 An Extension

We relax the parallel execution condition to concurrent execution of polynomially-many copies (of the GK-protocol) that satisfy the following two conditions:

C1: No copy enters Step 2 before all copies complete Step 1.

C2: No copy enters Step 4 before all copies complete Step 3.

In other words, the concurrent execution proceeds in three phases:

Phase 1: All copies perform Step 1 (in arbitrary order).

Phase 2: All copies perform Steps 2 and 3 (in arbitrary order except for the obvious local timing condition (i.e., each copy performs Step 3 after it has completed Step 2)).

Phase 3: All copies perform Step 4 (in arbitrary order).

Our treatment of parallel executions extends to the above (concurrent) case. The reason being that the simulator treats Steps 2–3 as one unit, and so the fact that these steps may be interleaving among copies is of no importance. Specifically, Step S0 of the extended simulator refers to Phase 1 (rather than to Step 1 of the protocol), its Steps S1–S3 refer to Phase 2 (rather than to Steps 2–3), and its Step S4 refers to Phases 2–3 (rather than to Steps 2–4).

4 Simulator for the case of Bounded-Simultaneity

Recall that the GK-protocol proceeds in four (abstract) steps:

1. The verifier commits to a challenge (i.e., Steps (P0) and (V1) in the protocol).
2. The prover commits to a sequence of values (i.e., Step (P1) in the protocol).
3. (Step (V2):) The verifier decommits (either properly or not).
4. (Step (P2):) Pending on the verifier's proper decommitment, the prover decommits to the corresponding values.

Here we consider (say n) concurrent executions in which up-to w copies of the GK-protocol run simultaneously at any given time, where w may be any fixed constant.

[19] An alternative approach may be to derive, in the contradiction argument, an expected polynomial-time algorithm that violates the hiding property of C, and to derive from it (via truncating long runs) a strict polynomial-time algorithm that violates the hypothesis that C is computationally-hiding.

4.1 Motivation

The case of $w = 1$ corresponds to sequential composition, and it is well-known that any zero-knowledge protocol maintains its security in this case. So let us turn (as a warm-up) to the case of $w = 2$. Trying to use the single-session simulator of [17] in this case, we encounter the following problem: when we try to deal with the simulation of one copy (by using the single-session simulator), the verifier may invoke another copy. A natural thing to do is to apply the single-session simulator also to the second copy. The good news is that the verifier cannot initiate yet another copy (before it terminates either the first or second copy, because this would violate the bounded-simultaneity condition (for $w = 2$)). Instead, eventually (actually, in a few steps), one of two things will happen (first):

1. The verifier may execute Step 3 in the *second* copy, in which case we make progress on treating the second copy (towards completing a simulation of it, which would put us back in the one-session case).
2. Alternatively, the verifier may execute Step 3 in the *first* copy, in which case we make progress on treating the first copy. For example, if we were trying to get the decommitment value for the first copy and we just got it, then we may abandon the treatment of the second copy and proceed by rewinding the first copy. (Note that in this case we lost all work done in the current simulation of the second copy.) Similarly, if we were trying to simulate the full run of the first copy then we just obtained one additional trial at a proper decommitment for Step 3 (which eventually will allow us to complete the simulation of the first copy).

Thus, in each of these cases, we make progress. Intuitively, the cost of dealing with two simultaneous copies is that we may have to invoke the single-session simulator (for the second copy) per each operation of the single-session simulator (for the first copy). As will be shown below, the above intuition remains valid also when we handle polynomially-many copies such that at most two are running simultaneously. Furthermore, it extends also to the case that at most w copies are running simultaneously, where w is any fixed constant. In that case, at most w copies of the single-session simulator will be active at any point during the simulation. Specifically, each operation in the emulation of the i-th copy will require to invoke the single-session simulator (for simulating the $i+1$st copy). Thus, the time-complexity of the simulation will be exponential in w, where the base of the exponent is the time-complexity of the case where $w = 1$.

4.2 The actual simulation

We start with a high level description of the simulation, next provide detailed specification and implementation of the procedures used by the simulator, and finally analyze them. Throughout the rest of the description we fix a (deterministic) adversarial verifier (and use black-box access to it).

4.2.1 A high level description

In correspondence to the three main steps of the single-copy simulator (cf. Section 2.3), we introduce three recursive procedures: `Scan`, `Approx` and `Generate`. Each of these procedures tries to handle a single copy (just as done by the corresponding step of the single-copy simulator), while making recursive calls when encountering a Step 2 message of some other copy.[20] The recursive call will take place before executing this Step 2, and the execution of this Step 2 will be the first thing that

[20] This is no typo; we do mean Step 2, not Step 1.

the invoked procedure will do. (Other steps of other copies may be handled by these procedures themselves.)

Before proceeding, let us recall the main steps of the *single-copy simulator*, and slightly modify them to provide a more convenient basis for our generalization. In particular, in this modification, Step 1 (of the protocol) is simulated separately (rather than as part of the Scan Step), and the Generation Step is used also in case the Scan encountered an improper decommitment. The resulting simulation steps are as follows:

A straightforward simulation of Step 1: The simulator emulates Step 1 of the protocol by obtaining the verifier's commitment (of Step (V1), after emulating Step (P0) in a straightforward manner).

The Scan Step: The simulator emulates Steps 2–3 of the protocol by using commitments to dummy values in Step 2, and obtains the verifier's decommitment for Step 3, which may be either proper or improper. We call this proper/improper bit the type of the decommitment. The simulator records the type of the decommitment as well as the decommitment information in case of proper decommitment.

The Approximation Step: The simulator approximates the probability that a single scan (as performs in the Scan Step) ends with a decommitment of the recorded type. (This is done by repeated trials, each as in the Scan Step, until some polynomial number of decommitments of the recorded type is encountered.)

The Generation Step: Using the decommitment information obtained in the Scan Step, the simulator repeatedly tries to generate a full transcript of the same type as encountered in the Scan Step. It does so by emulating Steps 2–4, using commitments to "pseudo-colorings" that do not "violate the coloring conditions imposed by the decommitted edges" (in case the Scan Step ended with a proper decommitment, and using commitments to dummy values otherwise). The number of trials is inversely proportional to the probability estimated in the approximation step, and if all fail then the simulator outputs a special failure symbol.

Analogously, the recursive procedures `Scan`, `Approx` and `Generate`, operate as follows, where the straightforward simulation of Step 1 (of each copy) is performed "en route" (by one of these procedures, while handling a different copy):

The `Scan` *procedure* is invoked to emulate Steps 2–3 of a certain copy that is scheduled to perform Step 2 at the current point (i.e., just following the current "simulation transcript"), provided that the current "simulation record" contains no trace of a prior handling of Step 2 of this copy. The procedure first emulates Step 2 of the said copy by using commitments to dummy values, and the hope is that it will reach Step 3 of the current copy and obtain the verifier's decommitment for this copy, which may be either proper or improper. When this happens, the procedure returns the relevant information (i.e., the decommitment value in case of proper decommitment and a special symbol in case of improper decommitment). However, other things may happen (due to the other copies):

- The procedure may encounter Step 1 of *some other copy*, in which case it emulates it in a straightforward manner (which results in augmenting the simulation transcript). Next, the procedure continues handling the current copy.
- The procedure may encounter Step 2 of *some other copy*, in which case it invokes either `Generate` or `Scan` to handle this other copy, depending on whether or not our current simulation record contains a trace of a prior handling of Step 2 of that copy. We stress that the invoked procedure may return an answer that refers to a copy that is not the one

for which the procedure was invoked (i.e., the copy to which the currently encountered Step 2 belongs). Following is a description of what the procedure does with the answer provided to it by the procedure it invokes, which indeed depends on which procedure was invoked.

  - When encountering a Step 2 of another copy (denoted j) that was not handled before, we invoke `Scan`, and handle the answer (of `Scan`) according to whether or not it refers to copy j. In the case that the answer relates to copy $k \neq j$ (which includes the case that k equals the current copy) we return the relevant information (as when we encounter Step 3 of the current copy), otherwise (i.e., $k = j$) we record the information and continue (as when handling other steps of other copies). In the latter case, we will next execute the following sub-case (which refers to the very same Step 2 (i.e., of copy j)). We stress that, regardless of the answer of `Scan`, we do not extend the simulation transcript in the current sub-case (and thus an execution of the following sub-case referring to copy j will necessarily follow the execution of the current sub-case).
  - When encountering a Step 2 of another copy that was already handled before, we repeatedly invoke `Generate`, until it either succeeds or an adequate number of trials was performed, and handle the answer (of `Generate`) as follows. If the answer provides an extension of the simulation transcript, we continue handling the current copy using that transcript. Otherwise (e.g., the answer is a decommitment information of yet some other copy) then we terminate returning this very answer.

  (Indeed, `Generate` corresponds to a single trial of the Generation Step, and the repeated attempts are done by the procedure that invokes it.)

- The procedure may encounter Step 3 of *some other copy*, which may happen when Step 2 of that copy was handled by an invocation that preceded the current one in the recursion path. Again, the action depends on whether or not our current simulation record contains information regarding a prior handling of Step 3 of that copy.
  - If no such prior handling exists (for this copy) then the procedure returns the corresponding decommitment information (although it is not the copy for which the current execution was invoked).
  - If such prior handling exists and the current emulation of Step 3 fits its type then the procedure augments the simulation transcript and continues handling the current copy. If the type does not fit then the procedure returns a special failure symbol.
- The procedure may encounter Step 4 of *some other copy*, which may happen when Step 2 of that copy was handled by an invocation that preceded the current one in the recursion path. Furthermore, in that case the recorded information allows to emulate this step in a straightforward manner, and `Scan` continues handling the current copy (after augmenting the simulation transcript).

Indeed, two key notions referred to above are the simulation transcript and the simulation record. The former is a prefix of a full transcript (of an execution) being generated by the simulator, and the latter provides auxiliary information regarding that (partial) transcript. In particular, the record contains information regarding copies that appear in the transcript, where this information was obtained in previous invocations of various procedures on prefixes of this transcript. For example, a successful `Scan` returns information regarding the decommitment of a certain copy.

The `Approx` *procedure* is invoked to approximate the probability that a certain invocation of `Scan` returns a certain value (i.e., the identity of the decommitting copy and the type (i.e., proper or improper) of that decommitment). This is done by repeated trials, where in each trial the procedure behaves similarly to `Scan`, until a sufficient number of trials return the value of interest.

The `Generate` *procedure* is invoked to emulate Steps 2–4 of a certain copy that is scheduled to perform Step 2 at the current point, provided that the current "simulation record" contains information regarding a prior handling of Step 2 of this copy (i.e., by `Scan`). The procedure behaves like `Scan` except that it emulates Step 2 using commitments to passing values (i.e., values that would pass w.r.t the corresponding proper decommitment, or arbitrary values in case the corresponding decommitment is improper). The hope is that the procedure will reach Step 4 of the current copy, and that the verifier's behavior at the corresponding Step 3 fits the recorded information. When this happens, the procedure emulates these steps in a straightforward manner (relying on the fact that a proper decommitment yields a challenge that can be met by the "passing values" used in emulating Step 2). Once the emulation of Step 4 is completed, the procedure returns the corresponding simulation transcript. However, as in case of `Scan`, other things may happen:

- The procedure may encounter steps of other copies. These are handled as in `Scan`.
- In addition, it may happen that Step 3 of the current copy decommits differently than in the simulation record (i.e., differently with respect to the proper/improper bit). In this case, the procedure returns a special failure symbol.

As mentioned above, the three procedures maintain (and pass along) the state of the currently handled copies as well as related auxiliary information. In particular, $\overline{h}$ will denote a partial transcript of the (concurrent) execution, and $\overline{a}$ will denote a corresponding list of currently active copies together with auxiliary information regarding each of them (e.g., decommitment information obtained in previous related runs). For sake of clarity, although the the identity of the copy that is responsible for the current procedure call (i.e., the copy that encountered Step 2) is implicit in $\overline{h}$, we pass this identity explicitly. The (simulator's) main program merely consists of a special invocation of `Generate` with empty history (i.e., $\overline{h} = \overline{a} = \lambda$).

4.2.2 The specification of the procedures

Let us first elaborate on the structure of the auxiliary information $\overline{a}$, which consists of records, each corresponding to some encountered copy of the protocol. The record corresponding to copy i consists of three fields:

1. The verifier decommitment field (of copy i) indicates whether the first encounter of Step 3 (i.e., the verifier's decommitment) of copy i was proper or improper (i.e., the type of decommitment), and in the former case the field includes also the value of the decommitment. That is, if non-empty, the field stores a pair (X, v), where $X \in \{\texttt{proper}, \texttt{improper}\}$ is a decommitment type and v is a decommitment value (which is meaningful only in case $X = \texttt{proper}$). This field (of the record of the i^{th} copy) is filled-up according to the answer returned by some invocation of $\texttt{Scan}(\overline{h}, \cdot, i)$.

2. The decommitment probability field (of copy i) holds an approximation of the probability that an invocation (with parameters as the one that filled-up the first field) actually turns out

returning same type. That is, suppose that the first field of record i (i.e., the record of the i^{th} copy) was filled-up according to the answer returned by $\texttt{Scan}(\overline{h}, \overline{a}, i)$, which resulted with a decommitment of type $X \in \{\texttt{proper}, \texttt{improper}\}$. Then the second field of record i should hold an approximation of the probability that $\texttt{Scan}(\overline{h}, \overline{a}, i)$ returns with an answer that encodes the same type of decommitment of copy i. (Jumping ahead, we hint that $\texttt{Scan}(\overline{h}, \overline{a}, i)$ may return with a decommitment to some other copy, and so the sum of the two probabilities corresponding to the two types is not necessarily 1.)

3. The prover decommitment field (of copy i) encodes the decommitment information corresponding to the prover's commitment in Step 2. This field (of the record of the i^{th} copy) is filled-up at the up-front of the execution of $\texttt{Generate}(\overline{h}, \overline{a}', i)$, which follows the invocation of $\texttt{Scan}(\overline{h}, \overline{a}, i)$, where $\overline{a}'$ is $\overline{a}$ augmented by the verifier decommitment information of copy i and the prover's commitment is performed so to passed the latter.

As hinted above, the fields are filled-up in the order they appear above (i.e., the verifier decommitment field is filled-up first). In reading the following specifications, it may be instructive to consider the special case of a single copy (in which case failure never occurs and $j = i$ always holds).

Specification of $\texttt{Scan}(\overline{h}, \overline{a}, i)$: This call produces a prefix of a "pseudorandom" execution transcript that extends the prefix $\overline{h}$, and *returns some related information*. The transcript is *pseudorandom* in the sense that it is computationally indistinguishable from a (prefix of a random) real continuation of $\overline{h}$ (by the adversary interacting with copies of the prover).[21] The extended transcript is truncated (i.e., the extended prefix ends) at the first point where one of the following holds:

1. Progress: This is a case where the (extended) execution reaches Step 3 of some copy j (possibly but not necessarily $j = i$) such that the first field of record j is empty. In this case, *the procedure should return the index j as well as the decommitment information* (provided in the current execution of Step 3 of copy j). That is, the answer is a pair (j, y), where j is a index of a copy and y is a decommitment information (which may be either proper or improper).

2. Failure: This is a case where the (extended) execution reaches Step 3 of some copy $j \neq i$ such that the first field of record j encodes a decommitment type different than the one occuring in the current extension. That is, the first field of record j encodes decommitment type $X \in \{\texttt{proper}, \texttt{improper}\}$, whereas in the current execution Step 3 of copy j has a decommitment type different from X (i.e., opposite to X). In this case, the procedure cannot continue (and should return a special `failure` symbol).

   (In contrast, in case the execution reaches Step 3 of some copy $j \neq i$ such that the first field of record j encodes a decommitment type that equals the one occuring in the current execution, the procedure may continue handling copy i.)

Furthermore, `Scan` should make progress with overwhelmingly high probability (equivalently, should fail only with negligible probability).

[21] The reader may wonder as to what will happen in case $\overline{h}$ itself is not consistent with any prefix of such a real interaction. The answer is that the extended execution will always be truncated before this fact becomes evident (i.e., we never perform Step 4 of a copy unless Step 2 of that copy was performed in a passing manner).

Specification of $\mathtt{Approx}(\overline{h}, \overline{a}, X, i)$**:** Always returns an approximation of the probability that $\mathtt{Scan}(\overline{h}, \overline{a}, i)$ answers with a pair (i, y) such that y has type $X \in \{\mathtt{proper}, \mathtt{improper}\}$. The approximation is required to be correct to within a factor of 2, with probability at least $1 - 2^{-n}$.

Specification of $\mathtt{Generate}(\overline{h}, \overline{a}, i)$**:** This call produces a prefix of a pseudorandom execution transcript that extends the prefix $\overline{h}$, and *returns either this extension or related information.* The notion of pseudorandom is the same as in case of `Scan`, and the extended transcript is truncated at the first point where one of the following holds:

1. Failure: Exactly as in the specification of `Scan`, except that here $j = i$ is possible too.
2. Progress: Here there are two sub-cases:
   (a) This is a case where the (extended) execution reaches Step 3 of some copy j such that the first field of record j is empty. This sub-case is handled exactly as the Progress Case of `Scan`. (Unlike in `Scan`, here $j = i$ cannot not possibly hold.)
   (b) This is a case where the (extended) execution reaches Step 4 of copy i. In this case, the procedure returns the currently extended transcript (including the execution of Step 4 of copy i), along with a corresponding update to the auxiliary information $\overline{a}$.

Furthermore, `Generate` should make progress with probability that is at most negligibly smaller than the probability approximated by the corresponding `Approx`-call. Thus, unlike in the presentation of the single-copy simulator, here `Generate` does not make progress almost always (not even in the case of a single copy), but rather makes progress with probability that is close to the one approximated by the corresponding `Approx`-call. That is, `Generate` is actually a generation-attempt, and the repetition of this attempt is made by the higher level invocation (rather than in the procedure itself).

4.2.3 The implementation of the procedures

We refer to the notion of a *passing commitment* as defined and used in Section 3. Recall that a passing commitment is a sequence of (prover's) commitments to values that pass for the corresponding challenge (encoded in the first field of the corresponding copy): See Footnote 14.

We start with the description of `Generate` (although $\mathtt{Generate}(\cdot, \cdot, i)$ is invoked after $\mathtt{Scan}(\cdot, \cdot, i)$). We note that $\mathtt{Generate}(\overline{h}, \overline{a}, i)$ is always invoked when the first field in the i^{th} record in $\overline{a}$ is not empty (but rather encodes some decommitment, of arbitrary proper/improper type), and the third field is empty (and will be filled-up at the very beginning of the execution).

Procedure $\mathtt{Generate}(\overline{h}, \overline{a}, i)$**:** Initializes $\overline{h}' = \overline{h}$ and $\overline{a}' = \overline{a}$, generates a passing commitment for (Step 2 of) copy i, and augments $\overline{h}'$ and $\overline{a}'$ accordingly. Specifically:

1. The procedure generates a random sequence of values, denoted $\overline{v}$, that pass the challenge described in the first field of the i^{th} record of $\overline{a}$. That is, $\overline{v}$ may be arbitrary if the said field encodes an improper decommitment; but in case of proper decommitment, $\overline{v}$ must pass with respect to the challenge value encoded in that field.
2. The procedure generates a random sequence of (prover's) commitments, denoted $\overline{c}$, to $\overline{v}$, augments $\overline{h}'$ by $\overline{c}$, and augments $\overline{a}'$ by placing the corresponding decommitment information in the third field of the i^{th} record.

Next, the procedure proceeds in iterations according to the following cases that refer to the next step taken in the concurrent execution.

Step 1 by some (new) copy: Just augment $\overline{h}'$ accordingly (and proceed to the next iteration).

Step 2 by some copy j (certainly $j \neq i$)**:** We consider two cases depending on whether or not $\overline{a}'$ contains the verifier's decommitment information for copy j (i.e., whether or not the first field of the j^{th} record is non-empty).

1. In case $\overline{a}'$ does contain such information, we generate a corresponding passing commitment (i.e., a prover commitment to values that pass w.r.t challenge encoded in the first field of the j^{th} record), augment $\overline{h}'$ and $\overline{a}'$ accordingly, and proceed to the next iteration. (Specifically, analogously to the up-front activity for (Step 2 of) the i^{th} copy, the third field in the j^{th} record of $\overline{a}'$ is augmented by the decommitment information corresponding to this prover commitment, and $\overline{h}'$ is augmented by the commitment itself.)
2. The case in which $\overline{a}'$ does not contain such information (i.e., the first field of the j^{th} record is empty (and certainly $j \neq i$)), is the most involved part of the procedure. In this case, we proceed as follows:
   (a) We invoke `Scan`$(\overline{h}', \overline{a}', j)$, and consider its answer, which is either `failure` or a *progress pair* (k, y). In case of progress, we determine the type $X \in \{$`proper`, `improper`$\}$ of the decommitment information y (with respect to the corresponding Step 1 commitment in $\overline{h}'$).
   (b) If the answer is either `failure` or is a progress pair (k, y) with $k \neq j$ then we `return` with the very same answer (i.e., either `failure` or (k, y)).
   (Here, in case of progress, $k \neq i$ must hold.)
   (c) We reach this step only if the answer obtained from `Scan` is a progress pair (k, y) with $k = j$. Letting X be the type of y, we let $\tilde{q} \leftarrow$ `Approx`$(\overline{h}', \overline{a}', X, j)$, and update the j^{th} record of $\overline{a}'$ placing (X, y) in the first field and $\tilde{q}$ in the second field. (Actually, it suffices to place (X, v) in the first field, where v is the decommitment value included in the decommitment information y.)
   (We comment that in case $X =$ `improper`, we could have skipped all subsequent sub-steps, and used instead the extended transcript generated by the above invocation of `Scan`, provided that `Scan` were modified to return this information as well. However, avoiding this natural modification makes the extension in Section 5 more smooth.)
   (d) Next, we repeatedly invoke `Generate`$(\overline{h}', \overline{a}', j)$ until getting a progress, but not more than $\text{poly}(n)/\tilde{q}$ times. (We will show that only with negligible probability can it happen that all calls return `failure`.) If all attempts have returned `failure` then we `return failure`, otherwise we act according to the sub-cases of the (first) progress answer (of `Generate`$(\overline{h}', \overline{a}', j)$), where the progress may be either a decommitment pair or an extended transcript:
      i. If the progress answer (of `Generate`$(\overline{h}', \overline{a}', j)$) provides a pair (k', y') (where certainly $k' \neq j$ as well as $k' \neq i$), then (analogously to sub-step 2b) we `return` with the very same answer (k', y').
      ii. If the progress answer (of `Generate`$(\overline{h}', \overline{a}', j)$) provides an updated history $\overline{h}''$ (together with updated auxiliary information $\overline{a}''$) then update $\overline{h}'$ and $\overline{a}'$ (i.e., $\overline{h}' \leftarrow \overline{h}''$ and $\overline{a}' \leftarrow \overline{a}''$), and proceed to the next iteration. (Note that in this case $\overline{h}''$ ends with execution of Step 4 of copy j.)

Note that in handling this case, we provide a full handling of copy j, invoking all three procedures. Indeed, this handling is analogous to the single-copy simulator.

Step 3 by copy i: Just as the first sub-case in the next case (i.e., Step 3 by some copy $j \neq i$ with a non-empty first field).

Step 3 by some copy $j \neq i$: We consider two cases depending on whether or not $\overline{a}'$ contains the verifier's decommitment information for copy j (i.e., the first field of the j^{th} copy is not empty).

1. In case $\overline{a}'$ does contain such information, we consider sub-cases according to the relation of the contents of the the first field of the j^{th} copy, denoted $(X, \cdot)$, and the current answer of the verifier.
   (a) If the decommitment type of the current Step 3 (of the j^{th} copy) fits X then we just augment $\overline{h}'$ accordingly (and proceed to the next iteration).
   (b) Otherwise (i.e., the decommitment type of the current Step 3 does not fit X), `return failure`.
2. In case $\overline{a}'$ does not contain such information (i.e., the first field of the j^{th} copy is empty), obtain the relevant decommitment information from the adversary (it may be either an improper or proper decommitment), and `return` (as progress) with this information only. That is, `return` with (j, y), where y encodes the decommitment information just obtained from the adversary.

Step 4 by some copy j (possibly $j = i$)**:** We will show that this case may happen only in case the corresponding (Step 2) prover commitment is passing and $\overline{a}'$ contains the corresponding decommitment (in the third field of the j^{th} record). Using the latter prover's decommitment information, we emulate Step 4 in the straightforward manner (and augment $\overline{h}'$ accordingly). In case $j = i$, `return` with the current $\overline{h}'$ and $\overline{a}'$ (otherwise proceed to the next iteration).

Note that Step 2 of copy i is handled up-front. In case of a single copy i, the above procedure degenerates to the basic handling of Steps 2–4 of copy i. In the fictitious invocation of `Generate` by the main program (i.e., with empty $\overline{h}$ and a fictitious i), only the handlings of Steps 2–4 for copies $j \neq i$ are activated (whereas, in handling Step 2, sub-steps 2b and 2(d)i are never activated). We now turn to procedure `Scan`, which is similar to `Generate`, except for its handling of the steps of copy i.

Procedure `Scan`$(\overline{h}, \overline{a}, i)$**:** Initializes $\overline{h}' = \overline{h}$ and $\overline{a}' = \overline{a}$, generates a dummy commitment for (Step 2 of) copy i, and augments $\overline{h}'$ accordingly. (Specifically, the procedure generates a random sequence of commitments, $\overline{c}$, to dummy values, and augments $\overline{h}'$ by $\overline{c}$.) Next, the procedure proceeds in iterations according to the following cases that refer to the next step taken in the concurrent execution.

Step 1 by some (new) copy: As in `Generate`.

Step 2 by some copy j (certainly $j \neq i$)**:** As in `Generate`.

(We comment that unlike in sub-step 2b of `Generate`, here $k = i$ is possible.)

Step 3 by copy i: Obtain the relevant decommitment information from the adversary (it may be either an improper or proper decommitment), and `return` (as progress) with this information. That is, `return` with (i, y), where y encodes the decommitment information just obtained from the adversary.

Step 3 by some copy $j \neq i$: As in `Generate`.

Step 4 by some copy $j \neq i$: As in `Generate`.

Note that we never reach Step 4 of copy i (and that Step 2 of copy i is handled up-front).

Procedure $\texttt{Approx}(\overline{h}, \overline{a}, X, i)$**:** This procedure merely invokes $\texttt{Scan}(\overline{h}, \overline{a}, i)$ until it obtains $m = \text{poly}(n)$ invocations that return a pair that is a decommitment of type X for copy i, and returns the fraction of m over the number of trials. Specifically, the procedure proceeds as follows:

Set $\texttt{cnt}_{\text{total}} = \texttt{cnt}_{\text{succ}} = 0$.
Until $\texttt{cnt}_{\text{succ}} = m$ do
  increment $\texttt{cnt}_{\text{total}}$ (unconditionally),
  $(j, y) \leftarrow \texttt{Scan}(\overline{h}, \overline{a}, i)$,
  increment $\texttt{cnt}_{\text{succ}}$ if and only if $j = i$ and y is of type X.
Output: $m/\texttt{cnt}_{\text{total}}$.

4.2.4 Analysis of the simulation

It is quite straightforward to show that the procedure `Approx` satisfies its specification. Ignoring the exponentially vanishing probability that any single approximation (by the procedure `Approx`) is off by more than a factor of 2, we may bound the total expected running-time by using the recursive structure of the simulation. (We start with bounding the running-time, because we will have to use this bound in analyzing the output of the simulator.)

Running-time analysis. Towards the running-time analysis, it is useful to pass among the procedures also the corresponding path in the tree of recursive calls. For example, instead of saying that $\texttt{Scan}(\overline{h}, \overline{a}, i)$ invokes $\texttt{Generate}(\overline{h}', \overline{a}', j)$, we may say that $\texttt{Scan}(\overline{h}, \overline{a}, i; \overline{p})$ invokes $\texttt{Generate}(\overline{h}', \overline{a}', j; (\overline{p}, i))$, where $\overline{p}$ denotes the path of recursive calls leading to the calling invocation (i.e., $\texttt{Scan}(\overline{h}, \overline{a}, i; \overline{p})$). Bounded-simultaneity implies that the depth of the recursive tree is a constant (i.e., equals the simultaneity bound w), because whenever a procedure is invoked with path $\overline{p}$ it must be the case that the copies with indices in $\overline{p}$ are still active (i.e., the corresponding transcript does not contain their last message). The fact that the depth of the recursive tree is a constant is the key to the analysis of the running-time of the simulation.

Considering oracle calls to the adversary's strategy as atomic steps, the expected running-time of $\texttt{Scan}(\overline{h}, \overline{a}, i; \overline{p})$ (resp., $\texttt{Generate}(\overline{h}, \overline{a}, i; \overline{p})$) is dominated by the time spent by the recursive calls invoked by $\texttt{Scan}(\overline{h}, \overline{a}, i; \overline{p})$ (resp., $\texttt{Generate}(\overline{h}, \overline{a}, i; \overline{p})$). Such calls are made only when handling Step 2 of a copy with no verifier decommitment information. Each of these handlings consists of first invoking $\texttt{Scan}(\overline{h}', \overline{a}', j; (\overline{p}, i))$, where $\overline{h}'$ is the current extension of the transcript $\overline{h}$, and, pending on its not returning `failure`, invoking `Approx` and `Generate` on $(\overline{h}', \cdot, j; (\overline{p}, i))$. (Specifically, the latter procedures are invoked only if $\texttt{Scan}(\overline{h}', \overline{a}', j; (\overline{p}, i)) = (j, \cdot)$.) In particular, $\texttt{Approx}(\overline{h}', \overline{a}', X, j; (\overline{p}, i))$ invokes $\texttt{Scan}(\overline{h}', \overline{a}', j; (\overline{p}, i))$ for an expected number of times that is inversely proportional to the probability that $\texttt{Scan}(\overline{h}', \overline{a}', j; (\overline{p}, i))$ answers with a type X decommitment to copy j, and $\texttt{Generate}(\overline{h}', \overline{a}', j; (\overline{p}, i))$ is invoked for the at most the same (absolute) number of times. That is, letting $\mathtt{Scan}'(\overline{h}',\overline{a}',j) \stackrel{\rm def}{=} (k,X)$ if $\mathtt{Scan}(\overline{h}',\overline{a}',j)$ answers with a type X decommitment to copy k, we conclude that the expected number of recursive calls made by $\mathtt{Scan}(\overline{h},\overline{a},i;\overline{p})$ (resp., $\mathtt{Generate}(\overline{h},\overline{a},i;\overline{p})$) when handling a Step 2 message of Copy j is

$$\sum_{X\in\{\mathtt{proper},\mathtt{improper}\}} \Pr[\mathtt{Scan}'(\overline{h}',\overline{a}',j)=(j,X)] \cdot \frac{\mathrm{poly}(n)}{\Pr[\mathtt{Scan}'(\overline{h}',\overline{a}',j)=(j,X)]} \;=\; \mathrm{poly}(n) \qquad (4)$$

The key point is that all these recursive calls (invoked by, say, $\mathtt{Scan}(\overline{h},\overline{a},i;\overline{p})$) have the longer path $(\overline{p},i)$. Furthermore, these calls refer to transcripts that are prefixes of one another (i.e., each recursive call refers either to the same transcript as the previous call or to an extension of it). Thus, each node in the (depth w) tree of recursive-calls has an expected polynomial number of children, and so the expected size of the tree is upper-bounded by $\mathrm{poly}(n)^w$. It follows that, the simulation terminates in expected polynomial-time. That is:

Claim 4.1 *For any polynomial-time adversary and any constant w that bounds the number of simultaneously active copies, the simulation terminates in expected polynomial-time.*

Output distribution analysis. We start the analysis (of the output distribution) by justifying the discarding of the (remote) possibility that during the (polynomial-time) simulation we ever get two conflicting proper decommitments to the same verifier commitment. (In fact, the above functional description suggests this assumption, although formally it is not needed in the functional description.) The justification is that the polynomial bound on the expected running-time implies that the computational-binding property of the verifier's commitment is violated during the simulation with negligible probability.

Next, we establish that the implementations of the various procedures satisfy the corresponding specification, by using backward induction on the depth of the recursive call. First, we establish that in sub-step 2d of the handling of a Step 2 message, it rarely happens that all invocations of `Generate` return `failure` (i.e., this bad event occurs with negligible probability). This is due to the specification of the procedures invoked at the current stage (assumed in the induction step or to the fact that no procedure is invoked in the base case of the induction). (Specifically, `Generate` is invoked for a number of times that is inversely proportional to the probability it succeeds.) This holds for a single handling of a Step 2 message, and we infer the same for all handlings that take place in the recursion tree by using a union bound and relying on the polynomial bound on the expected number of handlings (implied by Claim 4.1). The analysis of the other sub-steps in the handling of a Step 2 message is straightforward (from the code and specification). The analysis of the handling of Step 3 messages is similar, and the analysis of other handlings is straightforward. Thus, we obtain:

Claim 4.2 *For any polynomial-time adversary and any constant w that bounds the number of simultaneously active copies, the invocation of any procedure during the simulation behaves according to the corresponding specification.*

Once Claim 4.2 is established, we look at the initial (fictitious) invocation of `Generate`, which cannot possibly return with `failure`, and conclude that the simulator's output is computational indistinguishable from a real interaction of the cheating verifier with copies of the prover. Thus, we get

Theorem 4.3 *The* (constant-round) *GK-protocol is zero-knowledge under concurrent composition of bounded-simultaneity.*

5 Simulation under the Timing Model

Recall that the timing assumptions refer to two constants, Δ and ρ, such that Δ is an upper bound on the message handling-and-delivery time, and $\rho \geq 1$ is a bound on the relative rates of the local clocks. Specifically, each real-time period of Δ units elapses Δ' units of time on the local clock, where $\Delta/\rho \leq \Delta' \leq \rho\Delta$. For simplicity, we may assume without loss of generality that $\Delta/\rho \leq \Delta' \leq \Delta$ (i.e., that all clocks are at least as slow as the real time).[22]

5.1 The Time-Augmented GK-protocol

Recall that the GK-protocol proceeds in four abstract steps, but the actual implementation of the first step consists of the prover sending a preliminary message that is used as basis to the verifier's actual commitment. Thus, the GK-protocol is actually a 5-round protocol starting with a prover message. We augment this protocol with the following time-driven instructions, where all times are measured according to the prover's clock starting at the time of the invocation of the prover's program:

1. The prover time-outs Step 1 after $\Delta_1 \stackrel{\text{def}}{=} 2\Delta$ units of time (as measured on its clock).

   (By the timing assumption, this does not disrupt honest operation, because 2Δ real units of time suffice for the delivery of a message from the prover to the verifier and back.)

2. The prover delays its execution of Step 2 to time $\Delta_2 \stackrel{\text{def}}{=} \rho \cdot \Delta_1 + \Delta$. That is, it sends its message exactly when its clock shows that Δ_2 units of time have elapsed.

3. The prover time-outs Step 3 after $\Delta_3 \stackrel{\text{def}}{=} \Delta_2 + 2\Delta$ units of time.

   (Note that $\Delta_3 = (2\rho + 3) \cdot \Delta$.)

4. The prover delays its execution of Step 4 to time $\Delta_4 \stackrel{\text{def}}{=} \rho \cdot \Delta_3 + \Delta$.

We comment that, compared to Dwork *et. al.* [13], we are making a slightly more extensive use of the time-out and delay mechanisms: Specifically, they only used the last two items and did so while setting $\Delta_3 = 4\Delta$ and $\Delta_4 = \rho\Delta_3$. On the other hand, our use of the time-out and delay mechanisms is less extensive than the one suggested by Section 1.5: We only guarantee that for two copies that start at the same time, Step 2 (resp., Step 4) in one copy starts after Step 1 (resp., Step 3) is completed in the other copy, but we do not guarantee anything about the relative timing of Steps 2 and 3 (of different copies). Relying on special properties of the GK-protocol (as analyzed in Section 3.5), we can afford doing so, whereas the description in Section 1.5 is generic and refers to any c-round protocol. (However, in the typical case where $\rho \approx 1$, the difference between the various time-augmentations of the GK-protocol is quite small.)

Comment: A more general treatment can be derived by introducing an auxiliary parameter, denoted $\delta > 0$, which (in the description above) we have set to equal Δ. In the general treatment, Step 2 uses delay $\Delta_2 \stackrel{\text{def}}{=} \rho \cdot \Delta_1 + \delta$, whereas Step 4 uses $\Delta_4 \stackrel{\text{def}}{=} \rho \cdot \Delta_3 + \delta$, where $\Delta_1 \stackrel{\text{def}}{=} 2\Delta$ and $\Delta_3 \stackrel{\text{def}}{=} \Delta_2 + 2\Delta$ (as above). Doing so, in the decomposition, one may partition time to intervals of length δ (rather than length Δ). For $\rho = 1$, the number of overlapping blocks in Claim 5.1 changes by a factor of $(3\Delta + \delta)/4\delta > 1/4$, whereas the execution time of the protocol changes by a factor of $(4\Delta + 2\delta)/6\Delta > 2/3$. Observe that we do not gain much by setting $\delta \neq \Delta$. Specifically, by setting $\delta \ll \Delta$ we may reduce the the execution time by not more than a factor of $2/3$, whereas the effect

[22] We comment that although our formulation looks different than the one in [13], it is in fact equivalent to it.

on the simulation time is devastating (because the latter depends exponentially on the number of overlapping blocks, which in turn grows by a factor of approximately $3\Delta/4\delta$ for $\delta \ll \Delta$). On the other hand, setting $\delta \gg \Delta$ does not make the simulation significantly faster, whereas it delays the execution time considerably (i.e., by a factor of approximately $\delta/3\Delta$ for $\delta \gg \Delta$). Thus, we chose to set $\delta = \Delta$.

5.2 The Simulation

As mentioned in the introduction, the simulation relies on a decomposition of any schedule that satisfies the timing model into sub-schedules such that each sub-schedule resembles parallel composition, whereas the relations among the sub-schedules resembles bounded-simultaneity concurrent composition. In fact, we can prove something stronger:

Claim 5.1 *Consider an arbitrary scheduling of concurrent sessions of the time-augmented GK-protocol that satisfy the timing assumption. Place a session in* block i *if it is invoked within the real-time interval* $((i-1)\cdot\Delta, i\cdot\Delta]$. *Then, for every* i*:*

1. *Each session in block* i *terminates Step 1 by real-time* $i\cdot\Delta+\rho\Delta_1$, *starts Step 2 after real-time* $i\cdot\Delta+\rho\Delta_1$, *terminates Step 3 by real-time* $i\cdot\Delta+\rho\Delta_3$, *and starts Step 4 after real-time* $i\cdot\Delta+\rho\Delta_3$.
2. *The number of blocks that have a session that overlaps with some session in block* i *is at most* $16\rho^3$. *That is, the number of* $j \neq i$ *such that there exists a time* t, *a session* s *in block* i, *and a session* s' *in block* j *such that* s *and* s' *are both active at time* t *is at most* $16\rho^3$.

The first item corresponds to Conditions C1 and C2 in Section 3.5, and the second item corresponds to bounded-simultaneity.[23]

Proof: The latest and slowest possible session in block i is invoked by real-time $i\cdot\Delta$, and takes $\rho\Delta$ units of real-time to measure Δ local-time units. It follows that such a session terminates Step 1 (resp., Step 3) by real-time $i\cdot\Delta+\rho\cdot\Delta_1$ (resp., $i\cdot\Delta+\rho\cdot\Delta_3$). On the other hand, the earliest and fastest possible session in block i is invoked after real-time $(i-1)\cdot\Delta$, and takes Δ units of real-time to measure Δ local-time units. It follows that such a session starts Step 2 (resp., Step 4) after real-time $(i-1)\cdot\Delta+\Delta_2 = i\cdot\Delta+\rho\Delta_1$ (resp., $(i-1)\cdot\Delta+\Delta_4 = i\cdot\Delta+\rho\Delta_3$). The first item follows.

For the second item, note that the earliest possible session in block i is invoked after real-time $(i-1)\cdot\Delta$, whereas the latest and slowest possible session in block i terminates by real-time $i\cdot\Delta+\rho\Delta_4+\Delta = (i+1)\cdot\Delta+\rho\cdot(2\rho^2+3\rho+1)\cdot\Delta$. Thus, all sessions of each block are active during a time interval of length $(2\rho^3+3\rho^2+\rho+2)\cdot\Delta$, and therefore these sessions may overlap sessions of at most $2\cdot(2\rho^3+3\rho^2+\rho+2)\leq 16\rho^3$ other blocks. ■

5.2.1 Combining the simulation techniques – the perfect case

Given Claim 5.1, we extend the simulation strategy of Section 4 by showing how to handle blocks of "practically parallel" sessions rather than single sessions (which may be viewed as "singleton blocks"). To motivate the final construction, we consider first the special case in which each block is a perfect parallel composition of some sessions.

[23] The second item is actually stronger than bounded-simultaneity, because it upper-bounds the total number of blocks that overlap with a given block (rather than upper-bounding the number of blocks that are (simultaneously) active at any given time).

The key to the extension is to realize that all that changes is the types of verifier decommitment events (corresponding to Step 3 messages). Recall that in case of a single session, there were two possible events (i.e., proper and improper decommitment), and these were the two decommitment types we have considered. Here, for m parallel copies (of some block), we may have 2^m possible events corresponding to whether each of the m copies is proper or improper. However, the decommitment types we consider here are (not these 2^m events but rather) the $n+1$ events considered in Section 3: the events $E_0, E_1, ..., E_n$, where event E_j holds if all the properly decommitting sessions (in the current run) have proper-decommitment probability above the threshold $t_j \approx 2^{-j}$ but not all these sessions have proper-decommitment probability above the threshold $t_{j-1} \approx 2^{-(j-1)}$. Indeed, E_0 is the event that all sessions have improperly decommitted in the current run. (It is important that the number of decommitment types is bounded by a polynomial; this will be reflected when trying to extend the analysis captured in Eq. (4).)

Given the new notion of decommitment types, the three procedures of Section 4 (`Scan`, `Approx` and `Generate`) are extended by using the corresponding operations in Section 3. We stress that, in case of progress, the extended `Scan` (as well as the first progress case in the extended `Generate`) returns the decommitment information, which includes the indication of whether each session has properly decommitted, but not the decommitment type. The latter will be determined as in Section 3 (which is far more complex than the trivial case handled in Section 4, where decommitment type equals the decommitment indicator bit). The decommitment type (rather than the sequence of decommitment indicators) is what matters in much of the rest of the activities of the modified procedures.

We focus on the most interesting modifications to the main procedures (`Scan` and `Generate`), and ignore straightforward extensions (which apply also to other steps):

1. *The handling of Step 2 messages by a block j with a non-empty first information field* is analogous to the treatment in the original procedure, and we merely wish to clarify what this means here. The point is that the first field of block j encodes a decommitment type E_k as well as decommitment information for all sessions that properly decommit with probability at least $t_k \approx 2^{-k}$. The prover commitment produced here is designed to pass with respect to these decommitment values. (The same applies to the initial actions in `Generate`.)

2. *The handling of Step 2 messages by a block j with an empty first information field* (i.e., the only case that invokes recursive calls). The following sub-steps correspond to the sub-steps in the original procedures (`Scan` and `Generate`):

   (a) We invoke `Scan` with a block index j (rather than with a copy index), and consider its answer which is either `failure` or a progress pair (k, y), where k is a block index, and y is a list of decommitments corresponding to the various copies of block k. We refer to the above invocation of `Scan` as to the initial one, and note that many additional invocations (with the same parameters) will take place in handling the current step.

   If (the initial invocation of) `Scan` returned with a progress pair (k, y) such that $k = j$, then we turn to the complex task of determining the decommitment type E_ℓ (which holds with respect to y) as well as the corresponding sets T_ℓ and $T_{\ell-1}$. (If $k \neq j$ then the following activity will not be conducted here, but rather be conducted by the instance that invoked `Scan`$(\cdot, \cdot, k)$.) The decommitment type E_ℓ as well as the corresponding sets T_ℓ and $T_{\ell-1}$ are determined analogously to the main part of Step S1 (of Section 3), which needs to be implemented in the current context. In particular, the implementation of Step S1 calls for the approximation of the probabilities (denoted p_i's in Section 3) that each of the sessions properly decommits. This, in turn, amounts to multiple executions of Steps 2–3 of these sessions, which in our case should be handled by multiple invocation of $\texttt{Scan}(\cdot,\cdot,j)$. Details follow.

Let $I \subseteq [n]$ denote the set of sessions in which the verifier has properly decommitted in y. (Recall we are in the case where the initial invocation of $\texttt{Scan}(\overline{h}',\overline{a}',j)$ has returned the progress pair (j,y).) Our objective is to determine the corresponding event index ℓ as well as the sets T_ℓ and $T_{\ell-1}$. We consider the following cases (w.r.t I):

Case of empty I: Set $\ell = 0$ and $T_\ell = T_{\ell-1} = \emptyset$.

Case of non-empty I: Set $t_0 = 1$ and $T_0 = \emptyset$. We determine $\ell \geq 1$ (as well as T_ℓ), by iteratively considering $\ell = 1, ..., n$ (as in Section 3.2). That is, for $\ell = 1, ..., n$ do

i. We obtain t_ℓ by invoking a procedure analogous to $T(\ell, n)$ (of Section 3.2). Specifically, we approximate each of the p_s's by $\text{poly}(n) \cdot 2^\ell$ invocations of $\texttt{Scan}(\overline{h}',\overline{a}',j)$. Recall that each call of $\texttt{Scan}(\overline{h}',\overline{a}',j)$ specifies whether each session in Block j has properly decommitted, and approximations to the p_s's, denoted a_s's, are determined accordingly. We stress that p_s is the probability that $\texttt{Scan}(\overline{h}',\overline{a}',j)$ returns a progress pair (j,y') such that Session s properly decommits in y' (e.g., p_s is upper bounded by the probability that $\texttt{Scan}(\overline{h}',\overline{a}',j)$ returns a progress pair $(j,\cdot)$). Once all a_s's are determined, we determine t_ℓ just as in the second step of $T(\ell, n)$.

ii. Determine the set T_ℓ by determining, for each s, whether or not $p_s > t_\ell$. We use the above approximations to each p_s and rely on $|p_s - t_\ell| > (1/9n)2^{-\ell}$.

iii. Decide if event E_ℓ holds for y by using $T_{\ell-1}$ (of the previous iteration) and T_ℓ (just computed). Recall that event E_ℓ holds for y if $I \subseteq T_\ell$ but $I \not\subseteq T_{\ell-1}$.

iv. If event E_ℓ holds then exit the loop with the current value of ℓ as well as with the values of T_ℓ and $T_{\ell-1}$. Otherwise, proceed to the next iteration (i.e., the next value of ℓ).

In both cases (of I), we have determined the commitment type $X = E_\ell$ with respect to y (as obtained in the initial invocation of $\texttt{Scan}$) as well as the corresponding sets T_ℓ and $T_{\ell-1}$.

(This corresponds to Step S1 of the simulator of Section 3.)

(b) Exactly as in the original sub-step 2b. (That is, if the initial answer is either a `failure` or is a progress pair (k,y) with $k \neq j$ then `return` with the very same answer.)

(c) Recall that we reach this sub-step only if the answer of the initial invocation of `Scan` is a progress pair (j,y), and that we have already determined the event E_ℓ that holds (for y). By $\text{poly}(n) \cdot 2^\ell$ additional invocations of `Scan` (with the same parameters as above), we may obtain progress pairs of the form $(j,\cdot)$ several times. In each of these cases, the second component consists of a list of proper decommitment values. With overwhelmingly high probability, for each $s \in T_\ell$, we will obtain (from at least one of these lists) a proper decommitment for Session s (because $p_s > 2^{-\ell}$). Ignoring the question of what decommitment types hold in these lists,[24] we combine all these lists to a list v of all proper decommitment values (obtained in any of these lists). This list v together with T_ℓ and $T_{\ell-1}$ (as obtained in sub-step 2a) forms a new information string

[24] In particular, we do not care if the decommitment event happens to be of type E_ℓ or not. Furthermore, we may ignore y itself and not use it below (although we may also use y if we please).

$z = (v, T_\ell, T_{\ell-1})$, which will be used below (i.e., recorded in $\overline{a}'$ for future use). (This corresponds to Step S2 of the simulator of Section 3.)

Next, analogously to the original sub-step 2c, we obtain an approximation to the probability that $\texttt{Scan}(\overline{h}', \overline{a}', j) = (j, y)$ such that E_ℓ holds in y. Specifically, we let $\tilde{q} \leftarrow \texttt{Approx}(\overline{h}', \overline{a}', (E_\ell, T_\ell, T_{\ell-1}), j)$, where procedure Approx uses T_ℓ and $T_{\ell-1}$ in order to determine whether the event E_ℓ holds in each of invocations of $\texttt{Scan}(\overline{h}', \overline{a}', j)$. We update the j^{th} record of $\overline{a}'$ by placing (E_ℓ, z) in the first field and $\tilde{q}$ in the second field. (This corresponds to Step S3 of the simulator of Section 3.)

(d) Finally, analogously to the original sub-step 2d, we invoke $\texttt{Generate}(\overline{h}', \overline{a}', j)$ up-to $\text{poly}(n)/\tilde{q}$ times and deal with the outcomes as in the original sub-step 2d. (This corresponds to Step S4 of the simulator of Section 3.)

3. *The handling of Step 3 messages by a block j* (possibly $j = i$) is analogous to the treatment in the original procedure, and we merely wish to spell out what this means: We consider two cases depending on whether or not $\overline{a}'$ contains the verifier's decommitment information for block j (i.e., the first field of the j^{th} block is not empty).

   (a) In case $\overline{a}'$ does contain such information, we consider sub-cases according to the relation of the contents of the the first field of the j^{th} block, denoted (E_ℓ, z), and the current answer of the verifier. Specifically, we check whether the verifier's current answer is of type E_ℓ. We note that the type of the current verifier decommitment is determined using the sets T_ℓ and $T_{\ell-1}$ provided in z (i.e., $z = (v, T_\ell, T_{\ell-1})$, where v is a sequence of decommitment values not used here). The sub-cases (fit versus non-fit) are handled as in the original procedure.

   (b) In case $\overline{a}'$ does not contain such information (i.e., the first field of the j^{th} block is empty), we obtain the relevant decommitment information (i.e, a sequence of decommitments) from the adversary, and `return` (as progress) with this information only.

This completes the description of the modification to the main procedures for the current setting (of bounded-simultaneity of blocks of parallel sessions). We stress that here (unlike in Section 3) the events E_ℓ regarding the decommitment to block j are not the only things that may happen when we invoke Scan with block index j (which corresponds to Step S1 in Section 3). As in Section 4, the answer may be `failure` or progress with respect to a different block. Indeed, the latter may not occur in case there is only one block, in which case the above treatment reduces to the treatment in Section 3. It is also instructive to note that when each block consists of a single copy, the above modified procedures degenerate to the original one (i.e., in Section 4).

To analyze the current setting (of bounded-simultaneity of blocks of parallel sessions), we plug the analysis of Section 3 into the analysis of Section 4. The only point of concern is that we have introduced additional recursive calls (i.e., in the handling of Step 2, specifically in the handling sub-step 2a). However, as shown in Section 3, the expected number of these calls is bounded above by a polynomial (i.e., it is $\sum_{\ell=0}^{n} \Pr[E_\ell] \cdot 2^\ell \text{poly}(n)$, whereas $\Pr[E_\ell] = O(n \cdot 2^{-\ell})$). Thus, again, the tree of recursive calls has expected $\text{poly}(n)$ branching and depth at most w. Consequently, again, the expected running-time is bounded by $\text{poly}(n)^w$.

5.2.2 Combining the simulation techniques – the real case

In the real case the execution decomposes into blocks of almost parallel sessions (rather than perfectly parallel ones) such that (again) bounded-simultaneity holds with respect to the blocks. In view of the extension in Section 3.5, the non-perfect parallelism *within* each block does not raise any problems (as far as a single block is concerned). What becomes problematic is the relation between the (non-perfectly parallel) blocks, and in particular our references to the ordering of steps taken by the different blocks. That is, our treatment of the perfect-parallelism case treats the parallel steps of each block as an atom. Consequently we have related to an ordering of these steps such that if one "block step" comes before another then all sessions in the the first block take the said step before any session of the other block takes the other step. However, in general, we cannot treat the parallel steps of each block as an atom, and the following problem arises: what if one session of block i takes Step A, next one session of block $j \neq i$ takes Step B, and then a different session of block i takes Step A. This problem seems particularly annoying if handling the relevant steps requires passing control between recursive calls. In general, the problem is resolved by treating differently the first (resp., last) session and other sessions of each block that reach a certain step. Loosely speaking, the first (or last) such session will be handled similarly to the atomic case (i.e., as in Section 5.2.1), whereas in some cases other sessions (of the block) will be handled differently (in a much simpler manner). In particular, recursive calls are made only by the first session, and control is returned only by either the first or last such sessions. For sake of clarity, we present below the modification to the procedure $\texttt{Generate}(\overline{h}, \overline{a}, i)$. Note that this procedure is invoked when the immediate extension of $\overline{h}$ calls for execution of Step 2 by the *first* session in block i (i.e., $\overline{h}$ contains no Step 2 by any session that belongs to block i).

Initialization (upon invocation) step: Initializes $\overline{h}' = \overline{h}$ and $\overline{a}' = \overline{a}$, generates a passing commitment for (Step 2 of) the current (i.e., first) session of block i, and augments $\overline{h}'$ and $\overline{a}'$ accordingly. Specifically, the commitment is generated so that it passes the challenge corresponding to the current session (as recorded in the first field of record i), and only the corresponding part of the third field of the i^{th} record (in $\overline{a}'$) is updated.

In all the following cases, $\overline{h}'$ and $\overline{a}'$ denote the current history prefix and auxiliary information, respectively. (The following cases refer to the next message to be handled by the procedure, which handles such messages until it returns.)

Step 1 by some (new) session: Exactly as in the atomic case (i.e., augment $\overline{h}'$ and proceed to the next iteration).

Step 2 by the first session in block j (certainly $j \neq i$)**:** Analogous to the atomic case (see Section 5.2.1). Specifically, the handling depends on whether or not $\overline{a}'$ contains the verifier's decommitment information for copy j (i.e., whether or not the first field of the j^{th} record is non-empty).

1. In case $\overline{a}'$ does contain such information, we just generate a corresponding passing commitment (i.e., passing w.r.t the first field of the j^{th} record), augment $\overline{h}'$ and $\overline{a}'$ accordingly, and proceed to the next iteration.
2. In case $\overline{a}'$ does not contain such information (i.e., the first field of the j^{th} record is empty), we try to obtain such information. This is done analogously to the atomic case (see Section 5.2.1). We stress that this activity will yield the necessary information for all sessions in the j^{th} block, and not merely for the current (first) session in the block. Recall that the handling of this sub-case involves making recursive calls to the three procedures (with parameters $(\overline{h}', \overline{a}', j)$).

Step 2 by a non-first session in block j (here $j = i$ may hold)**:** We consider two cases depending on whether or not $\overline{a}'$ contains the verifier's decommitment information for copy j (i.e., whether or not the first field of the $j^{\rm th}$ record is non-empty).

1. In case $\overline{a}'$ does contain such information, we just generate a corresponding passing commitment, augment $\overline{h}'$ and $\overline{a}'$ accordingly, and proceed to the next iteration.

   (This is exactly as in the corresponding treatment of the first session of block j to reach Step 2.)

2. In case $\overline{a}'$ does not contain such information (i.e., the first field of the $j^{\rm th}$ record is empty), we generate a dummy commitment, augment $\overline{h}'$ accordingly, and proceed to the next iteration. (Recall that we count on the first session in the $j^{\rm th}$ block to find out the necessary information (for all sessions in the block).)

   (This is very different from the treatment of the first session of block j to reach Step 2.)

Step 3 by a non-last session of block j (possibly $j = i$)**:** Just augment $\overline{h}'$ accordingly (and proceed to the next iteration).

(This is very different from the treatment of the last session of block j to reach Step 3.)

Step 3 by the last session of block j (possibly $j = i$)**:** Analogous to the atomic case. We consider two cases depending on whether or not $\overline{a}'$ contains the verifier's decommitment information for block j (i.e., the first field of the $j^{\rm th}$ block is not empty).

1. In case $\overline{a}'$ does contain such information, we consider sub-cases according to the relation of the contents of the the first field of the $j^{\rm th}$ block, denoted (E_ℓ, z), and the Step 3 answer of the verifier (for all sessions in the $j^{\rm th}$ block). Specifically, we should consider the answers to previous sessions in the $j^{\rm th}$ block as recorded in $\overline{h}'$ and the answer to the last session in the block as just obtained. Recall that the type of the verifier decommitments (for the sessions in the $j^{\rm th}$ block) is determined using the sets T_ℓ and $T_{\ell-1}$ provided in the first field of the $j^{\rm th}$ block. The sub-cases (fit versus non-fit) are handled as in the original procedure. That is:

   (a) If the decommitment type of the Step 3 answers (of the $j^{\rm th}$ block) fits E_ℓ then we just augment $\overline{h}'$ accordingly (and proceed to the next iteration).

   (b) Otherwise (i.e., the decommitment type of the current Step 3 does not fit E_ℓ), `return failure`.

   (As in the atomic setting this case must hold if $j = i$.)

2. In case $\overline{a}'$ does not contain such information (i.e., the first field of the $j^{\rm th}$ block is empty), we obtain the relevant decommitment information as in the previous case, and `return` (as progress) with this information only. Specifically, the decommitment information for the previous sessions of the $j^{\rm th}$ block is recorded in $\overline{h}'$, whereas the the decommitment information for the last session has just been obtained (from the adversary).

Step 4 by a session of block j (possibly $j = i$)**:** Using the prover's decommitment information (as recorded in the third field of the $j^{\rm th}$ record), we emulate Step 4 in the straightforward manner (and augment $\overline{h}'$ accordingly). If this is the last session of block j and $j = i$, then `return` with the current $\overline{h}'$ and $\overline{a}'$ (otherwise proceed to the next iteration).

The modifications to procedure `Scan` are analogous. We stress that although the above description treats the schedule as if it is fixed, the treatment actually extends to a dynamic schedule where the membership of sessions in blocks is determined on-the-fly (i.e., upon their execution of Step 1).[25] The analysis of the perfect case can now be applied to the real case, and Theorem 1.1 follows. That is:

Theorem 5.2 *The Time-Augmented GK-protocol is* concurrent zero-knowledge under the timing model.

6 Other applications of our techniques

As stated in Section 1.3, our techniques are applicable also to several well-known protocols that have a structure similar to the GK-protocol. Notable examples include the (constant-round) zero-knowledge *arguments* of [15] and [4] as well as the *perfect* (constant-round) zero-knowledge proof of [5]. In fact, our techniques are applicable also to protocols with less apparent similarity to the GK-protocol. One such example is provided by the protocols that result from the transformation of Bellare, Micali and Ostrovsky [6].

In Section 6.1, we show that our techniques can be applied to the four-round argument system of Bellare, Jakobsson and Yung [4]. In Section 6.2, we describe a general class of protocols to which our techniques are applicable.

6.1 Application to the BJY-protocol

We start by briefly recalling the BJY-protocol (due to Bellare, Jakobsson and Yung [4], which in turn builds upon the work of Feige and Shamir [15]). Their protocol uses an adequate three-round witness indistinguishable proof system (e.g., parallel repetition of the basic zero-knowledge proof of [19]). Specifically, we consider a three-round witness indistinguishable proof system (e.g., for $G3C$) of the form:

Step WI1: The prover commits to a sequence of values (e.g., the colors of each vertex under several 3-colorings of the graph). This commitment scheme is perfectly-binding (and non-interactive; see Footnote 11).

Step WI2: The verifier send a random challenge (e.g., a random sequence of edges).

Step WI3: The prover decommits to the corresponding values.

(The implementation details are as in Construction 2.2.) For technical reasons, it is actually preferable to use protocols for which demonstrating a "proof of knowledge" property is easier (e.g., parallel execution of Blum's basic protocol; cf. [16, Sec. 4.7.6.3] and [16, Chap. 4, Exer. 28]). Given the above, the (four-round) BJY-protocol (for any language $L \in \mathcal{NP}$) proceeds as follows:

1. The verifier sends many hard "puzzles", which are unrelated to the common input x. These puzzles are random images of a one-way function f, and their solutions are corresponding preimages. In fact, the verifier selects these puzzles by uniformly selecting preimages of f, and applying f to obtain the corresponding images. Thus, the verifier knows solutions to all puzzles he has sent.

   In the rest of the protocol, the prover will prove (in a witness indistinguishable manner) that either it knows a solution to one of (a random subset of) these puzzles or $x \in L$. The latter proof is by reduction to some instance of an NP-complete language.

[25] Recall that by our assumption that the verifier never violates the time-out condition (cf. Sec. 2.2), the "last session in a block to reach a certain step" can be determined as well.

2. The prover performs Step WI1 *in parallel* to asking to see a random subset of the solutions to the above puzzles. Specifically, the puzzles are paired, and the prover asks to see a solution to one (randomly *selected*) puzzle in each pair. Furthermore, in executing Step WI1, the prover refers to a statement derived from the reduction of the assertion $x \in L$ or some of the *non-selected* puzzles has a solution.
3. The verifier performs Step WI2 *in parallel* to sending the required solutions (to the selected puzzles).
4. The prover verifies the correctness of the solutions provided by the verifier, and in case all solutions are correct it performs Step WI3.

As shown in [4], the BJY-protocol is a four-round zero-knowledge argument system for L. The simulator is similar to the one presented for the GK-protocol. Specifically, it starts by executing Steps 1–3, while using dummy commitments (in Step 2). Such a partial execution is called proper if the adversary has revealed all solutions to the selected puzzles (and is called improper otherwise). In case the partial execution is improper, the simulator halts while outputting it. Otherwise, the simulator moves to generating a full execution transcript by repeatedly rewinding to Step 2 and trying to emulate Steps 2–4 using the fact that (unless it selects the same set of puzzles again (which is highly unlikely)) it already knows a solution to one of the puzzles not selected (by it) in the current execution (but rather selected in the initial execution of Steps 1–3). Using such a solution, which yields an NP-witness to the reduced instance, the simulator can emulate the WI proof. As in the simulation of the GK-protocol (cf. [17]), the number of repetitions must be bounded by the reciprocal of the probability of a proper (initial) execution (as approximated by an auxiliary intermediate step).[26]

Given the similarity of the two simulators (i.e., the one here and the one for the GK-protocol), it is evident that our treatment of concurrent composition of the GK-protocol applies also to the BJY-protocol. Thus, recalling that the BJY-protocol is only based on one-way functions, we obtain:

Theorem 6.1 *Assuming the existence of one-way function, there exists a* (four-round) *argument system for $\mathcal{NP}$ that is* concurrent zero-knowledge under the timing model.

6.2 Application to a general class of protocols

In this section, we describe a general class of protocols to which our techniques are applicable. These protocols proceed in four main abstract steps:

1. The verifier "commits" to some secret information. Indeed, this "commitment" may be (as in the case of the GK-protocol) the result of applying a commitment protocol to the said information, but need not be so (cf., e.g., the BJY-protocol).
2. Some initial sub-protocol takes place such that its execution can be easily simulated by a computationally-bounded party that is only given the public information (i.e., the common input and the transcript of Step 1).

   In the GK-protocol, this step consists of the prover's commitment to a sequence of 3-colorings and can be simulated by producing commitments to dummy values. In other cases (e.g., [6]), this step may be vacuous.

[26] Unfortunately, this technical issue is avoided by Bellare *et. al.* [4], but it arises here (i.e., in [4]) similarly to the way it arises in [17], and it can be resolved in exactly the same manner. (The issue is that the prover commitments in the initial scan are distributed differently (but computational-indistinguishably) than its commitments in the generation process.)

3. The verifier proves knowledge of the secret information it has committed to in Step 1.

   In the GK-protocol, this step amounts to performing the corresponding decommitment step.
4. Pending on the prover being convinced, some residual sub-protocol takes place. The two sub-protocols (of Steps 2 and 4) are such that they can be easily simulated by a computationally-bounded party that is given the verifier's secret (as well as the the public information).

   In the GK-protocol, these two steps can be simulated by first sending commitments to corresponding "pseudo-colorings" and next performing the corresponding decommitments.

The single-session simulation of the above abstract protocol is similar to the simulator we have used in the previous sections. Specifically, the simulator starts by performing Step 1, and then performs Steps 2–3 (by using the corresponding guarantee regarding Step 2). In case the transcript is unacceptable by the prover, the simulator halts outputting the truncated transcript. Otherwise, the simulator invokes the knowledge-extractor that is guaranteed for Step 3, and obtains the verifier's secret information.[27] Once the simulator has this secret information, it can simulate Steps 2–4 (by the corresponding guarantee). We warn that indeed the actual implementation of the simulation procedure is more complex than the above description (e.g., as in [17], in some cases an approximation sub-step needs to be added). Still, the interested reader may verify that the techniques applied in Sections 3–5 extend to the above (abstract) simulation scheme. We conclude that *every protocol of the above type is concurrent zero-knowledge under the timing model.*

Acknowledgments

We are grateful to Uri Feige and Alon Rosen for helpful discussions at the initial stages of this research. We also wish to thank Rafi Ostrovsky for pointing out that that our techniques can be applied to the protocols in [5, 6], Boaz Barak and Daniele Micciancio for interesting discussions regarding the use of time-driven operations (see Footnote 6), and the anonymous referees for their valuable comments. Our research was partially supported by the Minerva Foundation, Germany.

[27] Actually, the simulator uses a knowledge-extractor that corresponds to Steps 2–3. Observe that if Step 3 is a proof-of-knowledge then so are Steps 2–3.

References

[1] B. Barak. How to Go Beyond the Black-Box Simulation Barrier. In *42nd FOCS*, pages 106–115, 2001.

[2] B. Barak and Y. Lindell. Strict Polynomial-time in Simulation and Extraction. In *34th ACM Symposium on the Theory of Computing*, pages 484–493, 2002.

[3] M. Bellare, R. Impagliazzo and M. Naor. Does Parallel Repetition Lower the Error in Computationally Sound Protocols? In *38th FOCS*, pages 374–383, 1997.

[4] M. Bellare, M. Jakobsson and M. Yung. Round-Optimal Zero-Knowledge Arguments based on any One-Way Function. In *EuroCrypt'97*, Springer-Verlag LNCS Vol. 1233, pages 280–305.

[5] M. Bellare, S. Micali, and R. Ostrovsky. Perfect Zero-Knowledge in Constant Rounds. In *22nd STOC*, pages 482–493, 1990.

[6] M. Bellare, S. Micali, and R. Ostrovsky. The (True) Complexity of Statistical Zero Knowledge. In *22nd STOC*, pages 494–502, 1990.

[7] G. Brassard, D. Chaum and C. Crépeau. Minimum Disclosure Proofs of Knowledge. *JCSS*, Vol. 37, No. 2, pages 156–189, 1988. Preliminary version by Brassard and Crépeau in *27th FOCS*, 1986.

[8] G. Brassard, C. Crépeau and M. Yung. Constant-Round Perfect Zero-Knowledge Computationally Convincing Protocols. *Theoretical Computer Science*, Vol. 84, pages 23–52, 1991.

[9] R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali. Resettable Zero-Knowledge. In *32nd STOC*, pages 235–244, 2000.

[10] R. Canetti, J. Kilian, E. Petrank and A. Rosen. Black-Box Concurrent Zero-Knowledge Requires (Almost) Logarithmically Many Rounds *SICOMP*, Vol. 32, No. 1, February 2002, pages 1–47. Preliminary version in *33rd STOC*, 2001.

[11] I. Damgård. Efficient Concurrent Zero-Knowledge in the Auxiliary String Model. In *Eurocrypt'00*, pages 418–430, 2000.

[12] D. Dolev, C. Dwork, and M. Naor. Non-Malleable Cryptography. *SICOMP*, Vol. 30, No. 2, April 2000, pages 391–437. Preliminary version in *23rd STOC*, 1991.

[13] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. In *30th STOC*, pages 409–418, 1998.

[14] C. Dwork, and A. Sahai. Concurrent Zero-Knowledge: Reducing the Need for Timing Constraints. In *Crypto98*, pages 442–457, Springer LNCS 1462.

[15] U. Feige and A. Shamir. Zero-Knowledge Proofs of Knowledge in Two Rounds. In *Crypto'89*, Springer-Verlag LNCS Vol. 435, pages 526–544, 1990.

[16] O. Goldreich. *Foundation of Cryptography – Basic Tools.* Cambridge University Press, 2001.

[17] O. Goldreich and A. Kahan. How to Construct Constant-Round Zero-Knowledge Proof Systems for NP. *J. of Crypto.*, Vol. 9, No. 2, pages 167–189, 1996. Preliminary versions date to 1988.

[18] O. Goldreich and H. Krawczyk. On the Composition of Zero-Knowledge Proof Systems. *SICOMP*, Vol. 25, No. 1, February 1996, pages 169–192. Preliminary version in *17th ICALP*, 1990.

[19] O. Goldreich, S. Micali and A. Wigderson. Proofs that Yield Nothing but their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *JACM*, Vol. 38, No. 1, pages 691–729, 1991. Preliminary version in *27th FOCS*, 1986.

[20] O. Goldreich and Y. Oren. Definitions and Properties of Zero-Knowledge Proof Systems. *J. of Crypto.*, Vol. 7, No. 1, pages 1–32, 1994.

[21] S. Goldwasser and S. Micali. Probabilistic Encryption. *JCSS*, Vol. 28, No. 2, pages 270–299, 1984. Preliminary version in *14th STOC*, 1982.

[22] S. Goldwasser, S. Micali and C. Rackoff. Knowledge Complexity of Interactive Proofs. In *17th STOC*, pages 291–304, 1985. This is a preliminary version of [23].

[23] S. Goldwasser, S. Micali and C. Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SICOMP*, Vol. 18, pages 186–208, 1989. Preliminary version in [22].

[24] J. Håstad, R. Impagliazzo, L.A. Levin and M. Luby. A Pseudorandom Generator from any One-way Function. *SICOMP*, Vol. 28, No. 4, pages 1364–1396, 1999. Preliminary versions by Impagliazzo *et. al.* in *21st STOC* (1989) and Håstad in *22nd STOC* (1990).

[25] J. Kilian and E. Petrank. Concurrent and resettable zero-knowledge in poly-logarithmic rounds. In *33rd STOC*, pages 560–569, 2001.

[26] J. Kilian, E. Petrank, and C. Rackoff. Lower Bounds for Zero-Knowledge on the Internet. In *39th FOCS*, pages 484–492, 1998.

[27] M. Naor. Bit Commitment using Pseudorandom Generators. *J. of Crypto.*, Vol. 4, pages 151–158, 1991.

[28] M. Prabhakaran, A. Rosen and A. Sahai. Concurrent Zero-Knowledge Proofs in Logarithmic Number of Rounds. In *43rd IEEE Symposium on Foundations of Computer Science*, pages 366–375, 2002.

[29] R. Richardson and J. Kilian. On the Concurrent Composition of Zero-Knowledge Proofs. In *EuroCrypt99*, Springer LNCS 1592, pages 415–413.

[30] A. Rosen. A Note on Constant-Round Zero-Knowledge Proofs for NP. In *1st TCC*, Springer LNCS 2951, pages 191–202, 2004.

[31] A.C. Yao. Theory and Application of Trapdoor Functions. In *23rd FOCS*, pages 80–91, 1982.