SPARKs: Succinct Parallelizable Arguments of Knowledge

Naomi Ephraim[1], Cody Freitag[1], Ilan Komargodski[2], and Rafael Pass[1]

[1] Cornell Tech, New York, NY 10044, USA
`{nephraim,cfreitag,rafael}@cs.cornell.edu`
[2] NTT Research, Palo Alto, CA 94303, USA
`ilan.komargodski@ntt-research.ac.il`

Abstract. We introduce the notion of a *Succinct Parallelizable Argument of Knowledge* (SPARK). This is an argument system with the following three properties for computing and proving a time T (non-deterministic) computation:

— The prover's (parallel) running time is $T + \operatorname{polylog} T$. (In other words, the prover's running time is essentially T for large computation times!)
— The prover uses at most $\operatorname{polylog} T$ processors.
— The communication complexity and verifier complexity are both $\operatorname{polylog} T$.

While the third property is standard in succinct arguments, the combination of all three is desirable as it gives a way to leverage moderate parallelism in favor of near-optimal running time. We emphasize that even a factor two overhead in the prover's parallel running time is not allowed.

Our main results are the following, all for non-deterministic polynomial-time RAM computation. We construct (1) an (interactive) SPARK based solely on the existence of collision-resistant hash functions, and (2) a non-interactive SPARK based on any collision-resistant hash function and any SNARK with quasi-linear overhead (as satisfied by recent SNARK constructions).

1 Introduction

Interactive proof systems, introduced by Goldwasser, Micali, and Rackoff [27], are one of the most fundamental concepts in theoretical computer science. Such systems consist of a prover who is able to convince a verifier of the validity of some statement if and only if it is true. The "if" direction is called *completeness* and the "only if" direction is called *soundness*. Proof systems where soundness is only guaranteed to hold for efficient (i.e., polynomial-time) provers are called *argument* systems.

We focus on *succinct* argument systems for NP: argument systems where the total communication is essentially independent of the size of the verification circuit of the language and even shorter than the statement. Since their introduction [31, 34, 12], succinct argument systems have drawn significant attention due to their appealing efficiency properties. Nowadays they are widely implemented and used in various systems, most notably in numerous blockchain platforms.

One aspect of such argument systems that has been the center of many recent works (e.g., [13, 18, 43, 28] to name a few) is *prover efficiency*. Consider the application of succinct arguments to delegating (possibly non-deterministic) computation, where a prover performs some expensive computation and then uses a succinct argument to convince an efficient verifier the validity of the output. If computing the proof takes much longer than the computation (even, say, a multiplicative factor of two), this would cause a significant delay making the system useless in various realistic settings. This motivates the following question:

Is it possible to compute the proof in parallel
to the computation while incurring no additional delay?

SPARKs. In this work, we answer the above question affirmatively. We introduce succinct *parallelizable* arguments of knowledge (SPARKs) where the prover's running time is "essentially" optimal. More precisely, an interactive argument (P, V) is a SPARK if instances solvable in (non-deterministic) sequential time T can be proven with the following efficiency requirements (ignoring dependence on the security parameter or statement size):

– The prover's parallel time is $T + \operatorname{polylog} T$.[3] (In other words, the prover's running time is essentially T for large computations!)
– The total prover complexity is $T \cdot \operatorname{polylog} T$ and only uses $\operatorname{polylog} T$ parallel threads.
– The communication complexity and verifier complexity are $\operatorname{polylog} T$.

Note that the third property is standard for succinct arguments. The first two properties stipulate that the running time of a prover with only a moderate amount of parallel processors is optimal—even a factor two overhead in terms of a prover running time is not allowed. Without the first property, there are existing succinct arguments with time $T \cdot \operatorname{polylog} T$ using only a single processor (e.g., [10, 7, 28]). Without the second property, there are existing constructions with parallel time $T + \operatorname{polylog} T$ using roughly T processors (e.g., [7]).

1.1 Our Results

For our main theorem, we show the existence of SPARKs for NP based on the existence of collision-resistant hash functions. The formal theorem and full details are deferred to the full version of the paper.

Theorem 1.1 (Informal). *Assuming collision resistant hash functions, there exists a four-round SPARK for non-deterministic polynomial-time RAM computation.*

[3] Only the additive $\operatorname{polylog} T$ term is allowed to additionally depend on the security parameter or statement size.

If we additionally assume succinct non-interactive arguments of knowledge (SNARKs) where the prover's sequential running time is quasi-linear in the verification time, then we obtain non-interactive SPARKs. The formal theorem and full details are deferred to the full version of the paper.

Theorem 1.2 (Informal). *Assuming collision resistant hash functions and a SNARK for* NP *with a quasi-linear prover, there exists a non-interactive SPARK for non-deterministic polynomial-time RAM computation.*

Our results are obtained by a generic construction that assumes collision resistant hash functions and *any* succinct argument of knowledge for a specific NP language, where the prover's sequential running time is quasi-linear (i.e. $T \cdot \text{polylog}\, T$ when using a single processor for T time computations), and results with a SPARK, where the prover's parallel time is essentially optimal. More precisely, we prove the following theorem.

Theorem 1.3 (Informal; see Theorem 5.6). *Assuming collision resistant hash functions, any succinct argument of knowledge for* NP *with a quasi-linear prover can be generically transformed into a SPARK for non-deterministic polynomial time RAM computation. Additionally, if the original succinct argument of knowledge is non-interactive, then so is the resulting SPARK.*

Applying the transformation to Kilian's protocol [31] instantiated with a quasi-linear size PCP [19, 10] yields a SPARK with poly-logarithmically many rounds. A simple modification to this transformation, when instantiated with Kilian's protocol, preserves the round complexity and yields Theorem 1.1. Theorem 1.2 follows by applying the above theorem to any SNARK where the prover has quasi-linear overhead (e.g., based on Micali's CS proofs [34] instantiated with a quasi-linear size PCP [19, 10]; see also [7, 28]).

Model of Computation. We define and build SPARKs for sequential RAM computations, whereas our construction of SPARKs is in the parallel RAM model. While the RAM model of computation is very expressive in theory, there is clearly not an exact one-to-one correspondence with real computers. For example, we do not take into account the performance of caches or other optimizations in modern processors that can easily result in additional overhead. As such, we view the results in this paper as showing a theoretical feasibility for practical implementations of SPARKs. We next briefly discuss and justify both the model of computation and the notion of time used in this work. For further details, see Section 3.1.

Recall that a RAM machine is a Turing machine with random access to its memory string. Between accesses, the machine applies some transition function to determine its next memory access. Each access is either a read or write, and we additionally assume that every time a process writes a value to a location in memory, it receives the previous value at that location. We define the running time of a RAM machine as the number of memory accesses it makes. For parallel RAM machines, we define the parallel running time as the number of "rounds" of memory accesses made by all processors, so if two processors access memory during the same logical round, we only count it as a single unit of parallel time. In other words, a SPARK proves a RAM computation that makes T sequential accesses in $T + \operatorname{polylog} T$ rounds of parallel accesses.

Similar models have been used in other contexts for delegating RAM computation (see e.g., [29, 28]), but they were much less sensitive to the model since they did not care about small multiplicative overheads. However, we believe that the above timing model we propose is reflective of real programs. For memory-intensive programs, our model captures the fact that memory accesses are practically the most time-consuming operations. For compute-intensive tasks, where the memory accesses are more sparse, it is only better that the overhead of a SPARK scales with the number of memory accesses and not the computation time itself.

1.2 Applications

SPARKs are a variant of succinct argument systems where the prover both computes and proves validity of the computation in parallel time which is essentially as efficient as possible. While our focus here is on establishing a theoretical feasibility result, we expect that our ideas may also be useful in practical constructions, which we leave for future work. Below we present applications of SPARKs.

Time-tight delegation of RAM computation. In the problem of verifiable delegation of computation [26, 39, 29], there is a client who wishes to outsource an expensive computation M on an input x to a powerful yet untrusted server. The server should not only produce the output y but also a proof that the computation was done correctly.

A non-interactive SPARK directly gives a delegation protocol for sequential RAM computation. This is because SPARKs satisfy a "delayed-output" property—the output y of the computation need not be known to the SPARK prover or verifier in advance, as it is computed in parallel to the proof. Therefore, using a non-interactive SPARK, a server can perform a RAM computation as well as a proof with essentially no overhead over the sequential running time. Specifically, for T-time computations, the server runs in time $T + \operatorname{polylog} T$ and uses at most $\operatorname{polylog} T$ processors. We call delegation schemes with this property *time-tight*. Previously, the best that was known was where the server uses a single processor and runs in time $T \cdot \operatorname{polylog} T$ [10, 7, 28], or where the server uses roughly T processors and runs in parallel time $T + \operatorname{polylog} T$ [7].

Our time-tight delegation protocol also works for *non-deterministic* computations. For example, consider the case where a client wants to outsource a RAM computation over a large database (stored at the server) but only knows a hash of the database. The server can perform the computation while proving both that the output is correct and the database is consistent with the client's hash. Furthermore, if both the server and the client have agreed upon the hash at the beginning of the protocol, the running time depends only on the time of the RAM computation (otherwise, the server will need to prove that the initial database hashes to the correct value, which requires computing a hash over the whole database and will be expensive if the database is large).

Towards VDFs from sequential functions. Verifiable delay functions (VDFs) are functions that require some "long" time T to compute (where T is a parameter given to the function), yet the answer to the computation can be efficiently verified given a proof that can be jointly generated with the output (with only small overhead) [14, 15, 38, 42]. The original work of Boneh et al. [14] suggests a theoretical construction of VDFs based on succinct non-interactive arguments (SNARGs) and any *iteratively sequential function* (ISF).[4] Other known constructions of VDFs [38, 42] rely on the repeated squaring assumption—a concrete ISF.

Let us recall what ISFs are. A *sequential* function (SF) is a function that takes a long time to compute, even if one has many parallel processors. An ISF is the *iteration* of some round function and the assumption is that iterating the round function is the fastest way to evaluate the ISF, even if one has many parallel processors. Clearly, any VDF implies an SF and so any construction of VDFs will necessarily rely on such (but this is not the case for an ISF[5]). It is thus a very natural question whether we can get a VDF based on only SFs and SNARGs. Note that the construction of Boneh et al. [14] inherently relies on the iterated structure of the underlying sequential function.[6]

Towards answering this question, we observe that any non-interactive SPARK for computing and proving an SF implies a VDF: simply compute the non-interactive SPARK for the SF. If the SF does not require any parallelism to compute, then by our main theorem, any SF, SNARK (with quasi-linear overhead), and collision-resistant hash function imply a VDF. However, in general, a moderate amount of parallelism may help to speed up the computation of an SF, and thus for this application, we would require a SPARK for (moderately) parallel computation. We defer this extension of our main theorem to the full version.

In fact, one way to view our main construction is by improving existing techniques for constructing verifiable computation for iterated functions from

[4] Actually, their original construction relied on *incremental verifiable computation* [41], which exists based on SNARKs [12], and any ISF. In an updated version they show that actually SNARGs, along with ISFs, are sufficient.

[5] However, a *continuous* VDF [24] does imply an ISF.

[6] In the construction based on SNARGs and ISFs, they need to be able to "break" the computation of the function in various mid-points of the computation and the internal "state" in those locations has to be small for efficiency of the construction. In the construction based on SNARKs and ISFs, they rely on a *tight* construction of incremental verifiable computation but the number of parallel processors required for the latter is as large as the cost of a single step [12, 8, 36], and so many steps are needed.

SNARGs to arbitrary functions using SNARKs (and collision resistant hash functions). An interesting open question is how to construct verifiable computation for arbitrary functions from only SNARGs, rather than SNARKs.

Memory-hard VDFs. A particularly appealing extension of the application above is to the existence of *memory-hard* VDFs. Recall that VDFs only guarantee that a long computation has been performed (and anyone can verify this publicly). It is very natural to require that not only a time-consuming computation was performed but also that the computation required many resources, for example, a large portion of the memory across time.

Clearly any VDF that is based on an ISF is not memory hard. The reason is that even if the basic round function is memory-hard, upon every iteration the memory consumption goes to 0! Since the VDF construction discussed above does not necessarily have to be instantiated with an ISF but rather any SF (and a SPARK for computing it), we can use a memory hard sequential function (e.g., [22, 23, 4, 3, 1, 2]) and get a VDF where the computation not only takes a long time, but also requires large memory throughout. As above, this requires a SPARK for a memory hard function, which may require using more than one parallel processor, and as such we give this extension in the full version.

1.3 Related Work

Succinct arguments with efficient provers. We elaborate on the existing succinct arguments that focus on prover efficiency. First, we recall that Kilian's succinct argument consists of a prover who commits to a PCP using a Merkle tree and locally opens a set of random locations specified by the verifier. As such, efficient PCP constructions immediately give rise to succinct arguments with an efficient prover. Specifically in [10, 7], they show how to construct PCPs in quasi-linear time, which yield succinct arguments with a prover running in $T \cdot \operatorname{polylog} T$ time for T-time computations. In [7], they show how to construct a quasi-linear size PCP where every bit can be computed in $\operatorname{polylog} T$ depth given the transcript of the computation. This results in a succinct argument where the prover runs in parallel time $T + \operatorname{polylog} T$ using roughly T processors (as opposed to $\operatorname{polylog} T$ processors as required by SPARKs). Furthermore, the above arguments can be made non-interactive by applying the Fiat-Shamir transformation [25, 34].

A different line of work has focused additionally on the prover's *space complexity*. Bitansky et al. [12] (following Valiant's [41] incrementally verifiable computation framework using recursive proof composition) construct complexity-preserving SNARKs, in which both the time and space of the underlying computation up to (multiplicative) polynomial factors in the security parameter. For the task of delegating deterministic T-time S-space computation, Holmgren and Rothblum [28] give a prover with $T \cdot \operatorname{polylog} T$ time and $S + o(S)$ space assuming sub-exponential LWE. We leave as future work the question of additionally reducing the prover's space complexity for SPARKs.

Tight VDFs. As we describe shortly in Section 2, our construction splits the computation into "chunks" and proves each of them in parallel. This idea is inspired by the recent transformations of Boneh et al. and Döttling et al. [14, 20] in the context of verifiable delay functions (VDFs) [14, 15]. Those works show how to use a VDF for an iterated sequential function where the honest evaluator has some overhead into a VDF where the honest evaluator uses multiple parallel processors and has essentially no parallel time overhead at all. However, iterated functions can be naturally split into chunks and so most of the technical difficulty in our work does not arise in that context. See Section 2 for more details.

IOPs. In an effort to bring down the quasi-linear overhead of PCPs, Ben-Sasson et al. [9] and Reingold et al. [39] introduced the concept of *interactive oracle proofs* (IOPs).[7] IOPs are a type of proof system that combines aspects of interactive proofs (IPs) and PCPs: in every round a prover sends a possibly long message but the verifier is allowed to read only a few bits. IOPs also generalize Interactive PCPs [30]. The most recent IOP is due to Ron-Zewi and Rothblum [40] (improving Ben-Sasson et al. [6]) and achieves nearly optimal overhead in proof length (i.e., a $1+\epsilon$ factor for an arbitrary $\epsilon > 0$) and constant rounds and query complexity, however the prover's running time is some unspecified polynomial.

2 Technical Overview

In this section, we present the main techniques underlying our transformation from succinct arguments of knowledge with quasilinear overhead to SPARKs.

2.1 Warmup: SPARKs for Iterated Functions

Our starting point stems from the recent works of Boneh et al. and Döttling et al. [14, 21]. For concreteness, we describe the setting of [14], which focuses on the simplified case of proving correctness of the output of an *iterated function* $g^{(T)}(x_0) = (g \circ \ldots \circ g)(x_0)$ for some $T \in \mathbb{N}$. Rather than proving that $g^{(T)}(x_0) = x_T$ directly, they split the computation into different sub-computations of geometrically decreasing size such that the proof for *every* sub-computation completes by time T.

To demonstrate this idea, suppose for simplicity that each iteration takes one unit of time to compute and that there is a succinct argument that can non-interactively prove any computation of k iterations of g in $2k$ additional time. Then, in order to prove that $g^{(T)}(x_0) = x_T$, they first perform $1/3$ of the computation to obtain $g^{(T/3)}(x_0) = x_{T/3}$ and then prove its correctness. Observe that $x_{T/3}$ can be computed in time $T/3$ and the proof can be generated in time $2T/3$

[7] To clarify notation, IOPs (introduced by [9]) are equivalent to the notion of Probabilistically Checkable Interactive Proofs (introduced concurrently and independently by [39]).

by assumption, so the proof that $g^{(T/3)}(x_0) = x_{T/3}$ completes by time T. In parallel to proving that $g^{(T/3)}(x_0) = x_{T/3}$, they additionally compute and prove $1/3$ *of the remaining computation* (namely, that $g^{((T-T/3)/3)}(x_{T/3}) = x_{5T/9}$) in a separate parallel thread, which also will finish by time T. They continue in this fashion recursively until the remaining computation can be verified directly.

In this construction, the prover only needs to start at most $O(\log T)$ parallel computation threads and finishes in essentially parallel time T. The final proof consists of $O(\log T)$ proofs of the intermediate sub-computations. The verifier checks each proof for the sub-computations independently and accepts if all checks pass and the proposed inputs and outputs are consistent with each other. More generally, if the given non-interactive argument had α multiplicative overhead, the resulting number of threads needed would be $O(\alpha \cdot \log T)$. So, when the overhead is quasi-linear (i.e. $\alpha \in \text{polylog}\, T$), the resulting argument is still succinct.

2.2 Extending SPARKs to Arbitrary Computations

The focus of this work is extending the above example to handle arbitrary non-deterministic polynomial-time computation (possibly with a long output) which introduces many complications. Specifically, suppose we are given an statement (M, x, T) with witness w, where M is a RAM machine and we want to prove that $M(x, w)$ outputs some value y within T steps. We emphasize that our goal is to capture general non-deterministic, polynomial-time computation where the output y is not known in advance, so we would like to simultaneously compute y given (M, x, T) and w, and prove its correctness. Since M is a RAM machine, it has access to some (potentially large) memory $D \in \{0,1\}^n$ where n consists of at most $2^{|x|}$ bits. To capture NP computation, we let the security parameter λ be roughly the input size $|x|$, and we let T be a arbitrary polynomial in λ. Let us try to employ the above strategy in this more general setting.

As M does not necessarily implement an iterated function, the first problem we encounter is that there is no natural way to split the computation into many sub-computations with small input and output. For intermediate statements, the naïve solution would be to prove that running the RAM machine M for k steps starting at some initial memory D_{start} results in final memory D_{final}. However, this is a problem because the size of the memory, n, may be large—perhaps even as large as the full running time T—so the intermediate statements we need to prove may be huge!

A natural attempt to mitigate this would be to instead provide a succinct commitment to the memory at the beginning and end of each sub-computation, and then have the prover additionally prove that it knows a memory string consistent with each commitment. Concretely, each sub-computation corresponding to k steps of computation would contain commitments $c_{\mathsf{start}}, c_{\mathsf{final}}$. The prover would show that there exist strings D_{start}, D_{final} such that (1) c_{start}, c_{final} are commitments to D_{start}, D_{final}, respectively, and (2) starting with memory D_{start} and running RAM machine M for k steps results in memory D_{final}. This seems like a step in the right direction, since the statement size for each sub-computation would only depend on the output size of the commitment and not the size of the memory. However, the prover's witness—and hence running time to prove each sub-computation—still scales linearly with the size of the memory in this approach. Therefore, the main challenge we are faced with is removing the dependence on the memory size in the witness of the sub-computations.

Using local updates. To overcome the above issues, we observe that in each sub-computation the prover only needs to prove that the transition from the initial commitment c_{start} to the final commitment c_{final} is consistent with k steps of computation done by M. At a high level, we do so by proving that there exists a sequence of k local updates to c_{start} which result in c_{final}. Then in order to verify a sub-computation corresponding to k steps, we can simply check the k local updates to the commitment of the memory, rather than checking the memory in its entirety. To formalize this idea, we rely on short commitments that allow for local updates which can be efficiently computed in parallel to the main computation. We call such commitments *concurrent locally updatable commitments.*

Given such commitments, will use a succinct argument of knowledge (P_{sARK}, V_{sARK}) for an NP language L_{upd} that corresponds to checking that a sequence of local updates are valid. Specifically, a statement $(M, x, k, c_{\mathsf{start}}, c_{\mathsf{final}}) \in L_{\mathsf{upd}}$ if and only if there exists a sequence of updates $u_1, \ldots, u_k$ such that, starting with short commitment c_{start}, running M on input x for k steps specifies the updates $u_1, \ldots, u_k$ that result in a commitment c_{final}. Then, as long as the updates are themselves succinct, the size of the witness scales only with the number of steps of the computation and not with the size of the memory.

In order to make the above approach work, we need locally updatable commitments that satisfy the following two properties:

1. Updates can be computed efficiently in parallel to the main computation.
2. Local updates can be verified as modifying at most a single location in the committed memory.

We next explain how we obtain the required commitments satisfying the above properties. We believe that this primitive and the techniques used to obtain it are of independent interest.

Concurrent locally updatable commitments. Roughly speaking, a concurrent locally updatable commitment is a standard computationally binding string commitment scheme with a local update property which supports updating a single bit in the underlying message without re-committing to the whole message. For efficiency we additionally require that one can perform several local updates concurrently. For soundness, we require that no efficient adversary can find two different openings for the same location even if it is allowed to perform polynomially-many update operations. A formal definition appears in Section 4.

Our construction relies on Merkle trees [33] and hence can be instantiated with any collision resistant hash function. Recall that a Merkle tree uses a compressing hash function, which we assume for simplicity is given by $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$, and is obtained via a binary tree structure where nodes are associated with values. The leaves are associated with arbitrary values and each internal node is associated with a value that is the hash of the concatenation of its children's values.

It is well known that Merkle trees, when instantiated with a collision resistant hash function h, act as short commitments with local opening. The latter property enables proving claims about specific blocks in the input without opening the whole input, by revealing the *authentication path* from some input bit to the root (i.e. the hashes corresponding to sibling nodes along the path from the leaf to the root). Not only do Merkle trees have the local opening property, but the same technique allows for *local updates*. Namely, one can update the value of a specific bit in the input and compute the new root value without recomputing the whole tree (by updating the hashes along the authentication path of the bit). All of these local procedures cost time which is proportional to the depth of the tree, $\log n$, as opposed to the full memory n. We denote this update time as β (which may additionally depend polynomially on λ, for example, to compute the hash function at each level in the tree).

Let us see what happens when we use Merkle trees as our commitment. Recall that the Merkle tree contains the hash of the memory at every step of the computation, and we update its value after each such step. The latter operation, as mentioned above, takes β time. So even with local updates, using Merkle trees naïvely incurs a β delay for every update operation which implies a β *multiplicative* delay for the whole computation (which we want to avoid)! To handle this, we use a *pipelining* technique to perform the local updates in parallel.

Pipelining local updates. Consider two updates u_1 and u_2 that we want to apply to the current Merkle tree sequentially. We observe that since Merkle trees updates work "level by level," we can first update the first level of the tree (corresponding to the leaves) according to u_1. Then, update the second layer according to u_1 and *in parallel* update the first layer using u_2. Continuing in this fashion, we can update the third layer according to u_1 and in parallel update the second layer using u_2, and so on. The idea can be generalized to pipeline $u_1, \ldots, u_k$, so that the final update u_k completes after $(k-1)+\beta$ steps, and the memory is consistent with the Merkle tree given by performing update operations $u_1, \ldots, u_k$ sequentially. The implementation of this idea requires β additional parallel threads since the computation for at most β updates will overlap at a given time. A key point that allows us to pipeline these concurrent updates is that the operations at each level in the tree are data-independent in a standard Merkle tree. Namely, each processor can perform all of the reads/writes to a given level in the tree at a single time step, and the next processor can continue in the next time step without incurring any delay.

Ensuring optimal prover run-time. Using the above ingredients, we discuss how to put everything together to ensure optimal prover run-time. Concretely, suppose we have a concurrent locally updatable commitment where each update takes time β, and a succinct non-interactive argument of knowledge with $\alpha \in \operatorname{polylog} T$ multiplicative overhead.

As discussed above, to prove that $M(x, w)$ output a value y in T steps, we split the computation into m sub-computations which all complete by time T. The ith sub-computation will consist of a "compute" phase, where we compute k_i steps of the total T computation steps, and a "proof" phase, where we use the succinct argument to prove correctness of those k_i steps. For the "compute" phase, recall that performing k_i steps of computation while also updating the commitment takes $k_i \cdot \beta$ total work. However, as described above, we can pipeline these updates so that the parallel time to compute these updates is only $(k_i - 1) + \beta$.

For the "proof" phase, recall that we that we use a succinct argument for the update language L_{upd} such that a statement $(M, x, k, c_{\mathsf{start}}, c_{\mathsf{final}}) \in L_{\mathsf{upd}}$ if there exists a sequence of k updates such that (1) the updates are consistent with the computation of M and (2) applying these updates to c_{start} results in c_{final}. To compute the proofs in the desired amount of time, we need to set the values of k_i appropriately. As the total work to compute k_i steps with updates is $k_i \cdot \beta$, this implies that each proof takes at most $k_i \cdot \alpha \cdot \beta$ time. Therefore, the largest "chunk" of computation we can compute and prove by time T time is $T/(\alpha\beta + 1)$. For convenience, let $\gamma \triangleq \alpha\beta + 1$. Then, in the first sub-computation, we can compute and prove $k_1 = T/\gamma$ steps of computation. In each subsequent computation, we compute and prove a γ fraction of the remaining computation. Putting everything together, we get that $k_i = (T/\gamma) \cdot (1 - 1/\gamma)^{i-1}$ for $i \in [m-1]$ and then $k_m < \gamma$ is the number of remaining steps such that $\sum_{i=1}^{m} k_i = T$.

In Figure 1 we show the structure of the compute and proof phases for all m sub-computations. We emphasize that the entire protocol completes within $T + \beta$ parallel time. As $\beta \in \operatorname{polylog} T$, this implies that only have a small additive rather than multiplicative overhead. This is tight in the sense that computing the commitment for T steps of computation with updates takes $T + \beta$ time, so all of the proofs about the updates to the commitments are computed completely in parallel. Next, we note that we have a β gap between the time that the "compute" phase ends and the "proof" phase begins for a particular sub-computation. This is because we have to wait β additional time to finish computing the updates before we can start the proofs. However, we can immediately start computing the next sub-computation without waiting for the updates to complete. Lastly, the number of processors used in the protocol is β at all times in the constantly running "compute" phase which is additionally computing updates to the commitment in parallel. Then we have at most $m - 1$ additional processors for the proofs of the first $m - 1$ sub-computations. The last sub-computation, we don't have the prover compute the proof, and instead the prover will send the updates in the clear for the verifier to check directly.

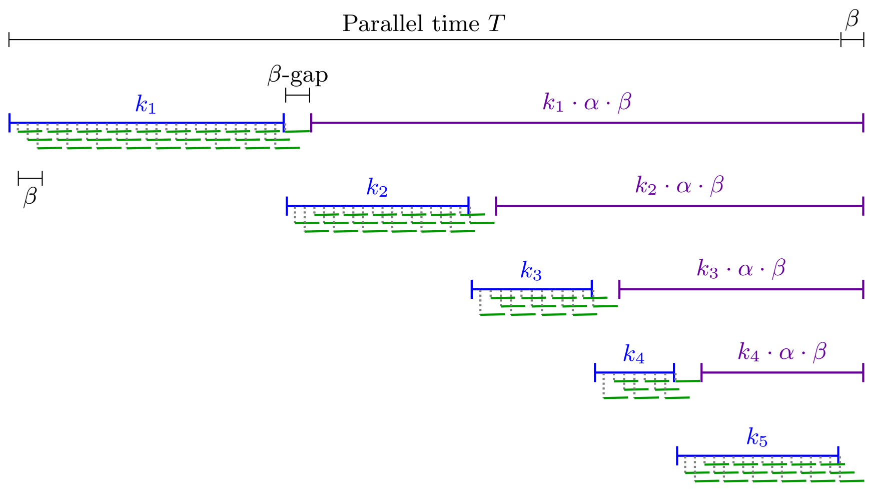
Parallel time T
β
β-gap
k_1
$k_1 \cdot \alpha \cdot \beta$
β
k_2
$k_2 \cdot \alpha \cdot \beta$
k_3
$k_3 \cdot \alpha \cdot \beta$
k_4
$k_4 \cdot \alpha \cdot \beta$
k_5

Fig. 1. The "compute" and "proof" phases for each of m sub-computations. For $i \in [m-1]$, the ith sub-computation consists of k_i steps, while pipelining updates which each take β time. After finishing all updates, the prover computes the proof which takes $k_i \cdot \alpha \cdot \beta$ time. In the final sub-computation, we send the updates to the verifier in the clear instead of giving a proof.

Computing the initial commitment. Before giving the full protocol, we address a final issue, which is for the prover to compute the commitment to the initial memory string. Specifically, the prover needs to commit to a string $D \in \{0,1\}^n$, which the RAM machine M assumes contains its inputs (x, w). Directly committing to the string $x||w$ would require roughly $|x| + |w|$ additional time, which could be as large as T. To circumvent the need to compute the initial commitment, we simply do not commit to the initial memory! Instead, we start with a commitment to an *uninitialized* memory that can be computed efficiently and allows each position to be initialized *exactly once* whenever it is first accessed. In Section 4, we discuss the full details of how we deal with this issue for our commitments.

2.3 Our SPARK Construction

We now summarize our full SPARK construction. Suppose that we have (1) a concurrent locally updatable commitment that starts as uninitialized where each update takes time β and (2) a succinct non-interactive argument of knowledge $(P_{\mathsf{sARK}}, V_{\mathsf{sARK}})$ for the update language L_{upd} with $\alpha \in \operatorname{polylog} T$ multiplicative overhead. Let $\gamma \triangleq \alpha\beta + 1$, as described above, which is the fraction of remaining computation done at each step. The protocol (P, V) for a statement (M, x, T) is as follows:

1. V samples public parameters pp for the commitment and sends them to P.

2. Using pp, P computes the commitment c_{start} for the uninitialized memory $D_{\mathsf{start}} = \perp^n$.
3. P computes T/γ steps of $M(x, w)$ while in parallel updating D_{start} and the corresponding local updates to $c_1 = c_{\mathsf{start}}$.
4. After completing the T/γ steps of the computation (but not necessarily completing all corresponding updates), P starts recursively computing and proving the remaining $T - T/\gamma$ steps in parallel.
5. Let $u_1, \ldots, u_{T/\gamma}$ be the current updates, which result in commitment c_1'. After computing the current updates, P uses $P_{\mathsf{sARK}}(u_1, \ldots, u_{T/\gamma})$ for language L_{upd} to prove that starting with commitment c_1, running M on input x for T/γ steps results in commitment c_1'.
6. P continues until there are at most γ steps of the computation. At this point, P computes the remaining steps and sends the corresponding updates to V in the clear to be verified directly.
7. After finishing the computation and all corresponding updates, P uses the final commitment to open the output y and give a proof of its correctness. V accepts if the proof certifying y verifies and V_{sARK} accepts all sub-protocols, which are consistent with each other.

Handling interactive protocols. The same transformation described above applies to interactive r-round succinct argument of knowledge. However, since the protocol is interactive, the prover starts an interactive protocol in order to prove that sub-computations were performed correctly. It is not necessarily the case that the messages in the various interactive arguments will be "synced" up, and so our transformation suffers from (at most) a $\operatorname{polylog} T$ factor increase in the round complexity. For specific underlying succinct argument, however, it may be the case that it is easy to synchronize the rounds in reduce the round complexity.

Security proof and argument of knowledge definition. We note that proving security in the above construction is somewhat non-trivial. The key issue is that we need to simultaneously extract witnesses from super logarithmically many *concurrent* or *parallel* arguments of knowledge, without causing a blow-up in the complexity of the resulting extractor. Towards resolving this issue, we introduce a new argument of knowledge definition, which 1) enables dealing with this issue in our proof of security, yet 2) is satisfied by known succinct arguments of knowledge for NP. We view this definition as an additional independent contribution. For more details, see Section 5.2.

3 Preliminaries

We defer some standard notation to the full version of the paper and instead focus on the necessary ingredients for our construction. We also defer the formal definition of succinct arguments of knowledge, as it is a natural analogue to the SPARK definition given in Section 5.2.

3.1 Random Access Memory

RAM computation consists of a machine M which keeps some local state state and has read/write access to memory $D \in (\{0,1\}^\lambda)^n$ (equivalent to the tape of a Turing machine). Here, λ is the security parameter and length of a word,[8] and $n \leq 2^\lambda$ is the number of words in memory used by M. When we write $M(x)$ to denote running M on input x, this means that M expects its initial memory D to be equal to $x||0^{n\lambda-|x|}$. The computation is defined using a function step, which has the following syntax:

$$(\mathsf{state}', \mathsf{op}, \ell, v^{\mathsf{wt}}) = \mathsf{step}(M, \mathsf{state}, v^{\mathsf{rd}}).$$

Specifically, step takes as input the description of the machine M, the current state state, and a word v^{rd} that was read in the last step from memory. Then, it outputs the next state state', the operation $\mathsf{op} \in \{\mathsf{rd}, \mathsf{wt}\}$ to do next, the next location $\ell \in [n]$ to access, and the word v^{wt} to write next if $\mathsf{op} = \mathsf{wt}$ (or $\perp$ if $\mathsf{op} = \mathsf{rd}$).

Using step, we can define each step of RAM computation to run step, and then either do a read or a write. We assume that each write operation returns the value in the memory location before the write. Formally, starting with an initially empty state state_0 and letting $b_0^{\mathsf{rd}} = \perp$, the ith step of computation for $i \geq 1$ is defined as:

1. Compute $(\mathsf{state}_i, \mathsf{op}_i, \ell_i, v_i^{\mathsf{wt}}) = \mathsf{step}(M, \mathsf{state}_{i-1}, v_{i-1}^{\mathsf{rd}})$.
2. If $\mathsf{op}_i = \mathsf{rd}$, let v_i^{rd} be the word in location ℓ_i of D.
3. If $\mathsf{op}_i = \mathsf{wt}$, let v_i^{rd} be the word at location ℓ_i in D and write v_i^{wt} to that location.

The computation halts when step outputs a special halting value with the output y of $M(x)$ written at the start of the memory, where we assume that M specifies its output length. Without loss of generality, we assume that the state size can hold $O(\log n)$ bits.

We also consider the parallel-RAM (PRAM) setting, where each step of the machine can potentially branch to multiple processors that have access to the same memory D. We formalize this by allowing step to output multiple values for $(\mathsf{state}', \mathsf{op}, \ell, v^{\mathsf{wt}})$, each associated with a process identifier specifying the process to continue the computation from that state. The computation halts when there are no running processors. We are in the exclusive-read exclusive-write (EREW) model, i.e., the most restrictive PRAM model, where if some process accesses a location (either a read or a write) in memory while another process accesses the same location (either a read or a write), there are no guarantees for the resulting effect. We also assume that n words in memory can be allocated and intialized to zeros for free.

[8] We note that the length of a word only needs to be greater than $\log n$, but can be as large as any fixed polynomial in λ. We set it to λ for simplicity.

(P)RAM Complexity. Each step of RAM computation is allowed to make a single access to memory. We think of step, which computes the transition function from state to state', as being implemented by an efficient CPU algorithm with access to a constant number of words. As a result, we define the running time of a RAM machine M as the number of accesses it makes to its working memory. For PRAM machines, each step of computation may make multiple parallel accesses to memory via different processors.

To model the complexity of a (P)RAM machine M, we consider two complexity measures: work and depth. Specifically, we let $\mathsf{work}_M(x)$ denote the total amount of computation done by all processors measured in steps (or equivalently memory accesses). When M is a non-deterministic machine, we denote this by $\mathsf{work}_M(x, w)$ where w is the witness. We let $\mathsf{depth}_M(x)$ (analogously, $\mathsf{depth}_M(x, w)$) denote the number of sequential steps until M halts, where steps that occur in parallel are counted as one step. For a (non-parallel) RAM machine, we simply denote its running time by $\mathsf{work}_M(x)$.

3.2 Universal and NP Relations

Next, we define a variant of the universal relation, introduced by [5]. For efficiency reasons, it will be helpful to define this relative to different computational models, so we give definitions for Turing machine computation and RAM machine computation.

Definition 3.1. *The universal relation for Turing machines $R_{\mathcal{U}}^{\mathsf{TM}}$ is the set of instance-witness pairs $((M, x, t, L, y), w)$ where M is a Turing machine such that $M(x, w)$ outputs y within t steps, and additionally $|y| \leq L$. We let $L_{\mathcal{U}}^{\mathsf{TM}}$ be the corresponding universal language. We similarly define $R_{\mathcal{U}}^{\mathsf{RAM}}$ and $L_{\mathcal{U}}^{\mathsf{RAM}}$ to the be universal relation and language, respectively, for RAM computation, where the given machine M is a RAM machine.*

Following [17, 11], we define the NP relation R_c^{TM} as follows. For every $c \in \mathbb{N}$, we let $R_c^{\mathsf{TM}} \subseteq R_{\mathcal{U}}^{\mathsf{TM}}$ be a subset of the universal relation consisting of pairs $((M, x, t, L, y), w)$ where $t \leq |x|^c$. We let L_c^{TM} be the corresponding language. The relation R_c^{RAM} and language L_c^{RAM} are defined analogously for the case where M is a RAM machine.

The main difference between our definition and the standard universal relation of [5] is that we consider computation with long outputs y, and we also include an upper bound L on the length of y. We include y so as to have a definition which captures both deterministic and non-deterministic polynomial-time computation. A similar relation was given in [17] to define a canonical relation for P. Moreover, the universal relation of [5] is linear-time reducible to our definition above. With regards to L, we include this because in our main construction of SPARKs, the output y of the computation will not be known in advance. However, the complexity of the scheme inherently depends on L (as the output of the protocol is y).

Finally, we note that for a statement (M, x, y, L, t) with respect to RAM computation, we do not place any restriction on the length of the witness w.

Specifically, the machine M may only access t positions in w, but it could be the case that $|w|$ is significantly greater than t.

4 Concurrent Locally Updatable Commitment

In this section we define and construct a commitment that allows for local updates. Furthermore, we require that these local updates can be computed concurrently using multiple processors in a pipelined fashion (described in more detail below). We define our construction in the PRAM model.

For a security parameter $\lambda \in \mathbb{N}$, our commitment will be for strings D consisting of $n \le 2^\lambda$ words of length λ. It will also be helpful for us to capture the case when D is not defined at every location, that is, some words are set to $\perp$. To formalize this, below we define the notion of a partial string, which is simply a succinct way to represent strings over $(\{0,1\}^\lambda \cup \{\perp\})^n$.

Definition 4.1 (Partial string). *For any string $s \in (\{0,1\}^\lambda \cup \{\perp\})^*$ of words, we define the* partial string *D which represents s as follows. D is given by tuple (n, I, A), where n is the number of words (or $\perp$ elements) in s, $I \subseteq [n]$ is the set of non-$\perp$ locations in s, and $A \in \{0,1\}^{|I|}$ is the assignment to those indices. We let D_i denote the ith word in s.*

4.1 Concurrent Locally Updatable Commitment

Our commitment scheme C consists of algorithms with the following syntax:[9]

- $\mathsf{pp} \leftarrow \mathsf{C.Gen}(1^\lambda)$**:** A PPT algorithm that on input the security parameter λ, outputs a key pp.
- $(\mathsf{ptr}, \mathsf{com}) = \mathsf{C.Commit}(\mathsf{pp}, D)$**:** A deterministic algorithm that on input a key pp and a partial string $D = (n, I, A)$, outputs a pointer ptr to a location in memory and a string com.
- $(v, \pi) = \mathsf{C.Open}(\mathsf{pp}, \mathsf{ptr}, \ell)$**:** A read-only deterministic algorithm that on input a key pp, a pointer ptr, and a location $\ell \in [n]$, outputs a value $v \in \{0,1\}^\lambda \cup \{\perp\}$, and a proof π.
- $(\mathsf{com}, \tau) = \mathsf{C.Update}(\mathsf{pp}, \mathsf{ptr}, \ell, v)$**:** A deterministic algorithm that on input a key pp, a pointer ptr, a location $\ell \in [n]$, and a word $v \in \{0,1\}^\lambda$, outputs a commitment com and a proof τ.
- $b' = \mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}, \ell, v, \pi)$**:** A deterministic algorithm that on input a key pp, a commitment com, a location $\ell \in [n]$, a value $v \in \{0,1\}^\lambda \cup \{\perp\}$, and a proof π, outputs a bit b'.
- $b' = \mathsf{C.VerUpd}(\mathsf{pp}, \mathsf{com}, \ell, v, \mathsf{com}', \tau)$**:** A deterministic algorithm that on input a key pp, a commitment com, a location $\ell \in [n]$, a word $v \in \{0,1\}^\lambda$, a commitment com', and a proof τ, outputs a bit b'.

[9] For simplicity, the only randomized algorithm in our definition is the key generation algorithm, and the rest are deterministic. However, with minor modifications to our main protocol, we could use a commitment where all algorithms may be randomized.

We require the following properties.

Definition 4.2 (Completeness). *Let $\lambda \in \mathbb{N}$, pp in the support of $\mathsf{C.Gen}(1^\lambda)$, and let $D = (n, I, A)$ be a partial string. For any $m \geq 0$, and $\ell_i \in [n]$, $v_i \in \{0,1\}^\lambda$ for $i \in [m]$, do the following:*

1. *Compute* $(\mathsf{ptr}, \mathsf{com}_0) = \mathsf{C.Commit}(\mathsf{pp}, D)$.
2. *For $i = 1, \ldots, m$, compute* $(\mathsf{com}_i, \tau_i) = \mathsf{C.Update}(\mathsf{pp}, \mathsf{ptr}, \ell_i, v_i)$.

Let D' be the partial string resulting from writing v_i to D_{ℓ_i} for $i = 1, \ldots, m$. Then, the following hold for any $\ell \in [n]$:

- **Open Completeness.** *Let* $(v, \pi) = \mathsf{C.Open}(\mathsf{pp}, \mathsf{ptr}, \ell)$*. Then,*

$$\mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}_m, \ell, v, \pi) = 1 \ \wedge \ D'_\ell = v.$$

- **Update Completeness.** *For any $v \in \{0,1\}^\lambda$, let* $(\mathsf{com}, \tau) = \mathsf{C.Update}(\mathsf{pp}, \mathsf{ptr}, \ell, v)$*. It holds that*

$$\mathsf{C.VerUpd}(\mathsf{pp}, \mathsf{com}_m, \ell, v, \mathsf{com}, \tau) = 1.$$

Definition 4.3 (Soundness). *For all non-uniform PPT adversaries $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$, there exists a negligible function negl such that for all $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l} \mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}_0, \ell_0, v_0, \pi_0) = 1 \wedge \\ \forall i \in [m] : \mathsf{C.VerUpd}(\mathsf{pp}, \mathsf{com}_{i-1}, \ell_i, v_i, \mathsf{com}_i, \tau_i) = 1 \wedge \\ \mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}_m, \ell_0, v, \pi) = 1 \wedge \\ v \neq v_j \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where j is the largest index with $\ell_j = \ell_0$, and the probability is over the choice of $\mathsf{pp} \leftarrow \mathsf{C.Gen}(1^\lambda)$ *and* $(m, \{(\mathsf{com}_i, \ell_i, v_i, \tau_i)\}_{i \in [m]}, \mathsf{com}_0, \ell_0, v_0, \pi_0, v, \pi) \leftarrow \mathcal{A}_\lambda(\mathsf{pp})$.

Lastly, we require the following efficiency properties, which at a high level say that any sequence of k updates can be computed (while opening the previous values) in a pipelined fashion with only additive overhead.

Definition 4.4 (Efficiency). *Let $\lambda \in \mathbb{N}$ and let $D = (n, I, A)$ be a partial string where $n \leq 2^\lambda$. We say that a concurrent locally updatable commitment satisfies* efficiency *if there exists a polynomial $\beta = \beta(\lambda, \log n)$ such that the following hold:*

- *The algorithms* $\mathsf{C.Open}$, $\mathsf{C.Update}$, $\mathsf{C.VerOpen}$, *and* $\mathsf{C.VerUpd}$ *can each be computed with β work.*
- *Computing* $\mathsf{C.Commit}(\mathsf{pp}, D)$ *can be done with $\beta \cdot (|I| + 1)$ work.*
- *For any key* pp, *pointer* ptr, *location $\ell \in [n]$, and word v, define* $(\pi, \mathsf{com}, \tau)$ *as follows:*
  - $(v', \pi) = \mathsf{C.Open}(\mathsf{pp}, \mathsf{ptr}, \ell)$
  - $(\mathsf{com}, \tau) = \mathsf{C.Update}(\mathsf{pp}, \mathsf{ptr}, \ell, v)$

  There exists an algorithm $\mathsf{OpenUpdate}(\mathsf{pp}, \mathsf{ptr}, \ell, v)$ *which outputs* $(v', \pi, \mathsf{com}, \tau)$*, such that k sequential calls to* $\mathsf{OpenUpdate}$ *can be computed with $k\beta$ work, which can be decoupled into depth $(k-1) + \beta$ using β processors.*

We say that a concurrent locally updatable commitment satisfies β-efficiency *if the above hold with respect to a particular function* β*.*

Remark 4.5. We emphasize that the completeness and soundness properties we give for concurrent locally updatable commitments must hold for any sequence of m "valid" local updates. At a high level, these notions stipulate that an opening will always give the correct value (with a proof) and that no adversary can find an opening for a value you wouldn't expect (based on the local updates). Furthermore, we require $\mathsf{C.VerUpd}$ to ensure that a local update a one location does not affect any other locations.

We note that our definition generalizes standard notions of completeness and position binding for vector commitments [16], as when there are no updates (i.e., $m = 0$), they are equivalent. Our definition also generalized the read and write security properties of other Merkle tree commitments, such as those in [29]. We note that it does not suffice to consider the properties to hold with respect to a single update (i.e., when $m = 1$). This is because our commitments keep state, so it may be the case that it internally keeps a counter and artificially breaks completeness or soundness after some $m > 1$ updates have occurred.

4.2 Construction

Before giving our construction, we discuss the building blocks we will be using.

Merkle trees. Let $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$ be a compressing hash function. A Merkle tree [33] for a string $D \in \{0,1\}^{n\lambda}$ consists of a complete binary tree of $\log n + 1$ levels labelled $0, \ldots, \log n$ where level i consists of $n/2^i$ nodes. Each node is associated with a value in $\{0,1\}^{\lambda}$. The leaves at level 0 correspond to D, split into n blocks of length λ. The value of each node at level $i > 0$ is defined to be the hash (using h) of the concatenation of its children's values at level $i - 1$. The single node at level $\log n$ is referred to as the root or commitment of the Merkle tree.

An authentication path $\pi = (\pi_0, \ldots, \pi_{\log n - 1})$ for a leaf $i \in [n]$ consists of the values in the tree corresponding to the siblings of all nodes along the path from the leaf to the root, ordered from level 0 to $\log n - 1$. An authentication path $\pi = (\pi_0, \ldots, \pi_{\log n - 1})$ for a leaf i is said to be a valid opening for $v \in \{0,1\}^{\lambda}$ with respect to a commitment com if when hashing the value v at leaf i with π_0, hashing the resulting value with π_1, and so on for all values in π, the final value equals com. Whenever updating the value of a leaf i with block block, we additionally re-compute the hash values along the path to the root using its authentication path. The overall size needed to store the Merkle tree in memory is $2n\lambda$ bits.

Assuming the underlying hash function h is collision resistant, it is well known that a Merkle tree is a binding commitment to a fully defined string that allows for local opening and updates. Moreover, it is known that a standard Merkle tree satisfies the standard completeness and binding properties of a commitment.

In our construction, we will want to use a Merkle tree for values $v \in \{0,1\}^\lambda \cup \{\perp\}$. Therefore, we will use a Merkle tree for 2λ-bit values, so that we can uniquely encode each element of $\{0,1\}^\lambda \cup \{\perp\}$ as a string of length 2λ and each node in the Merkle tree corresponds to two consecutive words in memory.

Segment Tree. A segment tree is a data structure that provides a way for the prover to efficiently check if a range of indices in the partial string $D = (n, I, A)$ are $\perp$. To this end, we want to represent the set I (which will be constantly updated) in a way that allows us to check if $[i_1, i_2] \cap I = \emptyset$ in $O(\log n)$ time and independent of $|I|$ and $|i_2 - i_1|$.

To do so, we use a segment tree which mirrors the Merkle tree and consists of a complete binary tree with n leaves. Each node has an associated bit which is 1 if the corresponding node in the Merkle tree has been initialized and 0 otherwise. Every time a leaf in the Merkle tree is updated, we initialize all nodes in the tree along the path to the root, meaning we set the corresponding bits in the segment tree to 1. Then, if any node in the segment tree has a bit of 0, it guarantees that all indices corresponding to the leaves that are descendants of this node are $\perp$. This implies that for any range $[i_1, i_2]$, we can check if $[i_1, i_2] \cap I = \emptyset$ by checking the bits of $O(\log n)$ nodes in the tree that cover this range of indices. This data structure only requires $2n$ additional bits to store.

Our Construction. Let $\mathcal{H} = \{\mathcal{H}_\lambda\}_{\lambda \in \mathbb{N}}$ be a collision-resistant hash function family ensemble with $h \colon \{0,1\}^{4\lambda} \to \{0,1\}^{2\lambda}$ for each $h \in \mathcal{H}_\lambda$. Let $t_{\mathsf{hash}}(\lambda)$ be an upper bound on the running time of each $h \in \mathcal{H}_\lambda$. We also assume that we have a canonical, deterministic encoding of each value in $\{0,1\}^\lambda \cup \{\perp\}$ to 2λ-bit strings, denoted by $\mathsf{block}(v)$ for $v \in \{0,1\}^\lambda \cup \{\perp\}$, which can efficiently decoded (for example, we could represent $v \in \{0,1\}^\lambda$ as $v || 0^\lambda$ and $\perp$ as $1^{2\lambda}$).

We now give our full concurrent updatable commitment construction $\mathsf{C} = (\mathsf{C.Gen}, \mathsf{C.Commit}, \mathsf{C.Open}, \mathsf{C.Update}, \mathsf{C.VerOpen}, \mathsf{C.VerUpd})$.

- $\mathsf{pp} \leftarrow \mathsf{C.Gen}(1^\lambda)$**:** Sample $h \leftarrow \mathcal{H}_\lambda$ and output $\mathsf{pp} = h$.
- $(\mathsf{ptr}, \mathsf{com}) = \mathsf{C.Commit}(\mathsf{pp}, D)$**:**
  1. Allocate $4n\lambda + 2n + 2\lambda \log n$ bits of memory at a pointer ptr, starting with a Merkle tree with n leaves at ptr, a corresponding segment tree at pointer $\mathsf{segtree}$, and $\log n$ extra blocks of size 2λ at pointer aux.
     We assume that all memory is initialized to 0.
  2. Define $\mathsf{dummy}(0) = \mathsf{block}(\perp)$. Let $h = \mathsf{pp}$, and for $j = 1, \ldots, \log n$, compute $\mathsf{dummy}(j) = h(\mathsf{dummy}(j-1) || \mathsf{dummy}(j-1))$ and write it to the next block of free memory at aux.
  3. Recall that $D = (n, I, A)$ specifies a set I of non-$\perp$ indices. For each location $\ell \in I$, run the update procedure defined below by $\mathsf{C.Update}(\mathsf{pp}, \mathsf{ptr}, \ell, D_\ell)$.
  4. Let com be the value of the root in ptr and output $(\mathsf{ptr}, \mathsf{com})$.
- $(v, \pi) = \mathsf{C.Open}(\mathsf{pp}, \mathsf{ptr}, \ell)$**:** Let $\mathsf{segtree}$ be the pointer to the segment tree in memory. For $j \in \{0, \ldots, \log(n) - 1\}$, let node_j be the ancestor of leaf ℓ at level j and let sib_j be its sibling.

For each level $j = 0, \dots, \log(n) - 1$:

1. Read node_j in ptr, and let its value be y_j.
2. Read node_j in $\mathsf{segtree}$, and if its value is 0, let $y_j = \mathsf{block}(\perp)$.
3. Read sib_j in ptr, and let its value be π_j.
4. Read sib_j in $\mathsf{segtree}$, and if its value is 0, set $\pi_j = \mathsf{dummy}(j)$.

Let $v \in \{0,1\}^\lambda \cup \{\perp\}$ be the value such that $y_0 = \mathsf{block}(v)$, or $\perp$ if there is no such value. Output (v, π) where $\pi = (\pi_0, \pi_1, \dots, \pi_{\log(n)-1})$.

- $(\mathsf{com}, \tau) = \mathsf{C.Update}(\mathsf{pp}, \mathsf{ptr}, \ell, v)$**:** Let $\mathsf{segtree}$ be the pointer to the segment tree in memory. For $j \in \{0, \dots, \log(n) - 1\}$, let node_j be the ancestor of leaf ℓ at level j and let sib_j be its sibling. Let $y_0 = \mathsf{block}(v)$.

  For each level $j = 0, \dots, \log(n) - 1$:

  1. **Access Step.** Do the following in parallel:
     (a) Write y_j to node_j in ptr, and let $z_j \in \{0,1\}^{2\lambda}$ be the value overwritten at that location.[10]
     (b) Write 1 to node_j in $\mathsf{segtree}$.
     (c) Read sib_j in ptr, and let its value be π_j.
     (d) Read sib_j in $\mathsf{segtree}$, and if its value is 0, set $\pi_j = \mathsf{dummy}(j)$.
  2. **Hash Steps.** Let y_{j+1} be the hash of the concatenation y_j and π_j (with the leftmost sibling first), using pp.

  Let $v' \in \{0,1\}^\lambda \cup \{\perp\}$ be the value such that $z_0 = \mathsf{block}(v')$, or $\perp$ if there is no such value. Output (com, τ) where $\mathsf{com} = y_{\log n}$ and $\tau = v' || (\pi_0, \pi_1, \dots, \pi_{\log(n)-1})$.
- $b' = \mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}, \ell, v, \pi)$**:** Verify that the authentication path π for leaf ℓ is valid for value $\mathsf{block}(v)$ with respect to com.
- $b' = \mathsf{C.VerUpd}(\mathsf{pp}, \mathsf{com}, \ell, v, \mathsf{com}', \tau)$**:** Output 1 if and only if the following hold:
  1. τ can be parsed as $v' || \pi$ where $v' \in \{0,1\}^\lambda \cup \{\perp\}$ and π is an authentication path.
  2. $\mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}, \ell, v', \pi) = 1$.
  3. $\mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}', \ell, v, \pi) = 1$.

We now prove that our construction satisfies the completeness, soundness, and efficiency properties above assuming collision-resistant hash functions.

Theorem 4.6. *Assuming the existence of collision-resistant hash function families, there exists a concurrently updatable commitment scheme.*

We prove this theorem by showing that C, as described above, satisfies completeness, soundness, and efficiency. The proofs are deferred to the full version.

[10] Note that this is one place where we use the fact that writing to a location in memory returns the value being overwritten. We use this to put the value v' at leaf ℓ in the Merkle tree before the update into the update proof τ, which is used to verify that the commitment before the update and the commitment after the update only differ at one location.

5 Succinct Parallelizable Arguments of Knowledge

In this section, we define SPARKs and show how to construct them from any concurrent locally updatable commitment and succinct argument of knowledge with quasilinear overhead, for a specific NP language, defined in Section 5.1. More precisely, we construct a succinct argument system where the prover runs in optimal parallel time (i.e., depth). We define Succinct Parallelizable Arguments of Knowledge formally below, using the following syntax for interactive protocols. We denote by $\langle P(w), V\rangle$ the output of V in the interaction, which may be of arbitrary (polynomial) length. Furthermore, we let V output $\bot$ to indicate reject, and output $y \neq \bot$ to accept the output y.

Definition 5.1 (SPARK). *A* Succinct Parallelizable Argument of Knowledge (SPARK) *for a relation $R \subseteq R_{\mathcal{U}}^{\mathsf{RAM}}$ is a tuple of probabilistic interactive machines (P, V) where P is a PRAM machine, satisfying the following properties:*

- **Completeness:** *For every $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R$,*

$$\Pr\left[\langle P(w), V\rangle(1^\lambda, (M, x, t, L)) = y\right] = 1,$$

where the probability is over the random coins of P and V.

- **Argument of Knowledge:** *There exists a probabilistic oracle machine $\mathcal{E}$ and a polynomial q such that for every non-uniform polynomial-time prover $P^\star = \{P_\lambda^\star\}_{\lambda\in\mathbb{N}}$, there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$, $(M, x, t, L) \in \{0,1\}^*$ with $|M, x, t| \leq \lambda$ and $L \leq \lambda$, and $z, s \in \{0,1\}^*$, the following hold.*
  Let $P_{\lambda,z,s}^\star$ denote the machine $P_\lambda^\star$ with auxiliary input z and randomness s fixed, let V_r denote the verifier V using randomness $r \in \{0,1\}^{\ell(\lambda)}$ where $\ell(\lambda)$ is a bound on the number of random bits used by $V(1^\lambda, \cdot)$. Then:
  1. *The expected running time of $\mathcal{E}^{P_{\lambda,z,s}^\star, V_r}(1^\lambda, (M, x, t, L))$ is bounded by $q(\lambda, t)$, where the expectation is over $r \leftarrow \{0,1\}^{\ell(\lambda)}$ and the random coins of $\mathcal{E}$.*
  2. *It holds that*

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{\ell(\lambda)} \\ y = \langle P_{\lambda,z,s}^\star, V_r\rangle(1^\lambda, (M, x, t, L)) \\ w \leftarrow \mathcal{E}^{P_{\lambda,z,s}^\star, V_r}(1^\lambda, (M, x, t, L)) \end{array} : y \neq \bot \wedge ((M, x, y, L, t), w) \notin R\right]$$

$$\leq \mathsf{negl}(\lambda).$$

- **Succinctness:** *There exist polynomials p and q such that for any $\lambda \in \mathbb{N}$ and $M, x, t, L \in \{0,1\}^*$, it holds that*

$$\mathsf{work}_V(1^\lambda, (M, x, t, L)) \leq p(\lambda, |(M, x)|, L, \log t)$$

and the length of the transcript produced in the interaction between $P(w)$ and V on common input $(1^\lambda, (M, x, t, L))$ is bounded by $q(\lambda, L, \log t)$.

- **Optimal prover depth:** *There exists a polynomial* p *such that for all* $\lambda \in \mathbb{N}$ *and* $((M, x, t, L, y), w) \in R$*, it holds that*

$$\mathsf{depth}_P(1^\lambda, (M, x, t, L), w) = t + p(\lambda, |(M, x)|, L, \log t)$$

and the total number of processors used by P *is in* $\mathrm{poly}(\lambda, L, \log t)$.

A SPARK for NP *is a uniformly computable ensemble* $\{(P_c, V_c)\}_{c \in \mathbb{N}}$ *where* (P_c, V_c) *is a SPARK for* R_c^{RAM}.

We next remark about some subtleties in our definition and compare to related notions.

Remark 5.2 (Delayed output). We note that our definition of SPARKs has a "delayed output" property where the prover picks the output of the protocol rather than it being known a priori to both the prover and verifier. For typical NP languages, this distinction is not important because the prover is always trying to prove that the relation outputs 1. However, for proving more general polynomial-time computation, the output may not be known in advance, so the prover must compute both the output and a proof.

Remark 5.3 (Execution by execution extraction). Since there may be many possible outputs y of the computation, it is very important that the extractor finds a witness for the actual output y that V accepts in the interaction. Morally, this definition should capture the fact that the prover actually knows a witness *for that output*, instead of a witness for an arbitrary output y' that the prover may never convince the verifier of. This is particularly relevant for NP relations, since when a prover convinces a verifier of an accepting witness (i.e., one where the relation outputs 1) it is not meaningful to extract a witness which makes the relation output 0. Note that it does not suffice to run the protocol and simply give the extractor y (and require the extractor to provide a witness for that output), as the malicious prover may only convince V of any particular y with small probability.

A similar challenge motivated the work on precise proofs of knowledge [35], where they defined arguments of knowledge where the extractor's behavior depended on a specific instance of the protocol.[11] To capture this, their extractor receives a uniformly sampled view of the prover in the protocol and extracts a consistent witness. In our definition above, we choose to give the extractor oracle access to the fixed prover *as well as* the verifier with fixed randomness which results in accepting a particular output y. This is akin to giving the extractor an interactive version of the view, while additionally making the extractor black-box in both the malicious prover and (fixed) verifier. As such, the extractor can emulate the interaction to deterministically figure out the output y it needs to extract for.

[11] They considered instances with different running times, whereas we consider instances with different outputs.

Remark 5.4 (On composition). It is often important for an argument of knowledge to be composable—that is, to be able to be used as a sub-protocol (possibly many times). Indeed, we require this for our transformation from arguments of knowledge to SPARKs. Often, the challenge with composing proofs of knowledge is obtaining the desired running time of the final extractor.

One definition which composes well is precise argument of knowledge [35]. As explained above, in that definition the extractor receives the prover's view in the protocol, and for *every* view, the running time of the extractor is a fixed polynomial (in the prover's running time on that view). However, this notion is quite strong, and hence is not known to hold for standard arguments of knowledge.

A more standard notion is witness-extended emulation [32], where the extractor is not given a view, but instead must output a uniformly distributed view of the verifier as well as a witness. Moreover, the extractor only needs to run in *expected* polynomial time, and may use rewinding. However, when this is used as a sub-protocol, the view picked by the extractor may not be compatible with the external view in the rest of the protocol.

To fix this issue, our definition essentially gives the extractor a uniformly sampled view, and we require that the extractor runs in expected polynomial time over the choice of the view. This can be seen as a relaxation of precise argument of knowledge, since it doesn't need to be efficient for every view, but also as a strengthening of witness-extended emulation, because the extractor must work on a given view, rather than being able to sample one itself.

Remark 5.5 (Standard arguments of knowledge). The definition we use for a succinct argument of knowledge (rather than SPARKs) can be obtained from the above definition by including y in the statement (as is standard for arguments) and making the necessary syntactic changes. The formal definition is deferred to the full version. We note that for succinct arguments of knowledge, the corresponding extraction definition is implied by the definition used in [37].

We our now ready to state our main result.

Theorem 5.6. *[Restatement of Theorem 1.3] Suppose there exists a succinct argument of knowledge for* NP *with quasilinear overhead and a concurrent locally updatable commitment. Then, there exists a SPARK for* NP*.*

Next, we discuss some implications and details of this theorem. Then, to prove Theorem 5.6, we describe a helper language (Section 5.1) and then give the protocol (Section 5.2). We defer the proofs to the full version. We also discuss various extensions of the protocol in the full version.

The round complexity, prover's space complexity, and verifier's efficiency in the SPARK from the above theorem are all preserved from the underlying succinct argument up to $\mathrm{poly}(\lambda, |M, x|, L, \log t)$ factors. Furthermore, we observe that our SPARK has universal completeness, prover runtime, and succinctness, meaning that these three properties hold with respect to the universal relation $R_{\mathcal{U}}^{\mathsf{RAM}}$. Our soundness guarantee, however, requires knowing a polynomial upper bound on t, and as such we construct a protocol for R_c^{RAM} for each c such that

$t = |x|^c$. Alternatively, we could have achieved universal soundness by relying on a superpolynomial assumption on the soundness of the commitment scheme.

We can instantiate Theorem 5.6 with Kilian's 4-round succinct argument of knowledge [31], which exists assuming only collision resistant hash functions. Furthermore, we can instantiate the PCP used by Kilian's succinct argument with an efficient PCP (say [10] which has quasilinear prover running time and polylogarithmic verifier running time). Since we already assume collision resistant hash functions for the commitment, this shows that we can achieve SPARKs for NP from collision resistance alone. Applying the transformation as specified, the round complexity of the resulting transformation would be $\mathrm{poly}(\lambda, |M, x|, \log t)$. However, we can use the fact that for the standard implementation of Kilian (where the prover stores the entire PCP), the prover can compute the last two rounds in $\mathrm{poly}(\lambda, \log t)$ time, so we can do the last two rounds of Kilian in parallel to reduce the round complexity to four. This gives Theorem 1.1. The full details of this modification are described in the full version.

By suitably modifying the SPARK definition to be non-interactive, and relying on any SNARK with quasi-linear overhead, the above transformation can be used to obtain a non-interactive SPARK. This gives Theorem 1.2, for which the formal details are also deferred to the full version.

5.1 The Update Language

For any $c \in \mathbb{N}$, we would like to give a SPARK for R_c^{RAM}. Let (M, x, y, L, t) be any statement in L_c^{RAM}, where M is a RAM program with access to a string $D \in \{0,1\}^{n\lambda}$ in memory for $n \leq 2^\lambda$. To help with our construction, we define the language L_{upd} in Figure 2. This language corresponds to k steps of a RAM computation where at each step we additionally update a commitment corresponding to the memory of M. Specifically, a statement

$$(M, x, k, \mathsf{pp}, \mathsf{state}_0, \mathsf{com}_0, v_0^{\mathsf{rd}}, \mathsf{state}_{\mathsf{final}}, \mathsf{com}_{\mathsf{final}}, v_{\mathsf{final}}^{\mathsf{rd}})$$

is in L_{upd} if there exists a sequence of k consistent updates starting at state state_0 and ending at $\mathsf{state}_{\mathsf{final}}$. The ith update specifies the commitment com_i after that step, the value v_i^{rd} read from memory during that step (if any), and proofs π_i, τ_i validating the operation (read or write) performed at that step.

The relation of this language is defined relative to the values given by $(\mathsf{state}_i, \mathsf{op}_i, \ell_i, v_i^{\mathsf{wt}}) = \mathsf{step}(M, \mathsf{state}_{i-1}, v_{i-1}^{\mathsf{rd}})$ for $i \in [k]$. The relation first checks that the final state state_k and commitment and com_k match those given by the statement. Then, for every step i, it checks (1) that the update from com_{i-1} to com_i is valid (using proof τ_i) and (2) in the case of a read operation, namely $\mathsf{op}_i = \mathsf{rd}$, there is a valid opening for com_{i-1} at position ℓ_i (using proof π_i). Specifically, this check guarantees that v_i^{rd} either already appeared in position ℓ_i in com_{i-1}, or that the position was $\perp$ before step i and was initialized correctly to v_i^{rd} in step i.

The key properties of this language are (1) the witness scales with the length of the computation and *not* the size of the memory, and (2) witnesses for consecutive L_{upd} computations can be merged into a single witness for a larger L_{upd}

Language L_{upd}:

Statement. $(M, x, k, \mathsf{pp}, \mathsf{state}_0, \mathsf{com}_0, v_0^{\mathsf{rd}}, \mathsf{state}_{\mathsf{final}}, \mathsf{com}_{\mathsf{final}}, v_{\mathsf{final}}^{\mathsf{rd}})$
Witness. $(u_1, \dots, u_k)$, where $u_i = (\mathsf{com}_i, v_i^{\mathsf{prev}}, v_i^{\mathsf{rd}}, \pi_i, \tau_i)$ for all $i \in [k]$
Relation R_{upd}. Let $(\mathsf{state}_i, \mathsf{op}_i, \ell_i, v_i^{\mathsf{wt}}) = \mathsf{step}(M, \mathsf{state}_{i-1}, v_{i-1}^{\mathsf{rd}})$ for $i \in [k]$. Then, $(\mathsf{state}_k, \mathsf{com}_k, v_k^{\mathsf{rd}}) = (\mathsf{state}_{\mathsf{final}}, \mathsf{com}_{\mathsf{final}}, v_{\mathsf{final}}^{\mathsf{rd}})$ and for all $i \in [k]$ the following hold:

1. $\mathsf{C.VerUpd}(\mathsf{pp}, \mathsf{com}_{i-1}, \ell_i, v_i^{\mathsf{op}_i}, \mathsf{com}_i, \tau_i) = 1$.
2. $\mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}_{i-1}, \ell_i, v_i^{\mathsf{prev}}, \pi_i) = 1$.
3. $v_i^{\mathsf{prev}} \in \{\perp, v_i^{\mathsf{rd}}\}$.
4. If $v_i^{\mathsf{prev}} = \perp$ and $\ell_i < |x|$, then $v_i^{\mathsf{rd}} = x_{\ell_i}$.

Fig. 2. A language for verifying k steps of a RAM computation M on input x from initial state state_0 to final state $\mathsf{state}_{\mathsf{final}}$.

computation. This allows us to prove that $(M, x, y, L, t) \in L_c^{\mathsf{RAM}}$ with witness w by splitting a proof that $M(x, w) = 1$ into proofs of many sub-computations, where the proof of each sub-computation will correspond to a statement in L_{upd}.

The complexity of L_{upd}. Note that the language L_{upd} is a standard NP language. In particular, verifying that an instance-witness pair corresponding to k updates is in the relation for L_{upd} can be done by a circuit C with $|C| = k \cdot p(\lambda, |M, x|, \log n)$ for a polynomial p. Since we will only be using the succinct argument to prove statements in L_{upd}, we only need it to have quasi-linear overhead with respect to the circuit (or Turing Machine) complexity of this language.

5.2 The Protocol

Before defining our protocol in Figures 3 and 4, we give an overview to introduce the necessary notation and emphasize certain aspects that were omitted for simplicity from the technical overview. Let $(P_{\mathsf{sARK}}, V_{\mathsf{sARK}})$ be the succinct argument of knowledge and let α be its prover efficiency. Let C be the concurrent locally updatable commitment and let β be its efficiency.

As mentioned in Section 5.1, to prove that $((M, x, y, L, t), w) \in R_c^{\mathsf{RAM}}$, we split the computation of $M(x, w)$ into m sub-computations in such a way that the proof of each sub-computation completes roughly by time t. The ith sub-computation consists of a "compute" phase, where we compute k_i steps of the total t steps of computation and maintain a commitment to the memory at each step, and a "proof" phase, where we use $(P_{\mathsf{sARK}}, V_{\mathsf{sARK}})$ to prove correctness of those k_i steps. For the "compute" phase, recall that performing k_i steps of computation while also updating the commitment takes $k_i \cdot \beta$ total work, yet computed in depth $(k_i - 1) + \beta$ using β processors by Theorem 4.6.

To complete the "proof" phase in the desired amount of time, suppose that the work of the prover in the interactive protocol $(P_{\mathsf{sARK}}, V_{\mathsf{sARK}})$ is bounded by a function α of the security parameter and total work of the computation (where we recall that the security parameter also upper bounds the statement size).

For any $k \leq t$ steps of computation, it will be convenient to consider $\alpha^\star$ to be an upper bound on the multiplicative overhead of computing a proof for a statement in L_{upd}. We define this formally below, but it can be roughly thought of as a value upper bounded by $\alpha(\lambda, \beta \cdot t)/(\beta \cdot t)$. Then, the largest number of steps of the computation that we can compute and prove and ensure we finish before time t is $k_1 = t/(\alpha^\star \cdot \beta + 1)$ steps. This is because it takes $k_1 + \beta$ steps to compute (with corresponding hash updates using β processors) and then can be proven in time $k_1 \cdot \alpha^\star \cdot \beta$. Put these together, computing and proving will finish roughly in time $t + \beta$. Furthermore, after computing the first k_1 steps, we can recursively carve out the next largest piece of computation we can finish in time t.

In general, let $\gamma \triangleq \alpha^\star \cdot \beta + 1$. The size of the ith sub-computation will be $k_i = (t/\gamma) \cdot (1 - 1/\gamma)^{i-1}$, which intuitively holds because at each sub-computation we are left with a $(1 - 1/\gamma)$ fraction of the total remaining computation. We continue recursively until the remaining computation is less than $\log \lambda$ steps, which the verifier can then compute directly given the witness, and thus in total recurse for $m = \gamma \log t$ steps. We formalize the above idea in Figure 3 with the algorithm $\mathsf{Compute\text{-}and\text{-}prove}$.

In the full protocol (formalized in Figure 4), the verifier V first sends public parameters for the commitment (which alternatively could be part of a trusted common reference string in the non-interactive setting). The prover P then hashes an initially empty string (corresponding to uninitialized memory) and allocates memory to store the memory D for use when emulating M. M expects D to start with x and w. One way to achieve this would be for P to copy x, w to the start of D in $|x| + |w|$ time, but we want to avoid having P run in time depending on $|w|$ since this could be large. To resolve this, we instead have P translate all accesses to D that correspond to the witness to instead access its own memory where w is located. Because w is only needed to emulate M, if M overwrites the memory containing w, it will not cause any other issues for P. Finally, the prover P runs $\mathsf{Compute\text{-}and\text{-}prove}$ with V as discussed above. After proving all sub-computations, the prover sends the output y and a proof authenticating each word in y. Finally, V accepts if all sub-protocols are valid, the claimed statements are consistent with each other, and if the proofs of the claimed output are valid.

Parameters. For ease of readability for the protocol and corresponding proofs, we define the parameters and assumptions for the protocol with respect to $\lambda \in \mathbb{N}$, the relation R_c^{RAM}, and $M, x, t, L \in \{0,1\}^*$ as follows:

- $\beta \triangleq \beta(\lambda, \log(n))$ is the efficiency of C.
- α is a function representing the prover efficiency of $(P_{\mathsf{sARK}}, V_{\mathsf{sARK}})$. For any security parameter Λ, machine and input of total length X, and time bound T, we assume that $\alpha(\Lambda, X, T)/T \in \mathrm{poly}(\Lambda, X, \log T)$ and is an increasing function in each of its inputs.
- $\alpha^\star \triangleq \alpha(\lambda, |M, x| + 6\lambda + \ell_{\mathsf{Gen}}(\lambda) + \log t, t\beta)/(t\beta)$ is the worst-case multiplicative overhead of running P_{sARK} to prove a statement in L_{upd} corresponding to at

Compute-and-prove

Input: $T, \mathsf{state}_0, \mathsf{com}_0, v_0^{\mathsf{rd}}$
Prover Input: Witness w, ptr
Hardcoded Values: $1^\lambda, M, x, \gamma, \mathsf{pp}$
Protocol:

1. If $T \geq \gamma$, set $k = \lceil T/\gamma \rceil$, which will be the number of steps to compute, and otherwise set $k = T$.
2. P does the following for $i = 1, \ldots, k$:
   (a) Compute $(\mathsf{state}_i, \mathsf{op}_i, \ell_i, v_i^{\mathsf{wt}}) = \mathsf{step}(M, \mathsf{state}_{i-1}, v_{i-1}^{\mathsf{rd}})$.
   (b) If $\mathsf{op}_i = \mathsf{wt}$, update D with v_i^{wt} in location ℓ_i^{wt} and let v_i^{rd} be the value at that location that was overwritten.
   (c) If $\mathsf{op}_i = \mathsf{rd}$, let v_i^{rd} be the value at location ℓ_i in D.
   (d) Spawn a parallel process to compute $\mathsf{OpenUpdate}(\mathsf{pp}, \mathsf{ptr}, \ell_i, v_i^{\mathsf{op}_i})$ (specified by Definition 4.4) and let $(v_i^{\mathsf{prev}}, \pi_i, \mathsf{com}_i, \tau_i)$ be the output.
3. Without waiting Step 2d to halt, if $T \geq \gamma$, P spawns a process to run Compute-and-prove with V on input $(T - k, \mathsf{state}_k, \mathsf{com}_k, v_k^{\mathsf{rd}})$.
4. Once step 2d halts, set $\mathsf{statement} = (M, x, k, \mathsf{pp}, \mathsf{state}_0, \mathsf{com}_0, v_0^{\mathsf{rd}}, \mathsf{state}_k, \mathsf{com}_k, v_k^{\mathsf{rd}})$ and $\mathsf{wit} = ((\mathsf{com}_1, v_1^{\mathsf{prev}}, v_1^{\mathsf{rd}}, \pi_1, \tau_1), \ldots, (\mathsf{com}_k, v_k^{\mathsf{prev}}, v_k^{\mathsf{rd}}, \pi_k, \tau_k))$.
5. If $T \geq \gamma$, P spawns a process to run an interactive argument of knowledge with V to prove that $\mathsf{statement} \in L_{\mathsf{upd}}$ using $(P_{\mathsf{sARK}}(\mathsf{wit}), V_{\mathsf{sARK}})$.
6. Otherwise, when $T < \gamma$, P sends wit to V, and V uses wit directly to verify that $\mathsf{statement} \in L_{\mathsf{upd}}$.

Fig. 3. A parallel algorithm, used in the SPARK in Figure 4, that computes and proves T steps of RAM computation.

most t steps of computation, where $\ell_{\mathsf{Gen}}(\lambda)$ is the output length of $\mathsf{C.Gen}(1^\lambda)$, and so $|M, x| + 6\lambda + \ell_{\mathsf{Gen}}(\lambda) + \log t$ is an upper bound on the length of the L_{upd} statements. Note that $\alpha^\star$ is a function of λ, M, x, t, and β.

– $\gamma \triangleq \alpha^\star \cdot \beta + 1$ is the fraction of remaining computation done at each recursive call to Compute-and-prove. Note that γ is a function of λ, M, x, t, and β.

We formalize the protocol in Figures 3 and 4. We prove Theorem 5.6, that this protocol is a SPARK by showing completeness, argument of knowledge, succinctness, and prover efficiency. The proofs are deferred to the full version.

Acknowledgements. This work was supported in part by NSF Award SATC-1704788, NSF Award RI-1703846, AFOSR Award FA9550-18-1-0267, and by NSF Award DGE-1650441. This research is based upon work supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2019-19-020700006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized

Protocol $\Pi(1^\lambda, (M, x, t, L))$ for R_c^{RAM} between $P(w)$ and V:

1. V computes $\mathsf{pp} \leftarrow \mathsf{C.Gen}(1^\lambda)$ and $(*, \mathsf{com}_{\mathsf{start}}) = \mathsf{C.Commit}(\mathsf{pp}, D_\perp)$, where where $D_\perp$ is the empty partial string. V sends pp to P.
2. Both parties compute γ (as in the parameters paragraph), initialize $\mathsf{state}_{\mathsf{start}}$ as the initial (empty) state of M, and set $v^{\mathsf{rd}}_{\mathsf{start}} = \perp$.
3. P computes $(\mathsf{ptr}, \mathsf{com}_{\mathsf{start}}) = \mathsf{C.Commit}(\mathsf{pp}, D_\perp)$. P additionally allocates memory for M, denoted D, and initialized to zeros (which we assume is free), and copies x to the start of the D. Whenever P needs to access a location ℓ in D that would correspond to the witness (i.e., $|x| < \ell < |x| + |w|$), it instead accesses the corresponding location in w in its own memory. For simplicity, when we write that P accesses a location in D, we implicitly assume it translates the location appropriately.
4. P and V run the sub-protocol $\mathsf{Compute\text{-}and\text{-}prove}(t, \mathsf{state}_{\mathsf{start}}, \mathsf{com}_{\mathsf{start}}, v^{\mathsf{rd}}_{\mathsf{start}})$. For $i \in [m]$, let Π_i be the ith sub-protocol proving $\mathsf{statement}_i := (M_i, x_i, k_i, \mathsf{pp}_i, \mathsf{state}_i, \mathsf{com}_i, v^{\mathsf{rd}}_i, \mathsf{state}'_i, \mathsf{com}'_i, v^{\mathsf{rd}\prime}_i)$.
5. P computes $(y_i, \pi_{i,\mathsf{final}}) = \mathsf{C.Open}(\mathsf{pp}, \mathsf{ptr}, i)$ for $i \in [L']$ where $L' = \lceil L/\lambda \rceil$. Then, P sends $y = y_1 \| \ldots \| y_{L'})$ and $\pi_{\mathsf{final}} = (\pi_{1,\mathsf{final}}, \ldots, \pi_{L',\mathsf{final}})$ to V.
6. V outputs y if the following hold, and outputs $\perp$ otherwise:
   (a) V_{sARK} accepts in $\Pi_1, \ldots, \Pi_{m-1}$ and V accepts in Π_m.
   (b) For all $i \in [m]$, it holds that $(M_i, x_i, \mathsf{pp}_i) = (M, x, \mathsf{pp})$.
   (c) $\sum_{i=1}^{m} k_i = t$ and $t \le |x|^c$.
   (d) $(\mathsf{state}_{\mathsf{start}}, \mathsf{com}_{\mathsf{start}}, v^{\mathsf{rd}}_{\mathsf{start}}) = (\mathsf{state}_1, \mathsf{com}_1, v^{\mathsf{rd}}_1)$.
   (e) $(\mathsf{state}'_i, \mathsf{com}'_i, v^{\mathsf{rd}\prime}_i) = (\mathsf{state}_{i+1}, \mathsf{com}_{i+1}, v^{\mathsf{rd}}_{i+1})$ for all $i \in [m-1]$.
   (f) state'_m is a halting state, $|y| \le L$, and $\mathsf{C.VerOpen}(\mathsf{pp}, \mathsf{com}_m, i, y_i, \pi_{i,\mathsf{final}})$ accepts for all $i \in [L']$.

Fig. 4. A SPARK for R_c^{RAM}.

to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

1. Alwen, J., Blocki, J., Pietrzak, K.: Depth-robust graphs and their cumulative memory complexity. In: Advances in Cryptology - EUROCRYPT. pp. 3–32 (2017)
2. Alwen, J., Blocki, J., Pietrzak, K.: Sustained space complexity. In: Advances in Cryptology - EUROCRYPT. pp. 99–130 (2018)
3. Alwen, J., Chen, B., Kamath, C., Kolmogorov, V., Pietrzak, K., Tessaro, S.: On the complexity of Scrypt and proofs of space in the parallel random oracle model. In: Advances in Cryptology - EUROCRYPT. pp. 358–387 (2016)
4. Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: STOC. pp. 595–603. ACM (2015)

5. Barak, B., Goldreich, O.: Universal arguments and their applications. SIAM J. Comput. 38(5), 1661–1694 (2008)
6. Ben-Sasson, E., Chiesa, A., Gabizon, A., Riabzev, M., Spooner, N.: Interactive oracle proofs with constant rate and query complexity. In: 44th International Colloquium on Automata, Languages, and Programming, ICALP. pp. 40:1–40:15 (2017)
7. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete efficiency of probabilistically-checkable proofs. In: STOC. pp. 585–594. ACM (2013)
8. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Snarks for C: verifying program executions succinctly and in zero knowledge. In: Advances in Cryptology - CRYPTO. pp. 90–108 (2013)
9. Ben-Sasson, E., Chiesa, A., Spooner, N.: Interactive oracle proofs. In: Hirt, M., Smith, A.D. (eds.) Theory of Cryptography - 14th International Conference, TCC. pp. 31–60 (2016)
10. Ben-Sasson, E., Sudan, M.: Short pcps with polylog query complexity. SIAM J. Comput. 38(2), 551–607 (2008)
11. Bitansky, N., Canetti, R., Chiesa, A., Goldwasser, S., Lin, H., Rubinstein, A., Tromer, E.: The hunting of the SNARK. J. Cryptology 30(4), 989–1066 (2017)
12. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: ITCS. pp. 326–349. ACM (2012)
13. Bitansky, N., Chiesa, A.: Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In: Advances in Cryptology - CRYPTO. pp. 255–272 (2012)
14. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Advances in Cryptology - CRYPTO. pp. 757–788 (2018)
15. Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. IACR Cryptology ePrint Archive 2018, 712 (2018)
16. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Public Key Cryptography. Lecture Notes in Computer Science, vol. 7778, pp. 55–72. Springer (2013)
17. Chung, K., Lin, H., Pass, R.: Constant-round concurrent zero knowledge from p-certificates. In: 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS. pp. 50–59 (2013)
18. Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: Versatile verifiable computation. In: IEEE Symposium on Security and Privacy. pp. 253–270. IEEE Computer Society (2015)
19. Dinur, I.: The PCP theorem by gap amplification. J. ACM 54(3), 12 (2007)
20. Döttling, N., Garg, S., Ishai, Y., Malavolta, G., Mour, T., Ostrovsky, R.: Trapdoor hash functions and their applications. IACR Cryptology ePrint Archive 2019, 639 (2019)
21. Döttling, N., Garg, S., Malavolta, G., Vasudevan, P.N.: Tight verifiable delay functions. IACR Cryptology ePrint Archive 2019, 659 (2019)
22. Dwork, C., Goldberg, A.V., Naor, M.: On memory-bound functions for fighting spam. In: Advances in Cryptology - CRYPTO. pp. 426–444 (2003)
23. Dwork, C., Naor, M., Wee, H.: Pebbling and proofs of work. In: Advances in Cryptology - CRYPTO. pp. 37–54 (2005)
24. Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: Continuous verifiable delay functions. IACR Cryptology ePrint Archive 2019, 619 (2019)
25. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Advances in Cryptology - CRYPTO. pp. 186–194 (1986)

26. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: Interactive proofs for muggles. J. ACM 62(4), 27:1–27:64 (2015)
27. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. 18(1), 186–208 (1989)
28. Holmgren, J., Rothblum, R.: Delegating computations with (almost) minimal time and space overhead. In: 59th IEEE Annual Symposium on Foundations of Computer Science, FOCS. pp. 124–135 (2018)
29. Kalai, Y.T., Paneth, O.: Delegating RAM computations. In: TCC (B2). pp. 91–118 (2016)
30. Kalai, Y.T., Raz, R.: Interactive PCP. In: Automata, Languages and Programming, 35th International Colloquium, ICALP. pp. 536–547 (2008)
31. Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing. pp. 723–732. ACM (1992)
32. Lindell, Y.: Parallel coin-tossing and constant-round secure two-party computation. J. Cryptology 16(3), 143–184 (2003)
33. Merkle, R.C.: A certified digital signature. In: CRYPTO. vol. 435, pp. 218–238. Springer (1989)
34. Micali, S.: Computationally sound proofs. SIAM Journal on Computing 30(4), 1253–1298 (2000)
35. Micali, S., Pass, R.: Local zero knowledge. In: Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006. pp. 306–315 (2006)
36. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. Commun. ACM 59(2), 103–112 (2016)
37. Pass, R., Rosen, A.: Concurrent nonmalleable commitments. SIAM Journal on Computing 37(6), 1891–1925 (2008)
38. Pietrzak, K.: Simple verifiable delay functions. In: 10th Innovations in Theoretical Computer Science Conference, ITCS. pp. 60:1–60:15 (2019)
39. Reingold, O., Rothblum, G.N., Rothblum, R.D.: Constant-round interactive proofs for delegating computation. In: 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC. pp. 49–62 (2016)
40. Ron-Zewi, N., Rothblum, R.D.: Local proofs approaching the witness length. IACR Cryptology ePrint Archive 2019, 1062 (2019)
41. Valiant, P.: Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In: Theory of Cryptography, TCC. pp. 1–18 (2008)
42. Wesolowski, B.: Efficient verifiable delay functions. In: Advances in Cryptology - EUROCRYPT. pp. 379–407 (2019)
43. Wu, H., Zheng, W., Chiesa, A., Popa, R.A., Stoica, I.: DIZK: A distributed zero knowledge proof system. In: USENIX Security Symposium. pp. 675–692. USENIX Association (2018)