SPARKs: Succinct Parallelizable Arguments of Knowledge

NAOMI EPHRAIM and CODY FREITAG, Cornell Tech, USA
ILAN KOMARGODSKI, Hebrew University and NTT Research, Israel
RAFAEL PASS, Cornell Tech, USA

We introduce the notion of a *Succinct Parallelizable Argument of Knowledge* (SPARK). This is an argument of knowledge with the following three eficiency properties for computing and proving a (non-deterministic, polynomial time) parallel RAM computation that can be computed in parallel time T with at most p processors:

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

The combination of all three is desirable, as it gives a way to leverage a moderate increase in 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 contribution is a generic construction of SPARKs from any succinct argument of knowledge where the prover's parallel running time is $T \cdot \text{polylog}(T \cdot p)$ when using p processors, assuming collision-resistant hash functions. When suitably instantiating our construction, we achieve a four-round SPARK for *any* parallel RAM computation assuming only collision resistance. Additionally assuming the existence of a succinct *non-interactive* argument of knowledge (SNARK), we construct a non-interactive SPARK that also preserves the space complexity of the underlying computation up to $\text{polylog}(T \cdot p)$ factors.

We also show the following applications of non-interactive SPARKs. First, they immediately imply delegation protocols with near optimal prover (parallel) running time. This, in turn, gives a way to construct verifiable delay functions (VDFs) from any sequential function. When the sequential function is also memory-hard, this yields the first construction of a memory-hard VDF.

CCS Concepts: • Theory of computation → Computational complexity and cryptography; Cryptographic protocols;

Additional Key Words and Phrases: Succinct arguments, parallelization, non-interactive

This is a major revision of a paper with the same title that was published in the proceedings of EUROCRYPT 2020 [33].
This work was supported in part by NSF Award SATC-1704788, NSF Award RI-1703846, AFOSR Award FA9550-18-1-0267, NSF Award DGE-1650441, and DARPA Award HR00110C0086. Ilan Komargodski is supported in part by an Alon Young Faculty Fellowship and by an ISF grant (No. 1774/20). This research is based upon work supported in part by the Ofice of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2019-19-020700006.
Authors' addresses: N. Ephraim, C. Freitag, and R. Pass, Cornell Tech, New York, NY, 10044; emails: {nephraim, cfreitag, rafael}@cs.cornell.edu; I. Komargodski, Hebrew University and NTT Research, Jerusalem, 9190401, Israel; email: ilank@cs.huji.ac.il.

0004-5411/2022/10-ART31 $15.00
https://doi.org/10.1145/3549523

ACM Reference format:
Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. 2022. SPARKs: Succinct Parallelizable Arguments of Knowledge. *J. ACM* 69, 5, Article 31 (October 2022), 88 pages.
https://doi.org/10.1145/3549523

1 INTRODUCTION

Interactive proof systems, introduced by Goldwasser, Micali, and Rackoff [36], 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 eficient (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 [15, 41, 44], succinct argument systems have drawn significant attention due to their appealing eficiency 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., References [18, 25, 38, 57] to name a few) is *prover eficiency*. 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 eficient verifier of the validity of the output. If computing a proof takes much longer than the computation (even, say, a multiplicative factor of two), then this would cause a significant delay making the system useless in various realistic settings. This is particularly relevant for computations that are already incredibly time-consuming, or for applications like verifiable delay functions where the overhead of the prover directly impacts security. This motivates the following question:

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

SPARKs. In this work, we answer the above question afirmatively for any non-deterministic parallel RAM computation. 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) parallel time T using p processors can be proven with the following eficiency requirements (ignoring dependence on the security parameter or statement size):

- The prover's parallel time is $T + \text{polylog}(T \cdot p)$.[1] (In other words, the prover's running time is essentially T for large computations!)
- The prover uses at most $p \cdot \text{polylog}(T \cdot p)$ processors. In other words, the prover preserves the total work and parallelism of the underlying computation up to polylogarithmic factors.
- The communication and verifier complexity are $\text{polylog}(T \cdot p)$.

We 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 number of parallel processors over those used by the computation, 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 $T \cdot p \cdot \text{polylog}(T \cdot p)$ using only a single processor (e.g., References [11, 14]). Without the

[1]Only the additive $\text{polylog}(T \cdot p)$ term is allowed to additionally depend on the security parameter or statement size.

second property, there are existing constructions with parallel time T + polylog$(T \cdot p)$ but require roughly $T \cdot p$ processors (e.g., Reference [11]). No prior construction achieves all three properties simultaneously.

1.1 Our Results

Our results consider succinct arguments for arbitrary non-deterministic polynomial-time PRAM computation. Specifically, we consider machines M that run in parallel time T when using p processors.

Our main contribution is a generic transformation that starts with any succinct argument of knowledge, and shows how to transform *multiplicative* prover overhead to only *additive* overhead. Specifically, given a succinct argument of knowledge where the prover has $\alpha^{?}$ multiplicative overhead (over the depth of the underlying computation) when using p processors, we show how to obtain an argument with poly$(\alpha^{?})$ additive overhead when using roughly $p \cdot \alpha^{?}$ processors. More precisely, we prove the following theorem:

Theorem 1.1 (Informal; see Theorems 6.1 and 6.18). *Assuming collision-resistant hash functions, any succinct argument of knowledge for* NP *where the prover runs in parallel time* $T \cdot \alpha^{?}$ *when using p processors can be generically transformed into a succinct argument where the prover runs in parallel time* $T + (\alpha^{?})^2 \cdot$ polylog$(T \cdot p)$ *when using* $p \cdot \alpha^{?} \cdot$ polylog$(T \cdot p)$ *processors. Additionally, if the original argument is non-interactive, then so is the resulting one.*

We refer to arguments with multiplicative prover overhead $\alpha^{?}$ ⍰ polylog$(T \cdot p)$ when using p processors as *depth-preserving* as they preserve the parallelism and depth of the underlying computation up to polylog$(T \cdot p)$ multiplicative factors. It immediately follows that any depth-preserving succinct argument of knowledge implies a SPARK, assuming collision resistance.

Theorem 1.2 (Informal; see Theorems 7.2 and 7.6). *Assuming collision-resistant hash functions, any depth-preserving succinct argument of knowledge for* NP *can be generically transformed into a SPARK. Additionally, if the underlying argument is non-interactive, then so is the resulting SPARK.*

By instantiating the underlying succinct arguments in the above theorem, we get the following main results: First, by using Kilian's succinct argument [41] with a depth-preserving PCP (which can be obtained from the PCP of Ben-Sasson et al. [11]), we construct four-round SPARKs based on the existence of collision-resistant hash functions alone.

Theorem 1.3 (Informal; see Theorem 7.4). *Assuming collision-resistant hash functions, there exists a four-round SPARK for non-deterministic polynomial-time PRAM computation.*

We additionally construct SPARKs in the non-interactive setting from a succinct *non-interactive* argument of knowledge (SNARK). Specifically, assuming the existence of *any* SNARK (not necessarily depth-preserving), we can construct depth-preserving SNARKs based on the construction of Bitansky et al. [16]. Their SNARK construction also has the property that it is space-preserving, meaning that the space used to construct the proof is at most a polylog$(T \cdot p)$ multiplicative overhead over the space of the computation. The resulting SPARK is therefore also space-preserving, which yields the following theorem:

Theorem 1.4 (Informal; see Theorem 7.8). *Assuming collision-resistant hash functions and any SNARK, there exists a space-preserving, non-interactive SPARK for non-deterministic polynomial-time PRAM computation.*

Model of Computation. We define and build SPARKs for PRAM computations, where our SPARK prover is also a PRAM machine. While the PRAM 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 article 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, then we only count it as a single unit of parallel time. In other words, a SPARK proves a PRAM compu-tation that makes T rounds of parallel memory accesses with $T + \text{polylog}(T \cdot p)$ rounds of parallel accesses.

Similar models have been used in other contexts for delegating RAM computation (see, e.g., References [38, 39]), but they were 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

Below, we present applications of SPARKs that rely on the fact that in a SPARK, the prover both computes and proves the validity of a computation in parallel time, which is essentially as eficient as possible. While our focus here is on establishing theoretical feasibility results, we expect that our ideas may also be useful in practical constructions, which we leave for future work.

Time-tight delegation of PRAM computation. In the problem of verifiable delegation of computation [35, 39, 52], there is a client who wishes to outsource an expensive (possibly non-deterministic) 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 for a class of PRAM computations directly gives a delegation protocol for the same class. 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 PRAM computation as well as compute a proof with essentially no overhead in running time. Specifically, for T-time computations with p processors, the server runs in time $T + \text{polylog}(T \cdot p)$ and uses at most $p \cdot \text{polylog}(T \cdot p)$ processors. We call delegation schemes with this property *time-tight*.

We emphasize that our non-interactive SPARK construction yields a time-tight delegation protocol for *non-deterministic* computations that use any amount of parallelism. For example, consider the case where a client wants to outsource a PRAM 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, then the running time depends only on the time of the PRAM 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).

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 eficiently verified given a proof that can be jointly generated with the output (with only small overhead) [20, 21, 51, 56]. The work of Boneh et al. [20] suggests a theoretical construction of VDFs based on succinct non-interactive arguments (SNARGs) and any *iteratively sequential function* (ISF).[2] Other known constructions of VDFs [51, 56] 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[3]). 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. [20] inherently relies on the iterated structure of the underlying sequential function.[4]

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. Therefore by our main result, any SF, SNARK, and collision-resistant hash function imply a VDF.

Theorem 1.5 (Informal; see Theorem 9.4 and Corollary 9.8). *Assuming the existence of a collision-resistant hash function, a SNARK, and a sequential function, there exists a VDF.*

In fact, one way to view our main construction is by improving existing techniques for constructing verifiable computation for iterated functions from 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 (essentially) zero! 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., References [1–3, 5, 28–30]) and get a VDF where the computation not only takes a long time, but also requires large memory throughout.

Theorem 1.6 (Informal; see Theorem 9.4 and Corollary 9.11). *Assuming the existence of a collision-resistant hash functions, a SNARK, and a memory-hard sequential function, there exists a memory-hard VDF.*

[2]Their original construction relied on *incremental verifiable computation* [55], which exists based on SNARKs [15], and any ISF. In an updated version they show that SNARGs, along with ISFs, are suficient.

[3]However, a *continuous* VDF [32] does imply an ISF.

[4]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 eficiency 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, 16, 48], and so many steps are needed.

Last, we note that sequentiality and memory-hardness are two examples of functions that are hard to compute with bounded resources. Since a SPARK computes a function and constructs the proof in parallel, then the above transformations can be used to preserve *any* hardness property of a PRAM computation, so long as the function remains hard after an additive increase in the parallel running time (and an small increase in the number of parallel processors). This enables generically turning hard functions into verifiable hard functions (see Theorem 9.4 for a formal version of this claim).

1.3 Related Work

Succinct arguments with eficient provers. We elaborate on the existing succinct arguments that focus on prover eficiency. We consider the general setting of proving computation that takes T parallel time using p processors (although most works only explicitly consider the setting where $p = 1$ and T is the total time).

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, eficient PCP constructions immediately give rise to succinct arguments with an eficient prover. Specifically in References [11, 14], they show how to construct PCPs in quasi-linear time, which yield succinct arguments with a prover running in $T \cdot p \cdot \text{polylog}(T \cdot p)$ time for computation with total work $T \cdot p$. In Reference [11], they show how to construct a quasi-linear size PCP that can be computed in $\text{polylog}(T \cdot p)$ depth with roughly $T \cdot p$ processors, when given the transcript of the computation. This results in a succinct argument where the prover runs in parallel time $T + \text{polylog}(T \cdot p)$ using roughly $T \cdot p$ processors. When restricting the prover to use at most $p \cdot \text{polylog}(T \cdot p)$ processors, as required by SPARKs, this yields a succinct argument where the prover runs in parallel time $T \cdot p \cdot \text{polylog}(T \cdot p)$. Furthermore, the above arguments can be made non-interactive by applying the Fiat-Shamir transformation [34, 44].

A different line of work has focused additionally on the prover's *space complexity*. Bitansky et al. [16] (following Valiant's [55] 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 $(T \cdot p)$-time S-space computation, Holmgren and Rothblum [38] give a prover with $T \cdot p \cdot \text{polylog}(T \cdot p)$ total time and $S + o(S)$ space assuming sub-exponential LWE.

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. [20, 26] in the context of verifiable delay functions (VDFs) [20, 21]. 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. However, iterated functions can be naturally split into chunks and so most of the technical dificulty 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. [13] and Reingold et al. [52] introduced the concept of *interactive oracle proofs* (IOPs).[5] 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

[5] To clarify notation, IOPs (introduced by Reference [13]) are equivalent to the notion of Probabilistically Checkable Interactive Proofs (introduced concurrently and independently by Reference [52]).

also generalize Interactive PCPs [40]. The most recent IOP is due to Ron-Zewi and Rothblum [54] (improving Ben-Sasson et al. [10]) 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 small multiplicative prover 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. [20, 27]. For concreteness, we describe the setting of Reference [20], which focuses on the simplified case of proving correctness of the output of an *iterated function* $\partial^{(T)}(x_0) = (\partial \circ \ldots \circ \partial)(x_0)$ for some $T \in \mathbb{N}$. Rather than proving that $\partial^{(T)}(x_0) = x_T$ 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 ∂ in $2k$ additional time. Then, to prove that $\partial^{(T)}(x_0) = x_T$, they first perform $1/3$ of the computation to obtain $\partial^{(T/3)}(x_0) = x_{T/3}$ 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 $2T/3$ by assumption, so the proof that $\partial^{(T/3)}(x_0) = x_{T/3}$ completes by time T. In parallel to proving that $\partial^{(T/3)}(x_0) = x_{T/3}$, they additionally compute and prove $1/3$ *of the remaining computation* (namely, that $\partial^{((T-T/3)/3)}(x_{T/3}) = x_{5T/9}$) 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 $\alpha^{?}$ multiplicative overhead, then 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) that introduces many complications. For now, we focus on the case of RAM computation that uses only a single processor (we later show how to extend this to arbitrary parallel RAM computations). Specifically, suppose we are given a 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 consisting of n words in memory. We let λ be the security parameter and size of a word, and T be an 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 digest of 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 digest. Concretely, each sub-computation corresponding to k steps of computation would contain digests $c_{\mathsf{start}}, c_{\mathsf{final}}$. The prover would show that there exist strings $D_{\mathsf{start}}, D_{\mathsf{final}}$ such that (1) $c_{\mathsf{start}}, c_{\mathsf{final}}$ are digests of $D_{\mathsf{start}}, D_{\mathsf{final}}$, 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 digest 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 digest c_{start} to the final digest 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} that result in c_{final}. Then, to verify a sub-computation corresponding to k steps, we can simply check the k local updates to the digest of the memory, rather than checking the memory in its entirety. To formalize this idea, we rely on compressing hash functions that allow for local updates that can be eficiently computed in parallel to the main computation. We call these *concurrently updatable hash functions*.

Given such hash functions, will use a succinct argument of knowledge $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ for an NP language $\mathsf{L}_{\mathsf{upd}}$ that corresponds to checking that a sequence of local updates are valid. Specifically, a statement $(M, x, k, c_{\mathsf{start}}, c_{\mathsf{final}})$ ⍰ $\mathsf{L}_{\mathsf{upd}}$ if and only if there exists a sequence of updates $u_1, \ldots, u_k$ such that, starting with short digest c_{start}, running M on input x for k steps specifies the updates $u_1, . . . , u_k$ that result in a digest 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.

To make the above approach work, we need updatable hash functions that satisfy the following two properties:

(1) Updates can be computed eficiently in parallel to the main computation.
(2) Updates can be verified as modifying only the specified locations in memory.

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

Concurrently Updatable Hash Functions. Roughly speaking, concurrently updatable hash functions are computationally binding hash functions that support updating parts of the underlying message without re-hashing the whole message. For eficiency, we additionally require that one can perform several sequential 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 5.

We focus on the case where each update is local (a single word per timestep), but we show how to extend this to updating many words in parallel in Section 5. Our construction relies on Merkle trees [43] 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 (binding) 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 block 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 word in the input and compute the new root value without recomputing the whole tree (by updating the hashes along the path from the updated block to the root). All of these local procedures cost time that is proportional to the depth of the tree, $\log_2 n$, 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 hash function. 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 that 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 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 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 timestep, and the next processor can continue in the next timestep without incurring any delay.

Verifying that updates are local. With regards to the soundness of this primitive, a subtle—yet important—point that we need in our application is that it must be possible to prove that a valid update only modifies the locations it specifies. For example, suppose a cheating prover updates the digest with respect to one location in memory while simultaneously rewriting other locations in memory in a way that does not correspond to the memory access done by the machine M. Then, the prover will later be able to open inconsistent values and prove that M computes whatever it wants. Moreover, the prover could gradually make these changes across many different updates. Fortunately, the structure of Merkle trees allow us to prove that a local update only changes a single location. At a high level, this is because the authentication path for a leaf in a Merkle tree effectively binds the root of the tree to the entire memory. Thus, we show that if a Merkle tree is updated at some location, then one can use the authentication path to prove that no other locations were modified. Furthermore, we show in the general case how to extend this for updating many locations in a single update.

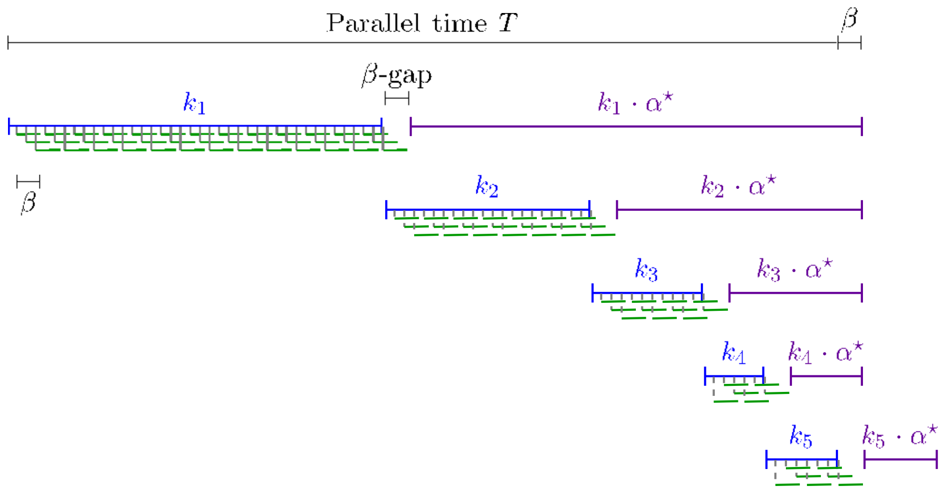

Fig. 1. The "compute" and "proof" phases for each of m sub-computations. The ith sub-computation consists of k_i steps, while pipelining updates which each take β time. After finishing all updates, the prover computes the proof that takes $k_i \cdot \alpha^\star$ time.

Ensuring optimal prover runtime. Using the above ingredients, we discuss how to put everything together to ensure optimal prover runtime. Concretely, suppose we have a concurrently updatable hash function where each update takes time β, and a succinct non-interactive argument of knowledge with quasilinear prover overhead for the language $\mathsf{L}_{\mathsf{upd}}$. Recall that a statement $(M, x, k, c_{\mathsf{start}}, c_{\mathsf{final}}) \in \mathsf{L}_{\mathsf{upd}}$ if there exists a sequence of k hash function 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}. Let $\alpha^\star$ be the multiplicative overhead of the succinct argument with respect to the number of updates (so a computation with $k \leq T$ updates takes time $k \cdot \alpha^\star$ to prove). Note that $\alpha^\star \in \mathrm{poly}(\beta, \log T)$, as we require that the total time to prove a $\mathsf{L}_{\mathsf{upd}}$ statement is quasilinear in the work, and a statement for at most T updates requires $T \cdot \beta$ total work.

As discussed above, to prove that $M(x, w)$ outputs a value y in T steps, we split the computation into m sub-computations that 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 digest takes $k_i \cdot \beta$ total work. However, as described above, we can pipeline these updates so the parallel time to compute these updates is only $(k_i - 1) + \beta$.

For the "proof" phase to complete in the desired amount of time, we need to set the values of k_i appropriately. Each proof for $k_i \leq T$ steps of computation takes at most $k_i \cdot \alpha^\star$ time. Therefore, the largest "chunk" of computation we can compute and prove by roughly time T is $T/(\alpha^\star + 1)$. For convenience, let $\gamma \triangleq \alpha^\star + 1$. 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 $\sum_{i=1}^{m} k_i = T$. This results in roughly $\gamma \log T \in \mathrm{poly}(\beta, \log T)$ total sub-proofs, meaning that the proof size depends only polylogarithmically on 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 $T + \alpha^\star \cdot \gamma + \beta$ parallel time, since the first

$m - 1$ sub-proofs complete by time $T + \beta$, all m sub-computations complete by time $T + \beta$, and the proof of the final γ steps takes roughly $\alpha^? \cdot \gamma$ time to prove. Since $\alpha^?$, γ, and β are in $\mathrm{poly}(\lambda, \log T)$, this implies that we only have a small additive rather than multiplicative overhead.

We note that in the overview above where we discuss SPARKs for iterated functions, correctness of the final sub-computation is proven by having the prover send the witness in the clear, and having the verifier check it directly. In our full construction, we instead have the prover give a succinct proof for the last sub-computation. The main reason for this is that for the case of general parallel RAM computations, we want the communication complexity and the complexity of the verifier to depend only poly-logarithmically on the depth T and processors ρ used in the original computation. However, the witness for the final sub-computation may have length linear in ρ (since at each step in the final sub-computation, the witness may specify the actions of each of the ρ processors). Having the prover instead provide a succinct proof solves this issue.

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. Last, the number of processors used in the protocol is β at all times in the constantly running "compute" phase that is additionally computing updates to the digest in parallel. Then, to run each of the m sub-proofs in parallel, we get at most a factor of m times the number of processors used by a single sub-proof.

Computing the initial digest. Before giving the full protocol, we address a final issue, which is for the prover to compute the digest of the initial memory string. Specifically, the prover needs to hash a string $D \in \{0,1\}^n$, which the RAM machine M assumes contains its inputs (x, w). Directly hashing 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 digest, we simply do not compute a digest of the initial memory! Instead, we start with a digest of an *uninitialized* memory that can be computed eficiently and allows each position to be initialized *exactly once* whenever it is first accessed.

We extend our hash function definition to enable this as follows: We start with a dummy value $\perp$ for the leaves of the Merkle tree. Because the leaves all have the same value, we can compute the root of the Merkle tree eficiently without touching all of the nodes in the tree. Specifically, if the leaves have the value $\mathrm{dummy}(0)$, then we can define the value of the nodes at level j recursively as $\mathrm{dummy}(j) = h(\mathrm{dummy}(j-1)||\mathrm{dummy}(j-1))$. Then the initial digest is just the root $\mathrm{dummy}(\log n)$. Note that here, the prover does not need to initialize the whole tree in memory with dummy values, it simply needs to compute $\mathrm{dummy}(\log n)$ as the initial digest.

Whenever the prover accesses a location in D for the first time, it performs the corresponding local update to the Merkle tree. However, performing this update is non-trivial as many of the nodes in the Merkle tree may still be uninitialized. What saves us is that any uninitialized node must correspond to leaves that are also uninitialized, so they still have the value $\perp$. As such, we can compute the value of any uninitialized node at level j eficiently as $\mathrm{dummy}(j)$. To main-tain eficiency, the prover can keep track of a bit for each node to check if it has been initialized or not.

Given a single authentication path for a newly initialized location in memory, the verifier can check that this path is a valid opening for $\perp$ with the previous digest and for the new value with the updated digest. This guarantees that only the newly initialized value was modified, and the verifier can make sure each location is updated at most once by disallowing the prover from updating locations to $\perp$. Furthermore, the verifier can check that any initialized value not part of the witness (corresponding to the input x) is consistent with what M expects.

2.3 Our SPARK Construction

We now summarize our full SPARK construction. Suppose that we have (1) a concurrently updatable hash function that starts as uninitialized where each update takes time β and (2) a succinct non-interactive argument of knowledge $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ for the update language $\mathsf{L}_{\mathsf{upd}}$ with $\alpha^{?}$ □ poly$(\lambda, \log T)$ multiplicative overhead. Let γ , $\alpha^{?} + 1$, 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 hash function and sends them to P.
(2) Using pp, P computes the digest c_{start} for the uninitialized memory $D_{\text{start}} =$ □^{n}.
(3) P computes T/γ steps of $M(x, w)$ while in parallel updating D_{start} and performing the corresponding local updates to digest $c_1 = c_{\text{start}}$.
(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 digest c_1^0. After computing the current updates, P uses $\mathsf{P}_{\mathsf{sARK}}(u_1, \ldots, u_{T/\gamma})$ for language $\mathsf{L}_{\mathsf{upd}}$ to prove that starting with digest c_1, running M on input x for T/γ steps results in digest c_1^0.
(6) P continues until there are at most γ steps of the computation, at which point P computes and proves the remaining steps and sends the proof to V.
(7) After finishing the computation and all corresponding updates, P uses the final digest to open the output y and give a proof of its correctness. V accepts if the proof certifying y verifies and $\mathsf{V}_{\mathsf{sARK}}$ 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 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 polylog T factor increase in the round complexity. For specific underlying succinct arguments, however, it may be the case that we can synchronize the rounds to reduce the round complexity. Indeed, this is the case for Kilian's succinct argument, which we discuss in Section 7.1.

Extending to PRAM computation. We next discuss how to extend the protocol given above to deal with parallel RAM computation with any number of processors. We assume for simplicity that in the given machine no two processors access the same location in memory concurrently. Suppose M is a PRAM machine where $M(x, w)$ runs in parallel time T using p processors. In the above protocol, we emulate each step of M while performing the corresponding hash function updates in parallel. The SPARK prover can use p processors to emulate M, but as M might access p locations in memory at each step, the hash function needs to support updating any set of p positions concurrently. We show how to generalize the updatable hash function scheme described above to handle such updates while still supporting pipelining for each set of updates. As for eficiency, we observe that this naively increases the overhead to compute each sub-proof by a factor of p (if the overhead scales with the total work rather than the depth of the underlying computation). As such, we need to use an underlying succinct argument that has overhead $\alpha^{?}$ □ polylog$(T \cdot p)$ in the *depth* of the underlying computation while using at most p processors. We refer to such arguments as depth-preserving and discuss how to construct them using known techniques in Sections 7.1 and 7.2.

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. In the non-interactive case, it is pretty straightforward to deal with this, since the statements are all "fixed" and so concurrent composition just works. However, the interactive setting is more challenging, since there are more dependencies. This issue came up and was resolved in previous works, e.g., References [42, 49], where new extraction techniques and definitions were introduced. In our case, we introduce yet another argument of knowledge definition, which (1) enables dealing with this issue in our proof of security, (2) is equivalent to common definitions of proofs of knowledge, and (3) we believe is conceptually simpler and much easier to work with. We view this definition as an additional independent contribution. See Section 4 for additional details in the context of SPARKs and Section 3.3 and Appendix A in the context of standard notions of succinct arguments of knowledge.

3 PRELIMINARIES

Basic notation. For a distribution X we denote by $x \leftarrow X$ the process of sampling a value x from the distribution X. For a set X, we denote by $x \leftarrow \mathsf{X}$ the process of sampling a value x from the uniform distribution on X. $\mathrm{Supp}(X)$ denotes the support of the distribution X. For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, 2, \ldots, n\}$. We use PPT as an acronym for *probabilistic polynomial time*.

A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}$ is *negligible* if it is asymptotically smaller than any inverse-polynomial function, namely, for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) \leq \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda \in \mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda \in \mathbb{N}}$ are *computationally indistinguishable* if for any non-uniform PPT algorithm $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ there exists a negligible function negl such that $|\Pr[\mathsf{A}_\lambda(1^\lambda, X_\lambda) = 1] - \Pr[\mathsf{A}_\lambda(1^\lambda, Y_\lambda) = 1]| \leq \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. For a language L with relation R_L, we let $R_L(x)$ denote the set of witnesses w such that $(x, w) \in R_L$. We say that an ensemble $\{X_n\}_{n \in \mathbb{N}}$ is *uniformly computable* if there exists a Turing Machine M such that $M(1^n)$ outputs X_n in time polynomial in n.

Interactive Protocols. We consider interactive (P)RAM machines and interactive protocols. Formally, we assume there is a specified part of a machine's memory for input from and output to another interactive machine, so the time for an interactive machine to send a message is simply the time to write it to its output tape. Given a pair of interactive machines P and V, we denote by $\langle \mathsf{P}(z_P), \mathsf{V}(z_V) \rangle(x)$ the random variable representing the output of V with common input x and private input z_V, when interacting with P with private input z_P, when the random tape of each machine is uniformly and independently chosen.

The round complexity of the protocol is the number of distinct messages sent between P and V. We say that a protocol is non-interactive if it consists of one message from P to V and then V computes its output. To define the complexity of an interactive machine A, we let $\mathrm{work}_A(x)$ denote the maximum amount of work done by $A(x)$ over any possible interactions.

3.1 RAM Model

Random Access Memory (RAM) computation consists of a machine M that 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,[6] and $n \leq 2^\lambda$ is the number of

[6]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.

words in memory used by M. We assume that M specifies n and that $|(M,x)| \leq n$. 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:

$$(state^0, op, \ell, v^{\mathrm{wt}}) = \mathrm{step}(M, state, v^{\mathrm{rd}}).$$

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 $state^0$, the operation $op \in \{\mathrm{rd}, \mathrm{wt}\}$ to do next, the next location $\ell \in [n]$ to access, and the word v^{wt} to write next if $op = \mathrm{wt}$ (or $\perp$ if $op = \mathrm{rd}$).

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 $v_0^{\mathrm{rd}} = \perp$, the ith step of computation for $i \geq 1$ is def ined as:

(1) Compute $(state_i, op_i, \ell_i, v_i^{\mathrm{wt}}) = \mathrm{step}(M, state_{i-1}, v_{i-1}^{\mathrm{rd}})$.
(2) If $op_i = \mathrm{rd}$, then let v_i^{rd} be the word in location ℓ_i of D.
(3) If $op_i = \mathrm{wt}$, then 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 the final state specifies the output length. Without loss of generality, we assume that the state size can hold $O(\log n)$ bits.

Parallel RAM Computation. Our main results will be in the parallel-RAM (PRAM) setting, where each step of the machine can potentially branch to multiple processes that have access to the same memory D. This can be formalized by allowing step to output multiple tuples $(state^0, op, \ell, v^{\mathrm{wt}})$, each associated with a process identifier specifying the process to continue the computation from that state. Then, each process continues by running step at each step, as above. The computation halts when there are no running processes.

For convenience, we define an algorithm parallel-step that logically runs step for all active processes. It has the following syntax:

$$(State^0, Op, S, V^{\mathrm{wt}}) = \text{parallel-step}(M, State, V^{\mathrm{rd}}).$$

Here, all inputs and outputs are tuples containing a value for each process. Specifically, if there are p active processes before the step, and p^0 resulting processes, then $State = (state_i)_{i\in[p]}$, $V^{\mathrm{rd}} = (v_i^{\mathrm{rd}})_{i\in[p]}$, $State^0 = (state_i^0)_{i\in[p^0]}$, $Op = (op_i)_{i\in[p^0]}$, $S = (\ell_i)_{i\in[p^0]}$, $V^{\mathrm{wt}} = (v_i^{\mathrm{wt}})_{i\in[p^0]}$. For each $i \in [p]$, in the previous step the ith process had state $state_i$ and read (or overwrote) value v_i^{rd}. For each $i \in [p^0]$, the ith process after the step has state $state_i^0$, and accesses location ℓ_i in memory by either writing v_i^{wt} to it when $op_i = \mathrm{wt}$, or reads from it when $op_i = \mathrm{rd}$. Note that V^{wt} contains $\perp$ for each element corresponding to a read operation. Also, note that if process i was spawned in this step, then $state_i^0$ will be its initial state.

For ease of notation, we will also define an algorithm access, which accesses a set of locations in memory and then reads and writes to them as specified. Specifically, $\mathrm{access}^D(Op, S, V^{\mathrm{wt}})$ has memory access to D, takes as input Op, S, and V^{wt} as defined above, and does the following for each $i \in [|Op|]$:

(1) If $op_i = \mathrm{rd}$, then let v_i^{rd} be the word at location ℓ_i of D.
(2) If $op_i = \mathrm{wt}$, then let v_i^{rd} be the word at location ℓ_i in D and write v_i^{wt} to that location.

It then outputs $V^{\mathrm{rd}} = (v_1^{\mathrm{rd}}, \ldots, v_{|Op|}^{\mathrm{rd}})$.

Using parallel-step and access, we can then formalize a full PRAM computation as follows: Starting with $State_0 = (state_0)$, where $state_0$ is an initially empty state, and $V^{\mathrm{rd}}_0 = (⍰)$, the ith step of the PRAM computation M for $i \geq 1$ is def ined as:

(1) Compute $(State_i, Op_i, S_i, V^{\mathrm{wt}}_i) = \text{parallel-step}(M, State_{i-1}, V^{\mathrm{rd}}_{i-1})$.
(2) Let $V^{\mathrm{rd}}_i = \text{access}^D(Op_i, S_i, V^{\mathrm{wt}}_i)$.

The computation halts when all running processes reach a halting state, and the output y of $M(x)$ is written to the start of the memory, where we additionally assume that the output length is encoded in the final state(s).

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), then there are no guarantees for the resulting effect. In addition to specifying the memory size n, we also assume that a PRAM machine specifies the number of concurrent processes p it uses, and that $p \leq n$, as we are in the EREW model. Last, we assume that all processes in a PRAM computation have local registers that can be used to communicate the results of each step.

(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*^{0}, as being implemented by an eficient 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 $\text{work}_M(x)$ 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 $\text{work}_M(x, w)$ where w is the witness. We let $\text{depth}_M(x)$ (analogously, $\text{depth}_M(x, w)$) 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 $\text{work}_M(x)$.

We also assume that n words in memory can be allocated and initialized to zeros for free.

3.2 Universal and NP Relations

Next, we define a variant of the universal relation, introduced by Reference [8]. For eficiency 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 (Universal Relation). The universal relation for Turing machines $\mathsf{R}^{\mathrm{TM}}_{\mathsf{U}}$ is the set of instance-witness pairs $((M, x, y, L, t), w)$ where M is a Turing machine such that $M(x, w)$ outputs y within t steps, and additionally $|y| \leq L$. We let $\mathsf{L}^{\mathrm{TM}}_{\mathsf{U}}$ be the corresponding universal language. We similarly define $\mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}$ and $\mathsf{L}^{\mathrm{PRAM}}_{\mathsf{U}}$ to the be universal relation and language, respectively, for PRAM computation, where the given machine M is a PRAM machine.

The main difference between our definition and the standard universal relation of Reference [8] 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 to have a definition that captures both deterministic and non-deterministic polynomial-time computation. A similar relation was given in Reference [24] to define a canonical relation for P. Moreover, the universal relation of Reference [8] 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 PRAM 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.

3.3 Succinct Arguments of Knowledge

In this section, we define succinct arguments of knowledge [8, 41, 44] for relations $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$. We focus on NP languages and relations, where the argument of knowledge definition is restricted to polynomial-time statements.

Definition 3.2 (Succinct Arguments of Knowledge for NP *Relations).* Let $\alpha : \mathbb{N}^3 \to \mathbb{N}$. A pair of interactive machines (P, V) is called a *succinct argument of knowledge with α-prover eficiency* for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ if the following conditions hold:

- Completeness: For every $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R$,
$$\Pr\left[\langle \mathsf{P}(w), \mathsf{V}\rangle(1^\lambda, (M, x, y, L, t)) = 1\right] = 1,$$
where the probability is over the random coins of P and V.
- Argument of Knowledge for NP: There exists a probabilistic oracle machine E and a polynomial q such that for every non-uniform probabilistic polynomial-time prover $\mathsf{P}^\star = \{\mathsf{P}^\star_\lambda\}_{\lambda \in \mathbb{N}}$ and every constant $c \in \mathbb{N}$, there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$, $M, x, t, L, y \in \{0,1\}^*$ with $|M, x, t, y| \le \lambda$, $L \le \lambda$, and $t \le |x|^c$, and every $z, s \in \{0,1\}^*$, the following hold:
  Let $\mathsf{P}^\star_{\lambda,z,s}$ denote the machine $\mathsf{P}^\star_\lambda$ 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 $\mathsf{V}(1^\lambda, \cdot)$. Then:
  (1) The expected running time of $\mathsf{E}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, y, L, t))$ is bounded by $q(\lambda, t)$, where the expectation is over $r \leftarrow \{0,1\}^{\ell(\lambda)}$ and the random coins of E.
  (2) It holds that
$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{\ell(\lambda)} \\ w \leftarrow \mathsf{E}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, y, L, t)) \end{array} : \begin{array}{l} \langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r\rangle(1^\lambda, (M, x, y, L, t)) = 1 \\ \wedge\ ((M, x, y, L, t), w) \notin R \end{array}\right] \le \mathsf{negl}(\lambda).$$
- Succinctness: There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$ and $(M, x, y, L, t) \in \{0,1\}^*$, it holds that
$$\mathrm{work}_{\mathsf{V}}(1^\lambda, (M, x, y, L, t)) \le q_1(\lambda, |(M, x, y, L)|, \log t)$$
and the length of the transcript produced in the interaction between $\mathsf{P}(w)$ and V on common input $(1^\lambda, (M, x, y, L, t))$ is bounded by $q_2(\lambda, \log t)$.
- α-Prover Runtime: For all $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R$, it holds that
$$\mathrm{work}_{\mathsf{P}}(1^\lambda, (M, x, y, L, t), w) \le \alpha(\lambda, |(M, x, y, L)|, t).$$

If the above holds for $R = \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$, then we say that (P, V) is a *succinct argument of knowledge for* NP.

We note that we could naturally relax the above definition so completeness and eficiency only hold for statements (M, x, y, L, t) where $t \le T(|x|)$ for some slightly super-polynomial function T, as in References [7, 24]. In our results, if we assume this weaker notion, then our resulting SPARK will satisfy the same notion.

We also note that the above definition captures succinct arguments of knowledge for any specific NP language L with relation R_{L} (not necessarily contained in $\mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$). The relation R_{L} implicitly determines an NP verification machine M_{L} with time bound $T \in \mathrm{poly}(|x|)$. Then, we can consider the relation $R = \{((M_{\mathsf{L}}, x, 1, 1, T(|x|)), w) : M_{\mathsf{L}}(x, w) = 1 \text{ within } T(|x|) \text{ steps}\} \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$.

Remark 1 (Comparison with Previous Definitions). In contrast to the definition of *universal* arguments of knowledge, the argument of knowledge definition above for NP holds only for all malicious provers $P^?$ and constants c where the statements (M, x, y, L, t) have $t \leq |x|^c$. We also define the extractor to run in expected polynomial time $q(\lambda, t)$ where q is a polynomial independent of $\mathsf{P}^?$ or the specific time bound $|x|^c$. This is in spirit of universal arguments [8] where they define a weak extractor that only extracts a single bit of the witness at a time (because they deal with t, which is not necessarily bounded by a polynomial).

We note that for NP, our extractor definition differs from the standard notion, which does not give the extractor oracle access to $\mathsf{V}_?$, runs in expected time proportional to $\epsilon(\lambda) - \kappa(\lambda)$, and always extracts a valid witness. Here, $\epsilon(\lambda)$ is the success probability of $\mathsf{P}^?_{\lambda,z,s}$ and $\kappa(\lambda)$ is the knowledge error (see Reference [9]). Nevertheless, we show in Section A that our definition for NP is implied by a definition of witness-extended emulation for NP arguments, which is in turn implied by the standard argument of knowledge definition for NP with negligible knowledge error [42] (with minor modifications to fit into our setting).

We emphasize that the above definition is given for relations in $\mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ where the time bound t represents the total work of the computation. We can readily extend this to relations for *parallel* computations where the machine M runs in depth t and uses p_M processors, by generically bounding the total work by $t \cdot p_M$ in the above definition. Below, we more precisely quantify the prover eficiency for parallel computations by decoupling the prover's depth and parallelism, which may depend on the parallelism and depth of the underlying computation.

Definition 3.3 (Decoupling Prover Eficiency for Succinct Arguments). Let $\alpha, \rho \colon \mathbb{N}^4 \to \mathbb{N}$. A succinct argument of knowledge (P, V) for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ satisfies (α, ρ)*-prover eficiency* if for all $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R$ where M uses at most p_M processors, it holds that

$$\mathrm{work}_{\mathsf{P}}(1^\lambda, (M, x, y, L, t), w) \leq \alpha(\lambda, |(M, x, y, L)|, t)$$

using $\rho(\lambda, |(M, x, y, L)|, t)$ processors.

We may also consider relations R consisting of parallel machines M that use p_M processors, in which case α and ρ may additionally depend on p_M.

We note that a succinct argument of knowledge with α-prover runtime immediately gives a succinct argument of knowledge satisfying $(\alpha^0, 1)$-prover eficiency where $\alpha^0(\lambda, |(M, x, y, L)|, t, p_M) = \alpha(\lambda, |(M, x, y, L)|, t \cdot p_M)$.

SNARKs. Next, we define succinct non-interactive arguments of knowledge.

Definition 3.4 (SNARKs for NP *Relations).* A *Succinct Non-interactive Argument of Knowledge (SNARK)* for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ is a tuple of probabilistic algorithms $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ with the following syntax:

- $(crs, \mathrm{st}) \leftarrow \mathsf{G}(1^\lambda)$: A PPT algorithm that on input a security parameter λ outputs a common reference string crs and a verification state st.
- $\pi \leftarrow \mathsf{P}(crs, (M, x, y, L, t), w)$: A probabilistic algorithm that on input a common reference string crs, a statement (M, x, y, L, t), and a witness w, outputs a proof π.
- $b \leftarrow \mathsf{V}(\mathrm{st}, (M, x, y, L, t), \pi)$: A PPT algorithm that on input a verification state st, a statement (M, x, y, L, t), and a proof π, outputs a bit b indicating whether to accept or reject.

We require the following properties:

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

$$\Pr\left[\begin{array}{l} (crs, \mathsf{st}) \leftarrow \mathsf{G}(1^\lambda) \\ \pi \leftarrow \mathsf{P}(crs, (M, x, y, L, t), w) \\ b \leftarrow \mathsf{V}(\mathsf{st}, (M, x, y, L, t), \pi) \end{array} : b = 1\right] = 1.$$

- Adaptive Argument of Knowledge for NP: For any non-uniform polynomial-time prover $\mathsf{P}^\star = \{\mathsf{P}^\star_\lambda\}_{\lambda \in \mathbb{N}}$, there exists a probabilistic machine E and a polynomial q, such that for every $c \in \mathbb{N}$, there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$ and $z, s \in \{0,1\}^*$, the following hold:

  Let $\mathsf{P}^\star_{\lambda,z,s}$ denote the machine $\mathsf{P}^\star_\lambda$ with auxiliary input z and randomness s fixed. Then:
  (1) The running time of $\mathsf{E}(crs, z, s)$ is bounded by $q(\lambda, t)$, where $(crs, st) \leftarrow \mathsf{G}(1^\lambda)$, and t is given by the statement output by $\mathsf{P}^\star_{\lambda,z,s}(crs)$.
  (2) It holds that

$$\Pr\left[\begin{array}{l} (crs, \mathsf{st}) \leftarrow \mathsf{G}(1^\lambda) \\ ((M, x, y, L, t), \pi) \leftarrow \mathsf{P}^\star_{\lambda,z,s}(crs) \\ b \leftarrow \mathsf{V}(\mathsf{st}, (M, x, y, L, t), \pi) \\ w \leftarrow \mathsf{E}(crs, z, s) \end{array} : \begin{array}{l} b = 1 \wedge \\ ((M, x, y, L, t), w) \notin R \wedge \\ t \leq |x|^c \end{array}\right] \leq \mathsf{negl}(\lambda).$$

- Succinctness: There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$, (crs, st) in the support of $\mathsf{G}(1^\lambda)$, $(M, x, y, L, t) \in \{0,1\}^*$ with $|y| \leq L$, witness w, and proof π in the support of $\mathsf{P}(crs, (M, x, y, L, t), w)$,[7] it holds that
  - $\mathsf{work}_\mathsf{V}(\mathsf{st}, (M, x, y, L, t), \pi) \leq q_1(\lambda, |(M, x, y, L)|, \log t)$ and
  - $|\pi| \leq q_2(\lambda, \log t)$.
- α-Prover Runtime: For all $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R$, it holds that

$$\mathrm{depth}_\mathsf{P}(crs, (M, x, y, L, t), w) = \alpha(\lambda, |(M, x, y, L)|, t).$$

If the above holds for $R = \mathsf{R}^{\mathrm{TM}}_\mathsf{U}$, then we say that $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ is a *SNARK for* NP. When $crs = \mathsf{st}$ for $\mathsf{G}(1^\lambda)$, we say that the SNARK is *publicly verifiable* and write $crs \leftarrow \mathsf{G}(1^\lambda)$.

We note that our definition of adaptive argument of knowledge for NP is implied by the definition of Reference [16] for NP. As in the interactive setting, we can similarly relax the completeness and eficiency properties to only hold for statements with t bounded by a slightly super-polynomial function $T(|x|)$ as in Reference [16].

Remark 2 (On the Distribution over the Auxiliary Input). With regards to auxiliary input, our SNARK definition follows the convention of Reference [15]. However, as they point out, it was shown by References [17, 22] that this definition is too strong. In particular, they show that it is impossible to achieve assuming indistinguishability obfuscation. As such, the argument of knowledge definition can be relaxed to consider security with respect to a particular distribution of auxiliary input appropriate for the specific application.

As with interactive arguments, we can also extend the above definition to decouple prover eficiency into prover depth and parallelism.

[7]Note that we could additionally require a verifier to be eficient for "dishonest" proofs that are not in the support of an honest prover P. However, given any verifier V that satisfies succinctness for honest proofs with universal polynomial p, we can construct an eficient verifier V' for any proof by running V for at most $p(\lambda, |(M, x, y, L)|, \log t)$ steps and rejecting otherwise.

Definition 3.5 (Decoupling Prover Efficiency for SNARKs). Let $\alpha, \rho \colon \mathbb{N}^3 \to \mathbb{N}$. A SNARK $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ satisfies (α, ρ)-*prover efficiency* if for all $\lambda \in \mathbb{N}$, (crs, st) in the support of $\mathsf{G}(1^\lambda)$, and $((M, x, y, L, t), w) \in R$, it holds that

$$\mathrm{work}_{\mathsf{P}}(crs, (M, x, y, L, t), w) \leq \alpha(\lambda, |(M,x,y,L)|, t)$$

using $\rho(\lambda, |(M,x,y,L)|, t)$ processors.

We may also consider relations R consisting of parallel machines M that use p_M processors, in which case α and ρ may additionally depend on p_M.

4 SUCCINCT PARALLELIZABLE ARGUMENTS OF KNOWLEDGE

In this section, we define succinct parallelizable arguments of knowledge for non-deterministic polynomial-time PRAM computation, using the following syntax for interactive protocols: We denote by $\langle \mathsf{P}(w), \mathsf{V} \rangle$ 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 4.1 (SPARKs for NP *Relations).* A *Succinct Parallelizable Argument of Knowledge (SPARK)* for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ 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 $((M, x, y, L, t), w) \in R$ where M has access to $n \leq 2^\lambda$ words in memory,

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

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

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

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ y = \langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r \rangle(1^\lambda, (M, x, t, L)) \\ w \leftarrow \mathsf{E}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, t, L)) \end{array} : y \neq \bot \wedge ((M, x, y, L, t), w) \notin R\right] \leq \mathsf{negl}(\lambda).$$

- Succinctness: There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$, $(M, x, t, L) \in \{0,1\}^\star$ where M has access to $n \leq 2^\lambda$ words in memory and p_M processors, it holds that

$$\mathrm{work}_{\mathsf{V}}(1^\lambda, (M, x, t, L)) \leq q_1(\lambda, |(M,x)|, L, \log(t \cdot p_M))$$

and the length of the transcript produced in the interaction between $\mathsf{P}(w)$ and V on common input $(1^\lambda, (M, x, t, L))$ is bounded by $q_2(\lambda, L, \log(t \cdot p_M))$.

- Optimal prover depth: There exist polynomials q_1, q_2 such that for all $\lambda \in \mathsf{N}$ and $((M, x, y, L, t), w) \in R$ where M has access to $n \leq 2^\lambda$ words in memory and p_M processors, it holds that

$$\mathrm{depth}_{\mathsf{P}}(1^\lambda, (M, x, t, L), w) \leq t + q_1(\lambda, |(M, x)|, L, \log(t \cdot p_M))$$

and the total number of processors used by P is at most $p_M \cdot q_2(\lambda, \log(t \cdot p_M))$.

If the above holds for $R = \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$, then we say that (P, V) is a *SPARK for non-deterministic polynomial-time PRAM computation.*

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

Remark 3 (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 4 (Execution-dependent 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^0 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 sufice 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 [45], where they defined arguments of knowledge where the extractor's behavior depended on a specific instance of the protocol.[8] 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 that results in accepting a particular output y. This is akin to giving the extractor an oracle 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.

Remark 5 (On Composition). It is often important for arguments 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 that composes well is precise argument of knowledge [45]. 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 [42], 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

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

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 does not need to be eficient for every view, but also as a (conceptual) strengthening of witness-extended emulation, because the extractor must work on a given view, rather than being able to sample one itself.

Remark 6 (On the Dependence on Parallelism). An important contribution of our SPARK definition is decoupling the time of a PRAM computation from the total work done. As such, we briefly discuss the dependence on the number of processors used by the underlying PRAM machine.

For a PRAM machine M that uses p_M processors and runs in time t, we note that the work of M can be generically bounded by $t \cdot p_M$. Therefore, we use $t \cdot p_M$ in place of the usual notion of work for succinctness and prover eficiency.

The only other dependence on p_M in our SPARK definition is in the amount of processors we allow the prover to use. As the prover must emulate $M(x, w)$ in roughly the same depth that M uses, the prover needs to at least use p_M processors. Furthermore, we require in our definition that the parallelism is preserved up to multiplicative $\text{poly}(\lambda, \log(t \cdot p_M))$ factors, following similar definitions for complexity-preserving arguments [18].

Non-interactive SPARKs. Next, we define non-interactive SPARKs for non-deterministic polynomial-time PRAM computation. Non-interactive SPARKs differ from SNARKs (Definition 3.4) in two key ways, analogously to the interactive setting. First, a non-interactive SPARK must compute the output of the (possibly non-deterministic) computation while computing the proof, and second, we require near-optimal prover eficiency. However, the other requirements, most notably the argument of knowledge definition, are nearly the same as in SNARKs.

Definition 4.2 (Non-interactive SPARKs for NP *Relations).* A *Non-interactive Succinct Parallelizable Argument of Knowledge (niSPARK)* for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ is a tuple of probabilistic algorithms $(\mathsf{G}_{\mathsf{ni}}, \mathsf{P}_{\mathsf{ni}}, \mathsf{V}_{\mathsf{ni}})$ with the following syntax:

- $(crs, \mathrm{st}) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda)$: A PPT algorithm that on input a security parameter λ outputs a common reference string crs and a verification state st.
- $(y, \pi) \leftarrow \mathsf{P}_{\mathsf{ni}}(crs, (M, x, t, L), w)$: A probabilistic algorithm that on input a common reference string crs, a statement (M, x, t, L), and a witness w, outputs a value y and a proof π.
- $b \leftarrow \mathsf{V}_{\mathsf{ni}}(\mathrm{st}, (M, x, y, L, t), \pi)$: A PPT algorithm that on input a verification state st, a statement (M, x, y, L, t), and a proof π, outputs a bit b indicating whether to accept or reject.

We require the following properties:

- Completeness: For every $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R$ where M has access to $n \leq 2^\lambda$ words in memory,

$$\Pr\left[\begin{array}{l} (crs, \mathrm{st}) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda) \\ (y, \pi) \leftarrow \mathsf{P}_{\mathsf{ni}}(crs, (M, x, t, L), w) \\ b \leftarrow \mathsf{V}_{\mathsf{ni}}(\mathrm{st}, (M, x, y, L, t), \pi) \end{array} : b = 1\right] = 1.$$

- Adaptive Argument of Knowledge for NP: For all non-uniform polynomial-time provers $\mathsf{P}^\star = \{\mathsf{P}^\star_\lambda\}_{\lambda \in \mathbb{N}}$, there exists a probabilistic machine E and a polynomial q such that for every constant $c \in \mathbb{N}$, there is a negligible function negl such that for every $\lambda \in \mathbb{N}$ and $z, s \in \{0, 1\}^*$, the following hold:

Let $\mathsf{P}^{\star}_{\lambda,z,s}$ denote the machine $\mathsf{P}^{\star}_{\lambda}$ with auxiliary input z and randomness s fixed. Then:

(1) The running time of $\mathsf{E}(crs, z, s)$ is bounded by $q(\lambda, t \cdot p_M)$, where t is given by the statement (M, x, y, L, t) output by $\mathsf{P}^{\star z}_{\lambda,\,,s}(crs)$ and p_M is the number of processors used by M.

(2) It holds that

$$\Pr\left[\begin{array}{l}(crs, \mathrm{st}) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^{\lambda})\\ ((M,x,y,L,t),\pi) \leftarrow \mathsf{P}^{\star}_{\lambda,z,s}(crs)\\ b \leftarrow \mathsf{V}_{\mathsf{ni}}(\mathrm{st},(M,x,y,L,t),\pi)\\ w \leftarrow \mathsf{E}(crs,z,s)\end{array} : \begin{array}{l} b = 1 \wedge\\ ((M,x,y,L,t),w) \notin R \wedge\\ t \cdot p_M \leq |x|^c\end{array}\right] \leq \mathsf{negl}(\lambda),$$

where p_M is the number of processors used by M.

- Succinctness: There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$, (crs, st) in the support of $\mathsf{G}_{\mathsf{ni}}(1^{\lambda})$, $(M, x, t, L) \in \{0,1\}^{*}$ where M uses $n \leq 2^{\lambda}$ words in memory and p_M processors, witness w, and (y, π) in the support of $\mathsf{P}_{\mathsf{ni}}(crs, (M, x, t, L), w)$, it holds that
  - $\mathrm{work}_{\mathsf{V}_{\mathsf{ni}}}(\mathrm{st}, (M, x, y, L, t), \pi) \leq q_1(\lambda, |(M, x)|, L, \log(t \cdot p_M))$,
  - $|y| \leq L$, and
  - $|\pi| \leq q_2(\lambda, L, \log(t \cdot p_M))$.
- Optimal prover depth: There exists polynomials q_1 and q_2 such that for all $\lambda \in \mathbb{N}$ and $((M, x, t, L, y), w) \in R$ where M has access to $n \leq 2^{\lambda}$ words in memory and p_M processors, it holds that

  $$\mathrm{depth}_{\mathsf{P}_{\mathsf{ni}}}(crs, (M, x, t, L), w) = t + q_1(\lambda, |(M, x)|, L, \log(t \cdot p_M))$$

  and the total number of processors used by P_{ni} is in $p_M \cdot q_2(\lambda, \log(t \cdot p_M))$.

If the above holds for $R = \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}$, then we say that $(\mathsf{G}_{\mathsf{ni}}, \mathsf{P}_{\mathsf{ni}}, \mathsf{V}_{\mathsf{ni}})$ is a *non-interactive SPARK for non-deterministic polynomial-time PRAM computation.* When $\mathrm{st} = crs$ for $\mathsf{G}_{\mathsf{ni}}(1^{\lambda})$, we say that the non-interactive SPARK is *publicly verifiable* and write $crs \leftarrow \mathsf{G}_{\mathsf{ni}}(1^{\lambda})$.

5 CONCURRENTLY UPDATABLE HASH FUNCTIONS

In this section, we define and construct a hash function that (1) allows concurrently updating arbitrary positions in the string underlying the digest, (2) has the property that different updates can be computed concurrently using multiple processors in a pipelined fashion (described in more detail below). This can be seen as a strengthening of locally updatable hash functions, with extra eficiency properties. We define our construction in the PRAM model.

For a security parameter $\lambda \in \mathbb{N}$, our hash function will be for strings D consisting of $n \leq 2^{\lambda}$ words of length λ. It will 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 5.1 (Partial String). For any string $s \in (\{0,1\}^{\lambda} \cup \{\perp\})^{*}$ of words, the *partial string* D representing s is defined 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.

Next, we define the hash functions used in this article. A *concurrently updatable hash function* is a tuple of algorithms (C.Gen, C.Hash, C.Open, C.Update, C.VerOpen, C.VerUpd) with the following syntax[9]:

[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 scheme where all algorithms may be randomized.

- $pp \leftarrow \mathsf{C.Gen}(1^\lambda, n)$: A PPT algorithm that on input the security parameter λ in unary and an integer n, outputs public parameters pp.
- $(ptr, digest) = \mathsf{C.Hash}(pp, D)$: A deterministic algorithm that on input public parameters pp and a partial string D, outputs a pointer ptr to a location in memory and a string $digest$.
- $(V, \pi) = \mathsf{C.Open}(pp, ptr, S)$: A read-only deterministic algorithm that on input public parameters pp, a pointer ptr, and an ordered set $S = (\ell_1, \ldots, \ell_p)$ of locations $\ell_i \in [n]$, outputs a tuple $V = (v_1, \ldots, v_p)$ of values $v_i \in \{0,1\}^\lambda \cup \{\bot\}$, and a proof π.
- $(digest, \tau) = \mathsf{C.Update}(pp, ptr, S, V)$: A deterministic algorithm that on input public parameters pp, a pointer ptr, an ordered set $S = (\ell_1, \ldots, \ell_p)$ of locations $\ell_i \in [n]$, and a tuple $V = (v_1, \ldots, v_p)$ of words $v_i \in \{0,1\}^\lambda$, outputs a digest $digest$ and a proof τ.
- $b = \mathsf{C.VerOpen}(pp, digest, S, V, \pi)$: A deterministic algorithm that on input public parameters pp, a digest $digest$, an ordered set $S = (\ell_1, \ldots, \ell_p)$ of locations $\ell_i \in [n]$, a tuple $V = (v_1, \ldots, v_p)$ of values $v_i \in \{0,1\}^\lambda \cup \{\bot\}$, and a proof π, outputs a bit b.
- $b = \mathsf{C.VerUpd}(pp, digest, S, V, digest', \tau)$: A deterministic algorithm that on input public parameters pp, a digest $digest$, an ordered set $S = (\ell_1, \ldots, \ell_p)$ of locations $\ell_i \in [n]$, a tuple $V = (v_1, \ldots, v_p)$ of words $v_i \in \{0,1\}^\lambda$, a digest $digest'$, and a proof τ, outputs a bit b.

We assume for simplicity that there are no duplicate locations specified by the set S in the above algorithms. We note that when S is a single location ℓ and V is a single word v, to simplify notation, we let C.Open, C.Update, C.VerOpen, and C.VerUpd take ℓ and v as input rather than the singleton ordered set (ℓ) and tuple (v). We require the following completeness, soundness, and eficiency properties.

At a high level, completeness says that opening or updating an honestly generated digest gives a valid proof, and that the string underlying the digest is correct. Moreover, this holds after any sequence of updates to the digest.

Definition 5.2 (Completeness). Let $\lambda, n \in \mathbb{N}$ with $n \le 2^\lambda$, pp be in the support of $\mathsf{C.Gen}(1^\lambda, n)$, $D = (n, I, A)$ be a partial string, and $m \ge 0$. For any ordered sets $S^{(i)} \subseteq [n]$ and tuples $V^{(i)} \in (\{0,1\}^\lambda)^{|S^{(i)}|}$ for $i \in [m]$, do the following:

(1) Compute $(ptr, digest^{(0)}) = \mathsf{C.Hash}(pp, D)$.
(2) For $i = 1, \ldots, m$, compute $(digest^{(i)}, \tau^{(i)}) = \mathsf{C.Update}(pp, ptr, S^{(i)}, V^{(i)})$.

Let D' be the partial string resulting from writing each word in $V^{(i)}$ to D at the corresponding location in $S^{(i)}$ for $i = 1, \ldots, m$. Then, the following hold for any $p \in \mathbb{N}$ and ordered set $S = (\ell_1, \ldots, \ell_p)$ of locations in $[n]$:

- Open Completeness. Let $(V, \pi) = \mathsf{C.Open}(pp, ptr, S)$ where $V = (v_1, \ldots, v_p)$. Then,

$$\mathsf{C.VerOpen}(pp, digest^{(m)}, S, V, \pi) = 1 \wedge D'_{\ell_i} = v_i \ \forall i \in [p].$$

- Update Completeness. For any tuple $V \in (\{0,1\}^\lambda)^p$, let $(digest, \tau) = \mathsf{C.Update}(pp, ptr, S, V)$. It holds that

$$\mathsf{C.VerUpd}(pp, digest^{(m)}, S, V, digest, \tau) = 1.$$

Next, we define soundness, which informally says that no PPT adversary can give a digest and a sequence of valid updates that update some position ℓ to a word v^{prev} and then open ℓ to a different value $v^{\mathsf{final}} \neq v^{\mathsf{prev}}$.

Definition 5.3 (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 for all with $n \leq 2^\lambda$,

$$\Pr\left[\begin{array}{l} \text{C.VerOpen}(pp, digest^{(0)}, S^{(0)}, V^{(0)}, \pi^{(0)}) = 1 \wedge \\ \forall i \in [m] : \text{C.VerUpd}(pp, digest^{(i-1)}, S^{(i)}, V^{(i)}, digest^{(i)}, \tau^{(i)}) = 1 \wedge \\ \text{C.VerOpen}(pp, digest^{(m)}, S, V, \pi) = 1 \wedge \\ \exists \ell \in S \cap S^{(0)} : v^{\text{prev}} \neq v^{\text{final}} \end{array}\right] \leq \text{negl}(\lambda),$$

the probability is over the choice of $pp \leftarrow \text{C.Gen}(1^\lambda, n)$ and $(m, \{(digest^{(i)}, S^{(i)}, V^{(i)}, \tau^{(i)})\}_{i \in [m]}, digest^{(0)}, S^{(0)}, V^{(0)}, \pi^{(0)}, S, V, \pi) \leftarrow \mathcal{A}_\lambda(pp)$, and v^{prev} and v^{final} are defined as follows:

- v^{prev} is the value in $V^{(i)}$ at the index of ℓ in $S^{(i)}$, where $i \in \{0, \ldots, m\}$ is the largest index with $\ell \in S^{(i)}$.
- v^{final} is the value in V at the index of ℓ in S.

Last, we require the following eficiency 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 5.4 (Parallel Eficiency). Let $\beta \colon \mathbb{N} \to \mathbb{N}$. We say that a concurrently updatable hash function satisfies *β-parallel eficiency* if the following hold for all $\lambda, n \in \mathbb{N}$ with $n \leq 2^\lambda$, pp in the support of $\text{C.Gen}(1^\lambda, n)$, and ordered sets $S \subseteq [n]$:

- The algorithms C.Open, C.Update, C.VerOpen, and C.VerUpd when given public parameters pp and locations S can each be computed with $|S| \cdot \beta(\lambda)$ work, which can be decoupled into depth $\beta(\lambda)$ with $|S| \cdot \beta(\lambda)$ processors.
- Computing $\text{C.Hash}(pp, D)$ for any partial string $D = (n, I, A)$ can be done with $|I| \cdot \beta(\lambda)$ work, which can be decoupled into depth $\beta(\lambda)$ with $|I| \cdot \beta(\lambda)$ processors.
- For any pointer ptr, and tuple $V \in (\{0,1\}^\lambda)^{|S|}$, def ine$(V^0, \pi, digest, \tau)$ as follows:
  – $(V^0, \pi) = \text{C.Open}(pp, ptr, S)$
  – $(digest, \tau) = \text{C.Update}(pp, ptr, S, V)$
  There exists an algorithm $\text{OpenUpdate}(pp, ptr, S, V)$ that outputs $(V^0, \pi, digest, \tau)$, such that k sequential calls to OpenUpdate, each on at most $p_{\max}$ locations, can be computed with $p_{\max} \cdot \beta(\lambda)$ work, which can be decoupled into depth $(k-1) + \beta(\lambda)$ using at most $p_{\max} \cdot \beta(\lambda)$ processors.

When β is a polynomial, we say the scheme satisfies parallel eficiency.

Remark 7. We emphasize that the completeness and soundness properties we give for concurrently updatable hash functions must hold for any sequence of m "valid" updates. At a high level, these notions stipulate that an opening will always give the correct values (with a proof) and that no adversary can find an opening for a value you would not expect (based on the updates). Furthermore, we require C.VerUpd to ensure that an update to a set of locations does not affect any other locations.

We note that even when viewed as a hash function with local updates (i.e., updates to a single location rather than a set) our definition generalizes some previous notions. Specifically, this applies to standard notions of completeness and position binding for vector commitments [23], 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 Reference [39].

We note that it does not sufice to consider the properties to hold with respect to a single update (i.e., when $m = 1$). This is because our hash functions 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.

5.1 Hash Function Building Blocks

Before giving our concurrently updatable hash function construction, we provide some preliminary definitions and building blocks.

Binary trees. When we discuss complete binary trees with n leaves, we refer to each node having a level, where the leaves are level 0 and the root is level $\log n$. For a node at level i, its children are the two nodes adjacent to it at level $i - 1$, and its parent is the node adjacent to it at level $i + 1$.

Definition 5.5 (Ancestor Nodes). For a complete binary tree and a set of leaves S, we define the set ancestors(S) to be the set containing all nodes that are ancestors of any node in S, including S. For a single node ℓ, we simply write ancestors(ℓ) to denote the ancestors of the node ℓ.

Definition 5.6 (Dangling Nodes). Let T be a complete binary tree and S be a set of leaves in MT. The *dangling nodes* with respect to S, denoted dangling(S), is the set consisting of all siblings of nodes in ancestors(S) that themselves are not contained in ancestors(S). For a single leaf ℓ, we simply write dangling(ℓ) to denote the dangling nodes relative to $\{\ell\}$.

We remark that the notion of dangling nodes for a set S is a generalization of an authentication path for a single location ℓ. Specifically, just like an authentication path gives a proof for opening a single location in a Merkle tree, the values for nodes in dangling(S) can similarly be used to certify an opening for the locations in S. Next, we bound the size of a dangling set.

Claim 5.7. *Consider a complete binary tree with n leaves and let $S \subseteq [n]$. If $0 < |S| \leq p$, then $|\text{dangling}(S)| \leq p \log(n/p)$.*

Proof. A similar observation and proof were given in Reference [46]. We give the full proof with our notation here for completeness.

We prove the claim by induction on i where $n = 2^i$ for any $p \in [n]$. In the base case, when $i = 0$ so $n = 2^0 = 1$, $|\text{dangling}(S)| = 0 \leq p \log(n/p)$ for all $p \in [2^0] = \{1\}$ as required. We next show the claim for $n = 2^i$ for $i > 0$ assuming it for $n/2 = 2^{i-1}$. Let $S \subseteq [n]$ be a set of leaves for the complete binary tree with n leaves. Let $S_L = S \cap \{1, \ldots, n/2\}$ and $S_R = S \cap \{n/2 + 1, \ldots, n\}$, where we consider S_L to be a set of leaves in the sub-tree of height $i - 1$ rooted at the left child of the root, and similarly S_R to be a set of leaves in the sub-tree rooted at the right child of the root.

We first consider the case when $|S_L|, |S_R| > 0$. By the inductive hypothesis, there are at most $|S_L| \log(n/(2|S_L|))$ nodes in dangling(S_L) and similarly at most $|S_R| \log(n/(2|S_R|))$ nodes in dangling(S_R). This implies that

$$\begin{aligned} &|\text{dangling}(S_L)| + |\text{dangling}(S_R)| \\ &\leq |S_L| \log \frac{n}{2|S_L|} + |S_R| \log \frac{n}{2|S_R|} \\ &= (|S_L| + |S_R|) \log n - (|S_L| \log |S_L| + |S_R| \log |S_R|) - (|S_L| + |S_R|). \end{aligned}$$

Using the fact that $a \log a + b \log b \geq (a+b)(\log(a+b) - 1)$ for any $a, b > 0$,[10] this implies that

$$\begin{aligned} |\text{dangling}(S_L)| + |\text{dangling}(S_R)| &\leq p \log n - p(\log p - 1) - p \\ &= p \log(n/p). \end{aligned}$$

Furthermore, note that this covers all nodes in dangling(S) as the roots of both ancestors(S_L) and ancestors(S_R) (when viewed as sub-trees) are in ancestors(S) (since $|S_L|, |S_R| > 0$), and there are no other siblings that cross between the two sub-trees ancestors(S_L) and ancestors(S_R).

Now consider the case where either $|S_L| = 0$ or $|S_R| = 0$. Note that because we assume $p > 0$, it cannot be the case that both $|S_L|$ and $|S_R|$ are 0. Without loss of generality, we consider the case where $|S_R| = 0$. In this case, it must be that $|S_L| = p \leq n/2$. Then by the inductive hypothesis there are at most $p \log(n/(2p))$ nodes in dangling(S_L). Furthermore, dangling(S) consists of all nodes in dangling(S_L) plus the root of S_R. So,

$$\begin{aligned} |\text{dangling}(S)| &\leq 1 + p \log(n/(2p)) = p \log(n/p) + (1-p) \\ &\leq p \log(n/p), \end{aligned}$$

which holds given that $p \geq 1$.

Next, we give the following helpful claim, which follows from the definition of a dangling set, which will be helpful in our concurrently updatable hash function construction. Recall that a *proper* tree is one where every node has either zero children or two children.

Claim 5.8. *For any set S of leaves in a complete binary tree with n leaves,* ancestors$(S) \cup$ dangling(S) *is a proper sub-tree with leaves $S \cup$* dangling(S).

Proof. Note that if S is empty, the claim holds vacuously, then so henceforth, we assume S is non-empty. Let T be the sub-tree consisting of ancestors$(S) \cup$ dangling(S). Note that T is a tree since ancestors(S) is a tree, and every node in dangling(S) is a child of a node in ancestors(S). To show that T is proper and that its leaves are $S \cup$ dangling(S), we will show that every node in T is either in $S \cup$ dangling(S), in which case it is a leaf, or is in ancestors$(S) \setminus S$ and has both of its children in T, which sufices for the claim. Consider any node *node* in T. If *node* $\in$ dangling(S), then its children are not in T, since neither child is an ancestor of S by definition, and hence neither can be in dangling(S). It follows that *node* is a leaf. If *node* $\in S$, then it is a leaf in the complete binary tree and is in T, so is a leaf in T. If *node* $\in$ ancestors$(S) \setminus S$, then its children are in ancestors$(S) \cup$ dangling(S), and so are both in T.

Merkle trees. Let $h\colon \{0,1\}^{2\lambda} \to \{0,1\}^{\lambda}$ be a compressing hash function. A Merkle tree [43] 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 def ined 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 digest 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

[10]~~This follows~~ by an application of Jensen's inequality to the function $f(x) = 2x \log x$, which is convex on all $x > 0$. Specifically,

$$a \log a + b \log b = \frac{f(a)}{2} + \frac{f(b)}{2} \geq f \left(\frac{a+b}{2}\right) = (a+b) \log \left(\frac{a+b}{2}\right) = (a+b)(\log(a+b) - 1).$$

opening for $v \in \{0,1\}^\lambda$ with respect to a digest *digest* 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 *digest*. 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. In our construction, rather than using an authentication path, we will use the notion of a dangling set (5.6) that generalizes an authentication path for multiple leaves.

Assuming the underlying hash function h is collision-resistant, it is well known that a Merkle tree is binding 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 $v \in \{0,1\}^\lambda \cup \{\bot\}$. Therefore, we will use a Merkle tree for 2λ-bit values, so we can uniquely encode each element of $\{0,1\}^\lambda \cup \{\bot\}$ 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 eficiently check if a range of indices in the partial string $D = (n, I, A)$ are $\bot$. 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 that mirrors the Merkle tree and consists of a complete binary tree with n leaves. Each node has an associated bit that 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, then it guarantees that all indices corresponding to the leaves that are descendants of this node are $\bot$. 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.

5.2 Construction

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

We now give our full concurrently updatable hash function construction C = (C.Gen, C.Hash, C.Open, C.Update, C.VerOpen, C.VerUpd).

- $pp \leftarrow$ C.Gen$(1^\lambda, n)$: Sample $h \leftarrow \mathsf{H}_\lambda$ and output $pp = (h, n)$.
- $(ptr, digest) =$ C.Hash(pp, D):
  1. Parse $pp = (h, n)$. 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 *segtree*, and $\log n$ extra blocks of size 2λ at pointer *aux*.
     We assume that all memory is initialized to 0.
  2. Define dummy$(0) =$ block$(\bot)$. Let $h = pp$, and for $j = 1, \ldots, \log n$, compute dummy(j) $= h($dummy$(j-1)||$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-$\bot$ indices with values given in A. Run the update procedure defined below by C.Update(pp, ptr, I, A).
  4. Let *digest* be the value of the root in *ptr*, or dummy$(\log n)$ if it is uninitialized, and output $(ptr, digest)$.

- $(V, \pi) =$ C.Open(pp, ptr, S): Parse $pp = (h, n)$. Let $p = |S|$ and let $S = (\ell^{(1)}, \ldots, \ell^{(p)})$. Let *segtree* be the pointer to the segment tree in memory.
  1. Compute the set dangling(S).
  2. Let R be an initially empty set, which will store all read values.
  3. For each level $j = 0, \ldots, \log n - 1$, do the following:
     (a) In parallel for each node $\ell \in S \cup$ dangling(S) at level j:
       - Read ℓ in *ptr*, and let its value be u^{rd}_ℓ.
       - Read ℓ in *segtree*, and let its value be b^{rd}_ℓ.

     (b) For every $\ell \in S \cup$ dangling(S) at level j, if $b^{\text{rd}}_\ell = 0$, let $u^{\text{rd}}_\ell =$ dummy(j). Add $(\ell, u^{\text{rd}}_\ell)$ to R.

  To form the output, do the following:
  1. For each $i \in [p]$, let $v^{(i)} \in \{0,1\}^\lambda \cup \{\perp\}$ be the value such that $(\ell^{(i)}, \text{block}(v^{(i)})) \in R$.
  2. Let π be a list containing all (ℓ, u) in R such that $\ell \in$ dangling(S).
  3. Note that the above values exist in R since it contains an entry for each node in $S \cup$ dangling(S). Output (V, π) where $V = (v^{(1)}, \ldots, v^{(p)})$.
- $(digest, \tau) =$ C.Update(pp, ptr, S, V): Let $p = |S|$, $S = (\ell^{(1)}, \ldots, \ell^{(p)})$, and $V = (v^{(1)}, \ldots, v^{(p)})$. Parse $pp = (h, n)$. Let *segtree* be the pointer to the segment tree in memory.

  Preprocessing Steps.
  1. Compute the sets of nodes dangling(S) and ancestors(S).
  2. Let R, W be sets, initially empty, which will contain the read and written values (respectively).
  3. Add $(\ell^{(i)}, \text{block}(v^{(i)}))$ to W for all $i \in [p]$.

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

  Access Step. Do the following in parallel:
  - For every node $\ell \in$ ancestors(S) at level j, in parallel:
    - Let u be the value with $(\ell, u) \in W$, and write u to ℓ in *ptr*. Let u^{prev}_ℓ be the value overwritten.
    - Write 1 to ℓ in *segtree*, and let the value overwritten be b^{prev}_ℓ.
  - For every $\ell \in$ dangling(S) at level j, in parallel:
    - Read ℓ in *ptr*, and let its value be u^{rd}_ℓ.
    - Read ℓ in *segtree*, and let its value be b^{rd}_ℓ.

  Compute Steps.
  1. In parallel for every $\ell \in$ ancestors(S) at level j, if $b^{\text{prev}}_\ell = 0$, then set $u^{\text{prev}} =$ dummy(j). Add (ℓ, u^{prev}) to R.
  2. In parallel for every $\ell \in$ dangling(S) at level j, if $b^{\text{rd}}_\ell = 0$, then set $u^{\text{rd}} =$ dummy(j). Add (ℓ, u^{rd}) to R.
  3. In parallel for every node $\ell \in$ ancestors(S) at level $j + 1$, do the following:
     (a) For its left child and right child, let u_L and u_R, respectively, be the values given by W if they exist and by R otherwise. If neither, then abort and output $\perp$.
     (b) Compute u as the hash of $u_L || u_R$ using h, and add (ℓ, u) to W.

  Form Output.
  1. For each $i \in [p]$, let $v^{(i)}_{\text{prev}} \in \{0,1\}^\lambda \cup \{\perp\}$ be the value such that $(\ell^{(i)}, \text{block}(v^{(i)}_{\text{prev}})) \in R$.
  2. Let π be a list containing all (ℓ, u) in R such that $\ell \in$ dangling(S).
  3. If any of the above values cannot be found, then output $\perp$. Otherwise, output $(digest, \tau)$ where *digest* is the value of the root given by W and $\tau = (v^{(1)}_{\text{prev}}, \ldots, v^{(p)}_{\text{prev}}, \pi)$.
- $b =$ C.VerOpen$(pp, digest, S, V, \pi)$: Parse $pp = (h, n)$ and output 1 if and only if the following steps are successful:

(1) Check that $|S| = |V|$, each element of S is in $[n]$, each value in V is in $\{0,1\}^\lambda \cup \{\bot\}$, and each element of π is a pair $(\ell, u) \in [n] \times \{0,1\}^{2\lambda}$.
(2) Compute dangling(S) and check that the set of locations in π is equal to dangling(S).
(3) Let R be a set, initialized with all elements π and $(\ell^{(i)}, \text{block}(v^{(i)}))$, where $\ell^{(i)}$ is the ith location in S and $v^{(i)}$ is the ith value in V.
(4) For each level $j = 0, \ldots, \log n - 1$, do the following:
  (a) For each pair of sibling nodes ℓ_L, ℓ_R in $S \cup$ dangling(S) at level j, let ℓ be the location of their parent node.
  (b) Compute u as the hash of the values for ℓ_L and ℓ_R given by R using h.
  (c) Add (ℓ, u) to R.
(5) Check that the value in R corresponding to the root is equal to *digest*.

- b = C.VerUpd($pp, digest, S, V, digest', \tau$): Parse $pp = (h, n)$ and output 1 if and only if the following hold:
  (1) τ can be parsed as $V' || \pi$ where $|V'| = |S|$.
  (2) Each value of V is in $\{0,1\}^\lambda$.
  (3) C.VerOpen($pp, digest, S, V', \pi$) = 1.
  (4) C.VerOpen($pp, digest', S, V, \pi$) = 1.

Theorem 5.9. *Assuming the existence of collision-resistant hash function families, there exists a concurrently updatable hash function.*

We prove Theorem 5.9 in Section 5.3, where we show that the construction C satisfies completeness in Lemma 5.10, soundness in Lemma 5.14, and eficiency in Lemma 5.18.

5.3 Proofs

Lemma 5.10 (Completeness). *The construction* C *satisfies completeness.*

Proof. Fix any $\lambda, n \in \mathbb{N}$ with $n \leq 2^\lambda$ and pp in the support of C.Gen($1^\lambda, n$). To show the completeness properties, recall that the hash function algorithms keep track of a Merkle tree at *ptr* and a segment tree at *segtree* to keep track of which nodes are initialized. We start by defining a notion that captures when memory at (*ptr*, *segtree*) is consistent with a Merkle tree for a partial string D. Formally, we say that (*ptr*, *segtree*) is *consistent* with a partial string $D = (n, I, A)$ if the following hold:

(1) For every $i \in I$, leaf i has value 1 in *segtree*,
(2) For every node with value 1 in *segtree*, the values of its ancestors in *segtree* are set to 1, and
(3) For every node *node* with value 1 in *segtree*, its value in *ptr* is equal to the value of *node* in the Merkle tree for block(D_1)$||\ldots||$block(D_n) using the hash function given by pp.

We start by showing that doing an update preserves consistency.

Claim 5.11. *Suppose that* (*ptr*, *segtree*) *is consistent with a partial string D. For any ordered set $S = (\ell^{(1)}, \ldots, \ell^{(p)})$ of locations $\ell^{(i)} \in [n]$ and tuple $V = (v^{(1)}, \ldots, v^{(p)})$ of words $v^{(i)} \in \{0,1\}^\lambda$, let* (*ptr'*, *segtree'*) *be pointers to memory after computing* C.Update(pp, ptr, S, V). *Then,* (*ptr'*, *segtree'*) *is consistent with the partial string D', where $D'_{\ell^{(i)}} = v^{(i)}$ for all $i \in [p]$, and D' agrees with D at all other locations.*

Proof. When C.Update(pp, ptr, S, V) is computed, the only nodes updated in *ptr* and *segtree* are those in ancestors(S). In *segtree*, every node in ancestors(S) is set to 1. This immediately gives the first two properties of consistency. To show the third property, let *MT* be the Merkle tree for the string block(D_1)$||\ldots||$block(D_n) using the hash function given by pp. We need to show that every node with value 1 in $segtree^0$ has the same value in ptr^0 and MT. Since $(ptr, segtree)$ are consistent with D, and the only changes are to nodes in ancestors(S), it sufices to show that this holds for every node in ancestors(S). Throughout this proof, we will refer to iteration j of C.Update as the iteration that updates the jth level of the tree, for $j = 0, \dots, \log n$.

Consider any node $node \in$ ancestors(S). We show by induction on the level of $node$ that its value in ptr^0 is equal to its value in MT. For the base case, when $node$ is at level 0 (i.e., it is a leaf), it follows that $node = \ell^{(i)}$ for some index i. It is only updated at iteration 0, where it is set to block$(v^{(i)}) =$ block$(D_{\ell^{(i)}})$, which gives the base case.

Next, assume that every node at level j has the same value in ptr^0 and MT, and suppose $node$ is at level j+1. For convenience, let ℓ_L, ℓ_R be the locations for the left and right child of $node$, respectively. During the update, $node$ is only written to in the $(j + 1)$st iteration, where it is set to the hash of the concatenation of values corresponding to its children, found in sets R, W maintained by the algorithm. Let u_L, u_R be the values used for the left and right child, respectively. To show that the value for $node$ is indeed its value in MT, it therefore sufices to show that u_L, u_R are the values for ℓ_L, ℓ_R in MT. Without loss of generality, we show this for the value u_L used for ℓ_L.

To prove that u_L is indeed the value of ℓ_L in MT, we claim the following:

Subclaim 5.12. *If ℓ_L is initialized before the $(j + 1)$st iteration, then u_L is the value of ℓ_L in ptr^0. If it is not initialized, then u_L is set to* dummy(j).

We complete the proof assuming Subclaim 5.12 and then show that the subclaim holds. The only time that ℓ_L is accessed by C.Update is during the jth iteration. There are two cases to consider:

- Case 1: ℓ_L is in ancestors(S). In this case, it is initialized during iteration j, so it follows by Subclaim 5.12 that u_L is its value in ptr^0. Since it is at level j, then by the inductive hypothesis, this is equal to the value in MT.
- Case 2: ℓ_L is in dangling(S). In this case, it is not changed by C.Update. If it was already initialized before the update, then the inductive hypothesis applies as in the previous case. If not, then $u_L =$ dummy(j) by Subclaim 5.12. Moreover, since $(ptr, segtree)$ is consistent with D before the update, then the fact that ℓ_L is uninitialized in $segtree$ implies that $D_\ell = \perp$ for every leaf ℓ that is a descendant of ℓ_L. Therefore, the value of ℓ_L in MT is dummy(j), so u_L is indeed equal to the value of ℓ_L in MT.

Since $node$ is an ancestor of a leaf in S, these are the only two cases. Therefore, assuming Subclaim 5.12, the value u_L agrees with MT. To complete the proof, it remains to show Subclaim 5.12.

To prove that Subclaim 5.12 holds, recall that the algorithm C.Update first checks if ℓ_L is in the set W and then checks the set R. Both children are only accessed and modified in R, W in iteration j. Between the two children, at least one child must be in ancestors(S). In this case, in iteration j it is initialized and its final value in memory is added to W, which is the value used. If either child is not in ancestors(S), then it is in dangling(S) by definition. In this case, it follows that in iteration j it is added to R (and not W), where either its value in memory is used, or dummy(j) if it is not initialized. This completes the proof of Subclaim 5.12, which in turn gives the claim.

Next, we show that the memory after initially hashing a partial string is consistent with that partial string.

Claim 5.13. *Let $D_{\text{start}} = (n, I, A)$ be a partial string, and let $(ptr, segtree)$ be the pointers to the Merkle tree and segment tree in memory after running* C.Hash(pp, D_{start}). *Then, $(ptr, segtree)$ are consistent with D_{start}.*

Proof. Running C.Hash(pp, D_{start}) results in the same memory as running:

(1) $(ptr, digest) =$ C.Hash$(pp, D_\perp)$, where $D_\perp$ is the empty partial string.
(2) C.Update(pp, ptr, I, A), where we recall that I specifies the set of non-$\perp$ locations in D_{start} and A is the assignment to those locations.

After C.Hash$(pp, D_\perp)$, it is vacuously true that the resulting memory is consistent with $D_\perp$, since there are no non-$\perp$ words in $D_\perp$. Therefore, by Claim 5.11, the memory after C.Update(pp, ptr, I, A) is consistent with D_{start}.

We are now ready to prove completeness. Fix any partial string $D_{\mathsf{start}} = (n, I, A)$, integer $m \geq 0$, ordered sets $S^{(i)} \subseteq [n]$ and tuples $V^{(i)} \in (\{0,1\}^\lambda)^{|S^{(i)}|}$ for $i \in [m]$. Compute

(1) $(ptr, digest_0) =$ C.Hash(pp, D_{start}).
(2) For $i = 1, \ldots, m$, compute $(digest^{(i)}, \tau^{(i)}) =$ C.Update$(pp, ptr, S^{(i)}, V^{(i)})$.

Let D be the partial string formed by writing each word in $V^{(i)}$ to D_{start} at the corresponding location in $S^{(i)}$ for $i = 1, \ldots, m$, and let MT be the Merkle tree for D. We start by noting that $(ptr, segtree)$ is consistent with D after all m updates. This following by induction on m: For the base case, when $m = 0$, this follows from Claim 5.13. For the inductive step, assuming this holds for m updates, then Claim 5.11 implies that it holds after the $(m + 1)$st update. Using the fact that $(ptr, segtree)$ is consistent with D, we proceed to show open completeness and update completeness.

Open Completeness. Fix any $p \geq 0$ and ordered set $S = (\ell^{(1)}, \ldots, \ell^{(p)})$. Compute

$$(V, \pi) = \text{C.Open}(pp, ptr, S),$$

and parse $V = (v^{(1)}, \ldots, v^{(p)})$. To show open completeness, we first make the following assertions about the values in MT:

- For all $\ell^{(i)} \in S$, the value at leaf $\ell^{(i)}$ in MT is equal to block$(v^{(i)})$.
- For all $\ell \in$ dangling(S), the value in MT is equal to the value u such that $(\ell, u) \in \pi$.
- The value of the root in MT is equal to $digest^{(m)}$.

These assertions hold by consistency of $(ptr, segtree)$ with D. Specifically, each of these values is either given by the node's value in ptr, or is set to dummy(j) if uninitialized and at level j. Each initialized node agrees with MT by consistency, and for any uninitialized node, consistency implies that all of the leaves that are descendants of that node must be uninitialized and thus have the value $\perp$. Therefore, dummy(j) is the value at the corresponding location in MT. Therefore, in either case, the value given above is equal to the corresponding value in MT.

Using this, we proceed to show open completeness. We need to show (1) that D agrees with V at the locations in S, and (2) that C.VerOpen$(pp, digest^{(m)}, S, V, \pi)$ accepts. (1) follows immediately from our observation that V correspond to the values at S in MT.

For (2), recall that C.VerOpen does syntactic checks on V and π and then iteratively hashes values down the tree to obtain a digest $digest^?$. It accepts if all syntactic checks pass and $digest^? = digest^{(m)}$. By construction, V consists of a value $v^{(i)}$ for $i \in [p]$, and the proof π contains a pair (ℓ, u) for each $\ell \in$ dangling(S), so the syntactic checks pass.

To show that $digest^? = digest^{(m)}$, we have that $digest^?$ is derived from the values in V and π, which constitute a set of values for $S \cup$ dangling(S). Specifically, $digest^?$ is obtained by iteratively hashing each pair of siblings at each level until reaching the root. By Claim 5.8, there is a sub-tree containing ancestors(S) whose leaves are all in $S \cup$ dangling(S). It follows that having values for every node in $S \cup$ dangling(S) sufices to obtain a value for the root. Moreover, since the values given for $S \cup \text{dangling}(S)$ are equal to the corresponding values in MT, then $digest^?$ is equal to the root of MT. Since $digest^{(m)}$ is also equal to the root of MT, then $digest^? = digest^{(m)}$, which concludes the proof of open completeness.

Update Completeness. Fix any $p \geq 0$, ordered set $S = (\ell^{(1)}, \ldots, \ell^{(p)})$, and tuple $V = (v^{(1)}, \ldots, v^{(p)})$. Compute

$$(digest, \tau) = \text{C.Update}(pp, ptr, S, V).$$

To show update completeness, we need to show that $\text{C.VerUpd}(pp, digest^{(m)}, S, V, digest, \tau) = 1$, which consists of syntactic checks and two inner verifications. The syntactic checks pass by definition of C.Update, which in particular state that τ can be parsed as $V^0||\pi$ where V^0 is a tuple of p values. For the verifications, we need to show that both of the following hold:

(A) $\text{C.VerOpen}(pp, digest^{(m)}, S, V^0, \pi) = 1$
(B) $\text{C.VerOpen}(pp, digest, S, V, \pi) = 1$

For Equation (A), we claim that (V^0, π) would be the output of $\text{C.Open}(pp, ptr, S)$, had it been run *before* the final update. Specifically, for each $i \in S$, V^0 consists of a value $v^{(i)}_{\text{prev}}$ with $\text{block}(v^{(i)}_{\text{prev}})$ equal to the value in memory at each leaf in S before the update, or $\perp$ if the leaf is uninitialized, just as what would be output by C.Open. For π, it consists of the values read for each node in $\text{dangling}(S)$, or the dummy values if uninitialized. Since C.Update never writes to the nodes in $\text{dangling}(S)$, then these values are exactly what would be returned by C.Open. Therefore, Equation (A) holds by open completeness.

For Equation (B), we claim that (V, π) would be the output of running $\text{C.Open}(pp, ptr, S)$ *after* this final update. To see this, we observe that V consists of a value $v^{(i)}$ for each $\ell^{(i)} \in S$ where $\text{block}(v^{(i)})$ is equal to its value in ptr after the update. Moreover, each of these nodes is initialized, and so these are the values that would be returned by C.Open. For π, the same logic as above holds (namely, that the nodes in $\text{dangling}(S)$ are not changed by C.Update, and so are determined exactly as by C.Open). Therefore, Equation (B) accepts by open completeness, concluding the proof.

Lemma 5.14 (Soundness). *The construction* C *satisfies soundness.*

Proof. Suppose for contradiction there exists a non-uniform PPT adversary $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ and a polynomial q such that for infinitely many $\lambda \in \mathbb{N}$, there exists an integer $n \leq 2^\lambda$ such that

$$\Pr\left[\begin{array}{l} \text{C.VerOpen}(pp, digest^{(0)}, S^{(0)}, V^{(0)}, \pi^{(0)}) = 1 \wedge \\ \forall i \in [m] : \text{C.VerUpd}(pp, digest^{(i-1)}, S^{(i)}, V^{(i)}, digest^{(i)}, \tau^{(i)}) = 1 \wedge \\ \text{C.VerOpen}(pp, digest^{(m)}, S, V, \pi) = 1 \wedge \\ \exists \ell \in S \cap S^{(0)} : v^{\text{prev}} \neq v^{\text{final}} \end{array}\right] \geq \frac{1}{q(\lambda)}, \tag{5.1}$$

the probability is over $pp \leftarrow \text{C.Gen}(1^\lambda, n)$ and $(m, \{(digest^{(i)}, S^{(i)}, V^{(i)}, \tau^{(i)})\}_{i \in [m]}, digest^{(0)}, S^{(0)}, V^{(0)}, \pi^{(0)}, S, V, \pi) \leftarrow \mathsf{A}_\lambda(pp)$ and v^{prev} and v^{final} are defined as follows:

- v^{prev} is the value in $V^{(j)}$ at the position of ℓ in $S^{(j)}$, where $j \in \{0, \ldots, m\}$ is the largest index with $\ell \in S^{(j)}$.
- v^{final} is the words in V at the index of ℓ in S.

We show that whenever A succeeds, we can construct authentication paths certifying that ℓ can be opened to two different values in $digest^{(m)}$, which breaks the binding of standard Merkle trees assuming collision resistance.

The outline of the proof is as follows: First, in Claim 5.15, we will show that given a valid opening for many locations, we can eficiently construct a valid with respect to each individual location, which in fact is just a single Merkle tree authentication path. This claim actually sufices for the case of no updates, i.e., $m = 0$. To deal with $m > 0$, we show in Claim 5.16 how, given an opening under for ℓ under $digest^{(i)}$ and a valid update proof to $digest^{(i+1)}$, we can construct an opening for ℓ under $digest^{(i+1)}$ (or otherwise break collision resistance). At a high level, applying Claim 5.15 and then Claim 5.15 m times yields two Merkle tree authentication paths for $v^{\mathsf{prev}} \neq v^{\mathsf{final}}$ with respect to $digest^{(m)}$, which contradicts collision resistance of H as required.

We next formally state these general claims, then prove the lemma assuming they hold, and finally prove each of the claims to complete the proof of the lemma.

Claim 5.15. *For any $\lambda, n \leq 2^{\lambda}, p \in \mathbb{N}$, pp in the support of* $\mathsf{C.Gen}(1^{\lambda}, n)$, *ordered set* $S = (\ell^{(1)}, \ldots, \ell^{(p)})$, *tuple* $V = (v^{(1)}, \ldots, v^{(p)})$, *digest digest, and proof* π, *if*

$$\mathsf{C.VerOpen}(pp, digest, S, V, \pi) = 1,$$

then there exist proofs $\pi^{(1)}, \ldots, \pi^{(p)}$ *such that*

$$\mathsf{C.VerOpen}(pp, digest, \ell^{(i)}, v^{(i)}, \pi^{(i)}) = 1$$

for all $i \in [p]$. *Moreover, they can be computed from* (S, V, π) *in polynomial time.*

Claim 5.16. *There exists a polynomial-time algorithm* A' *that on input* $(pp, digest, \ell, v, \pi, digest', S, V, \tau)$, *if*

(1) $\mathsf{C.VerOpen}(pp, digest, \ell, v, \pi) = 1$ *and*
(2) $\mathsf{C.VerUpd}(pp, digest, S, V, digest', \tau) = 1$,

then A' *either outputs a collision in* H *under h, where h is given by pp, or outputs a proof* $\pi^{\star}$ *such that*

$$\mathsf{C.VerOpen}(pp, digest', \ell, v^{\star}, \pi^{\star}) = 1,$$

where $v^{\star} = v$ *if* $\ell \notin S$ *and otherwise* $v^{\star}$ *is the value in V at the index of* ℓ *in S.*

Proving the lemma assuming the above claims. We next prove the lemma assuming that Claims 5.15 and 5.16 hold. We condition on the event that A succeeds, which occurs with probability at least $1/p(\lambda)$.

First for the case of $m = 0$, we apply Claim 5.15 for $(S^{(0)}, V^{(0)}, \pi^{(0)})$ and $\ell \in S^{(0)}$ to eficiently compute a proof $\pi_{\ell}^{(0)}$ such that $\mathsf{C.VerOpen}(pp, digest^{(0)}, \ell, v_{\ell}^{(0)}, \pi_{\ell}^{(0)}) = 1$ where $v_{\ell}^{(0)}$ is the value in $V^{(0)}$ corresponding to location $\ell \in S^{(0)}$. Note that, since $m = 0$, then $v_{\ell}^{(0)} = v^{\mathsf{prev}}$ by definition. Next, we apply the claim for (S, V, π) and $\ell \in S$ to eficiently compute a proof $\pi_{\ell}^{\mathsf{final}}$ such that $\mathsf{C.VerOpen}(pp, digest^{(0)}, \ell, v^{\mathsf{final}}, \pi_{\ell}^{\mathsf{final}}) = 1$. By definition of $\mathsf{C.VerOpen}$, $\pi_{\ell}^{(0)}$ and $\pi_{\ell}^{\mathsf{final}}$ both give valid Merkle tree authentication paths with respect to the same location but different values $v^{\mathsf{prev}} \neq v^{\mathsf{final}}$. This contradicts collision resistance of H as this event occurs with probability $1/p(\lambda)$ by assumption.

Next, we consider the case when $m > 0$. Again, we start by applying Claim 5.15 for $(S^{(0)}, V^{(0)}, \pi^{(0)})$ and $\ell \in S^{(0)}$ to eficiently compute $\pi^{(0)}$ such that $\mathsf{C.VerOpen}(pp, digest^{(0)}, \ell, v^{(0)}, \pi^{(0)}) = 1$ where $v_{\ell}^{(0)}$ is the value for ℓ in $V^{(0)}$. Now we apply Claim 5.16 for $i = 1, \ldots, m$ to either find a collision or construct a proof $\pi^{(i)}$ for the value $v_{\ell}^{(i)}$ specified by the first i updates. Specifically, for the first case of $i = 1$, note that $(pp, digest^{(i-1)}, \ell, v_{\ell}^{(i-1)}, \pi_{\ell}^{(i-1)}, digest^{(i)}, S^{(i)}, V^{(i)}, \tau^{(i)})$ satisfy the conditions for Claim 5.16. As a result, we either find a collision or compute a proof $\pi_{\ell}^{(i)}$ for the value $v_{\ell}^{(i)}$ with respect to $digest^{(i)}$. Assuming we do not find a collision, this implies that the conditions for the claim also hold for general $i > 1$ as well. As such, after applying the claim at most m times, we will either find a collision or have computed a proof $\pi_{\ell}^{(m)}$ such that

C.VerOpen(pp, $digest^{(m)}, \ell, \nu_\ell^{(m)}, \pi_\ell^{(m)}$) = 1. Note that $\nu_\ell^{(m)} = \nu^{\text{prev}}$ by definition. Finally, we apply Claim 5.15 for (S, V, π) and $\ell \in S$ to eficiently compute eficiently compute a proof π_ℓ^{final} such that C.VerOpen(pp, $digest^{(m)}, \ell, \nu^{\text{final}}, \pi_\ell^{\text{final}}$) = 1. Again, by definition of C.VerOpen, $\pi_\ell^{(m)}$ and π_ℓ^{final} both give valid authentication paths for ℓ but for different values $\nu^{\text{prev}} \neq \nu^{\text{final}}$. Thus, in the case where applying Claim 5.16 does not directly find a collision with respect to H, we still find a collision by the binding property of Merkle trees. As this event occurs with probability $1/p(\lambda)$ by assumption, this contradicts the collision resistance of H.

Proving the claims. We now continue to prove Claims 5.15 and 5.16. Towards this, we start by defining a helpful criteria for when C.VerOpen accepts. This requires defining an algorithm extend and the notion of an induced value. To define these, fix any $\lambda, n, p \in \mathbb{N}$ with $p \leq n \leq 2^\lambda$, pp in the support of C.Gen$(1^\lambda, n)$, ordered set $S = (\ell^{(1)}, \ldots, \ell^{(p)})$, tuple $V = (\nu^{(1)}, \ldots, \nu^{(p)})$, and list π of values for nodes in dangling(S).

Define extend(pp, S, V, π) to do the following: Parse $pp = (h, n)$ and let T be the proper sub-tree of the complete binary tree given by Claim 5.8 whose leaves are $S \cup$ dangling(S). Assign values to the nodes in T as follows:

- For each leaf $\ell^{(i)}$ in S, let its value be given by block$(\nu^{(i)})$.
- For each node in dangling(S), let its value be given by π.
- For the remaining nodes, iteratively hash each pair of siblings using h at each level to assign a value to their parent, until reaching the root.

Let MT be the resulting (proper) Merkle tree on T, and define extend$(pp, S, V, \pi) = MT$.

Using this algorithm, we define an induced value as follows: For any node ℓ and value u, we say that (ℓ, u) is *induced* by (S, V, π) if the value of ℓ in MT is u, where $MT =$ extend(S, V, π). Note that this implies that u is the value of ℓ in any Merkle tree that agrees with the above values at $S \cup$ dangling(S). Our main observation for this proof is that when S, V, π have the correct syntax, the following subclaim holds:

Subclaim 5.17. *C.VerOpen(pp, $digest$, S, V, π) accepts if and only if digest is the value for the root induced by (S, V, π).*

This follows immediately from the definition of C.VerOpen. Specifically, C.VerOpen(pp, $digest$, S, V, π) implicitly runs extend(pp, S, V, π), compares the value of the resulting root to $digest$, and accepts when they are equal. Using Subclaim 5.17, we are now ready to prove the two claims above.

Proof of Claim 5.15. Fix λ, pp, S, V, $digest$, and π as in the statement of the claim. Let $MT =$ extend(pp, S, V, π). For each $i \in [p]$, let $\pi^{(i)}$ contain all pairs (ℓ, u) such that $\ell \in$ dangling$(\ell^{(i)})$ and u is the value of $\ell^{(i)}$ in MT. Note that these values exist as dangling$(\ell^{(i)}) \subseteq$ ancestors$(S) \cup$ dangling(S), and MT contains the latter nodes.

For each $i \in [p]$, we show first that $\pi^{(i)}$ is eficiently computable, and then we show that it gives a valid opening proof. For eficiency, note that extend(pp, S, V, π) runs in time poly$(\lambda, p, \log n)$, since it requires computing at most $|S \cup \text{dangling}(S)|$ hashes, each taking time polynomial in λ, and $|S \cup \text{dangling}(S)| \leq$ poly$(p, \log n)$ by Claim 5.7. Moreover, the input to extend has length polynomial in λ, p, and $\log n$, so it follows that $\pi^{(i)}$ can be computed in polynomial time based on S, V, π.

Next, we show that C.VerOpen(pp, $digest$, $\ell^{(i)}, \nu^{(i)}, \pi^{(i)}$) = 1. By Subclaim 5.17, this accepts whenever $digest$ is the value for the root induced by $(\ell^{(i)}, \nu^{(i)}, \pi^{(i)})$. Let $MT' =$ extend$(pp, \ell^{(i)}, \nu^{(i)}, \pi^{(i)})$. We want to show that $digest$ is the value of the root in MT'. Note that the values of $\ell^{(i)}$ and of dangling$(\ell^{(i)})$ agree between MT and MT' by definition. It follows that the values for each ancestor of $`^{(i)}$ agree between the two Merkle trees. Finally, we note that, since C.VerOpen$(pp, digest, S, V, \pi)$ accepts, then *digest* is the value of the root of MT, and hence is the value of the root of MT^0, which completes the proof.

Proof of Claim 5.16. Since C.VerUpd$(pp, digest, S, V, digest^0, \tau) = 1$, then τ can be parsed as $V_{\text{prev}}||\pi^0$ such that C.VerOpen$(pp, digest, S, V_{\text{prev}}, \pi^0) = 1$ and C.VerOpen$(pp, digest^0, S, V, \pi^0) = 1$. In the case that $` □ S$, then by Claim 5.15, $\mathsf{A}^0$ can use $(S, V, \pi^0)$ to compute and output a proof $\pi^?$ in polynomial time such that C.VerOpen$(pp, digest^0, `, v^?, \pi^?)$ accepts, where $v^?$ is the value of $`$ given by $V$. As a result, we focus on the case that $` < S$, and thus $v^? = v$.

Consider running the verifications C.VerOpen$(pp, digest, `, v, \pi)$, C.VerOpen$(pp, digest, S, V_{\text{prev}}, \pi^0)$, and C.VerOpen$(pp, digest^{C}, S, V, \pi^0)$. They all accept by assumption, and from the inputs to each we can define a Merkle tree with all the induced values. Specifically, let $MT$ = extend$(pp, `, v, \pi)$, let MT^{prev} = extend$(pp, S, V_{\text{prev}}, \pi^0)$, and let MT^{final} = extend(pp, S, V, π^0). By Subclaim 5.17, the root of MT and MT^{prev} is *digest*, and the root of MT^{final} is $digest^0$. Note that MT contains all nodes in ancestors$(`)$ □ dangling$(`)$, and both MT^{prev} and MT^{final} contain ancestors(S) □ dangling(S).

To construct a proof $\pi^?$ corresponding to opening location $`$ to value $v^?$ in $digest^0$, we need to construct values for dangling$(`)$, which are simply the nodes in the authentication path for $`$. Before defining $\pi^?$, we introduce some notation. For $j$ □ $\{0, \ldots, \log n - 1\}$, let $node_j$ be the ancestor of $`$ at level j and let sib_j be its sibling. Also, let i □ $[\log n]$ be the level in a binary tree containing the closest common ancestor of leaf $`$ and any leaf in S.

Next, define $\pi^?$ to contain all pairs (sib_j, u_j) for j □ $\{0, \ldots, \log n - 1\}$ where u_j is defined as follows:

- If $j < i - 1$, then u_j is the value of sib_j in MT (or □ if it does not exist).
- If $j \geq i - 1$, then u_j is the value of sib_j in MT^{final} (or □ if it does not exist).

We claim that either C.VerOpen$(digest^0, `, v^?, \pi^?) = 1$, in which case A^0 outputs $\pi^?$, or we can find a collision in the hash function. Recall that C.VerOpen can be split into syntactic checks, and checking the value of $digest^0$. We first show that the syntactic checks done by C.VerOpen pass, and then we show that either A^0 outputs a collision, or the rest of the verification succeeds.

For the syntactic checks, it follows that the inputs to C.VerOpen are formatted correctly, so we only need to show that $\pi^?$ contains a value for all nodes in dangling$(`) = (sib_1, \ldots, sib_{\log n-1})$. To show this, we have the following:

- For $j < i - 1$, sib_j □ dangling$(`)$ by definition and so is successfully found in MT.
- For $j = i - 1$, we note that $node_i$ is the closest common ancestor of $`$ and $S$, and is not a leaf, since $` < S$. Therefore, the children of $node_i$, namely, sib_{i-1} or $node_{i-1}$, must be in ancestors(S) □ dangling(S). This implies that sib_j is found successfully in MT^{final}.
  We note that this also implies that $node_{i-1}$ is in dangling(S), since it cannot be in ancestors(S) by definition of i, which is will be helpful later on in the proof.
- For $j > i - 1$, we have that $node_j$ □ ancestors$(node_i)$ □ ancestors(S), and so its sibling sib_j □ ancestors(S) □ dangling(S).

This shows that $\pi^?$ contains a value for every node in dangling$(`)$, so the syntactic checks done by verification pass.

Next, C.VerOpen$(digest^0, `, v^?, \pi^?)$ checks $digest^0$ by computing the root induced by $(`, v^?, \pi^?)$. Along the way, it computes a value for each node in ancestors$(`^{(i)})$. Let $c_1, \ldots, c_{\log n}$ be these values. We will show that either $c_{\log n} = digest^{C}$ and so verification accepts, or we can find a collision. Towards this, we have the following observations:

(1) c_{i-1} is the value of $node_{i-1}$ in MT.
This holds, since c_{i-1} is computed based on leaf values for ` and for $sib_0, \ldots, sib_{i-2}$ from MT, and so it agrees with MT.

(2) Either $node_{i-1}$ has the same value in MT and MT^{prev}, or we can find a collision.
Both Merkle trees MT and MT^{prev} have *digest* as the root. They also both contain $node_{i-1}$, since it is in both ancestors(`) by definition and in dangling(S) as shown above. This implies that they also contain the nodes in its authentication path. If the values for $node_{i-1}$ between the two trees are not the same, then this would give two different openings for $node_{i-1}$ relative to *digest*, which can be used to find a collision.

(3) $node_{i-1}$ has the same value in MT^{prev} and MT^{final}.
MT^{prev} is induced by $(S, V_{\text{prev}}, \pi^0)$, while MT^{final} is induced by (S, V, π^0). Therefore, these trees agree at all nodes in π^0, which consists of all nodes in dangling(S), and in particular contains $node_{i-1}$ as shown above. Therefore, MT^{prev} and MT^{final} have the same value for $node_{i-1}$.

(4) $(c_i, \ldots, c_{\log n})$ are the values for $node_i, \ldots, node_{\log n}$, respectively, in MT^{final}.
By combining observation 1, 2, and 3, we have that c_{i-1} is the value of $node_{i-1}$ in MT^{final}. Moreover, the values for $sib_{i-1}, \ldots, sib_{\log n-1}$ in $\pi^?$ are defined to be the values from MT^{final}. For $j = i, \ldots, \log n$, the value c_j is computed as the hash of these values for sib_{j-1} and $node_{j-1}$, so c_j is the value of $node_j$ in MT^{final}.

Observation 4 implies that $c_{\log n} = digest^0$, so C.VerOpen($pp, digest^0$, `, $v^?, \pi^?$) = 1, as required.

This completes the proof of Lemma 5.14.

Lemma 5.18 (Parallel Efficiency). *There exists a polynomial β such that the construction* C *satisfies β-parallel eficiency.*

Proof. We show the three required eficiency properties in the following claims. The lemma then follows by letting the polynomial β be any polynomial larger than q_1, q_2, and q_3 given in the claims.

For the following claims, let $t_{\mathsf{H}}(\lambda)$ denote the time it takes to hash each pair of 2λ-bit words, and note that $t_{\mathsf{H}}(\lambda)$ □ poly(λ). It will also be helpful to note that for any set S of p locations, ancestors(S) □ dangling(S) contains at most $p \log n$ nodes by definition.

Claim 5.19. *There exists a polynomial q_1 such that for any λ, n □ N with $n \leq 2^\lambda$ and pp in the support of* C.Gen($1^\lambda, n$)*, the algorithms* C.Open, C.Update, C.VerOpen, *and* C.VerUpd, *when given a set S of p locations and public parameters pp, can each be computed in with work $p \cdot q_1(\lambda)$, or with depth $q_1(\lambda)$ using $p \cdot q_1(\lambda)$ processors.*

Proof. We analyze C.Update, and we observe that the analyses for C.Open and C.VerOpen follow similarly as the algorithms have the same overall structure. Furthermore, C.VerUpd simply calls C.VerOpen twice. Thus, it suffices to argue the claim for C.Update.

We note that C.Update can be split into (1) preprocessing, (2) access and compute steps at each level in the tree, and (3) forming the output. Before analyzing the complexity of each of these, we discuss how to implement each of the relevant sets to achieve eficiency. The sets S, dangling(S), ancestors(S), R, and W each contain at most $p \cdot \log n$ □ $p \cdot$ poly(λ) nodes, and R,W additionally contain 2λ-bit values for each node. We would like each set to support concurrent reads and writes to distinct locations. This is done by allocating $2n \cdot$ poly(λ) bits in memory for each set (initialized to zeroes) and using an indicator bit to say if an element is in the set or not followed by its value (if any).

This can be done as there are $2n$ nodes in the tree, and each location can be encoded with $\log(2n)$ bits (and so with the above implementation, there are $\text{poly}(\lambda)$ bits in memory for each node). Specifically, the root is encoded as 0, and for each node with index i, its left and right children are encoded as $2i + 1$ and $2i + 2$, respectively. The exact encoding is not important for our application, only that each location requires $\log(2n)$ bits and that it gives a way to find a node's parent or child in time $\text{poly}(\lambda)$. Note that with this encoding and at most $p \cdot \text{poly}(\lambda)$ processors for each of the above sets, every location in each set can be accessed concurrently.

Next, we analyze the running time of (1), (2), and (3). For (1), the preprocessing steps require computing the relevant sets, which can be done in depth $\text{poly}(\lambda)$ using p processors with the implementation described above. Specifically, computing R and W is straightforward, where for W, we assume that each $\text{block}(v^{(i)})$ can be encoded (and decoded) in $\text{poly}(\lambda)$ time. For $\text{ancestors}(S)$, we can use p processors as follows: Each of the p processors can start at the leaf nodes (where each processor know its starting leaf index). Subsequently, they can move down the tree and update the sets. To make sure only one process is accessing a single location at a time, after each processor adds node at level i of the tree, it can check if that node's sibling was also added to $\text{ancestors}(S)$. If so, then only the processor accessing the sibling with the larger index can move on to the next level. Once a node stops (because its corresponds to the smaller of the two nodes), it can stop checking nodes further down the tree. Thus, at most two processors might be trying to access a node at each step, and each processor can eficiently check if it should continue. After determining $\text{ancestors}(S)$, the set $\text{dangling}(S)$ can be computed in depth $\text{poly}(\lambda)$ with p processors, where each processor is initially assigned to a leaf node in S, and adds that node's sibling to $\text{dangling}(S)$ whenever the sibling is not given by $\text{ancestors}(S)$. Each processor can stop making updates exactly as above, so each memory location is only accessed by a single process.

For (2), we would like each access step to take a single time slot, as specified by the algorithm. To do this, at the end of the pre-processing steps, we can compute the locations for each leaf in $S \cup \text{dangling}(S)$, which only adds an additional $\text{poly}(\lambda)$ depth using $p \log n$ processors, and then spawn $p \log n$ processors to access these locations in Merkle tree in the subsequent access step. Then, during the compute steps, using depth $\text{poly}(\lambda)$ and at most p processors, the locations for the next access step can be computed as above. Continuing in this fashion ensures that each access step is indeed a single step, with at most p processors. The compute steps additionally require updating R and W, as well as computing a hash per each of the p processors. This takes depth $\text{poly}(\lambda)$ using $p \cdot \text{poly}(\lambda)$ processors, where $\text{poly}(\lambda)$ extra processors are possibly needed to compute the hash eficiently. These access and compute steps are repeated $\log n \leq \lambda$ times for each level in the tree.

For (3), forming the output requires reading R with at most $\text{poly}(\lambda)$ work per element in the set, which can be distributed as above. Obtaining the value of *digest* from W requires an additional $O(\lambda)$ depth.

Thus, it holds that there is a polynomial q_1 such that C.Update, C.Open, C.VerOpen, and C.VerUpd can be computed with work $p \cdot q_1(\lambda)$, or with depth $q_1(\lambda)$ using at most $p \cdot q_1(\lambda)$ processors.

Claim 5.20. *There exists a polynomial q_2 such that for any $\lambda, n \in \mathbb{N}$ with $n \leq 2^\lambda$, pp in the support of* $\text{C.Gen}(1^\lambda, n)$, *and partial string $D = (n, I, A)$ computing* $\text{C.Hash}(pp, D)$ *can be done in work $|I| \cdot q_2(\lambda)$, or with depth $q_2(\lambda)$ with $|I| \cdot q_2(\lambda)$ processors.*

Proof. Recall that computing $\text{C.Hash}(pp, D)$ consists of allocating memory initialized to 0 (which we assume is free), computing $\log n$ hashes to compute dummy values, and running $\text{C.Update}(pp, ptr, I, A)$. As shown in the previous claim, running C.Update takes either work $|I| \cdot q_1(\lambda)$, or depth $q_1(\lambda)$ using $|I| \cdot q_1(\lambda)$ processors, and computing $\log n \leq \lambda$ hashes requires

$t_H(\lambda) \cdot \log n \in \mathrm{poly}(\lambda)$ work. Thus, we let q_2 be a polynomial such that $q_2(\lambda)$ is at least as large as $q_1(\lambda) + t_H(\lambda) \cdot \lambda$ to cover the depth requirement.

Claim 5.21. *There exists a polynomial q_3 and an algorithm* OpenUpdate *such that the following holds: For any $\lambda, p, n \in \mathbb{N}$ with $n \leq 2^\lambda$, pp in the support of* C.Gen$(1^\lambda, n)$, *pointer ptr, ordered set $S \subseteq [n]$ of p locations, and tuple of words $V \in (\{0,1\}^\lambda)^p$, define$(V^0, \pi, digest, \tau)$ as follows:*

- $(V^0, \pi) =$ C.Open(pp, ptr, S) *and*
- $(digest, \tau) =$ C.Update(pp, ptr, S, V).

It holds that OpenUpdate(pp, ptr, S, V) *outputs* $(V^0, \pi, digest, \tau)$ *and computing k sequential calls to* OpenUpdate, *each on at most $p_{\max}$ locations, can be done with $k \cdot p_{\max} \cdot q_3(\lambda)$ work, or with depth $(k - 1) + q_3(\lambda)$ using at most $p_{\max} \cdot q_3(\lambda)$ processors.*

Proof. For the algorithm OpenUpdate, we note that C.Update already computes the values for S before the update and the values for dangling(S). We therefore define OpenUpdate to run C.Update to obtain $(digest, \tau)$, parse $\tau = V^0 || \pi$ where $V^0 \in (\{0,1\}^\lambda \cup \{\bot\})^p$ and output $(V^0, \pi, digest, \tau)$. Since V^0 gives value for each location in S in the Merkle tree before being updated (or $\bot$ is uninitialized), then V^0 is the tuple of values for S given by C.Open$(pp, ptr, `)$ before the update. Additionally, because the node values for dangling(S) are unchanged by C.Update, the proof π output by OpenUpdate will be the same as in C.Open. Therefore, the output of OpenUpdate is correct.

To perform k sequential updates to the Merkle tree, we observe that it is possible to pipeline them, as we describe next. Note that each update only needs to share memory corresponding to the Merkle tree and segment tree. All other memory used by the algorithm specified in Claim 5.19 can be allocated per updated. Consider a sequence of k sequential calls to OpenUpdate, denoted Upd^i for $i \in \{0, \ldots, k-1\}$, each updating at most $p_{\max}$ locations. Recall that OpenUpdate preprocesses its input, then iterates over the levels of a binary tree doing a single access step and then compute steps at each level, and then forms its output. In what follows, it will be helpful to denote the phases of computation done by Upd^i as the sequence:

$$P^i, A^i_0, C^i_0, A^i_1, C^i_1, \ldots, A^i_{\log(n)-1}, C^i_{\log(n)-1}, F^i,$$

where P^i denotes the pre-processing steps, A^i_j is the access step at iteration j, C^i_j denotes the compute steps at iteration j, and F^i corresponds to the steps for forming the output.

To perform the updates in parallel, we will pipeline them in different processes so one starts after the other: Specifically, Upd^0 will start at time 0, Upd^1 will start at time 1, and in general Upd^i will start at time i. Each process remembers the node values it sees during the procedure. The value of the root node, when all operations finish, is the new digest. Additionally, even if some update is given less than $p_{\max}$ positions, we require that certain phases of the update whose running time depends on $p_{\max}$ (namely, the preprocessing steps and compute steps) still take time as if they were given $p_{\max}$ positions. Namely, each of these takes fixed polynomial time in λ and $p_{\max}$, so this can be easily implemented by doing dummy operations until the right amount of time has elapsed. This ensures that for each update P^i takes the same amount of time for each update i, and C^j_i takes the same amount of time for each i, j.

In terms of correctness, we want to show that for every $i \in [k]$, the output of Upd^i in the concurrent execution is the identical to its output in a sequential execution where the operations are run sequentially (using the number of processors specified by the C.Update description). To do so, we will show that for each block of memory shared between different operations, the memory accesses to that block occur in the same order in both executions. The shared memory is that in *ptr* and *segtree*. Note that the only steps that access this memory are the access steps A^i_j.

Consider any memory location in level j of *ptr* or *segtree*. This is only accessed by A^i_j for each i. Therefore, consider any A^i_j and $A^{i^0}_j$ such that such that A^i_j occurs before $A^{i^0}_j$ in the sequential execution. We will show that this is preserved in the concurrent execution.

To show this, let t_P be the depth of the preprocessing steps in single call to C.Update and let t_C be the depth of the compute steps in a single C.Update, and note that t_P, t_C are functions of $\lambda, p_{\max}$. In the concurrent execution, A^i_j occurs at time t , $i + t_P + j \cdot (t_C + 1)$. This is because Upd^i starts at time i, and before A^i_j occurs, there are t_P steps for the pre-processing P^i, j access steps $A^i_0, \ldots, A^i_{j-1}$, and j groups of t_C compute steps $C^i_0, \ldots, C^i_{j-1}$. Let t^0 , $i^0 + t_P + j \cdot (t_C + 1)$ be the time that $A^{i^0}_j$ occurs. Since A^i_j occurs first in the sequential execution, then $i < i^0$, which implies that $t < t^0$. Since this holds for every i , i^0, it follows that every memory access to level j of the tree occurs in the same order in both the concurrent and sequential executions, which implies correctness. Note that this crucially relied on the fact that each access step indeed is a single step.

Last, we show eficiency for the pipelined operations. We note that, since OpenUpdate requires running C.Update and then formatting the output, a single invocation to OpenUpdate requires depth $2 \cdot q_1(\lambda)$ using at most $p_{\max} \cdot q_1(\lambda)$ processors by Claim 5.19, and can be done with $2p_{\max} \cdot q_1(\lambda)$ total work. This implies that the total work to do all k operations is $k \cdot p_{\max} \cdot 2q_1(\lambda)$. To decouple this into depth and processors, we note that, since we pipeline the operations such that in every step a new OpenUpdate begins, the total depth of this sequence of operations can be bounded by $2 \cdot q_1(\lambda) + (k - 1)$. Moreover, there can be a total of $2 \cdot q_1(\lambda)$ operations occurring concurrently, and so $(2 \cdot q_1(\lambda)) \cdot (p_{\max} \cdot q_1(\lambda))$ bounds the total number of processors needed at any given time. Letting $q_3(\lambda) = 2 \cdot (q_1(\lambda))^2$ completes the proof.

This completes the proof of Lemma 5.18.

6 FROM SUCCINCT ARGUMENTS TO SPARKS

In this section, we present our main transformation, which will be instrumental in our construction of SPARKs. Specifically, we show a generic transformation from any concurrently updatable hash function and succinct argument of knowledge for NP, to an argument that satisfies the SPARK completeness and argument of knowledge properties, and where the provers overhead depends *additively* on the multiplicative overhead of the original succinct argument. As we show in Section 8, when instantiating this transformation with a succinct argument whose prover overhead is suficiently small (which is indeed satisfied by existing succinct arguments), this transformation yields a SPARK.

We first give the transformation in the interactive setting. To do so, we start by describing a helper language in Section 6.1 and then give the interactive protocol in Section 6.2. We then prove completeness, argument of knowledge, optimal prover depth, and succinctness in Section 6.3. Finally, we show the transformation in the non-interactive setting in Section 6.4.

6.1 The Update Language

Let (M, x, y, L, t) be any statement in $\mathsf{L}^{\mathrm{PRAM}}_{\mathsf{U}}$, where M is a PRAM 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 $\mathsf{L}_{\mathsf{upd}}$ in Figure 2. This language corresponds to k steps of a PRAM computation where at each step we additionally update a digest corresponding to the memory of M. Specifically, a statement

$$(M, x, k, pp, h, digest_0, hash_0, digest_k, hash_k)$$

is in $\mathsf{L}_{\mathsf{upd}}$ if there exists a sequence of k consistent updates that start with digest $digest_0$ and end with digest $digest_k$. Here, each update may correspond to concurrently reading or writing multiple positions. The ith update $(digest_i, V^{\mathsf{prev}}_i, V^{\mathsf{rd}}_i, \pi_i, \tau_i)$ specifies the digest $digest_i$ after that step, the

Language $\mathcal{L}_{\mathsf{upd}}$:

Statement. $(M, x, k, pp, h, digest_0, hash_0, digest_k, hash_k)$

Witness. $(State_0, V_0^{\mathsf{rd}})$ and updates $(u_1, \ldots, u_k)$, where $u_i = (digest_i, V_i^{\mathsf{prev}}, V_i^{\mathsf{rd}}, \pi_i, \tau_i)$ for all $i \in [k]$.

Relation R_{upd}. For each $i \in [k]$:

– Compute $(State_i, Op_i, S_i, V_i^{\mathsf{wt}}) = \textbf{parallel-step}(M, State_{i-1}, V_{i-1}^{\mathsf{rd}})$.

– Let $v_j^{\mathsf{rd}}, v_j^{\mathsf{wt}}, v_j^{\mathsf{prev}}, \ell_j$, and op_j be the jth values in $V_i^{\mathsf{rd}}, V_i^{\mathsf{wt}}, V_i^{\mathsf{prev}}, S_i$, and Op_i, respectively, for each $j \in [|S_i|]$. The following hold:

(1) $\mathsf{C}.\textbf{VerUpd}(pp, digest_{i-1}, S_i, V_i, digest_i, \tau_i) = 1$, where V_i is a tuple of $|S_i|$ values, where the jth element of V_i is $v_j^{op_j}$.

(2) $\mathsf{C}.\textbf{VerOpen}(pp, digest_{i-1}, S_i, V_i^{\mathsf{prev}}, \pi_i) = 1$.

(3) For each $j \in [|S_i|]$, $v_j^{\mathsf{prev}} \in \left\{\perp, v_j^{\mathsf{rd}}\right\}$.

(4) For each $j \in [|S_i|]$, if $v_j^{\mathsf{prev}} = \perp$ and $\ell_j < |x|$, then $v_j^{\mathsf{rd}} = x_{\ell_j}$.

Last, it holds that $hash_0 = h(State_0, V_0^{\mathsf{rd}})$, $hash_k = h(State_k, V_k^{\mathsf{rd}})$.

Fig. 2. A language for verifying k steps of a RAM computation M on input x from initial state $State_0$ to final state $State_{\mathsf{final}}$.

values V_i^{prev} at the updated locations in the digest before the update, the values V_i^{rd} read from or overwritten in D during that step, and proofs π_i, τ_i validating the operations performed at that step.

The relation of this language is defined relative to a starting PRAM configuration $(State_0, V_0^{\mathsf{rd}})$ and the values given by

$$(State_i, Op_i, S_i, V_i^{\mathsf{wt}}) = \text{parallel-step}(M, State_{i-1}, V_{i-1}^{\mathsf{rd}})$$

for i □ $[k]$. For every step i, the relation checks (1) that the update from $digest_{i-1}$ to $digest_i$ is valid (using proof τ_i and the values V^{rd} and V^{wt}) and (2) there is a valid opening for $digest_{i-1}$ at locations in S_i (using proof π_i and the values V_i^{prev}). Specifically, this check guarantees that the value in V_i^{rd} claimed to have been read for each position either already appeared there under $digest_{i-1}$, or that the position was □ before step i and was initialized correctly in step i. Last, it checks that the values before the sequence of updates $State_0, V_0^{\mathsf{rd}}$ and those after the final update $State_k, V_k^{\mathsf{rd}}$ hash (using h) to the values $hash_0$, $hash_k$, respectively, given by the statement.

We emphasize that for each step i, the values V_i^{rd}, V_i^{wt}, and V_i^{prev} each serve a difference purpose: for each wt operation in the update, V_i^{wt} contains the value written to D, and V_i^{rd} contains the value overwritten in D. For each rd operation, V_i^{rd} contains the read value (and V_i^{wt} contains □). Finally, V_i^{prev} contains the values underlying the digest before the update, at all the positions in question.

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 $\mathsf{L}_{\mathsf{upd}}$ computations can be merged into a single witness for a larger $\mathsf{L}_{\mathsf{upd}}$ computation. This allows us to prove that (M, x, y, L, t) □ $\mathsf{L}_{\mathsf{U}}^{\mathsf{PRAM}}$ 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 $\mathsf{L}_{\mathsf{upd}}$.

The complexity of $\mathsf{L}_{\mathsf{upd}}$. Note that the language $\mathsf{L}_{\mathsf{upd}}$ is a standard NP language. In particular, verifying that an instance-witness pair corresponding to $k \leq t$ updates is in the relation for $\mathsf{L}_{\mathsf{upd}}$ can be done by a circuit with depth $k \cdot \beta(\lambda) \cdot q(\lambda, |(M,x)|, \log t)$ for a polynomial q using $\beta(\lambda) \cdot p_M$ processors, where β is the eficiency of the concurrently updatable hash function, whenever the

Compute-and-prove

Input: T, $State_0$, V_0^{rd}
Prover Input: Witness w, ptr
Hardcoded Values: 1^λ, M, x, t, γ, pp, h
Protocol:

(1) If $T \geq \gamma \log t + 1$, than set $k = \lfloor T/\gamma \rfloor$, which will be the number of steps to compute, and otherwise set $k = T$.

(2) $\mathcal{P}$ does the following for $i = 1, \ldots, k$:[a]
   (a) Compute $(State_i, Op_i, S_i, V_i^{\mathsf{wt}}) = \textbf{parallel-step}(M, State_{i-1}, V_{i-1}^{\mathsf{rd}})$.
   (b) Update D by running $V_i^{\mathsf{rd}} = \textbf{access}^D(Op_i, S_i, V_i^{\mathsf{wt}})$.
   (c) Spawn a parallel process to compute $\textbf{OpenUpdate}(pp, ptr, S_i, V_i)$, where V_i is a tuple of $|S_i|$ values where the jth value is given by either V_i^{rd} or V_i^{wt} according to the corresponding operation in Op_i. Let $(V_i^{\mathsf{prev}}, \pi_i, digest_i, \tau_i)$ be the output.

(3) Without waiting for Step 2c to halt, if $T \geq \gamma \log t + 1$ then $\mathcal{P}$ spawns a process to run **Compute-and-prove** with $\mathcal{V}$ on input $(T - k, State_k, V_k^{\mathsf{rd}})$.

(4) Without waiting for Step 2c to halt, $\mathcal{P}$ computes $hash_k = h(State_k, V_k^{\mathsf{rd}})$.

(5) Once step 2c halts, $\mathcal{P}$ sets $statement = (M, x, k, pp, h, digest_0, hash_0, digest_k, hash_k)$ and $wit = ((State_0, V_0^{\mathsf{rd}}), (digest_1, V_1^{\mathsf{prev}}, V_1^{\mathsf{rd}}, \pi_1, \tau_1), \ldots, (digest_k, V_k^{\mathsf{prev}}, V_k^{\mathsf{rd}}, \pi_k, \tau_k))$. In the statement, $digest_0$, $hash_0$ are the final digest and hash computed in the previous call to **Compute-and-prove**, or $digest_{\mathsf{start}}$, $hash_{\mathsf{start}}$ in the case that this is the first one.

(6) $\mathcal{P}$ spawns a process to run an interactive argument of knowledge with $\mathcal{V}$ to send $statement$ to $\mathcal{V}$ and prove that $statement \in \mathcal{L}_{\mathsf{upd}}$ using $(\mathcal{P}_{\mathsf{sARK}}(wit), \mathcal{V}_{\mathsf{sARK}})$.

[a]The definitions of **parallel-step** and **access** can be found in Section 3.1, and the definition of **OpenUpdate** is specified by Definition 5.4.

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

number of positions changed in each update is at most p_M (this follows from the eficiency of the concurrently updatable hash function). When using a succinct argument to prove statements in $\mathsf{L_{upd}}$, we can either view the relation as a circuit, Turing machine, or PRAM machine that uses $\beta(\lambda) \cdot p_M$ processors.

6.2 Interactive Protocol

In this section, we give our protocol in Figures 3 and 4. It relies on the following ingredients:

- A succinct argument of knowledge $(\mathsf{P_{sARK}}, \mathsf{V_{sARK}})$ for $\mathsf{L_{upd}}$ with (α, ρ)-prover eficiency.
- A concurrently updatable hash function C with β-parallel eficiency.
- A collision-resistant hash function family ensemble $\mathsf{H} = \{\mathsf{H}_\lambda\}_{\lambda \in \mathbb{N}}$ with $h\colon \{0,1\}^* \to \{0,1\}^\lambda$ for each $h \in \mathsf{H}_\lambda$. We note that this is implied by C.

We refer to Section 2 for a high-level overview of the construction and next give the formal details.

Parameters. For ease of readability for the protocol and corresponding proofs, we define the parameters for the protocol with respect to the relation $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$, security parameter $\lambda \in \mathbb{N}$, and statement $(M, x, t, L) \in \{0,1\}^*$ as follows: Note that we assume that all functions defined below are computable in polynomial time in their input length.

- $n \leq 2^\lambda$ is the amount of words in memory needed to run M, and p_M is the number of parallel processors used by M.
- $\beta \triangleq \beta(\lambda)$ is the "hash efficiency" of our construction. Namely, β upper bounds the parallel efficiency of C on security parameter λ and the time to compute a hash from H_λ. Specifically, we will be using the hash function $h \in \mathsf{H}_\lambda$ on inputs containing k RAM states and k words, for $k \in \mathbb{N}$, and we require that this takes time β using $k \cdot \beta$ processors. For example, this can be achieved by using C for H.
- α and ρ are functions defining the prover efficiency of $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$. For any security parameter Λ, machine, input, and output of total length X, and bound on time T to verify a statement in $\mathsf{L}_{\mathsf{upd}}$ using P processors, without loss of generality, we assume $\alpha(\Lambda, X, T, P)/T$ and $\rho(\Lambda, X, T, P)$ are increasing functions in X, T, and P.[11] For any statement in $\mathsf{L}_{\mathsf{upd}}$ corresponding to k updates, we note that T can be written as $k \cdot f(k)$ where f is increasing in k (and also depends on λ, $|(M,x)|$), and so $\alpha(\Lambda, X, T, P)/k$ is also increasing as a function of k.
- $\ell_{\mathsf{upd}}, t_{\mathsf{upd}}, p_{\mathsf{upd}}$ are functions determining the complexity of an $\mathsf{L}_{\mathsf{upd}}$ instance on at most t updates. Define $\ell_{\mathsf{upd}} \triangleq \ell_{\mathsf{upd}}(\lambda, |(M,x)|, t)$ to be an upper bound on the statement length, and note that $\ell_{\mathsf{upd}} \in |(M,x)| + \log t + \mathrm{poly}(\lambda)$ by definition of $\mathsf{L}_{\mathsf{upd}}$. We let $t_{\mathsf{upd}} \triangleq t_{\mathsf{upd}}(\lambda, |(M,x)|, t)$ upper bound the time to verify the instance using $p_{\mathsf{upd}} \triangleq p_{\mathsf{upd}}(\lambda, p_M)$ processors. Note that $t_{\mathsf{upd}} \in t \cdot \beta \cdot |(M,x)| \cdot \mathrm{poly}(\lambda, \log t)$ when $p_{\mathsf{upd}} = \beta \cdot p_M$.
- $\alpha^* \triangleq \alpha(\lambda, \ell_{\mathsf{upd}}, t_{\mathsf{upd}}, p_{\mathsf{upd}})/t$ is the worst-case multiplicative overhead (with respect to the depth t) of the depth of running $\mathsf{P}_{\mathsf{sARK}}$ to prove a statement in $\mathsf{L}_{\mathsf{upd}}$ corresponding to at most t steps of computation, when using $\rho^* \triangleq \rho(\lambda, \ell_{\mathsf{upd}}, t_{\mathsf{upd}}, p_{\mathsf{upd}})$ processors. Note that this implies that any valid $\mathsf{L}_{\mathsf{upd}}$ statement with $k \leq t$ steps can be proven in parallel time $\alpha^* \cdot k$ with ρ^* processors.
- $\gamma \triangleq \alpha^* + 1$ is such that a $1/\gamma$ fraction of remaining computation is done at each recursive call to Compute-and-prove. We note that γ can be efficiently computed as a function of the common inputs to the protocol.

We formalize the protocol in Figures 3 and 4. We are now ready to state our main theorem.

Theorem 6.1 (Restatement of Theorem 1.1). *Suppose there exists a concurrently updatable hash function and a succinct argument of knowledge* $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ *with* (α, ρ)*-prover efficiency for the* NP *language* $\mathsf{L}_{\mathsf{upd}}$. *Then, there exists an interactive protocol* (P, V) *for* $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ *satisfying SPARK completeness and argument of knowledge for* NP*, as well as the following efficiency properties:*

There exists a polynomial q such that for all $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ where M has access to $n \leq 2^\lambda$ words in memory and p_M processors, the following hold: Let α^* and ρ^* (formally defined above based on α and ρ) be the multiplicative overhead in depth (with respect to the number of steps) and number of parallel processors used, respectively, by $\mathsf{P}_{\mathsf{sARK}}$ *to prove a statement in* $\mathsf{L}_{\mathsf{upd}}$ *corresponding to at most t steps of computation. Then:*

- *The depth of the prover is bounded by* $t + (\alpha^*)^2 \cdot |(M,x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M))$ *when using* $(p_M + \alpha^* \cdot \rho^*) \cdot q(\lambda, \log(t \cdot p_M))$ *processors.*
- *The work of the verifier is bounded by* $\alpha^* \cdot |(M,x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M))$*, and the length of the transcript produced in the interaction between* $\mathsf{P}(w)$ *and* V *is bounded by* $\alpha^* \cdot L \cdot q(\lambda, \log(t \cdot p_M))$.

[11]For example, if $\alpha(\Lambda, X, T, P)/T$ were not increasing in T, then we could define an upper bound $\alpha'(\Lambda, X, T, P) = T \cdot \max_{t \leq T}(\alpha(\Lambda, X, t, P)/t)$ that is increasing in T and preserves asymptotic behavior.

Protocol $(\mathcal{P}, \mathcal{V})$ for $\mathcal{R}_{\mathcal{U}}^{\mathrm{PRAM}}$ between $\mathcal{P}(w)$ and $\mathcal{V}$ on common input $(1^\lambda, (M, x, t, L))$:

(1) $\mathcal{V}$ samples $pp \leftarrow \mathbf{C.Gen}(1^\lambda, n)$ and $h \leftarrow \mathcal{H}_\lambda$, and then computes $(*, digest_{\mathsf{start}}) = \mathbf{C.Hash}(pp, D_\perp)$, where $D_\perp$ is the empty partial string. $\mathcal{V}$ sends (pp, h) to $\mathcal{P}$.

(2) Both parties compute γ (as in the parameters paragraph), initialize $State_{\mathsf{start}}$ as a tuple containing the initial (empty) state of M, set $V_{\mathsf{start}}^{\mathsf{rd}} = (\perp)$, and compute $hash_{\mathsf{start}} = h(State_{\mathsf{start}}, V_{\mathsf{start}}^{\mathsf{rd}})$.

(3) $\mathcal{P}$ computes $(ptr, digest_{\mathsf{start}}) = \mathbf{C.Hash}(pp, D_\perp)$. $\mathcal{P}$ 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 $\mathcal{P}$ 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 $\mathcal{P}$ accesses a location in D, we implicitly assume it translates the location appropriately.

(4) $\mathcal{P}$ runs the sub-protocol **Compute-and-prove** with $\mathcal{V}$ on input $(t, State_{\mathsf{start}}, V_{\mathsf{start}}^{\mathsf{rd}})$. For $i \in [m]$, let Π_i be the ith sub-protocol proving $statement_i := (M_i, x_i, k_i, pp_i, h_i, digest_i, hash_i, digest_i', hash_i')$.

(5) $\mathcal{P}$ computes $(Y, \pi_{\mathsf{final}}) = \mathbf{C.Open}(pp, ptr, [L'])$, where $L' = \lceil L/\lambda \rceil$, and sends $(Y, \pi_{\mathsf{final}}, State_{\mathsf{final}}, V_{\mathsf{final}}^{\mathsf{rd}})$ to $\mathcal{V}$ where $State_{\mathsf{final}}, V_{\mathsf{final}}^{\mathsf{rd}}$ are the final PRAM states and words read in the last iteration of **Compute-and-prove**.

(6) $\mathcal{V}$ lets y be the concatenation of the first $outlen$ bits of Y, where $outlen$ is the output length specified by $State_{\mathsf{final}}$. Then, $\mathcal{V}$ outputs y if the following hold, and outputs $\perp$ otherwise:
 (a) $\mathcal{V}_{\mathsf{sARK}}$ accepts in $\Pi_1, \ldots, \Pi_m$.
 (b) For all $i \in [m]$, it holds that $(M_i, x_i, pp_i, h_i) = (M, x, pp, h)$.
 (c) $\sum_{i=1}^{m} k_i = t$.
 (d) $(digest_1, hash_1) = (digest_{\mathsf{start}}, hash_{\mathsf{start}})$, and $hash_m' = h(State_{\mathsf{final}}, V_{\mathsf{final}}^{\mathsf{rd}})$.
 (e) $(digest_i', hash_i') = (digest_{i+1}, hash_{i+1})$ for all $i \in [m-1]$.
 (f) $State_{\mathsf{final}}$ is a halting state, Y consists of $L' = \lceil L/\lambda \rceil$ words, and $\mathbf{C.VerOpen}(pp, digest_m', [L'], Y, \pi_{\mathsf{final}})$ accepts.

Fig. 4. Protocol (P, V) for $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$.

We prove Theorem 6.1 by showing that the protocol in Figure 4 is a SPARK for $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ with ρ-succictness for every ρ with $\rho(\lambda, t) \in \mathrm{poly}(\lambda, \log t)$. The proof is given in Section 6.3. Specifically, we show completeness in Lemma 6.2, argument of knowledge in Lemma 6.3, prover eficiency in Lemma 6.13, and succinctness in Lemmas 6.16 and 6.17. Before giving the proofs, we give the following remarks about the construction:

Remark 8 (On the Size of M and x). We note that when we bound the communication complexity (Lemma 6.17), we assume without loss of generality that the machine M and input x are *a priori* bounded by a f ixed polynomial inλ. This enables us to bound the number of sub-protocols, and hence the communication complexity, independently of $|(M,x)|$. A similar observation was made by Reference [16] to achieve succinctness. This assumption is without loss of generality, since P, when given a large input (M, x), could instead compute $digest = h(M, x)$ where h is a hash function and prove the instance $(U_h, (h, digest), t^0, L)$ using witness (M, x, w). Here, U_h is a universal RAM machine for p_M bounded parallelism with the hash function h hardcoded. U_h receives input *digest*, witness (M, x, w), and checks that $digest = h(M, x)$ and if so, computes and outputs $y = M(x, w)$. U has size poly(λ) independent of $|(M,x)|$, and because it is a RAM machine, it can perform the hash and simulate M in time $t^0 = t + |(M,x)| \cdot \text{poly}(\lambda)$. Additionally, U uses the same amount of parallelism as M and $n + |(M,x)| \cdot \text{poly}(\lambda)$ memory, where the additional memory is used to compute the hash (note that if the resulting memory size is larger than 2^λ, then P and V can simply use a polynomially larger security parameter to prove the resulting statement).

To formalize this transformation, both P and V would be changed to compute *digest* and run the SPARK protocol with statement $(U_h, (h, digest), t^0, L)$. As such, the running times of the prover and verifier incur a delay of $|(M,x)| \cdot \text{poly}(\lambda)$, but the remaining complexity would be based on having a statement of size poly(λ) and a time bound of $t^0 = t + |(M,x)| \cdot \text{poly}(\lambda)$.

Remark 9 (On the Dependence on t and p_M). We note that our construction when used for a PRAM machine M needs to know the time bound t and the bound on number of processors p_M ahead of time. Specifically, the parameter γ, which determines how the prover divides up the computation, depends on both t and p_M. This assumption is standard for universal arguments [8], but for some applications, a bound on time or processors may not be *a priori* known. Existing techniques for constructing eficient SNARKs based on incremental verifiable computation (e.g., References [16, 55]) do not require this assumption, but it is not clear how to extend this approach to the interactive setting (starting from weaker assumptions). We leave it as an open question to construct a SPARK where the prover does not know t and p_M in advance.

6.3 Proofs

In this section, we prove completeness, argument of knowledge, succinctness, and prover eficiency.

Lemma 6.2 (Completeness). *For every $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in \mathsf{R}_{\mathsf{U}}^{\text{PRAM}}$ where M has access to $n \leq 2^\lambda$ words in memory, it holds that*

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

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

Proof. Let Π_i be as defined by the protocol for $i \in [m]$, with statement

$$statement_i = (M_i, x_i, k_i, pp_i, h_i, digest_i, hash_i, digest_i^0, hash_i^0).$$

Recall that V accepts and outputs $y \neq \bot$ if and only if conditions 6a through 6f from Figure 4 are valid with respect to these statements. Conditions 6b, 6c, 6d, and 6e follow immediately by definition of P. Therefore, we focus on conditions 6a and 6f.

For conditions 6a and 6f, we first show that the sequence of t updates $u_i = (digest_i, V_i^{\text{prev}}, V_i^{\text{rd}}, \pi_i, \tau_i)$ for $i \in [t]$ that the prover computes at each step (across all statements) are valid. In particular, let $(State_i, Op_i, S_i, V_i^{\text{wt}}) = \text{parallel-step}(M, State_{i-1}, V_{i-1}^{\text{rd}})$ for all $i \in [t]$ where we initialize $State_0, V_0^{\text{rd}}$ as $State_{\text{start}}, V_{\text{start}}^{\text{rd}}$ as in the protocol. We show that all conditions specified in $\mathsf{L}_{\mathsf{upd}}$ hold for each update u_i according to the computation of M.

To show this, recall that the digest and proofs in each update i of the full computation are computed as $(V_i^{\text{prev}}, \pi_i, digest_i, \tau_i) = \text{OpenUpdate}(pp, ptr, S_i, V_i)$, where V_i is defined from $V_i^{\text{rd}}, V_i^{\text{wt}}$ as in the protocol. By the eficiency property of Definition 5.4, the values computed are equivalent to computing $(V_i^{\text{prev}}, \pi_i) = \text{C.Open}(pp, ptr, S_i)$ and then $(digest_i, \tau_i) = \text{C.Update}(pp, ptr, S_i, V_i)$ sequentially at each step. Given this, it holds that before step i of the full computation, the prover has computed $(ptr, digest_0) = \text{C.Hash}(pp, D_\emptyset)$, where $D_\emptyset$ is the empty partial string, and then computed $i - 1$ updates. Let D^* be the true string resulting from the first $i - 1$ updates, and let D^C be the partial string underlying the digest. Namely, D^* starts as $x||w||0^{n_\lambda - |x| - |w|}$, D^C starts as $D_\perp$, and we apply the same $i - 1$ logical updates to both strings. Note that $D^C = \perp$ for all positions j that have not yet been accessed, and $D^C_j = D^*_j$ for all other locations.

Next, we will use D^* and D^C to show that update u_i satisfies conditions 1, 2, 3, and 4 of $\mathsf{L}_{\mathsf{upd}}$. First, by update completeness, since V_i is defined from $V^{\mathrm{rd}}_i, V^{\mathrm{wt}}_i$ exactly as in the definition of $\mathsf{L}_{\mathsf{upd}}$, and $(digest_i, \tau_i) = \mathrm{C.Update}(pp, ptr, S_i, V_i)$ then it follows that $\mathrm{C.VerUpd}(pp, digest_{i-1}, S_i, V_i, digest_i, \tau_i)$ accepts as required by condition 1. Next, by open completeness of C, since $(V^{\mathrm{prev}}_i, \pi_i) = \mathrm{C.Open}(pp, ptr, S_i)$, then $\mathrm{C.VerOpen}(pp, digest_{i-1}, S_i, V^{\mathrm{prev}}_i, \pi_i)$ accepts. This satisfies condition 2 of $\mathsf{L}_{\mathsf{upd}}$. Open completeness also implies that V^{prev}_i are the values of S_i in D^C. This gives condition 3, since the value of each location in D^C is equal to $\perp$ if it has not been accessed yet, and otherwise P sets it to the corresponding value in V^{rd} given for that location in D^*. Last, for each location $\ell_j \in S_i$, when the corresponding value in V^{prev}_i is set to $\perp$ and $\ell_j \leq |x|$, then $D^C = \perp$ and so location ℓ_i has never been accessed. This implies that $D^*_{\ell_i} = x_{\ell_i}$, which gives condition 4. Thus, all conditions specified by $\mathsf{L}_{\mathsf{upd}}$ hold for each update $u_i = (digest_i, V^{\mathrm{prev}}_i, V^{\mathrm{rd}}_i, \pi_i, \tau_i)$ as required.

We now show that V accepts condition 6a for the full protocol of Figure 4. Because each update is valid with respect to $\mathsf{L}_{\mathsf{upd}}$, it follows that the prover $\mathsf{P}_{\mathsf{sARK}}$ for sub-protocol Π_i receives a valid witness with respect to $statement_i$ for $i \in [m]$. Specifically, it receives the k_i consecutive updates corresponding to the ith sub-computation performed by P, where the starting hash corresponds to the starting states and words read in the witness, and the ending hash corresponds to the final states and words read resulting from the sequence of updates, both by definition of P. Completeness of $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ implies that $\mathsf{V}_{\mathsf{sARK}}$ accepts in protocols Π_i.

For condition 6f, we have that P honestly steps through the computation of $M(x, w)$. To see that P reaches the final state, recall that each sub-computation corresponds to k_i steps of the original computation and $\sum_{i=1}^m k_i = t$ (by condition 6c). Therefore, the final state $State_m$ corresponds to the state of $M(x, w)$ after t steps. Since $((M, x, y, L, t), w) \in \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}$, then after t steps the final state will be the halting state. We showed above that the prover performs all updates correctly and consistent with memory, so it follows by open completeness that $\mathrm{C.VerOpen}(pp, digest^0_m, [\lceil L/\lambda \rceil], Y, \pi_{\mathrm{final}}) = 1$ and that Y is the right length, and hence that the output is equal to y.

Lemma 6.3 (Argument of Knowledge). *(P, V) satisfies the argument of knowledge for* NP *property of Definition 4.1.*

Proof. To show that (P, V) is an argument of knowledge for $\mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}$, consider any non-uniform polynomial-time prover $\mathsf{P}^* = \{\mathsf{P}^*_\lambda\}_{\lambda \in \mathbb{N}}$, integer $c \in \mathbb{N}$, security parameter $\lambda \in \mathbb{N}$, and statement (M, x, t, L) where M accesses at most $n \leq 2^\lambda$ memory and p_M processors, with $|M,x,t| \leq \lambda$, $L \leq \lambda$, and $t \cdot p_M \leq |x|^c$. Let $\mathsf{P}^*_{\lambda, z, s}$ denote P^*_λ with auxiliary input z and hardcoded randomness s for any $z, s \in \{0, 1\}^*$. Let V_r denote the verifier V with hardcoded randomness $r \in \{0, 1\}^{l(\lambda)}$, where $l(\lambda)$ is an upper bound on the randomness used by the verifier. Note that l is a function of λ, since by Lemma 6.16, the verifier runs in time polynomial in λ, $|(M,x)|$ $L, p_M, \log t$, each of which are bounded by a fixed polynomial in λ.

Recall that (P, V) consists of m sub-protocols $\Pi_1, \ldots, \Pi_m$, where each is an instance of the protocol $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$. Let $\mathsf{E}_{\mathsf{sARK}}$ be the extractor for $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ with expected running time bounded by a polynomial q_{sARK}, which exists by assumption that $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ is an argument of knowledge. As a subroutine to our full extractor, we first construct a probabilistic oracle machine $\mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^*_{\lambda, z, s}, \mathsf{V}_r}$ that uses $\mathsf{E}_{\mathsf{sARK}}$ to extract witnesses for the statements in each sub-protocol defined by the interaction $(\mathsf{P}^*_{\lambda, z, s}, \mathsf{V}_r)$, as follows:

<u>$\mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^{\star}_{\lambda,z,s},\mathsf{V}_r}(1^\lambda,(M,x,t,L))$:</u>

(1) Emulate the interaction between $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r on common input $(1^\lambda,(M,x,t,L))$, which uniquely determines the statement $statement_i$ used for sub-protocol Π_i for all $i \in [m]$. Let Y be the values in the opening sent in the final message of the protocol.

(2) For all $i \in [m]$, define the prover $\mathsf{P}^{\star}_i$ and verifier $\mathsf{V}_{\mathsf{sARK},r_i}$ for the protocol $(\mathsf{P}_{\mathsf{sARK}},\mathsf{V}_{\mathsf{sARK}})$ on common input $(1^\lambda, statement_i)$ as follows:
- $\mathsf{P}^{\star}_i$ emulates the interaction between $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r on common input $(1^\lambda,(M,x,t,L))$ until the start of Π_i. $\mathsf{P}^{\star}_i$ then interacts with $\mathsf{V}_{\mathsf{sARK}}$ as part of Π_i for statement $statement_i$ while continuing to use $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r to emulate messages for all other sub-protocols.
- $\mathsf{V}_{\mathsf{sARK},r_i}$ is the verifier V_r on common input $(1^\lambda,(M,x,t,L))$ restricted to its interaction in sub-protocol Π_i. Namely, $\mathsf{V}_{\mathsf{sARK},r_i}$ uses fixed randomness r_i determined by r for Π_i.

Note that $\mathsf{P}^{\star}_i$ and $\mathsf{V}_{\mathsf{sARK},r_i}$ can be emulated using oracles $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r.

(3) For $i \in [m]$, let $wit_i \leftarrow \mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^{\star}_i,\mathsf{V}_{\mathsf{sARK},r_i}}(1^\lambda, statement_i)$, where all queries made by $\mathsf{E}_{\mathsf{sARK}}$ to $\mathsf{P}^{\star}_i$ and $\mathsf{V}_{\mathsf{sARK},r_i}$ are emulated by $\mathsf{E}_{\mathsf{inner}}$ using its own oracles $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r.

(4) Output $(wit_1,\ldots,wit_m,Y)$.

In the following claims, we show that (1) $\mathsf{E}_{\mathsf{inner}}$ runs in expected polynomial time (over r and its own random coins) and (2) with all but negligible probability (over r and the coins of $\mathsf{E}_{\mathsf{sARK}}$), either $\mathsf{P}^{\star}_{\lambda,z,s}$ fails to convinces V_r or for all $i \in [m]$ the witness wit_i extracted by $\mathsf{E}_{\mathsf{sARK}}$ is valid for $statement_i$ with respect to $\mathsf{L}_{\mathsf{upd}}$:

Claim 6.4. *There exists a polynomial q_{inner} such that for every non-uniform probabilistic polynomial-time prover $\mathsf{P}^{\star} = \{\mathsf{P}^{\star}_\lambda\}_{\lambda\in\mathbb{N}}$, $\lambda, c \in \mathbb{N}$, statement (M,x,t,L) where M has access to $n \le 2^\lambda$ words in memory and p_M processors, with $|M,x,t| \le \lambda$, $L \le \lambda$, and $t \cdot p_M \le |x|^c$, and $z,s \in \{0,1\}^*$, the expected running time (with a single processor) of $\mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^{\star}_{\lambda,z,s},\mathsf{V}_r}(1^\lambda,(M,x,t,L))$ is at most $q_{\mathsf{inner}}(\lambda, t\cdot p_M)$.*

Proof. We first analyze the time to emulate a full interaction between $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r, which is used to determine the statements $statement_i$ and to emulate the oracle calls of $\mathsf{E}_{\mathsf{sARK}}$ to $\mathsf{P}^{\star}_i$ and $\mathsf{V}_{\mathsf{sARK},r_i}$. Since each oracle call takes a single step by assumption, it follows that the emulation time is at most $\mathsf{work}_{\mathsf{V}}(1^\lambda,(M,x,t,L))$ to receive and read each message. By the succinctness of (P,V) (given by Lemma 6.16) this is bounded by a polynomial $q_{\mathsf{V}}(\lambda, |(M,x)|, L, p_M, \log(t\cdot p_M))$ independent of $\mathsf{P}^{\star}$ and the statement.

Next, we analyze the expected running time of $\mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^{\star}_i,\mathsf{V}_{\mathsf{sARK},r_i}}$ for each $i \in [m]$. Recall that $t_{\mathsf{upd}} \cdot p_{\mathsf{upd}}$ is an upper bound on the amount of work to verify a statement with at most t updates in $\mathsf{L}_{\mathsf{upd}}$. Since $\mathsf{E}_{\mathsf{sARK}}$ is extracting a witness for an $\mathsf{L}_{\mathsf{upd}}$ statement, then for each $i \in [m]$, the expected running time of $\mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^{\star}_i,\mathsf{V}_{\mathsf{sARK},r_i}}$ is at most $q_{\mathsf{sARK}}(\lambda, t_{\mathsf{upd}}\cdot p_{\mathsf{upd}})$ for some polynomial q_{sARK} when given oracle access to $\mathsf{P}^{\star}_i$ and $\mathsf{V}_{\mathsf{sARK},r_i}$ assuming r_i is uniformly distributed. As the random coins for V_r are uniform and V invokes m independent instances of $\mathsf{V}_{\mathsf{sARK}}$, this implies that the randomness r_i used by $\mathsf{V}_{\mathsf{sARK},r_i}$ is uniform. Thus, the expected running time of $\mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^{\star}_i,\mathsf{V}_{\mathsf{sARK},r_i}}$ is at most $q_{\mathsf{sARK}}(\lambda, t_{\mathsf{upd}}\cdot p_{\mathsf{upd}})$.

Putting everything together, we have that $\mathsf{E}_{\mathsf{inner}}$ first emulates the interaction between $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r and then runs $\mathsf{E}_{\mathsf{sARK}}$ to extract a witness m times while emulating the oracle calls of $\mathsf{E}_{\mathsf{sARK}}$ (and the resulting oracle calls made to $\mathsf{P}^{\star}_{\lambda,z,s}$ and V_r). Thus, the full expected running time is bounded by

$$q_{\mathsf{V}}(\lambda, L, p_M, \log(t\cdot p_M)) + m \cdot q_{\mathsf{V}}(\lambda, L, p_M, \log(t\cdot p_M)) \cdot q_{\mathsf{sARK}}(\lambda, t_{\mathsf{upd}} \cdot p_{\mathsf{upd}}).$$

We can bound $t_{\mathsf{upd}}(\lambda, |(M,x)|, t) \in \mathrm{poly}(\lambda, |(M,x)|, t)$ and $p_{\mathsf{upd}}(\lambda, p_M) \in \mathrm{poly}(\lambda, p_M)$, as well as $|(M,x)| \leq \lambda$, and $L \leq \lambda$. For m, by succinctness (Lemma 6.16), we have that $m \leq \alpha^\star \cdot \mathrm{poly}(\lambda, |(M,x)|, L, \log(t\cdot p_M))$ and $\alpha^\star$ can be bounded by a polynomial in $\lambda, |(M,x)|, t, p_M$ by definition. Putting these bounds together, this implies that the expected running time is bounded by a polynomial $q_{\mathsf{inner}}(\lambda, t\cdot p_M)$.

Claim 6.5. *For every non-uniform probabilistic polynomial-time prover $\mathsf{P}^\star = \{\mathsf{P}^\star_{\lambda,z,s}\}_{\lambda\in\mathbb{N}}$ and constant $c \in \mathbb{N}$, there exists a negligible function* $\mathsf{negl}_{\mathsf{inner}}$ *such that for all $\lambda \in \mathbb{N}$, statement (M, x, t, L) where M has access to $n \leq 2^\lambda$ and p_M processors, and with $|M,x,t| \leq \lambda$, $L \leq \lambda$, and $t\cdot p_M \leq |x|^c$, and every $z, s \in \{0,1\}^*$, it holds that*

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ y = \langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r\rangle(1^\lambda, (M,x,t,L)) \\ (wit_1, \ldots, wit_m, Y) \leftarrow \mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M,x,t,L)) \end{array} : \begin{array}{l} y \neq \bot \wedge \\ \exists i \in [m] : (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right]$$
$$\leq \mathsf{negl}_{\mathsf{inner}}(\lambda),$$

where $statement_i$ is defined to be the statement of the ith sub-protocol in the interaction $\langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r\rangle$.

Proof. To analyze the above probability, we start by formalizing an algorithm S, which is implicit in the description of $\mathsf{E}_{\mathsf{inner}}$. The algorithm S takes as input $r \in \{0,1\}^{l(\lambda)}$, and emulates the interaction $\langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r\rangle$. It then outputs $(y, statement_1, \ldots, statement_m)$, where $statement_i$ is the ith statement in the interaction and y is the output of the protocol. Note that these statements are the same as the ones computed by $\mathsf{E}_{\mathsf{inner}}$ in the first step of its description. We can then write the above probability as

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ (y, statement_1, \ldots, statement_m) = \mathsf{S}(r) \\ (wit_1, \ldots, wit_m, Y) \leftarrow \mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M,x,t,L)) \end{array} : \begin{array}{l} y \neq \bot \wedge \\ \exists i \in [m] : (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right].$$

Next, we apply a union bound to upper bound this by

$$\sum_{i\in[m]} \Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ (y, statement_1, \ldots, statement_m) = \mathsf{S}(r) \\ (wit_1, \ldots, wit_m, Y) \leftarrow \mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M,x,t,L)) \end{array} : \begin{array}{l} y \neq \bot \wedge \\ (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right]. \quad (6.1)$$

We now upper bound the above probability for any particular $i \in [m]$. We notice that whenever $y \neq \bot$, that implies that V accepts in protocol Π_i for $statement_i$.

By definition of $\mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r}$, for each $i \in [m]$, the witness wit_i is computed by running $\mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^\star_i, \mathsf{V}_{\mathsf{sARK}, r_i}}$, where $\mathsf{E}_{\mathsf{inner}}$ uses its oracles $\mathsf{P}^\star_{\lambda,z,s}$ and V_r to emulate all queries that $\mathsf{E}_{\mathsf{sARK}}$ makes to $\mathsf{P}^\star_i$ and $\mathsf{V}_{\mathsf{sARK}, r_i}$. Specifically, emulating $\mathsf{P}^\star_i$ requires querying $\mathsf{P}^\star_{\lambda,z,s}$ for every sub-protocol, and querying V_r for all protocols other than i.

Let r_{-i} be the randomness of V_r used in all protocols other than i, where it uses r_i. Note that $\mathsf{P}^\star_i$ only depends on r_{-i}, since it only uses V_r in protocols other than i. Another way to state this is to view $\mathsf{P}^\star_i$ as an *randomized* prover that emulates the verifier in all sub-protocols other than i using its internal randomness, where in the above execution, its internal randomness is r_{-i}. To make this clear, let $\mathsf{P}^\star_{i,r_{-i}}$ denote the prover $\mathsf{P}^\star_i$ (viewing it as a randomized algorithm) that uses randomness r_{-i} to emulate the verifier in all protocols other than i, and note that $\mathsf{P}^*_{i,r_{-i}}$ can still be emulated using the oracles $\mathsf{P}^*_{\lambda,z,s}$ and V_r. We can then write the above probability as

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ (y, statement_1, \ldots, statement_m) = \mathsf{S}(r) \\ wit_i \leftarrow \mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^*_{i,r_{-i}}, \mathsf{V}_{\mathsf{sARK},r_i}}(1^\lambda, statement_i) \end{array} : \begin{array}{l} y \neq \bot \,\wedge \\ (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right].$$

Whenever $y \neq \bot$, it must be the case that V accepts in all sub-protocols, and therefore by definition of P^*_i, it follows that $\mathsf{V}_{\mathsf{sARK},r_i}$ accepts in protocol Π_i with $\mathsf{P}^*_{i,r_{-i}}$. We can therefore upper bound the above probability by

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ (y, statement_1, \ldots, statement_m) = \mathsf{S}(r) \\ wit_i \leftarrow \mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^*_{i,r_{-i}}, \mathsf{V}_{\mathsf{sARK},r_i}}(1^\lambda, statement_i) \end{array} : \begin{array}{l} \langle \mathsf{P}^*_{i}, \mathsf{V}_{\mathsf{sARK},r_i}\rangle(1^\lambda, statement_i) = 1 \\ \wedge\, (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right]. \tag{6.2}$$

We can now use the argument of knowledge property of $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$. Let $l'(\lambda)$ be the length of the randomness used by $\mathsf{V}_{\mathsf{sARK}}$. For any $r = (r_{-i}, r_i) \in \{0,1\}^{l(\lambda)}$, using r_{-i} as the randomness for $\mathsf{P}^*_{i,r_{-i}}$, by the argument of knowledge property of $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ there exists a negligible function μ_i (which depends on the algorithm P^*_i but is independent of its randomness) such that for every randomness r_{-i} for P^*_i, and for the statement $statement_i$ (which in this case is determined by r_{-i}) it holds that

$$\Pr\left[\begin{array}{l} r_i \leftarrow \{0,1\}^{l'(\lambda)} \\ wit_i \leftarrow \mathsf{E}_{\mathsf{sARK}}^{\mathsf{P}^*_{i,r_{-i}}, \mathsf{V}_{\mathsf{sARK},r_i}}(1^\lambda, statement_i) \end{array} : \begin{array}{l} \langle \mathsf{P}^*_{i,r_{-i}}, \mathsf{V}_{\mathsf{sARK},r_i}\rangle(1^\lambda, statement_i) = 1 \\ \wedge\, (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right] \le \mu_i(\lambda).$$

By using the law of total probability in Equation (6.2) (to sum over each choice of r_{-i}), and by applying the above inequality, we obtain that Equation (6.2) is bounded above by $\mu_i(\lambda)$. Finally, by plugging this back into Equation (6.1), we obtain that the probability in the statement of the claim is upper bounded by $\sum_{i\in[m]} \mu_i(\lambda)$. As in the analysis of the previous claim, we can bound m by $\mathrm{poly}(\lambda, |(M,x)|, L, t, p_M)$. As $|(M,x)| \le \lambda$, $L \le \lambda$, and $t \cdot p_M \le |x|^c$, then $m \in \mathrm{poly}(\lambda)$, so this is negligible as required.

Using $\mathsf{E}_{\mathsf{inner}}$ to extract the witnesses in the sub-protocols, we now define the full extractor E that outputs a witness w for (M, x, y, L, t) given oracle access to $\mathsf{P}^*_{\lambda,z,s}$ and V_r, where y is the value output by V_r when interacting with $\mathsf{P}^*_{\lambda,z,s}$.

$\mathsf{E}^{\mathsf{P}^*_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, t, L))$:

(1) Run $(wit_1, \ldots, wit_m, Y) \leftarrow \mathsf{E}_{\mathsf{inner}}^{\mathsf{P}^*_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, t, L))$.
(2) Parse each wit_i as containing an initial set of states and values read $(State^{(i)}, V^{\mathsf{rd},(i)})$ as well as a sequence of updates, where the updates across all m witnesses together yield an overall sequence of t updates $u_j = (digest_j, V^{\mathsf{prev}}_j, V^{\mathsf{rd}}_j, \pi_j, \tau_j)$ for $j \in [t]$ (abort if this is not the case).
(3) For $j = 1, \ldots, t$, compute $(State_j, Op_j, S_j, V^{\mathsf{wt}}_j) = \text{parallel-step}(M, State_{j-1}, V^{\mathsf{rd}}_{j-1})$ where $State_0$ is the tuple containing the initial RAM state and $V^{\mathsf{rd}}_0 = (\bot)$.
(4) Let $D^{\mathsf{Init}} \in \{0,1\}^{n\lambda}$ be the string where for each $\ell \in [n]$, the ℓth word is set to its value in V^{rd}_i, where i is the first iteration with $\ell \in S_i$, or the ℓth word in Y if ℓ is never accessed and $\ell \le \lceil L/\lambda \rceil$, or 0^λ otherwise.
(5) Output w to be the string of length $n\lambda - |x|$ starting at position $|x|$ in D^{Init}.

We note that while D^{Init} and w above may be as large as $n \cdot \lambda$ bits, they can be specified while running M by using at most $\lambda + \log n$ bits for each non-zero value. Furthermore, they can have at most $t + \lceil L/\lambda \rceil$ non-zero values, since M makes at most t memory accesses, and at most $\lceil L/\lambda \rceil$ additional positions are accessed in specifying the output. Thus, D^{Init} and w can be computed with at most $\mathrm{poly}(\lambda, L, t, \log n)$ additive overhead in time and space.

Claim 6.6. *There exists a polynomial q such that $\mathsf{E}^{\mathsf{P}^{\star}_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, t, L))$ has expected running time at most $q(\lambda, t \cdot p_M)$.*

Proof. E first runs $\mathsf{E}_{\mathsf{inner}}$, which has expected running time bounded by a polynomial $q_{\mathsf{inner}}(\lambda, t \cdot p_M)$ by Claim 6.4. We bound the remaining running time of E by a polynomial in λ and $t \cdot p_M$, which completes the claim.

E parses the output as containing m sets of states and words that together have size $m \cdot p_M \cdot \mathrm{poly}(\lambda)$, as well as a sequence of t updates, where each update has size at most $2\beta \cdot p_M \cdot \lambda \in \mathrm{poly}(\lambda)$ by the eficiency of the underlying concurrently updatable hash function. As $m \in \mathrm{poly}(\lambda)$ as discussed in the previous claims, together this takes time $t \cdot p_M \cdot \mathrm{poly}(\lambda)$. Using these updates to determine which values to read, E emulates M for t steps, which can be done in time $t \cdot p_M \cdot \mathrm{poly}(\lambda)$. Finally, E computes the initial memory D^{Init} to output a witness w, which, as discussed above, requires specifying at most $t + \lceil L/\lambda \rceil$ positions and therefore takes at most $\mathrm{poly}(\lambda, L, t) \subseteq \mathrm{poly}(\lambda, t)$ time. Altogether, E runs in expected time at most $q_{\mathsf{inner}}(\lambda, t \cdot p_M) + t \cdot p_M \cdot \mathrm{poly}(\lambda) + t \cdot p_M \cdot \mathrm{poly}(\lambda) + \mathrm{poly}(\lambda, t)$, which can be bounded by a polynomial $q(\lambda, t \cdot p_M)$. ∎

Claim 6.7. *For every non-uniform probabilistic polynomial-time prover $\mathsf{P}^{\star} = \mathsf{P}^{\star}$ and constant $c \in \mathbb{N}$, there exists a negligible function negl such that for all $\lambda \in \mathbb{N}$, statement $(M, x, t, L)_{\lambda \in \mathbb{N}}$ where M has access to $n \leq 2^\lambda$ and p_M processors, and with $|(M, x, t)| \leq \lambda$, $L \leq \lambda$, and $t \cdot p_M \leq |x|^c$, and all $z, s \in \{0,1\}^*$, it holds that*

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{l(\lambda)} \\ y = \langle \mathsf{P}^{\star}_{\lambda,z,s}, \mathsf{V}_r \rangle(1^\lambda, (M, x, t, L)) \\ w \leftarrow \mathsf{E}^{\mathsf{P}^{\star}_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, t, L)) \end{array} : \begin{array}{l} y \neq \bot \land \\ ((M, x, y, L, t), w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Proof. In the following, all probabilities are over $r \leftarrow \{0,1\}^{l(\lambda)}$ and $w \leftarrow \mathsf{E}^{\mathsf{P}^{\star}_{\lambda,z,s}, \mathsf{V}_r}(1^\lambda, (M, x, t, L))$, and we let y and $statement_i$ for $i \in [m]$ be determined by r in each probability, namely, $y = \langle \mathsf{P}^{\star}_{\lambda,z,s}, \mathsf{V}_r \rangle(1^\lambda, (M, x, t, L))$ and $statement_i$ is the statement used by $\mathsf{P}^{\star}_{\lambda,z,s}$ for the ith sub-protocol with V_r. Additionally, we let $wit_1, \ldots, wit_m, Y$ be the output of $\mathsf{E}_{\mathsf{inner}}$ during the execution of E in each probability.

Suppose by way of contradiction that there exists a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[y \neq \bot \land ((M, x, y, L, t), w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}\right] > 1/p(\lambda).$$

We can rewrite this probability as

$$\Pr\left[\begin{array}{l} y \neq \bot \land \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \land \\ ((M, x, y, L, t), w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right] + \Pr\left[\begin{array}{l} y \neq \bot \land \\ \exists i \in [m]\ (statement_i, wit_i) \notin R_{\mathsf{upd}} \land \\ ((M, x, y, L, t), w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right]$$

$$\leq \Pr\left[\begin{array}{l} y \neq \bot \land \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \land \\ ((M, x, y, L, t), w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right] + \mathsf{negl}_{\mathsf{inner}}(\lambda),$$

by Claim 6.5 above. As $\mathsf{negl}_{\mathsf{inner}}(\lambda) < 1/(2p(\lambda))$ for infinitely many $\lambda \in \mathbb{N}$, this implies that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} y \neq \bot \,\wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \end{array}\right] > \frac{1}{2p(\lambda)}.$$

Furthermore, by a standard averaging argument, it holds that

$$\Pr\left[\begin{array}{l} y \neq \bot \,\wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ \mathsf{E}\ \text{halts after}\ 4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M)\ \text{steps} \end{array}\right] \leq \frac{1}{4p(\lambda)}.$$

Otherwise, the expected work done by E must be greater than $q(\lambda, t \cdot p_M)$, in contradiction with Claim 6.4. This implies that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} y \neq \bot \,\wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ \mathsf{E}\ \text{halts within}\ 4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M)\ \text{steps} \end{array}\right] > \frac{1}{4p(\lambda)}. \tag{6.3}$$

Given this, consider the following non-uniform adversary $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$. At a high level, we will show that on input $pp \leftarrow \mathrm{C.Gen}(1^\lambda, n)$ and $h \leftarrow \mathsf{H}_\lambda$, A will either break the soundness of C or the collision-resistance of H with at least the probability above. In its non-uniform advice, A_λ will have hardcoded the code of $\mathsf{P}^*_{\lambda,z,s}$, the statement (M,x,t,L), and the value of $p(\lambda)$.

$\mathsf{A}_\lambda(pp,h)$:

(1) Sample $r \leftarrow \{0,1\}^{l(\lambda)}$. Let $\mathsf{V}_{pp,h,r}$ be the verifier that uses (pp,h) as its first message and the string r for all other random bits needed.
(2) Run the interaction $y = \langle \mathsf{P}^*_{\lambda,z,s}, \mathsf{V}_{pp,h,r}\rangle(1^\lambda,(M,x,t,L))$. If $y = \bot$, then abort and output $\bot$. Otherwise, let $Y, \pi, State_{\mathsf{final}}, V^{\mathsf{rd}}_{\mathsf{final}}$ be the final message sent by $\mathsf{P}^*_{\lambda,z,s}$.
(3) For at most $4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M)$ steps, run $w \leftarrow \mathsf{E}^{\mathsf{P}^*_{\lambda,z,s}, \mathsf{V}_{pp,r}}(1^\lambda,(M,x,t,L))$. If E does not output within $4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M)$ steps, then abort and output $\bot$. Otherwise, let $wit_1, \ldots, wit_m$ be the witnesses output by $\mathsf{E}_{\mathsf{inner}}$ for statements $statement_1, \ldots, statement_m$.
(4) If there exists an $j \in [m]$ such that $(statement_j, wit_j) \notin R_{\mathsf{upd}}$, then abort and output $\bot$. Otherwise, parse each witness wit_j as containing an initial set of states and words read $(State^{(j)}, V^{\mathsf{rd},(j)})$, as well as a sequence of updates. Let $u_1, \ldots, u_t$ be the sequence of t updates obtained across all m witnesses. For each update $i \in [t]$ we now have the following values and notation:
   - The values $State_i, Op_i, S_i, V^{\mathsf{wt}}_i$ from each step of E's emulation.
   - The extracted update $u_i = (digest_i, V^{\mathsf{prev}}_i, V^{\mathsf{rd}}_i, \pi_i, \tau_i)$.
   - Let V_i be a tuple of $|S_i|$ values, where the jth value is that of V^{rd}_i or V^{wt}_i according to the corresponding operation given by Op_i.

   Last, we have the following starting values:
   - The starting values $(State_0, V^{\mathsf{rd}}_0)$ defined by E.
   - The initial digest computed by V, denoted $digest_0$.

   We will be using this notation throughout the proof.
(5) Check that E's emulation is consistent with the extracted updates. Specifically, let $K_0 = 0$ and let K_j be the number of updates in sub-statements 1 through j for each $j \in [m]$.

If there exists a $j \in [m]$ such that $(State^{(j)}, V^{\mathrm{rd},(j)})$ is not equal to $(State_{K_{j-1}}, V^{\mathrm{rd}}_{K_{j-1}})$, then let j be the smallest such index and output $((State^{(j)}, V^{\mathrm{rd},(j)}), (State_{K_{j-1}}, V^{\mathrm{rd}}_{K_{j-1}}))$. Similarly, if $(State_{\mathrm{final}}, V^{\mathrm{rd}}_{\mathrm{final}}) \neq (State_t, V^{\mathrm{rd}}_t)$, then output these four values.

(6) Next, A_λ emulates the computation of $M(x, w)$. To avoid confusion with the values in the extracted update, we will use a superscript "?" to denote the values computed in this emulation. Let $State^?_0$ be a tuple containing the initial RAM state, $V^{\mathrm{rd}?}_0 = (\perp)$, and $D^? = x||w$ be the initial memory string for use by M.

For $i = 1, \ldots, t$, do the following:

(a) Compute $(State^?_i, Op^?_i, S^?_i, V^{\mathrm{wt}?}_i) = \text{parallel-step}(M, State^?_{i-1}, V^{\mathrm{rd}?}_{i-1})$.

(b) Read from and write to $D^?$ by running $V^{\mathrm{rd}?}_i = \mathrm{access}^{D^?}(Op^?_i, S^?_i, V^{\mathrm{wt}}_i)$.

Let $Y^?$ be the tuple containing the first $L^0 = \lceil L/\lambda \rceil$ words of $D^?$, and let $y^?$ be the concatenation of the first *outlen* bits from $Y^?$, where *outlen* is the output length specified by $State^?_t$.

(7) If there exists an index i such that $V^{\mathrm{rd}}_i \neq V^{\mathrm{rd}?}_i$, then let i be the smallest such index. Compute a digest of the empty partial string $(ptr^?, digest^?_0) = \mathrm{C.Hash}(pp, D_\perp)$ and then compute $(\perp, \pi^?) = \mathrm{C.Open}(pp, ptr^?, S_i)$. Output

$$\left(i-1, \left\{(digest_j, S_j, V_j, \tau_j)\right\}_{j\in[i-1]}, digest_0, S_i, (\perp)^{|S_i|}, \pi^?, V^{\mathrm{prev}}_i, \pi_i\right).$$

(8) If $Y \neq Y^?$, then compute a digest of the empty partial string $(ptr^?, digest^?_0) = \mathrm{C.Hash}(pp, D_\perp)$ and then compute $(\perp, \pi^?) = \mathrm{C.Open}(pp, ptr^?, [L^0])$. Output

$$\left(t, \left\{(digest_j, S_j, V_j, \tau_j)\right\}_{j\in[t]}, digest_0, [L^0], (\perp)^{L^0}, \pi^?, Y, \pi_{\mathrm{final}}\right).$$

(9) Otherwise, abort and output $\perp$.

To analyze the success of A in breaking the soundness of H and C, in the subsequent subclaims, we argue that (1) A_λ runs in (strict) polynomial time; (2) if A_λ outputs in step 5, then A_λ finds a collision in h; (3) if A_λ outputs in steps 7 or 8, then A_λ finds values that breaking the soundness of C; and (4) if A_λ reaches step 9, then it must be the case that $((M, x, y, L, t), w) \in \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}$.

Given these subclaims, we can conclude the proof as follows: First, note that A_λ outputs in steps 5, 7, 8, or 9 whenever $y \neq \perp$, $(statement_i, wit_i) \in R_{\mathrm{upd}}$ for all $i \in [m]$, and E halts within $4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M)$ steps. We can break this event into two cases as

$$\Pr\left[\begin{array}{l} y \neq \perp \wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathrm{upd}} \wedge \\ \mathsf{E} \text{ halts within } 4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M) \text{ steps} \wedge \\ ((M, x, y, L, t), w) \in \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right] + \Pr\left[\begin{array}{l} y \neq \perp \wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathrm{upd}} \wedge \\ \mathsf{E} \text{ halts within } 4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M) \text{ steps} \wedge \\ ((M, x, y, L, t), w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right].$$

By Subclaim 6.12, the first term is greater than the probability that A_λ outputs in step 9. By Equation (6.3), the second term is greater than $1/(4p(\lambda))$. Putting these together, we get that the probability that A_λ outputs in step 5, 7, or 8 is greater than $1/(4p(\lambda))$. It then follows from Subclaims 6.8, 6.9, and 6.11 that for infinitely many $\lambda \in \mathbb{N}$, A_λ runs in polynomial time and either outputs a collision in H or in C with probability at least $1/(4p(\lambda))$. As A directly implies an adversary A' that either gets pp or h as input and simulates the other input for A, this implies that A can be used to break the soundness of H or of C with probability at least $1/(4p(\lambda))$, in contradiction.

Subclaim 6.8. *There exists a polynomial q_{A} such that for every $h \in \mathsf{H}_\lambda$ and pp in the support of* C.Gen$(1^\lambda, n)$*, the running time of $\mathsf{A}_\lambda(pp, h)$ is at most $q_{\mathsf{A}}(\lambda)$ for all $\lambda \in \mathbb{N}$.*

Proof. The running time of A_λ is bounded by the sum of (1) the time to run $\langle \mathsf{P}^?_{\lambda,z,s}, \mathsf{V}_{pp,h,r}\rangle(1^\lambda, (M, x, t, L))$, (2) the total amount of time A_λ spends running E, (3) the time to check that all $(statement_i, wit_i)$ pairs are in R_{upd}, (4) the time to check for an output in step 5, (5) the time to emulate the execution of M, and (6) the time to check for and compute an output in steps 7 and 8. We separately argue that each of these run in at most polynomial time in λ, $|(M,x)|$, L, p_M, t, which are each bounded by a fixed polynomial in λ as $|(M,x)| \leq \lambda$, $L \leq \lambda$, and $t \cdot p_M \leq |x|$.

First, (1) is bounded by a polynomial in λ, since $\mathsf{P}^?_{\lambda,z,s}$ runs in polynomial time for any $z, s \in \{0,1\}^*$ and both the communication complexity and running time of $\mathsf{V}_{pp,h,r}$ are bounded by a fixed polynomial poly$(\lambda, |(M,x)|, L, p_M, t)$ by Lemmas 6.16, 6.17, and by definition of $\alpha^?$. Next, (2) is bounded by $4 \cdot p(\lambda) \cdot q(\lambda, t \cdot p_M)$ by definition of A_λ, and p, q are polynomials. For (3), it requires checking the at most t updates are valid where each check requires a polynomial amount of work in λ, $|(M,x)|$, p_M, $\log t$ by definition of $\mathsf{L}_{\mathsf{upd}}$ and the eficiency of C. Next, (4) requires comparing $m + 1$ values of containing at most p_M states, where each state is a constant number of words, and p_M words of length λ, and so takes time poly(λ, p_M). Next, (5) takes t steps of computation, each of which takes time bounded by a fixed polynomial in λ, p_M by the definition of PRAM computation. Last, (6) requires $(t + L) \cdot p_M \cdot \lambda$ time to check equality of all corresponding values. Computing the initial digest and opening requires $2\beta(\lambda) \cdot p_M \in$ poly(λ) by eficiency of C. Then, the full output has size at most $t \cdot p_M \cdot$ poly$(\lambda) \in$ poly(λ) and takes at most $t \cdot p_M \cdot$ poly$(\lambda) \in$ poly(λ) time to compute.

As $|(M,x)|$, L, p_M, t are bounded fixed polynomials in λ as above, the (strict) running time of A_λ is bounded by some polynomial $q_{\mathsf{A}}(\lambda)$ for all $\lambda \in \mathbb{N}$.

Subclaim 6.9. *If $\mathsf{A}_\lambda(pp, h)$ outputs in step 5, then A_λ finds values that break the collision-resistance of H.*

Proof. Let $K_0 = 0$ and let K_j for $j \in [m]$ be the number of updates in witnesses 1 through j. Let $hash^{(j)}, hash^{(j)\prime}$ be the hashes given in $statement_j$ for all $j \in [m]$. Suppose that A_λ outputs in step 5, meaning that either there exists some $j \in [m]$ such that $(State^{(j)}, V^{\text{rd},(j)}) \neq (State_{K_{j-1}}, V^{\text{rd}}_{K_{j-1}})$ or $(State_{\text{final}}, V^{\text{rd}}_{\text{final}}) \neq (State_t, V^{\text{rd}}_t)$. We first discuss the former case and then the latter.

In the first case, let j be the smallest index with $(State^{(j)}, V^{\text{rd},(j)}) \neq (State_{K_{j-1}}, V^{\text{rd}}_{K_{j-1}})$. Since A_λ reached step 5, then the output y of $\mathsf{V}_{pp,h,r}$ is not equal to $\perp$, which in particular implies that $hash^{(j-1)\prime} = hash^{(j)}$, and that all m extracted witnesses are valid. Since wit_{j-1} is a valid witness for $statement_{j-1}$ (and $State_{K_{j-1}}$ corresponds to the state after the first K_{j-1} updates), then by definition of $\mathsf{L}_{\mathsf{upd}}$ it holds that $h(State_{K_{j-1}}, V^{\text{rd}}_{K_{j-1}}) = hash^{(j-1)\prime}$. Since wit_j is a valid witness for $statement_j$, then $hash^{(j)} = h(State^{(j)}, V^{\text{rd},(j)})$. Last, since $y \neq \perp$, then $hash^{(j-1)\prime} = hash^{(j)}$. Therefore, A_λ successfully finds a collision.

In the second case, $y \neq \perp$ implies that $hash^{(m)\prime} = h(State_{\text{final}}, V^{\text{rd}}_{\text{final}})$, and the fact that wit_m is valid for $statement_m$ implies that $h(State_t, V^{\text{rd}}_t) = hash^{(m)\prime}$, so this also results in a collision.

Next, we show that whenever A reaches step 6, rather than viewing the extracted witnesses as m separate $\mathsf{L}_{\mathsf{upd}}$ instances, they can be viewed as a single instance corresponding to all t updates. This will show that all t updates are in fact being applied to consecutive digests, which will help us show the subsequent claims analyzing A's attack. In the subsequent claims, we say that $(State, V^{\text{rd}})$

is a *PRAM configuration* if during any step of a PRAM evaluation, the set of states after that step are *State* and the words read in that step are V^{rd}.

Subclaim 6.10. *Let* $hash_{\mathrm{start}} = h(State_0, V_0^{\mathrm{rd}})$ *and* $hash_{\mathrm{final}} = h(State_{\mathrm{final}}, V_{\mathrm{final}}^{\mathrm{rd}})$. *Define*

$$statement_{\mathrm{comb}} = (M, x, t, pp, h, digest_0, hash_{\mathrm{start}}, digest_t, hash_{\mathrm{final}}),$$

and $wit_{\mathrm{comb}} = (State_0, V_0^{\mathrm{rd}}, u_1, \ldots, u_t)$. *If* $\mathsf{A}_\lambda(pp, h)$ *reaches step 6, then* $(statement_{\mathrm{comb}}, wit_{\mathrm{comb}}) \in \mathsf{L}_{\mathsf{upd}}$.

Proof. We start with an independent fact about the $\mathsf{L}_{\mathsf{upd}}$ language, which we will then apply to show that the combined statement in the claim is indeed a valid $\mathsf{L}_{\mathsf{upd}}$ statement. Consider any two R_{upd} instances

$$(statement_1, wit_1) = ((M, x, k_1, pp, h, digest_0, hash_0, digest_1, hash_1), (State_1, V_1^{\mathrm{rd}}, u_1^1, \ldots, u_{k_1}^1)),$$

$$(statement_2, wit_2) = ((M, x, k_2, pp, h, digest_1, hash_1, digest_2, hash_2), (State_2, V_2^{\mathrm{rd}}, u_1^2, \ldots, u_{k_2}^2))$$

that agree on M, x, pp, h, and such that the final digest and hash $(digest_1, hash_1)$ in the first statement matches the initial ones in the second statement. Let $State_{1,\mathrm{final}}$ be the final state computed when verifying $(statement_1, wit_1)$ and let $V_{1,\mathrm{final}}^{\mathrm{rd}}$ be the final words read, given by update $u_{k_1}^1$. In other words, $(State_{1,\mathrm{final}}, V_{1,\mathrm{final}}^{\mathrm{rd}})$ is the final PRAM configuration in the first R_{upd} instance. We claim that if $(State_{1,\mathrm{final}}, V_{1,\mathrm{final}}^{\mathrm{rd}})$ is equal to $(State_2, V_2^{\mathrm{rd}})$ (that is, the initial configuration of the second statement), then we can combine the statements together to get a new valid instance with statement

$$statement' = (M, x, k_1 + k_2, pp, h, digest_0, hash_0, digest_2, hash_2)$$

and witness $wit' = (State_1, V_1^{\mathrm{rd}}, u_1^1, \ldots, u_{k_1}^1, u_1^2, \ldots, u_{k_2}^2)$.

To show this, we first show that every update $i \in [k_1 + k_2]$ in wit' satisfies conditions 1, 2, 3, and 4, of $\mathsf{L}_{\mathsf{upd}}$. These four conditions are defined by starting with $(State_1, V_1^{\mathrm{rd}})$ as the starting PRAM configuration for M and using the updates in the witnesses to iteratively compute $k_1 + k_2$ PRAM steps. Then, for each step i, checks are done that depend on the values of the ith step, the input x, the initial digest $digest_1$ given by the statement, and the values in the ith update. Since $(statement', wit')$ and $(statement_1, wit_1)$ have the same machine M and input x, start with the same initial values $digest_1, State_1, V_1^{\mathrm{rd}}$, and agree at the first k_1 updates, this implies that conditions 1, 2, 3, and 4 hold for the first k_1 steps. For the remaining k_2 steps, we observe that, since the final state PRAM configuration of the first statement matches $(State_2, V_2^{\mathrm{rd}})$, then the values computed for each step of verifying $(statement_2, wit_2)$ are the same as those computed when verifying the final k_2 steps of the combined statement. It follows that every all k_1+k_2 updates satisfy the required conditions.

It remains to show that the $\mathsf{L}_{\mathsf{upd}}$ requirements for $hash_0, hash_2$ are satisfied. We have that $hash_0 = h(State_1, V_1^{\mathrm{rd}})$ as this is a requirement of the first statement being valid. We have that $hash_2$ is a hash of the final configuration for the combined statement, because this final configuration is the same as that of the second statement, as shown above. It follows that $(statement', wit')$ is a valid $\mathsf{L}_{\mathsf{upd}}$ instance.

We observe that the above holds for any number of statements by the same logic, which we will use to show the claim. Suppose that A_λ reaches step 6 and consider any $j \in [m]$. Let

$$statement_j = (M^{(j)}, x^{(j)}, k^{(j)}, pp^{(j)}, h^{(j)}, digest^{(j)}, hash^{(j)}, digest^{(j)\prime}, hash^{(j)\prime})$$

be the jth statement in the interaction between $\mathsf{P}^{?}_{\lambda,z,s}$ and $\mathsf{V}_{pp,h,r}$. Since A_λ did not output in step 2, then the output y of the interaction is not equal to $\bot$, which implies that $(M^{(j)}, x^{(j)}, pp^{(j)}, h) = (M, x, pp, h)$. Since A_λ did not output in step 4, then wit_j is valid for $statement_j$. Since A_λ did not output in step 5, then the PRAM configuration $(State_{K_{j-1}}, V^{\mathrm{rd}}_{K_{j-1}})$ before the start of the j sub-statement matches $(State^{(j)}, V^{\mathrm{rd},(j)})$. Therefore, the m witnesses satisfy all conditions above to "combine" them into a new witness. Based on our claim above, the new statement is

$$statement^0_0 = (M, x, t, pp, h, digest^{(1)}, hash^{(1)}, digest^{(m)0}, hash^{(m)0})$$

with witness

$$wit^{00} = (State^{(1)}, V^{\mathrm{rd},(1)}, u_1, \ldots, u_t),$$

where the new instance corresponds to t updates since $\sum_{j=1}^{m} k^{(j)} = t$ by the fact that $y \neq \bot$.

Recall that our goal is to show that $(statement_{\mathrm{comb}}, wit_{\mathrm{comb}})$ (given in the claim statement) is in R_{upd}. The difference between $statement_{\mathrm{comb}}$ and $statement^{00}$ is in the digests and hashes.

The initial digest and both hashes in $statement^{00}$ are equal to those in $statement_{\mathrm{comb}}$, since these are included in the checks done by the verifier, and so are implied by $y \neq \bot$. For the final digest, we have that $digest^{(m)0}$ (given by $statement_m$) is equal to the digest given by update u_t, which is $digest_t$, since the extracted witnesses are valid. It follows that $statement^{00} = statement_{\mathrm{comb}}$.

For the witnesses, the difference between wit^{00} and wit_{comb} is that for the initial configuration, wit^{00} has $(State^{(1)}, V^{\mathrm{rd},(1)})$, while wit_{comb} has $(state_0, V^{\mathrm{rd}}_0)$. Since A_λ did not output in step 5, these are equal, which concludes the claim.

Subclaim 6.11. *If* $\mathsf{A}_\lambda(pp, h)$ *outputs in step 7 or 8, then* A_λ *finds values that violate the soundness of* C.

Proof. We first show the claim for the case that A_λ outputs in step 7 and at the end discuss how to modify the proof in the case that A_λ outputs in step 8.

Suppose A_λ outputs in step 7, meaning that there exists an index i such that $V^{\mathrm{rd}}_i \neq V^{\mathrm{rd}\star}_i$. Moreover, let i be the smallest such index. In this case, A_λ outputs

$$\left(i-1, \left\{(digest_j, S_j, V_j, \tau_j)\right\}_{j \in [i-1]}, digest_0, S_i, (\bot)^{|S_i|}, \pi^\star, V^{\mathrm{prev}}_i, \pi_i\right).$$

Informally, the values output by A_λ correspond to $i-1$ updates, an opening of locations S_i in $digest_0$, and an opening of location S_i in $digest_{i-1}$ (the digest before the ith update). To show that this breaks the soundness of C, we need to show that (A) all updates and openings are valid, yet (B) V^{prev}_i is not equal to set of values at locations S_i consistent with the $i-1$ updates.

For (A), we first show that the initial opening of $digest_0$ at locations S_i to $(\bot)^{|S_i|}$ with proof $\pi^\star$ is valid. Note that A_λ computes $(ptr^\star, digest^\star_0) = \mathrm{C.Hash}(pp, D_\bot)$ and then $\pi^\star = \mathrm{C.Open}(pp, ptr^\star, S_i)$. By completeness of C, it follows that $\pi^\star$ is valid for $(\bot)^{|S_i|}$ at locations S_i with respect to the digest $digest^\star_0$ computed by A_λ. Since C.Hash is deterministic, it follows that $digest_0 = digest^\star_0$, so we conclude that $\mathrm{C.VerOpen}(pp, digest_0, S_i, (\bot)^{|S_i|}, \pi^\star) = 1$.

Next, using the fact that A reaches step 6, by Claim 6.10 the extracted witnesses form a combined statement $(statement_{\mathrm{comb}}, wit_{\mathrm{comb}})$ in R_{upd} containing all t extracted updates. This implies that the sequence of updates from $digest_0$ up until $digest_{i-1}$ are all valid. Namely,

$$\mathrm{C.VerUpd}(pp, digest_{j-1}, S_j, V_j, digest_j, \tau_j) = 1 \text{ for all } j \in [i-1].$$

This also implies that the proof π_i is a valid opening proof for V^{prev}_i at locations S_i with respect to $digest_{i-1}$. Namely,

$$\mathrm{C.VerOpen}(pp, digest_{i-1}, S_i, V^{\mathrm{prev}}_i, \pi_i) = 1.$$

Thus, the openings and updates output by A_λ are valid.

For the rest of the proof, it will be helpful to define the following notation: Let IND be an index where V_i^{rd} and $V_i^{\mathsf{rd}?}$ are not equal (which must exist by assumption). Let v^{rd}, $v^{\mathsf{rd}?}$, v^{prev}, and ` be the corresponding values at index IND of V_i^{rd}, $V_i^{\mathsf{rd}?}$, V_i^{prev}, and S_i, respectively. Before showing (B), we make the following simplifying observations, which make use of the assumption that $V_i^{\mathsf{rd}} \ , \ V_i^{\mathsf{rd}?}$:

(1) The first i updates and the first i steps in the emulation correspond to the same values, that is, $(Op_j^?, S_j^?, V_j^{\mathsf{wt}?}) = (Op_j, S_j, V_j^{\mathsf{wt}})$ for all $j \leq i$.

    This holds because of the following: The values $(Op_j^?, S_j^?, V_j^{\mathsf{wt}?})$ are computed as a deterministic function of the initial configuration $(State_0^?, V_0^{\mathsf{rd}?})$ and the words read in every step of the emulation done by A . The values $(Op_j, S_j, V_j^{\mathsf{wt}})$ are computed as a deterministic function of the initial configuration $(State_0, V_0^{\mathsf{rd}})$ and words read in the emulation done by E . The initial configurations are equal by definition, that is, $(State_0^?, V_0^{\mathsf{rd}?}) = (State_0, V_0^{\mathsf{rd}})$. Since i is the first iteration where $V_i^{\mathsf{rd}?} \ , \ V_i^{\mathsf{rd}}$, then the words read in both are the same. The observation follows.

(2) There exists an update $i^0 < i$ with ` □ S_{i^0}.

    This means that update i cannot be the first iteration that accesses location `. This holds, because if $i$ was the first such iteration, then $v^{\mathsf{rd}}$ would be the value in location ` of $x||w$ by definition of w, as would $v^{\mathsf{rd}?}$, in contradiction. Note that this relies on observation 1 above to use the fact that the locations accessed in the extracted updates and the emulation are the same.

Now, to show (B), we claim that $v^{\mathsf{prev}} \ , \ v^{\mathsf{rd}?}$, yet $v^{\mathsf{rd}?}$ is the value consistent with the updates that we "expect" to open at location ` with respect to $digest_{\,i-1}$. To show that $v^{\mathsf{prev}} \ , \ v^{\mathsf{rd}?}$, we note that because the ith update is valid with respect to R_{upd}, this implies that v^{prev} is either equal to □ or v^{rd}. However, $v^{\mathsf{rd}?}$ is not equal to v^{rd} by assumption. Additionally, $v^{\mathsf{rd}?}$ cannot be equal to □, since it starts off as a value in $x||w$ and is never updated to a non-□ value.

To formalize the notion that $v^{\mathsf{rd}?}$ is the value we expect to open, recall that $v^{\mathsf{rd}?}$ is the value in location ` of $D^?$ at step $i$. We argue that at every step starting from the first time ` is accessed, the value at location ` in $D^?$ is consistent with the $i-1$ extracted updates above. This will show that $v^{\mathsf{rd}?}$ is the value we expect to be at `. Consider the first such update $i_0 < i$ that accesses location `, which is guaranteed to exist by observation 2 above. If ` is read during update i_0 (as specified by Op_{i_0}), then the corresponding update value is the value at location ` in $x||w$ by definition of $w$, which by definition is given by $V_{i_0}^{\mathsf{rd}}$. Otherwise (when update $i_0$ writes to `), the value written to $D^?$ at location ` is given by $V_{i_0}^{\mathsf{wt}?}$, and $V_{i_0}^{\mathsf{wt}?} = V_{i_0}^{\mathsf{wt}}$ by observation 1 above. By the way that $\mathsf{A}_\lambda$ updates $D^?$ throughout the emulation, all subsequent reads and writes to ` in $D^?$ are consistent with the extracted updates. This implies that $v^{\mathsf{rd}?}$ is the value read from or written to ` during the last update $i^0 < i$ that accessed `, which in turn is the value to which we expect ` to open.

This completes the proof of the claim that if A_λ outputs in step 7, then it finds values that violate soundness of C. We conclude by discussing the case where A_λ outputs in step 8, which follows by similar logic. In this case, we are given that $Y \ , \ Y^?$ (rather than $V_i^{\mathsf{rd}} \ , \ V_i^{\mathsf{rd}?}$ as above). To show that the output in step 8 violates soundness, we need to argue all updates and openings are valid, yet for some ` $\leq L^0$, the `th value Y is not the value we expect to open at location i with respect to $digest_t$. Let $Y_`$ be this value, and let $Y_`^?$ be the corresponding value in $Y^?$. The initial opening and all t updates (before location ` is opened to its value in $Y$) are valid by identical logic as above. The final opening $\pi_{\mathsf{final}}$ is accepting, since $\mathsf{V}_{p\ p,h,r}$ outputs a non-□ value. Next, to argue that $Y_`$ is not the value we expect to open, we can show that $Y^?$ is the value we expect to open. If location ` is never accessed, then it would follow that $Y_` = Y_`^?$, since both would be the corresponding word in $x||w$, so it follows that there must be some previous access for location ℓ. Therefore, the same logic used in the above argument holds.

Subclaim 6.12. *If A_λ outputs ⊥ in step 9, then it holds that* $((M, x, y, L, t), w) \in R_U^{PRAM}$.

Proof. When A_λ does not output in step 2, it holds that $y \neq \bot$, so the final state $State_{final}$ must be halting. Since A_λ does not output in step 5, then $State_{final}$ is equal to the final state $State_t$ computed by E in the extraction. It follows that $State_t$ is a halting state.

When A_λ does not output in step 7, it holds that $V_i^{rd} = V_i^{rd\star}$ for all $i \in [t]$. Since $State_0 = State_0^\star$, $V_0^{rd} = V_0^{rd\star}$, and parallel-step is a deterministic function, this implies that $State_t^\star$ computed by A_λ is equal to $State_t$, which corresponds to a halting state as argued above. Moreover, the emulation done by A_λ perfectly emulates the computation of $M(x, w)$, so it is the case that $M(x, w) = y^\star$ within t steps, so $((M, x, t, y^\star), w) \in R_U^{PRAM}$. To show that $y = y^\star$, recall that $y^\star$ is the first *outlen* bits of $Y^\star$, where *outlen* is the output length specified by $State_t^\star = State_t$. We have that $Y = Y^\star$ whenever A_λ does not output in step 8. Moreover, y is the concatenation of the first *outlen* bits of Y, which follows because A does not abort in step 2. It follows that $y = y^\star$. Putting everything together, if A_λ outputs in step 9, then it follows that $((M, x, y, L, t), w) \in R_U^{PRAM}$, as required.

This completes the proof of Claim 6.7.

This completes the proof of Lemma 6.3.

Lemma 6.13 (Prover Efficiency). *There exists a polynomial q such that for any $\lambda \in \mathbb{N}$ and $((M, x, y, L, t), w) \in R_U^{PRAM}$ where M uses p_M processors and has access to $n \leq 2^\lambda$ words in memory, it holds that*

$$\mathrm{depth}_P(1^\lambda, (M, x, t, L), w) \leq t + (\alpha^\star)^2 \cdot |(M,x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M))$$

using at most $4 \cdot p_M \cdot \beta + \rho^\star \cdot \gamma \log t \leq (p_M + \alpha^\star \cdot \rho^\star) \cdot q(\lambda, \log(t \cdot p_M))$ *processors.*

Proof. The work of P can be split into initialization, running Compute-and-prove, and then proving the output. We first focus on the prover's complexity for initialization and proving the output, specified in Steps 2, 3, and 5 in Figure 4.

For Step 2 of initialization, the prover computes the initial state $State_{start}$ for M, the set $V_{start}^{rd} = (\bot)$, the parameter γ, and the hash $hash_{start}$. Both $State_{start}$ and V_{start}^{rd} can be computed in time $O(\lambda)$, and since $hash_{start}$ corresponds to hashing a single state and word, it can also be in time $\beta \in \mathrm{poly}(\lambda)$ (see parameters paragraph). To compute γ, the prover needs to compute the following parameters: First, the prover can compute the hash eficiency parameter $\beta = \beta(\lambda)$ given the security parameter λ. Next, we recall the following parameters based on the definition of L_{upd}, which can be eficiently computed at the start of the protocol given the security parameter λ, time bound t, and processors p_M used by the machine M: The length of L_{upd} statements for at most t updates is at most $\ell_{upd}(\lambda, |(M,x)|, t) \in \log t + |(M,x)| + \mathrm{poly}(\lambda)$, and when using $p_{upd}(\lambda, p_M) = \beta \cdot p_M$ processors, the verification procedure takes time at most $t_{upd}(\lambda, |(M,x)|, t) = t \cdot \beta \cdot |(M,x)| \cdot \mathrm{poly}(\lambda, \log t)$. Given these parameters, we can compute $\alpha^\star = \alpha(\lambda, \ell_{upd}, t_{upd}, p_{upd})/t$ and finally $\gamma = \alpha^\star + 1$, which the rest of the protocol depends on. All of these parameters can be eficiently computed in polynomial time in the input length on a single processor, so in total this step requires $\mathrm{poly}(\lambda) + \mathrm{polylog}(\lambda, |(M,x)|, p_M, t) \in \mathrm{poly}(\lambda, \log(t \cdot p_M))$ work with a single processor.

For Step 3 of initialization, the prover needs to compute the initial digest $digest_{start}$ and allocate memory to run M. By Definition 5.4, the work to compute $digest_{start}$ is β. To allocate memory and copy the input x, this takes at most $|x| \in \mathrm{poly}(\lambda)$ time.

In Step 5, the prover needs to open $\lceil L/\lambda \rceil$ locations in the concurrently updatable hash function, which takes $\lceil L/\lambda \rceil \cdot \beta$ work by Definition 5.4. The prover additionally sends $State_{final}$ and V_{final}^{rd} that have size $O(\lambda)$ as they correspond to a halting state for the PRAM computation M. As $\beta \in \mathrm{poly}(\lambda)$, this step takes at most $L \cdot \mathrm{poly}(\lambda)$ time to compute.

Combining the above, everything other than Compute-and-prove requires an additive overhead in depth (with just a single processor) of at most $L \cdot |x| \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$.

It remains to analyze Compute-and-prove. Recall that Compute-and-prove starts m sub-protocols $\Pi_1, \ldots, \Pi_m$. We start by bounding the number of sub-protocols m by $\gamma \log t$ in Claim 6.14. We then argue in Claim 6.15 that, starting at Step 4, P completes all sub-protocols in depth at most $t + \gamma^2 \cdot (\log t + 1) + \beta$ while using a total of $3 \cdot p_M \cdot \beta + m \cdot \rho^{?}$ processors. As $\gamma = \alpha^{?} + 1$, this implies that the total depth of P is $t + (\alpha^{?})^2 \cdot |x| \cdot L \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$ when using a total of $3 \cdot p_M \cdot \beta + \rho^{?} \cdot \gamma \log t \le (p_M + \alpha^{?} \cdot \rho^{?}) \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$ processors.

Last, we recall Remark 8, which states that we can assume without loss of generality that $|(M,x)|$ is bounded by an *a priori* fixed polynomial in λ when proving $\mathsf{L}_{\mathsf{upd}}$ statements regarding M, x, as long as the statements are proven relative to the time bound $t^0 = t + |(M,x)|$ rather than t. If not, then the prover (and verifier) can incur an additive $|(M,x)| \cdot \mathrm{poly}(\lambda)$ delay in depth using a single processor and prove a related statement where it is the case. Therefore, combining this with the above, it follows that there exists a polynomial q such that the total depth of P can be bounded by $t + (\alpha^{?})^2 \cdot |(M,x)| \cdot L \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$ when using a total of $(p_M + \alpha^{?} \cdot \rho^{?}) \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$ processors, as required.

Claim 6.14. *The number of protocols m started by P is at most $\gamma \log t$.*

Proof. Recall that k_i is the number of steps in the ith sub-protocol. By definition of the protocol, it holds that $k_1 = \lfloor t/\gamma \rfloor$. Then at each subsequent call to Compute-and-prove, the number of steps k_i in the ith sub-protocol is equal to $1/\gamma$ times the number of remaining steps (rounding down if necessary), until the number of remaining steps is less than γ. Thus, we can recursively define $k_i = \lfloor (1/\gamma) \cdot (t - \sum_{j=1}^{i-1} k_j) \rfloor$ for all i less than the number of sub-protocols m. For notational convenience, we define $K_i = \sum_{j=1}^{i} k_j$ to be the number of steps computed in the first i sub-protocols.

To bound m, we lower bound the number of steps computed by the first i sub-protocols before hitting the base case. Namely, we show for any $i \in [m-1]$,

$$K_i \ge t \cdot \left(1 - \left(1 - \frac{1}{\gamma}\right)^i\right) - i.$$

We prove this lower bound on K_i via induction on i. The base case of $i = 1$ holds as $(1 - (1 - 1/\gamma)^1) - 1 = 1/\gamma - 1$ and $K_1 = k_1 = \lfloor t/\gamma \rfloor \ge t/\gamma - 1$. For the inductive step, assume the bound holds for $j = i - 1$. Then, the claim follows by the following set of inequalities:

$$\begin{aligned} K_i = K_{i-1} + k_i &= K_{i-1} + \frac{1}{\gamma}(t - K_{i-1}) \\ &\ge K_{i-1}\left(1 - \frac{1}{\gamma}\right) + \frac{t}{\gamma} - 1 \\ &\ge t \cdot \left(1 - \left(1 - \frac{1}{\gamma}\right)^{i-1}\right)\left(1 - \frac{1}{\gamma}\right) + \frac{t}{\gamma} - (i-1) \cdot \left(1 - \frac{1}{\gamma}\right) - 1 \\ &\ge t \cdot \left(1 - \left(1 - \frac{1}{\gamma}\right)^i\right) - i. \end{aligned}$$

Note that m is then defined to be the smallest value such that the number of steps remaining is smaller than $\gamma \log t + 1$, so $t - K_m < \gamma \log t + 1$. Using the above lower bound, we note that for any arbitrary value $m^{?}$, it holds that $t - K_{m^{?}} \le t \cdot (1 - 1/\gamma)^{m^{?}} + m^{?}$. Therefore, if

$t \cdot (1 - 1/\gamma)^{m^\star} + m^\star \le \gamma \log t + 1$ for some $m^\star$, then $t - K_{m^\star} \le \gamma \log t + 1$. As m is the smallest value for which $t - K_m \le \gamma \log t + 1$, this would imply that $m \le m^\star$, since we would have hit the base case before $m^\star$.

Plugging in $m^\star = \gamma \log t$, we get that $t \cdot (1 - 1/\gamma)^{m^\star} + m^\star \le \gamma \log t + 1$. Thus, it follows that $m \le \gamma \log t$.

Claim 6.15. *The prover completes all protocols* $\Pi_1, \ldots, \Pi_m$ *in depth at most* $t + \gamma_2 \cdot (\log t + 1) + \beta$ *while using at most* $3 \cdot p_M \cdot \beta + m \cdot \rho^\star$ *processors in total.*

Proof. For $i \in [m]$, let the ith sub-protocol Π_i have statement $statement_i$ and witness wit_i as defined by the protocol. The prover's depth when considering Π_i consists of (1) k_i steps of computation corresponding to running M, (2) computing the witness wit_i for $\mathsf{P_{sARK}}$, (3) computing the hash $hash^{\mathsf{c}}_i$ for the statement $statement_i$, and (4) running $\mathsf{P_{sARK}}$ to prove the computation.

To compute wit_i for $\mathsf{P_{sARK}}$, P makes k_i pipelined calls to OpenUpdate in parallel to the computation. It follows that performing the computation in (1) and the k_i concurrent calls to OpenUpdate in (2) can together be computed in depth $k_i + \beta$ using $p_M + p_M \cdot \beta$ processors by Definition 5.4. For (3), we note that this corresponds to hashing at most p_M states and words, and so $hash_k$ can be computed in parallel in time β with $p_M \cdot \beta$ processors (see parameters paragraph). As the hash is computed in parallel to the final update in (2) (which also takes β steps), it follows that together (1), (2), and (3) can be done in depth $k_i + \beta$ with $3p_M \cdot \beta$ processors.

We note that steps (1)–(3) happen consecutively for all m protocols, which all together consist of t steps of computation while computing the corresponding updates and hash on the side. Thus, across all protocols, a total of $3 \cdot p_M \cdot \beta$ processors are used for these three steps and these steps all finish by time $t + \beta$.

For (4), we claim that any valid $\mathsf{L_{upd}}$ statement corresponding to $k \le t$ updates can be proven in time $k \cdot \alpha^\star$ using $\rho^\star$ processors. Let $\mathsf{d}(k)$ and $\mathsf{p}(k)$ be the function representing the depth and processors used to prove valid statements corresponding to k updates. It follows that the time to prove such a statement is $(\mathsf{d}(k)/k) \cdot k \le (\mathsf{d}(t)/t) \cdot k$ with $\mathsf{p}(k) \le \mathsf{p}(t)$ processors, since it holds without loss of generality that $\mathsf{d}(k)/k$ is an increasing function in k and that p is increasing (see parameters paragraph for more discussion). However, we defined $\alpha^\star$ as $\mathsf{d}(t)/t$ and $\rho^\star = \mathsf{p}(t)$, which implies the claim. Furthermore, all of these proofs happen simultaneously, so this adds a factor of $m \cdot \rho^\star$ processors to the total computation. It follows that running $\mathsf{P_{sARK}}$ requires depth $k_i \cdot \alpha^\star$ (with one processor) for sub-protocol i.

Putting everything together, at the start of some sub-computation i with T steps remaining, we compute and prove $\lfloor T/\gamma \rfloor$ steps of computation for $i \le m - 1$, or T steps when $i = m$, where recall that we defined $\gamma \triangleq \alpha^\star + 1$. By the above, for each $i \le m - 1$ this requires depth bounded by

$$\lfloor T/\gamma \rfloor + \beta + \alpha^\star \cdot \lfloor T/\gamma \rfloor = \lfloor T/\gamma \rfloor (\alpha^\star + 1) + \beta \le T + \beta.$$

For $i = m$, this requires depth bounded by

$$T + \beta + \alpha^\star \cdot T = T \cdot (\alpha^\star + 1) + \beta = T \cdot \gamma + \beta.$$

For all sub-protocols Π_i, we start recursively computing and proving the remaining $t - \sum_{j=1}^{i-1} k_j$ steps at depth $\sum_{j=1}^{i-1} k_j$. By the above, this implies that protocol Π_i for $i \le m - 1$ finishes at depth

$$\sum_{j=1}^{i-1} k_j + \left(t - \sum_{j=1}^{i-1} k_j \right) + \beta = t + \beta.$$

For protocol Π_m, note that it starts at depth $\sum_{i=1}^{m-1} k_i$ and takes $k_m \cdot \gamma + \beta$ depth to compute and prove by the above. Thus, it completes at depth

$$\sum_{i=1}^{m-1} k_i + k_m \cdot \gamma + \beta = t + k_m \cdot (\gamma - 1) + \beta \leq t + (\gamma \log t + 1)(\gamma - 1) + \beta \leq t + \gamma^2 \cdot (\log t + 1) + \beta$$

as $k_m \leq \gamma \log t + 1$. Thus, all protocols finish within depth $t + \gamma^2 \cdot (\log t + 1) + \beta$, as required.

This completes the proof of Lemma 6.13.

Lemma 6.16 (Verifier Efficiency). *There exists a polynomial q such that for any $\lambda \in \mathbb{N}$, $(M, x, t, L) \in \{0,1\}^*$ where M has access to $n \leq 2^\lambda$ words in memory and p_M processors, it holds that*

$$\mathsf{work}_{\mathsf{V}}(1^\lambda, (M, x, t, L)) \leq \alpha^? \cdot |(M,x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M)).$$

Proof. To bound the work of the verifier, we note that a bound on the length of each message is known to the verifier in advance (as they depend on $\alpha^?$, L, γ, and β, which are all known), so we can assume that the verifier aborts if it receives a message of the wrong length.

To analyze the verifier's eficiency, we have that the verifier first samples $pp \leftarrow \mathsf{C.Gen}(1^\lambda, n)$ and $h \leftarrow \mathsf{H}_\lambda$ and then computes γ, $State_{\mathsf{start}}$, and $V^{\mathsf{rd}}_{\mathsf{start}}$. Sampling pp, h take time $\mathrm{poly}(\lambda)$, and as discussed in the proof of Lemma 6.13, the rest of the values can be computed in time $\mathrm{poly}(\lambda, \log(t \cdot p_M))$.

The rest of the verifier's running time is in running and checking consistency of the m sub-protocols, checking the starting and ending hashes, and verifying the output. The m sub-protocols are computed using $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$, and by succinctness, there exists a polynomial q_{sARK} such that $\mathsf{V}_{\mathsf{sARK}}$ runs in time

$$q_{\mathsf{sARK}}(\lambda, \ell_{\mathsf{upd}}, \log(t_{\mathsf{upd}} \cdot p_{\mathsf{upd}})) \in \mathrm{poly}(\lambda, |(M,x)|, \log(t \cdot p_M)),$$

where we recall that $\ell_{\mathsf{upd}}(\lambda, |(M,x)|, t) \in \mathrm{poly}(\lambda, |(M,x)|, \log t)$ upper bounds the $\mathsf{L}_{\mathsf{upd}}$ statement length, $t_{\mathsf{upd}}(\lambda, |(M,x)|, t) \in t \cdot \mathrm{poly}(\lambda, |(M,x)|, \log t)$ upper bounds the depth to verify a $\mathsf{L}_{\mathsf{upd}}$ statement with at most t updates when using $p_{\mathsf{upd}}(\lambda, p_M) = p_M \cdot \beta$ processors.

Next, checking consistency between the sub-protocols is mostly syntactic and can be done in time $m \cdot |(M,x)| \cdot \mathrm{poly}(\lambda, \log t)$. Checking $digest_0$ and $hash_0$ can be done in time $\mathrm{poly}(\lambda)$, as well as checking $hash^{\mathsf{C}}_m$, since it corresponds to hashing a halting state and single word for the end of the PRAM computation. Similarly, checking that $state_{\mathsf{final}}$ is a halting state can be done in time $O(\lambda)$, as halting states consist of a single PRAM state with a constant number of words. Verifying the output y can be done in time $\lceil L/\lambda \rceil \cdot \beta \in L \cdot \mathrm{poly}(\lambda)$ by the eficiency of the hash function and the fact that $|y| \leq L$.

Putting everything together, we get that the verifier runs in time $m \cdot L \cdot \mathrm{poly}(\lambda, |(M,x)|, \log(t \cdot p_M))$. Since $m \leq \gamma \log t$ by Claim 6.14 and $\gamma = \alpha^? + 1$, this is bounded by

$$\alpha^? \cdot L \cdot \mathrm{poly}(\lambda, |(M,x)|, \log(t \cdot p_M)).$$

Last, by Remark 8, we note that we can assume without loss of generality that the $\mathsf{L}_{\mathsf{upd}}$ statements are relative to a machine M^0 and input x^0 with length bounded by a fixed polynomial in λ and a time bound $t^0 = t + \mathrm{poly}(\lambda, |(M,x)|)$, so long as V has an additional $|(M,x)| \cdot \mathrm{poly}(\lambda)$ factor in its running time to hash (M, x). Therefore, combining this with the above and noting that $\log(t^0) \in \mathrm{poly}(\lambda, \log t)$, the verifier's total running time is at most $\alpha^? \cdot |(M,x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M))$ for a fixed polynomial q.

Lemma 6.17 (Communication Complexity). *There exists a polynomial q such that for any $\lambda \in \mathbb{N}$, $(M, x, t, L) \in \{0,1\}^*$ where M has access to $n \leq 2^\lambda$ words in memory and p_M processors, it holds* *that the length of the transcript produced between* $\mathsf{P}(w)$ *and* V *on common input* $(1^\lambda, (M, x, t, L))$ *is bounded by*

$$\alpha^? \cdot L \cdot q(\lambda, \log(t \cdot p_M)).$$

Proof. The dominating part of the communication comes from the communication in all sub-protocols defined by Compute-and-prove. The rest of the communication has size at most $\text{poly}(\lambda)$ to send pp and h, size $\text{poly}(\lambda)$ to send $State_{\text{final}}, V^{\text{rd}}_{\text{final}}$ (as they correspond to the final state of the computation) and at most $\lambda \cdot \mathsf{d}L/\lambda\mathsf{e} + L \cdot \beta$ ⍰ $L \cdot \text{poly}(\lambda)$ to send the final proof. Put together, this is at most $L \cdot \text{poly}(\lambda)$.

The m sub-protocols in Compute-and-prove are computed using $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$, so they have communication bounded by some fixed polynomial q_{sARK} in λ and $\log(t_{\mathsf{upd}} \cdot p_{\mathsf{upd}})$ by succinctness of $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$, where recall t_{upd} upper bounds the tome to verify the ith $\mathsf{L}_{\mathsf{upd}}$ statement when using p_{upd} processors. Since $t_{\mathsf{upd}}(\lambda, |(M,x)|, t) \leq t \cdot \text{poly}(\lambda, |(M,x)|, \log t)$ when $p_{\mathsf{upd}}(\lambda, p_M) = p_M \cdot \beta$, then this implies that the communication across all protocols is at most $m \cdot q_{\mathsf{sARK}}(\lambda, \log(t_{\mathsf{upd}} \cdot p_{\mathsf{upd}}))$ ⍰ $m \cdot \text{poly}(\lambda, \log(t \cdot p_M))$, where we additionally used the fact that $|(M,x)| \leq n \leq 2^\lambda$. The prover also has to send the statement for each sub-protocol, which adds $m \cdot \grave{}_{\mathsf{upd}}$ ⍰ $m \cdot \text{poly}(\lambda, |(M,x)|, \log t)$ to the communication complexity, where we recall that $\grave{}_{\mathsf{upd}}$ is the upper bound on the $\mathsf{L}_{\mathsf{upd}}$ statement length. By Claim 6.14, $m \leq \gamma \log t$ and $\gamma = \alpha^? + 1$, so all together Compute-and-prove adds

$$\alpha^? \cdot \text{poly}(\lambda, |(M,x)|, \log(t \cdot p_M))$$

to the communication complexity.

Putting everything together, we get a bound of $\alpha^? \cdot L \cdot \text{poly}(\lambda, |(M,x)|, \log(t \cdot p_M))$. Finally, by Remark 8, without loss of generality, we can assume that $|(M,x)|$ is bounded by a fixed polynomial in λ when used in the $\mathsf{L}_{\mathsf{upd}}$ statements, as long as the statements are proven relative to a time bound $t^0 = t + \text{poly}(\lambda, |(M,x)|)$ (rather than t) and the prover and verify incur an additional delay of $|(M,x)| \cdot \text{poly}(\lambda)$ delay (which was taken into account in the proofs of prover and verifier eficiency). Therefore, this implies that $\log t^0$ ⍰ $\text{poly}(\lambda, \log t)$ (since $|(M,x)| \leq n \leq 2^\lambda$), and so the number of rounds and total communication is bounded by $\alpha^? \cdot L \cdot q(\lambda, \log(t \cdot p_M))$ for a fixed polynomial q.

6.4 Non-interactive Protocol

In this section, we give the protocol from Section 6 in the non-interactive setting. Specifically, we show a transformation from any concurrently updatable hash function and succinct non-interactive argument of knowledge (SNARK) to an argument where the multiplicative overhead of the SNARK prover translates into only additive overhead for the resulting prover. Our construction is nearly the same as in the interactive case, though we additionally need to assume that the underlying succinct argument is a SNARK. We formally define SNARKs in Section 3.3.

Let C be a concurrently updatable hash function, let $(\mathsf{G}_{\mathsf{snark}}, \mathsf{P}_{\mathsf{snark}}, \mathsf{V}_{\mathsf{snark}})$ be a SNARK for $\mathsf{L}_{\mathsf{upd}}$ with (α, ρ)-prover eficiency, and let $\mathsf{H} = \{\mathsf{H}_\lambda\}_{\lambda ⍰ \mathbb{N}}$ be a collision-resistant hash function family ensemble. When we mention the prover and verifier (P, V), we refer to the construction of Section 6.2. We now give the high-level details of our construction $(\mathsf{G}_{\mathsf{ni}}, \mathsf{P}_{\mathsf{ni}}, \mathsf{V}_{\mathsf{ni}})$ for $\mathsf{R}_{\mathsf{U}}^{\mathsf{PRAM}}$, emphasizing the key differences from our interactive construction.

- $(crs, \mathsf{st}) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda)$: Let $pp \leftarrow \mathsf{C.Gen}(1^\lambda, n)$ where $n = 2^\lambda$, $h \leftarrow \mathsf{H}_\lambda$, and $(crs_{\mathsf{snark}}, \mathsf{st}_{\mathsf{snark}}) \leftarrow \mathsf{G}_{\mathsf{snark}}(1^\lambda)$. Output (crs, st) where $crs = (crs_{\mathsf{snark}}, pp, h)$ and $\mathsf{st} = (\mathsf{st}_{\mathsf{snark}}, pp, h)$.[12]
- $(y, \pi) \leftarrow \mathsf{P}_{\mathsf{ni}}(crs, (M, x, t, L), w)$: Let $crs = (crs_{\mathsf{snark}}, pp, h)$. Let M^0 be the same as the machine M, except that it specifies $n = 2^\lambda$ as the amount of words in memory it has access to. Without

[12]Note that if the underlying SNARK is publicly verifiable, then $\mathsf{st}_{\mathsf{snark}} = crs_{\mathsf{snark}}$. Then, $crs = \mathsf{st}$, so the resulting non-interactive argument is also publicly verifiable.

sending any messages, run the prover $\mathsf{P}(w)$ on common input (M^0, x, t, L) using (pp, h) as the verifier's first message and crs_{snark} as the common reference string for all underlying SNARKs. Let y be the output of the computation and π be all messages that would have been sent in the protocol. Output (y, π).

- $b \leftarrow \mathsf{V}_{\mathsf{ni}}(\mathsf{st}, (M, x, y, L, t), \pi)$: Let $\mathsf{st} = (\mathsf{st}_{\mathsf{snark}}, pp, h)$. If M uses more than 2^λ words in memory, then output $b = 0$. Otherwise, let M^0 be the same as the machine M, except that it specifies $n = 2^\lambda$ as the amount of words in memory it has access to. Parse π as all messages from P, and run the verifier V for statement (M^0, x, t, y, L) using (pp, h) as the verifier's first message and $\mathsf{st}_{\mathsf{snark}}$ to verify all underlying SNARKs. Let y^0 be the value that V would have output. Output $b = 1$ if $y = y^0$ and $b = 0$ otherwise.

We get the following theorem:

Theorem 6.18. *Suppose there exists a concurrently updatable hash function and a SNARK* $(\mathsf{G}_{\mathsf{snark}}, \mathsf{P}_{\mathsf{snark}}, \mathsf{V}_{\mathsf{snark}})$ *with* (α, ρ)*-prover eficiency for the* NP *language* $\mathsf{L}_{\mathsf{upd}}$*. Then, there exists a tuple* $(\mathsf{G}_{\mathsf{ni}}, \mathsf{P}_{\mathsf{ni}}, \mathsf{V}_{\mathsf{ni}})$ *satisfying niSPARK completeness and argument of knowledge for* NP*, as well as the same eficiency properties as Theorem 6.1.*

Specifically, there exists a polynomial q *such that for all* $\lambda \in \mathbb{N}$ *and* $((M, x, y, L, t), w) \in \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ *where* M *has access to* $n \leq 2^\lambda$ *words in memory and* p_M *processors, the following hold: Let* $\alpha^?$ *and* $\rho^?$ *(formally defined based on* α *and* ρ*) be the multiplicative overhead in depth (with respect to the number of steps) and number of parallel processors used, respectively, by* $\mathsf{P}_{\mathsf{snark}}$ *to prove a statement in* $\mathsf{L}_{\mathsf{upd}}$ *corresponding to at most* t *steps of computation. Then:*

- *The depth of* P_{ni} *is bounded by* $t + (\alpha^?)^2 \cdot |(M, x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M))$ *when using* $(p_M + \alpha^? \cdot \rho^?) \cdot q(\lambda, \log(t \cdot p_M))$ *processors.*
- *The work of* V_{ni} *is bounded by* $\alpha^? \cdot |(M, x)| \cdot L \cdot q(\lambda, \log(t \cdot p_M))$*, and the length of the transcript produced in the interaction between* $\mathsf{P}(w)$ *and* V *is bounded by* $\alpha^? \cdot L \cdot q(\lambda, \log(t \cdot p_M))$*.*

To prove Theorem 6.18, we note that completeness, succinctness, and optimal prover depth follow identically as in the proof of the construction in Section 6. The proof of adaptive argument of knowledge is conceptually similar yet differs in the technical details as the definition of the extractor for both the underlying SNARK and niSPARK are different. As such, we give the full proof of adaptive argument of knowledge in Appendix B.

As we discuss in Remark 2 in Section 3.3, the argument of knowledge property of the underlying SNARK may only hold for certain distributions over the auxiliary input of the malicious prover. In this case, the argument of knowledge property in our construction holds for any distribution Z over the auxiliary input of the malicious prover so long as the SNARK is secure with auxiliary input drawn from $(Z, \mathrm{C.Gen}(1^\lambda), \mathsf{H}_\lambda)$.

7 MAIN RESULTS

We first construct a four-round SPARK in Section 7.1 assuming only collision resistance. Additionally assuming the existence of a SNARK, we construct a *space-preserving*, non-interactive SPARK in Section 7.2.

7.1 Four-round SPARKs

We consider general parallel RAM computations consisting of statements (M, x, y, L, t) where M is a parallel RAM machine using any p_M number of processors. If we instantiate our transformation from Section 6 with a succinct argument where the prover has $\alpha^? = \mathrm{poly}(\lambda, \log(t \cdot p_M))$ overhead in depth while using at most $\rho^? = p_M$ processors, then the transformation of Theorem 6.1 yields a

SPARK for $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$. To capture the requirements we need, we first formalize and define this notion as a depth-preserving succinct argument of knowledge.

Definition 7.1 (Depth-Preserving Succinct Argument of Knowledge). We say that a succinct argument of knowledge (P, V) for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ is *depth-preserving* if there exists a polynomial q such that (P, V) satisfies (α, ρ)-prover eficiency for $\alpha(\lambda, |(M,x,y,L)|, t, p_M) = (t + |(M,x,y,L)|) \cdot q(\lambda, \log(t \cdot p_M))$ and $\rho(\lambda, |(M,x,y,L)|, t, p_M) = p_M$.

In the following theorem, we show that a depth-preserving succinct argument of knowledge yields a SPARK:

Theorem 7.2. *Suppose there exists a concurrently updatable hash function and a depth-preserving succinct argument of knowledge for* NP. *Then, there exists a SPARK for non-deterministic polynomial time PRAM computation.*

Proof. Let $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ be a depth-preserving succinct argument of knowledge for the language $\mathsf{L}_{\mathsf{upd}}$ where $|(M,x)| \in \mathrm{poly}(\lambda)$ by Remark 8. Let α and ρ be the eficiency of $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$. We recall that the length of $\mathsf{L}_{\mathsf{upd}}$ statements for at most t updates is at most $\ell_{\mathsf{upd}}(\lambda, |(M,x)|, t) \in \log t + \mathrm{poly}(\lambda)$, and when using $p_{\mathsf{upd}}(\lambda, p_M) \in p_M \cdot \mathrm{poly}(\lambda)$ processors, the verification procedure takes depth at most $t_{\mathsf{upd}}(\lambda, |(M,x)|, t) = t \cdot \mathrm{poly}(\lambda, \log t)$. Since $(\mathsf{P}_{\mathsf{sARK}}, \mathsf{V}_{\mathsf{sARK}})$ is depth-preserving, this implies that there exists a polynomial q such that

$$\alpha^{?} = \alpha(\lambda, \ell_{\mathsf{upd}}, t_{\mathsf{upd}}, p_{\mathsf{upd}})/t \leq q(\lambda, \log(t \cdot p_M))$$

and $\rho^{?} = \rho(\lambda, \ell_{\mathsf{upd}}, t_{\mathsf{upd}}, p_{\mathsf{upd}}) = p_{\mathsf{upd}} \leq p_M \cdot q(\lambda, \log(t \cdot p_M))$. Theorem 6.1 implies that there exists an interactive protocol (P, V) for $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ that satisfies SPARK completeness and argument of knowledge for NP. Furthermore, plugging the values for $\alpha^{?} = q(\lambda, \log(t \cdot p_M))$ and $\rho^{?} = p_M \cdot q(\lambda, \log(t \cdot p_M))$ into the theorem, this implies that there exists a polynomial q^0 such that the following eficiency properties hold:

- The depth of the prover is bounded by $t + |(M,x)| \cdot L \cdot q^0(\lambda, \log(t \cdot p_M))$ when using $p_M \cdot q^0(\lambda, \log(t \cdot p_M))$ processors.
- The work of the verifier is bounded by $|(M,x)| \cdot L \cdot q^0(\lambda, \log(t \cdot p_M))$ and the length of the transcript produced in the interaction between $\mathsf{P}(w)$ and V is bounded by $L \cdot q^0(\lambda, \log(t \cdot p_M))$.

This immediately implies the SPARK optimal prover depth and succinctness properties.

We will instantiate the transformation in Theorem 7.2 by showing that Kilian's protocol [41] with the parallelizable PCP construction of Reference [11] is a four-round depth-preserving succinct argument of knowledge. This results in a SPARK from collision resistance alone. Furthermore, we will describe how to improve the round complexity of this instantiation. Specifically, when applying the above theorem generically to an r-round succinct argument of knowledge, the round complexity of the resulting SPARK is roughly $r \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$. We will show that when using Kilian's protocol, we can instead preserve the round complexity, resulting in a four-round SPARK.

We next recall Kilian's argument and the PCP we use and then show how this yields a four-round SPARK.

PCPs and Kilian's succinct argument. At a high level, Kilian's argument gives a way to compile a probabilistically checkable proof (PCP) of knowledge into a four-round succinct argument of knowledge. To show that Kilian's argument can be made to be depth-preserving, we will need the PCP to also have a depth-preserving property. We therefore start by defining a depth-preserving PCP of knowledge. In what follows, we use the notation V^{π} to denote a verifier V with oracle access to a proof string π.

Definition 7.3 (Depth-Preserving PCP of Knowledge). A depth-preserving PCP of knowledge for an NP relation R is a pair $(\mathsf{P}_{\mathsf{pcp}}, \mathsf{V}_{\mathsf{pcp}})$ satisfying the following:

- Completeness: For any $\lambda \in \mathbb{N}$, $(x, w) \in R$, and $\pi \leftarrow \mathsf{P}_{\mathsf{pcp}}(1^\lambda, x, w)$, it holds that $\mathsf{V}^{\pi}_{\mathsf{pcp}}(1^\lambda, x) = 1$.
- Proof of Knowledge: There exists a PPT extractor E and a negligible function negl such that for any $\lambda \in \mathbb{N}$, $x \in \{0,1\}^*$, and proof $\pi \in \{0,1\}^*$,
$$\Pr\left[w \leftarrow \mathsf{E}(x, \pi) : \begin{array}{c} \mathsf{V}^{\pi}_{\mathsf{pcp}}(1^\lambda, x) = 1 \\ \wedge (x, w) \notin R \end{array} \right] \leq \mathsf{negl}(\lambda).$$
- Depth-Preserving Prover Efficiency: Let M be the Turing machine that verifies the relation R using p_M processors. There exists a polynomial q such that for any $\lambda \in \mathbb{N}$ and $(x, w) \in R$, the depth of $\mathsf{P}_{\mathsf{pcp}}(x, w)$ is bounded by $t \cdot q(\lambda, |x|, \log(t \cdot p_M))$ when using p_M processors, where t is the depth of $M(x, w)$.
- Verifier Efficiency: Let M be the Turing machine that verifies the relation R using p_M processors. There exists a polynomial q such that for any $\lambda \in \mathbb{N}$, input $(x, w) \in \{0,1\}^*$, and oracle string $\pi \in \{0,1\}^*$, $\mathsf{V}^{\pi}_{\mathsf{pcp}}(1^\lambda, x)$ runs in time $q(\lambda, |x|, \log(t \cdot p_M))$, where t is the running time of $M(x, w)$.

Throughout this section, we will be focusing only on non-adaptive PCPs. In such a PCP, both the set of queries made by the verifier and the decision algorithm used by the verifier do not depend on the answers to previous PCP queries. Thus, a non-adaptive PCP verifier can be viewed as an interactive algorithm without access to π that first outputs a set of query indices I, receives the corresponding answers according to π, and then indicates whether to accept or reject. It will be convenient to view the PCP verifier as such when we specify Kilian's protocol.

We next observe that the PCP of Ben-sasson et al. [11] gives a depth-preserving, non-adaptive PCP of knowledge. To see this, we note that when viewing their construction as a PCP for a specific NP language, it has the property that the PCP can be computed from the tableau of the computation in depth $\mathrm{poly}(\lambda, \log(t \cdot p_M))$ using $t \cdot p_M$ processors. Such a PCP implies a depth-preserving PCP of knowledge by restricting the prover to only use p_M processors at a time, which increases its depth by a factor of t and satisfies the above definition.

To put everything together, we next recall Kilian's succinct argument and discuss its efficiency when instantiated with a depth-preserving PCP. Given any non-adaptive PCP system $(\mathsf{P}_{\mathsf{pcp}}, \mathsf{V}_{\mathsf{pcp}})$ for NP, Kilian's transformation yields a four-round interactive protocol (P, V) defined as follows: Let L be a language with witness relation R_L. The common input to the protocol is $(1^\lambda, x)$ and P receives private input w such that $(x, w) \in R_L$.

(1) V samples a function h from a collision-resistant hash function family and sends h to P.
(2) P computes $\pi \leftarrow \mathsf{P}_{\mathsf{pcp}}(1^\lambda, x, w)$, computes a Merkle tree hash of π using h, and sends the root to V.
(3) V samples a set I of query indices from $\mathsf{V}_{\mathsf{pcp}}(1^\lambda, x)$ and sends it to P.
(4) P opens up the locations in I in the Merkle tree and sends the openings along with the authentication paths to V.
(5) The verifier accepts if and only if (a) $\mathsf{V}_{\mathsf{pcp}}(1^\lambda, x)$ accepts given the openings and (b) all authentication paths are valid.

The above protocol is a four-round succinct argument of knowledge if $(\mathsf{P}_{\mathsf{pcp}}, \mathsf{V}_{\mathsf{pcp}})$ is a PCP of knowledge and h is a collision-resistant hash function. We note that the second message where P computes the PCP proof π with a Merkle tree is the most time-consuming step and is why we need a PCP with an efficient prover. All other steps can be computed in time $\mathrm{poly}(\lambda, |x|, \log(t \cdot p_M))$ for any PCP.

Next, we sketch why Kilian's protocol is depth-preserving when using a depth-preserving PCP. The prover $\mathsf{P}_{\mathsf{kilian}}$ consists of computing a PCP π, computing the Merkle tree root of π, and then opening up locations in the Merkle tree corresponding to the verifier's queries. By definition of a depth-preserving PCP, computing the PCP can be done in depth $t \cdot \mathrm{poly}(\lambda, |x|, \log(t \cdot p_M))$ with p_M processors. This results in a PCP of length $t \cdot p_M \cdot \mathrm{poly}(\lambda, |x|, \log(t \cdot p_M))$. The Merkle root can then be computed in depth $t \cdot \mathrm{poly}(\lambda, |x|, \log(t \cdot p_M))$ with p_M processors. By the bound on the length of the PCP combined with the PCP verifier's efiency, the query locations can be opened in time $\mathrm{poly}(\lambda, |x|, \log(t \cdot p_M))$. It follows that instantiating Kilian's protocol in this way results in a depth-preserving succinct argument of knowledge.

Constructing a four-round SPARK. We now describe our four-round SPARK construction. We assume familiarity with the protocol of Section 6.2, which we denote by $(\mathsf{P}_{\mathsf{spark}}, \mathsf{V}_{\mathsf{spark}})$, and which serves as the basis for the construction. For the underlying succinct argument of knowledge in that protocol, we use Kilian's succinct argument with a depth-preserving PCP of knowledge as described above, which we denote by $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$.

The protocol (P, V) for $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ is defined as follows: The common input to the protocol is $(1^\lambda, (M, x, t, L))$ and P receives private input w such that $((M, x, y, L, t), w) \in \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$ where y is the output of $M(x, w)$ within t steps. When we refer to protocol $(\mathsf{P}_{\mathsf{spark}}, \mathsf{V}_{\mathsf{spark}})$, we mean the protocol with the same inputs.

(1) V computes the first message msg_1 for $\mathsf{V}_{\mathsf{spark}}$ and a hash function h for $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$. V sends (msg_1, h) to P.
(2) Using msg_1, P runs the prover algorithm $\mathrm{Prove}_{\mathsf{spark}}$ through the Compute-and-prove step, which determines statements for the m sub-protocols. For each of the sub-protocols, P uses h to compute the second message of $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$ for the given statements. Recall that this consists of a Merkle tree digest of the PCP for that part of the computation, which P stores explicitly for all protocols. After computing all second messages in parallel, P sends them to V at the same time.
(3) V responds with the third message of $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$ for the m sub-protocols, consisting of indices to open in each PCP.
(4) P opens all relevant locations with authentication paths in the PCPs and sends the results to V along with the final message msg_{final} sent by $\mathsf{P}_{\mathsf{spark}}$.
(5) V accepts and outputs the value y specified by $\mathsf{V}_{\mathsf{spark}}$ if all of the underlying $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$ protocols accept and all conditions checked by $\mathsf{V}_{\mathsf{spark}}$ hold.

As $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$ is a depth-preserving succinct argument of knowledge assuming only the existence of collision-resistant hash functions, the above construction yields the following theorem:

Theorem 7.4 (Restatement of Theorem 1.3). *Suppose there exists a family of collision-resistant hash functions. Then, there exists four-round SPARK for non-deterministic polynomial-time PRAM computation.*

Proof. We consider the protocol (P, V) defined above which uses a depth-preserving succinct argument of knowledge and a collision-resistant hash function family.

The proofs of completeness and argument of knowledge for (P, V) follow identically to the analysis of Theorem 6.1, and the protocol above is defined in four rounds.

Succinctness follows from Theorem 7.2, since the underlying argument is depth-preserving. We briefly discuss prover eficiency. The prover complexity in $(\mathsf{P}_{\mathsf{kilian}}, \mathsf{V}_{\mathsf{kilian}})$, which dominates the prover complexity in the four-round SPARK, comes from the second and fourth messages of the protocol. All other messages by the prover and the verifier can be computed in time $\mathrm{poly}(\lambda, |(M,x)|, L, \log(t \cdot p_M))$. Without waiting for all messages of the protocol, all sub-protocols would have finished by depth $t + \gamma^2 \cdot (\log t + 1) + \beta$ by the analysis of Lemma 6.13. Thus, the second messages of the sub-protocols will finish by this time, so the second message will be sent by time $(t + \gamma^2 \cdot (\log t + 1) + \beta) \leq t + (\alpha^{?})^2 \cdot |(M,x)| \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$. The fourth message simply consists of opening locations in the Merkle trees with authentication paths. Assuming the entire PCP is stored, this can be computed in time $\mathrm{poly}(\lambda, \log(t \cdot p_M))$ for each of m PCPs in parallel. Thus, the total time for the protocol to finish is $t + (\alpha^{?})^2 \cdot |(M,x)| \cdot L \cdot \mathrm{poly}(\lambda, \log(t \cdot p_M))$. Again, as the underlying argument is depth-preserving, this implies that $\alpha^{?} \leq \mathrm{poly}(\lambda, \log(t \cdot p_M))$ and $\rho^{?} = p_M \cdot \mathrm{poly}(\lambda)$ as in Theorem 7.2, so the protocol satisfies optimal prover depth. Thus, the resulting protocol is a valid SPARK for $\mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}}$.

7.2 Non-interactive SPARKs

If we instantiate our transformation with a SNARK, as in Section 6.4, then the resulting protocol is non-interactive. Furthermore, if the SNARK is depth-preserving as in Definition 7.1, then this implies a non-interactive SPARK. For completeness, we define a depth-preserving SNARK and formally state this result below. We note that the proof follows identically to that of Theorem 7.2.

Definition 7.5 (Depth-Preserving SNARK). We say that a SNARK $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ is *depth-preserving* if there exists a polynomial q such that $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ satisfies (α, ρ)-prover eficiency for

$$\alpha(\lambda, |(M,x,y,L)|, t, p_M) = (t + |(M,x,y,L)|) \cdot q(\lambda, \log(t \cdot p_M))$$

and $\rho(\lambda, |(M,x,y,L)|, t, p_M) = p_M$.

Theorem 7.6. *Assuming there exists a concurrently updatable hash function and a depth-preserving SNARK for* NP. *Then, there exists a non-interactive SPARK for non-deterministic polynomial-time PRAM computation.*

Assuming the existence of collision-resistant hash functions, Bitansky et al. [16] show how to transform any (possibly ineficient or preprocessing) SNARK into a complexity-preserving SNARK using recursive proof composition (following ideas of Valiant [55]). We show that, for parallel RAM machines M using p_M processors, their construction gives a depth-preserving SNARK when allowing the prover to use p_M processors as well. The fact that the SNARK is complexity-preserving means that it also preserves the space complexity of the underlying computation up to $\mathrm{poly}(\lambda, \log(t \cdot p_M))$ factors. We isolate this property and refer to it as space-preserving, defined as follows:

Definition 7.7 (Space-preserving). We say that a succinct argument (P, V) for a relation $R \subseteq \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ is *space-preserving* if there exists a polynomial q such that for any $\lambda \in \mathbb{N}$, and $((M, x, y, L, t), w) \in R$ where $M(x, w)$ uses $n \leq 2^{\lambda}$ space and p_M processors, it holds that the space of P is at most $n \cdot q(\lambda, \log(t \cdot p_M))$. We analogously define *space-preserving* for succinct non-interactive arguments $(\mathsf{G}, \mathsf{P}, \mathsf{V})$.

At a high level, the transformation of Reference [16] splits the t-time computation into roughly t parts of size $\mathrm{poly}(\lambda)$ and constructs proofs for each part separately. Each of these proofs are treated as independent of each other and can be computed in parallel. At first, this does not provide any benefit, since the verifier would need to check roughly t distinct proofs. However, they show how to combine multiple proofs by proving the existence of a set of "lower-level" proofs that the verifier would have accepted. Using this idea, they combine proofs recursively in a tree-like fashion of constant-depth until the verifier only has to verify a single proof.

We briefly discuss the proof of this transformation and discuss why the resulting SNARK is depth-preserving. Completeness is straightforward. Proving that this transformation preserves the argument of knowledge property is more subtle and relies on the fact that the SNARK composition only has constant depth (without making stronger assumptions about the knowledge extractor for the underlying SNARK). Succinctness follows as the final proof is simply a single SNARK proof. To show that the resulting SNARK is depth-preserving and space-preserving, we note that even if the underlying SNARK has $\text{poly}(t)$ overhead in time and space for a t-time computation, each individual proof will only require $\text{poly}(\lambda)$ overhead, since the size of each sub-computation is only $\text{poly}(\lambda)$. Thus, the "layer one" proofs (corresponding to the proofs of the main computation) only incur a $\text{poly}(\lambda)$ multiplicative overhead in the underlying depth and space, and at most $\text{poly}(\lambda)$ proofs will be processed in parallel at any time. Furthermore, the composed proofs at higher levels of the tree can be computed as soon as they are ready, and only $\text{poly}(\lambda)$ proofs will be computed at any time. Once computed, the prover can "forget" the previous parts of the computation, so it only needs to keep information about $\text{poly}(\lambda)$ proofs around, consisting of the current "frontier" in this tree. We refer the curious reader to Reference [16] for more details of this proof.

Using the above SNARK transformation in our non-interactive SPARK construction of Section 6.4, we get the following theorem assuming collision-resistance and *any* SNARK. We emphasize that if the underlying SNARK is publicly verifiable, then so is the resulting SPARK.

Theorem 7.8. *Suppose there exists a family of collision-resistant hash functions and a SNARK. Then, there exists a space-preserving, non-interactive SPARK for non-deterministic polynomial-time PRAM computation.*

Completeness, argument of knowledge, succinctness, and optimal prover depth all follow directly from the analysis of Theorems 6.1, 6.18, and 7.2. As a result, we focus on the space complexity of the prover. The space complexity is dominated by the sub-protocols. The space used by the computation is defined to be n, and all other parts are bounded by $(|M,x| + L) \cdot \text{poly}(\lambda, \log(t \cdot p_M))$. If the underlying SNARK is space-preserving, then it holds that each subprotocol uses at most $n \cdot \text{poly}(\lambda, \log(t \cdot p_M))$ space. There are at most $m \leq (\alpha^? + 1) \log t$ sub-protocols, which are bounded by $\text{poly}(\lambda, \log(t \cdot p_M))$, since the protocol is depth-preserving. Thus, the space used by all sub-protocols is at most $n \cdot \text{poly}(\lambda, L, \log(t \cdot p_M))$ as required.

8 EXTENSIONS

In this section, we discuss various extensions of our main result.

8.1 Space-preserving Interactive SPARKs

In Section 7.2, we gave a transformation from SNARKs to non-interactive SPARKs that are also space-preserving. As discussed in that section, this relies on a transformation from SNARKs to complexity-preserving SNARKs due to Reference [16], which only works in the non-interactive setting. Specifically, if each intermediate argument in that transformation requires interaction, then this would make the round complexity, and hence communication complexity, depend at least linearly on t. This raises the question, can we construct space-preserving (interactive) SPARKs from weaker assumptions than space-preserving non-interactive SPARKs? We emphasize that the four-round SPARK protocol given in Section 7.1 is not space-preserving. In particular, that construction requires storing an entire PCP for each sub-protocol, so it requires space that depends on the time bound t of the underlying computation rather than the space bound.

Bitanksy and Chiesa [18] posed this question for succinct arguments of knowledge (without the optimal prover depth requirement). They construct four-round complexity-preserving succinct arguments of knowledge by adapting Kilian's four-round argument. Instead of relying on PCPs in Kilian's blueprint, they make use of a one-round complexity-preserving multi-prover interactive proof (MIP) of knowledge. In a MIP, there are many provers, and they are crucially not allowed to interact with each other (otherwise, it would be equivalent to the setting of a single prover). They show how to compile such a MIP into a succinct argument using function commitments. At a high level, function commitments allow the prover to commit to a function without evaluating it at every point, so they use the function commitments to commit to the MIP prover algorithms. In contrast, to commit to a PCP string in Kilian's protocol, the prover needs to compute the full PCP string.

In Reference [18], they show how to construct the required function commitments based only on fully homomorphic encryption (FHE), and so the resulting complexity-preserving succinct argument of knowledge is based only on FHE. By instantiating our SPARK construction of Section 6.2 with their succinct argument, we get the following theorem assuming collision resistance and FHE:

Theorem 8.1. *Suppose there exists a collision-resistant hash function family and a secure FHE scheme. Then, there exists a space-preserving SPARK for non-deterministic polynomial-time (sequential) RAM computations.*

The space-preserving property follows from the same observations as in the non-interactive case. However, we note that the round complexity of the resulting SPARK is $\text{poly}(\lambda, L, \log(t \cdot p_M))$. In short, the trick used in Section 7.1 to construct a four-round SPARK using Kilian's succinct argument does not immediately work to collapse rounds, as the prover needs to do quasi-linear work both to commit to the functions of the MIP provers and to homomorphically evaluate their responses. Additionally, we note that the complexity-preserving succinct argument is private-coin, so the resulting space-preserving SPARK is also private-coin.

Last, we remark that the complexity-preserving succinct argument of Reference [18] is only given for RAM (rather than PRAM computations), so the above theorem is also only stated for sequential RAM computations. We note that it actually holds for computations with moderate parallelism—namely, machines computable in time t with $\text{poly}(\lambda, \log t)$ parallelism. At a high level, this follows, because SPARKs for sequential RAM computation generically give depth-preserving succinct arguments for computation with moderate parallelism by ignoring the parallelism of the underlying computation and treating it as a $t \cdot \text{poly}(\lambda, \log t)$-time sequential computation. Applying our transformation to this results in a SPARK for moderately parallel computations. We leave the extension to full PRAM computation as future work.

Open problems. We comment on open problems left by Bitansky and Chiesa [18], which if resolved would immediately give results for space-preserving SPARKs. The first is to construct complexity-preserving PCPs. Using such a PCP in Kilian's argument would yield a complexity-preserving, *public-coin*, succinct argument. In turn, this can be used to construct a space-preserving, public-coin, four-round SPARK, by the techniques described in Section 7.1. Next, is it possible to construct a complexity-preserving, public-coin, succinct argument without going through PCPs and Kilian's transformation? Again, this would at least give a space-preserving, public-coin SPARK, although not necessarily with constant round complexity.

8.2 Proof Composition

We recall that in the transformation from succinct arguments to SPARKs, the prover proves $m \leq (\alpha^{?} + 1) \cdot \log t$ separate sub-protocols, where recall $\alpha^{?}$ is the overhead in depth of the underlying argument and t is the depth of the computation. This requires that the prover communicate

m proofs, and the verifier needs to check all of them. Even when the underlying argument is depth-preserving, the number of protocols $m \in \text{poly}(\lambda, \log(t \cdot p_M))$ may be undesirable.

In the non-interactive setting, the prover can generically compose proofs such that the prover only has to send—and the verifier only has to verify—a single SNARK proof. Specifically, let $\Pi_1, \ldots, \Pi_m$ be the m underlying SNARK protocols with statements $\mathit{statement}_i$ and witnesses wit_i for each $i \in [m]$. The prover will initially compute proofs $\pi_1, \ldots, \pi_m$ for each statement, which takes at most $t + \text{poly}(\lambda, \log(t \cdot p_M))$ time. At this point, the prover can send a hash of all m statements, witnesses, and proofs to the verifier and additionally use the SNARK to prove that it knows a set of statements, witnesses, and proofs that (1) the original SPARK verifier would have accepted and (2) are consistent with the provided hash. This additional work only incurs an additive $\text{poly}(\lambda, \log(t \cdot p_M))$ delay by the prover, so the resulting protocol still satisfies the optimal prover depth property required by a SPARK. This is a standard proof composition technique (see References [16, 55] for more details), and because this only requires one level of recursive composition, the argument of knowledge property is preserved.

In the interactive setting, proof composition does not generically work as described above to reduce communication and verifier complexity. However, in the case of Kilian's protocol and our construction in Section 7.1, we can do proof composition to reduce communication and verifier complexity at the cost of two extra messages of communication. At a high level, instead of sending the roots of the Merkle tree for all m PCPs, the prover hashes all of the statements and roots together and sends it to the verifier. This takes at most $t + \text{poly}(\lambda, \log(t \cdot p_M))$ time to finish the first prover message. At this point in time, the verifier sends randomness to specify challenge queries for the m PCPs (which can be compressed using a pseudo-random generator). The prover then uses a four-round succinct argument of knowledge to prove that it knows a set of openings consistent with the hash answering all of the PCP queries that the verifier would have accepted. The complex-ity of this statement is only $\text{poly}(\lambda, \log(t \cdot p_M))$, so it only incurs an additional $\text{poly}(\lambda, \log(t \cdot p_M))$ delay in the protocol as required. The argument of knowledge analysis follows similarly to the non-interactive setting. Furthermore, at the end of the protocol, the verifier only needs to check a single succinct argument of knowledge at the cost of an extra round of communication.

8.3 Efficiency Tradeoffs

We note that for some applications, requiring optimal prover depth may not be necessary. There may be a hard constraint on the time to finish the proof (e.g. compute the proof within 1 hour) or on the number of processors (e.g. compute the proof as fast as possible using p processors). We emphasize that the construction in Section 6.2 is flexible to these varying needs, depending on the specific application. Specifically, by choosing γ appropriately (which recall corresponds to the fraction of the remaining computation to compute and prove at each step), we can handle any pre-specified prover running time or achieve the best-possible running time given a fixed number of processors.

9 APPLICATIONS TO VERIFIABLE HARD FUNCTIONS

We observe that any non-interactive SPARK for deterministic computations gives a way to turn any function implemented in the parallel RAM model into a verifiable function that can be computed in roughly the same parallel time. In particular, this implies that any sequential function (one that can be computed in time T but not much faster) can be made into a verifiable delay function (VDF). Furthermore, if the underlying sequential function satisfies some hardness property, such as memory-hardness, then this is preserved in the transformation. In the following, we formally define verifiable hard functions and then show how to construct them using publicly verifiable non-interactive SPARKs for deterministic computations:

9.1 Defining Verifiable Hard Functions

In the subsequent definitions, we make use of the following algorithms with the specified syntax:

- $pp \leftarrow \mathrm{Gen}(1^\lambda)$: A PPT algorithm that on input a security parameter λ outputs public parameters pp. We assume for simplicity that pp contains 1^λ.
- $x \leftarrow \mathrm{Sample}(pp)$: A PPT algorithm that on input a security parameter λ and public parameters pp outputs a string $x \in \{0,1\}^*$.
- $y = \mathsf{Eval}(pp, x)$: A deterministic algorithm that on input a security parameter λ, public parameters pp, and an input $x \in \{0,1\}^*$, outputs a value $y \in \{0,1\}^*$.
- $(y, \pi) \leftarrow \mathrm{EvalWithProof}(pp, x)$: An algorithm that on input a security parameter λ, public parameters pp, and an input $x \in \{0,1\}^*$, outputs a value $y \in \{0,1\}^*$ and a proof $\pi \in \{0,1\}^*$. The value y can be generated by the deterministic algorithm $\mathrm{Eval}(pp, x)$. The second output π can be generated using randomness, so it may not be unique.
- $b \leftarrow \mathsf{Vf}(pp, x, (y, \pi))$: A probabilistic algorithm that on input a security parameter λ, public parameters pp, an input $x \in \{0,1\}^*$, a value $y \in \{0,1\}^*$, and a proof $\pi \in \{0,1\}^*$, outputs a bit b indicating whether to accept or reject.

Using the above syntax, we define a verifiable function in the public parameters model.

Definition 9.1 (Verifiable Function). A *verifiable function* is a a tuple $(\mathrm{Gen}, \mathrm{EvalWithProof}, \mathsf{Vf})$ of algorithms such that the following hold:

- Completeness: For every $\lambda \in \mathbb{N}$, $pp \in \mathrm{Supp}\left(\mathrm{Gen}(1^\lambda)\right)$, and $x \in \{0,1\}^*$, it holds that

$$\Pr\left[\mathsf{Vf}(pp, x, \mathrm{EvalWithProof}(pp, x)) = 1\right] = 1.$$

- Soundness: For every non-uniform PPT algorithm $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$, there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{l} pp \leftarrow \mathrm{Gen}(1^\lambda) \\ (x, y^0, \pi^0) \leftarrow \mathsf{A}_\lambda(pp) \\ (y, \pi) \leftarrow \mathrm{EvalWithProof}(pp, x) \\ b \leftarrow \mathsf{Vf}(pp, x, y^0, \pi^0) \end{array} : \begin{array}{l} b = 1 \\ \wedge\, y \neq y^0 \end{array}\right] \leq \mathsf{negl}(\lambda).$$

Before defining a hard function, we define the notion of a class of algorithms. Recall that an algorithm $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ is a actually sequence of algorithms for each $\lambda \in \mathbb{N}$. A class C is a set of algorithms satisfying some predicate as a function of λ. Also, we recall the distinction between uniform and non-uniform algorithms $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$. A is uniform if for all $\lambda \in \mathbb{N}$, A_λ can be computed by a constant-size PPT Turing machine on input 1^λ. A non-uniform algorithm may not have a constant-size description to eficiently generate A_λ for all $\lambda \in \mathbb{N}$. At a high level, a hard function *can* be computed by a uniform algorithm in an "honest" class whereas it *cannot* be computed even by non-uniform algorithms in an "adversarial" class.

Definition 9.2 (Hard Function). Let $\mathsf{C}^{\mathsf{Honest}}$ and $\mathsf{C}^{\mathsf{Adv}}$ be classes of algorithms. A $(\mathsf{C}^{\mathsf{Honest}}, \mathsf{C}^{\mathsf{Adv}})$-*hard function* is a tuple of algorithms $(\mathrm{Gen}, \mathrm{Sample}, \mathrm{Eval})$ such that the following hold:

- Honest Evaluation: There exists a uniform algorithm $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}} \in \mathsf{C}^{\mathsf{Honest}}$ such that for all $\lambda \in \mathbb{N}$, $pp \in \mathrm{Supp}\,(\mathrm{Gen}(1^\lambda))$, and $x \in \mathrm{Supp}(\mathrm{Sample}(pp))$,

$$\mathsf{A}_\lambda(pp, x) = \mathsf{Eval}(pp, x).$$

- Hardness: For every non-uniform PPT algorithm $\mathsf{A}_0 = \{\mathsf{A}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$,[13] there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{l} pp \leftarrow \mathrm{Gen}(1^\lambda) \\ \mathsf{A}_1 \leftarrow \mathsf{A}_{0,\lambda}(pp) \\ x \leftarrow \mathrm{Sample}(pp) \\ y \leftarrow \mathsf{A}_1(x) \end{array} : \begin{array}{l} \mathsf{Eval}(pp,x) = y \\ \wedge \mathsf{A}_1 \in \mathsf{C}^{\mathsf{Adv}} \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We say that a hard function has *bounded output* if for any pp in the support of $\mathrm{Gen}(1^\lambda)$ and x in the support of $\mathrm{Sample}(1^\lambda)$, it holds that $|\mathsf{Eval}(pp,x)| \leq \lambda$.

In the above definition, we emphasize that for hardness, the non-uniform algorithm A_0 is allowed to do arbitrary polynomial-time pre-processing on the public parameters and then must output a valid algorithm A_1 in the class $\mathsf{C}^{\mathsf{Adv}}$ that breaks security. In particular, this is stronger than a definition where the same adversary must work for all public parameters while also coming from the restricted class $\mathsf{C}^{\mathsf{Adv}}$.

Combining the above two notions, we can define a verifiable hard function in the public parameters model.

Definition 9.3 (Verifiable Hard Function). Let $\mathsf{C}^{\mathsf{Honest}}$ and $\mathsf{C}^{\mathsf{Adv}}$ be classes of algorithms. A *verifiable* $(\mathsf{C}^{\mathsf{Honest}}, \mathsf{C}^{\mathsf{Adv}})$*-hard function* is a tuple $(\mathrm{Gen}, \mathrm{Sample}, \mathrm{EvalWithProof}, \mathrm{Vf})$ such that $(\mathrm{Gen}, \mathrm{Sample}, \mathsf{Eval})$ is a $(\mathsf{C}^{\mathsf{Honest}}, \mathsf{C}^{\mathsf{Adv}})$-hard function and $(\mathrm{Gen}, \mathrm{EvalWithProof}, \mathrm{Vf})$ is a verifiable function.

Comparison with Reference [6]. Alwen and Tackmann [6] propose a definitional framework for moderately hard functions, which has been used in subsequent works defining various notions of memory-hard function (e.g., Reference [2]). The main goal of Reference [6] is to come up with a definition that composes nicely in applications. As such, they assume that both the honest and adversarial executions of a moderately hard function have bounded access to an idealized oracle. They propose an indifferentiability-style definition so when analyzing applications using moderately hard functions, it sufices to consider only the resource usage in an "ideal world" scenario. In contrast, our main goal is to show that applying SPARKs to an arbitrary moderately hard function preserves hardness in a "real world" setting, so we do not want to assume that the function has access to an idealized oracle. However, when applying SPARKs to a specific hard function in an idealized model, it would be beneficial to analyze the specific construction within the indifferentiability framework of Reference [6]. We leave this as important and interesting future work when using specific verifable hard functions in further applications.

9.2 Verifiable Hard Functions from Non-interactive SPARKs

We next give a generic theorem that, at a high level, shows that *any* hard function that can be implemented by a parallel RAM algorithm in parallel time T can be bootstrapped into a verifiable hard function using a publicly verifiable non-interactive SPARK for deterministic computations while nearly preserving the parallel running time and number of processors.

To formalize this, we define a class of parallel RAM algorithms that can be computed in roughly time T with p processors. For any functions $T, p, q \colon \mathbb{N} \to \mathbb{N}$, let $\mathsf{P}^{T,p,q}$ be the class of algorithms such that an algorithm $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ is in $\mathsf{P}^{T,p,q}$ if for all $\lambda \in \mathbb{N}$, A_λ is a parallel RAM algorithm running in parallel time $T(\lambda) + q(\lambda)$ with at most $p(\lambda) \cdot q(\lambda)$ processors. For any $T, p \colon \mathbb{N} \to \mathbb{N}$, we

[13]We note that we could naturally extend this definition to model hardness with respect to a more expensive preprocessing attack, but we define polynomial-time attackers for simplicity.

define
$$\mathsf{HonestP}^{T,p} = \bigcup_{q \in \mathrm{poly}(\lambda+\log(T(\lambda)\cdot p(\lambda)))} \mathsf{P}^{T,p,q}.$$

We assume that for algorithms $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda\in\mathbb{N}}$ in $\mathsf{HonestP}^{T,p}$, the value of $q(\lambda)$ is given by A_λ. We note that other definitions (e.g. the definition of a sequential function from Reference [20]) consider honest algorithms that run in time exactly $T(\lambda)$ with exactly $p(\lambda)$ processors. We allow for additive $\mathrm{poly}(\lambda + \log(T(\lambda)\cdot p(\lambda)))$ terms in the depth (and multiplicative ones in the number of processors) to capture overheads roughly independent of the length of the computation. In particular, we do this to make the class robust under application of a SPARK, which we formalize in the following theorem. One could also separate q into two functions q_1 and q_2 defining the additional overheads in the depth and processors, respectively, but for simplicity, we treat these as a single function.

Theorem 9.4. *Let $T, p \colon \mathbb{N} \to \mathbb{N}$ be eficiently computable functions and let $\mathsf{C}^{\mathsf{Adv}}$ be any class of algorithms. Assuming the existence of publicly verifiable non-interactive SPARKs for deterministic parallel computations, if there exists a $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function with bounded output, then there exists a verifiable $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function.*

By combining this with Theorem 7.8, we get the following:

Corollary 9.5. *Let $T, p \colon \mathbb{N} \to \mathbb{N}$ be eficiently computable functions and let $\mathsf{C}^{\mathsf{Adv}}$ be any class of algorithms. Assuming the existence of collision-resistant hash function families, publicly verifiable SNARKs for* NP*, and a $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function with bounded output, then there exists a verifiable $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function.*

Proof of Theorem 9.4. Let $(\mathsf{Gen}_{\mathsf{hard}}, \mathsf{Sample}_{\mathsf{hard}}, \mathsf{Eval}_{\mathsf{hard}})$ be a $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function with bounded output. Let (G, P, V) be a non-interactive SPARK for deterministic computations. We construct $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{EvalWithProof}, \mathsf{Vf})$ to be a verifiable $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function, defined as follows:

- $pp \leftarrow \mathsf{Gen}(1^\lambda)$: Run $crs_{\mathrm{SPARK}} \leftarrow G(1^\lambda)$ and $pp_{\mathsf{hard}} \leftarrow \mathsf{Gen}_{\mathsf{hard}}(1^\lambda)$. Output $pp = (crs_{\mathrm{SPARK}}, pp_{\mathsf{hard}})$.
- $x \leftarrow \mathsf{Sample}(pp)$: Let $(crs_{\mathrm{SPARK}}, pp_{\mathsf{hard}}) = pp$, and output $x \leftarrow \mathsf{Sample}_{\mathsf{hard}}(pp_{\mathsf{hard}})$.
- $(y, \pi) \leftarrow \mathsf{EvalWithProof}(pp, x)$: Let $(crs_{\mathrm{SPARK}}, pp_{\mathsf{hard}}) = pp$ that specifies security parameter λ, $M = \{M_\lambda\}_{\lambda\in\mathbb{N}}$ be the uniform algorithm from the honest evaluation property of $(\mathsf{Gen}_{\mathsf{hard}}, \mathsf{Sample}_{\mathsf{hard}}, \mathsf{Eval}_{\mathsf{hard}})$, and let $q(\lambda)$ be the value such that M_λ runs in time $T'(\lambda) = T(\lambda) + q(\lambda + \log(T(\lambda)\cdot p(\lambda)))$, where q is a polynomial guaranteed to exist by the definition of $\mathsf{HonestP}^{T,p}$. Output $(y, \pi) \leftarrow P(crs_{\mathrm{SPARK}}, (M_\lambda, (pp_{\mathsf{hard}}, x), \lambda, T^0(\lambda)))$. We additionally define $\mathsf{Eval}(pp, x)$ as $M_\lambda(pp_{\mathsf{hard}}, x)$.
- $b \leftarrow \mathsf{Vf}(pp, x, (y, \pi))$: Let $(crs_{\mathrm{SPARK}}, pp_{\mathsf{hard}}) = pp$ and $M = \{M_\lambda\}_{\lambda\in\mathbb{N}}$ be the uniform algorithm from the honest evaluation property of $(\mathsf{Gen}_{\mathsf{hard}}, \mathsf{Sample}_{\mathsf{hard}}, \mathsf{Eval}_{\mathsf{hard}})$. Output $b \leftarrow V(crs_{\mathrm{SPARK}}, (M_\lambda, (pp_{\mathsf{hard}}, x), y, \lambda, T^0(\lambda)), \pi)$.

We now show that (1) $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval})$ is a $(\mathsf{HonestP}^{T,p}, \mathsf{C}^{\mathsf{Adv}})$-hard function and (2) $(\mathsf{Gen}, \mathsf{EvalWithProof}, \mathsf{Vf})$ is a verifiable function, which completes the proof of the lemma.

For (1), note that, by completeness of (G, P, V), if $(y, \pi) \leftarrow P(crs_{\mathrm{SPARK}}, (M_\lambda, (pp_{\mathsf{hard}}, x), \lambda, T^0(\lambda)))$, then $y = M_\lambda(pp_{\mathsf{hard}}, x) = \mathsf{Eval}_{\mathsf{hard}}(pp_{\mathsf{hard}}, x)$ where $|y| \le \lambda$, since $\mathsf{Eval}_{\mathsf{hard}}$ has bounded output.

We first argue honest evaluation. Since $M \in \mathsf{HonestP}^{T,p}$, it follows that for all $\lambda \in \mathbb{N}$, M_λ runs in time $T^0(\lambda) = T(\lambda) + q(\lambda + \log(T(\lambda)\cdot p(\lambda)))$ using at most $p^0(\lambda) = p(\lambda)\cdot q(\lambda + \log(T(\lambda)\cdot p(\lambda)))$ processors. By eficiency of the non-interactive SPARK, it holds that P runs in time $T^0(\lambda) + \mathrm{poly}(\lambda, |(M_\lambda, x)|, \log(T^0(\lambda)\cdot p^0(\lambda)))$ using at most $p^0(\lambda)\cdot \mathrm{poly}(\lambda, \log(T^0(\lambda)\cdot p^0(\lambda)))$ processors. As $x \in \mathrm{Supp}(\mathsf{Sample}(pp))$, it holds that $|x| \in \mathrm{poly}(\lambda)$. Furthermore, $|M_\lambda|$ is a constant that q may depend on, so we can assume that $|M_\lambda| \le \text{poly}(\lambda, \log T(\lambda))$. It follows that there exist a polynomial q^0 such that for all $\lambda \in \mathbb{N}$, EvalWithProof runs in time $T(\lambda) + q^0(\lambda + \log(T(\lambda) \cdot p(\lambda)))$ using at most $p(\lambda) \cdot q^0(\lambda + \log(T(\lambda) \cdot p(\lambda)))$ processors.

For hardness, suppose there exists a non-uniform PPT adversary $\mathsf{A}_0 = \{\mathsf{A}_{0,\lambda}\}_{\lambda \in \mathbb{N}}$ and a polynomial p_{A} such that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} pp \leftarrow \text{Gen}(1^\lambda) \\ \mathsf{A}_1 \leftarrow \mathsf{A}_{0,\lambda}(pp) \\ x \leftarrow \text{Sample}(pp) \\ y \leftarrow \mathsf{A}_1(x) \end{array} : \begin{array}{l} \mathsf{Eval}(pp, x) = y \\ \wedge\ \mathsf{A}_1 \in \mathsf{C}^{\mathsf{Adv}} \end{array}\right] > 1/p_{\mathsf{A}}(\lambda).$$

Since x is sampled from $\text{Sample}_{\text{hard}}(\text{Gen}_{\text{hard}}(1^\lambda))$ and $y = \mathsf{Eval}_{\text{hard}}(pp, x)$, this implies that A_0 also breaks the hardness of $(\text{Gen}_{\text{hard}}, \text{Sample}_{\text{hard}}, \mathsf{Eval}_{\text{hard}})$, in contradiction.

For (2), we note that completeness of $(\text{Gen}, \text{EvalWithProof}, \mathsf{Vf})$ follows immediately by completeness of (G, P, V). Soundness follows, since the argument of knowledge property of (G, P, V) implies soundness. Specifically, suppose there exists a non-uniform PPT algorithm $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ and a polynomial p such that for all $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} pp \leftarrow \text{Gen}(1^\lambda) \\ (x, y^0, \pi^0) \leftarrow \mathsf{A}_\lambda(pp) \\ (y, \pi) \leftarrow \text{EvalWithProof}(pp, x) \\ b \leftarrow \mathsf{Vf}(pp, x, y^0, \pi^0) \end{array} : \begin{array}{l} b = 1 \\ \wedge\ y \neq y^0 \end{array}\right] > 1/p(\lambda).$$

We construct the adversary $P^\star = \{P^\star_\lambda\}_{\lambda \in \mathbb{N}}$ for the non-interactive SPARK, which has A hardcoded as non-uniform advice. For all $\lambda \in \mathbb{N}$, $P^\star_\lambda(crs_{\text{SPARK}})$ samples $pp_{\text{hard}} \leftarrow \text{Gen}_{\text{hard}}(1^\lambda)$, computes $(x, y^0, \pi^0) \leftarrow \mathsf{A}_\lambda((crs_{\text{SPARK}}, pp_{\text{hard}}))$, computes M_λ and $T^0(\lambda)$, and outputs $((M_\lambda, (pp_{\text{hard}}, x), y^0, \lambda, T^0(\lambda)), \pi^0)$. Because A_λ is PPT, and $\text{Gen}_{\text{hard}}(1^\lambda)$, M_λ, and $T^0(\lambda)$ can be computed in polynomial-time, this implies that $P^\star$ is PPT. Furthermore, by definition of A, we can rewrite the above probability statement to conclude that

$$\Pr\left[\begin{array}{l} crs_{\text{SPARK}} \leftarrow G(1^\lambda) \\ ((M_\lambda, (pp_{\text{hard}}, x), y^0, \lambda, T^0(\lambda)), \pi^0) \leftarrow P^\star_\lambda(crs_{\text{SPARK}}) \\ (y, \pi) \leftarrow P(crs_{\text{SPARK}}, (M_\lambda, (pp_{\text{hard}}, x), \lambda, T^0(\lambda))) \\ b \leftarrow V(crs_{\text{SPARK}}, (M_\lambda, (pp_{\text{hard}}, x), y^0, \lambda, T^0(\lambda)), \pi^0) \end{array} : \begin{array}{l} b = 1 \\ \wedge\ y \neq y^0 \end{array}\right] > 1/p(\lambda).$$

Because Eval is deterministic, the "witness" w is empty, so the output of the computation is unique. By completeness, the output is the value y output by P. Therefore, the argument of knowledge property of (G, P, V) stipulates that any $y^0 \neq y$ cannot be accepted with greater than $1/p(\lambda)$ probability for any polynomial p, in contradiction.

9.3 Applications to VDFs

At a high level, a T-sequential function is a function that can be computed in roughly T time with "moderate parallelism" but cannot be computed any quicker with much more parallelism.

To capture this notion, for any $T : \mathbb{N} \to \mathbb{N}$, define

$$\text{Honest}\mathsf{P}^{T,\text{polylog}} = \bigcup_{p, q \in \text{poly}(\lambda + \log T(\lambda))} \mathsf{P}^{T,p,q}.$$

Note that this is simply $\text{Honest}\mathsf{P}^{T,p}$ restricted to the case where the number of processors p is logarithmic in T, and so this class captures the honest execution of a T-sequential function. We next define an adversarial analog, which is allowed to use many parallel processors,

$$\mathsf{AdvP}^{T} = \bigcup_{\substack{p \in \mathrm{poly}(\lambda + \log T(\lambda)),\\ q \in \mathrm{poly}(\lambda + T(\lambda))}} \mathsf{P}^{T,p,q}.$$

We now formally define a sequential function.

Definition 9.6 (Sequential Function). For any $T\colon \mathbb{N} \to \mathbb{N}$, the tuple (Gen, Sample, Eval) is a T-*sequential function* if there exists an $\epsilon \in (0,1)$ such that it is a $(\mathsf{HonestP}^{T,\mathrm{polylog}}, \mathsf{AdvP}^{(1-\epsilon)\cdot T})$-hard function. We say that a sequential function has *bounded output* if for any $pp \in \mathrm{Supp}\ \ \mathrm{Gen}(1^{\lambda})$ and $x \in \mathrm{Supp}\ \ \mathrm{Sample}(1^{\lambda})$, it holds that $|\mathsf{Eval}(pp, x)| \leq \lambda$.

Next, a verifiable delay function is simply a sequential function that is additionally verifiable, formalized as follows:

Definition 9.7 (Verifiable Delay Function). Let $T\colon \mathbb{N} \to \mathbb{N}$. A T*-verifiable delay function (T-VDF)* is a tuple (Gen, Sample, EvalWithProof, Vf) such that (Gen, Sample, Eval) is a T-sequential function and (Gen, EvalWithProof, Vf) is a verifiable function. In the case where each algorithm takes as input a time bound T, we say the tuple is simply a *verifiable delay function* if it is a T-verifiable delay function for any input T.

We note that previous definitions of VDFs are functions that take as input a time bound T and require that the resulting function is a T-VDF for any valid input T. This models the scenario in practice where you want to "tune" a function that can be computed in a particular time T but not faster with more parallelism. We define a T-VDF with respect to a particular time bound T to capture the case where the underlying sequential function may not be able to be tuned to run in any given time bound.

Corollary 9.8. *Let $T\colon \mathbb{N} \to \mathbb{N}$. Assuming the existence of publicly verifiable non-interactive SPARKs for deterministic computations with moderate parallelism, if there exists a T-sequential function with bounded output, then there exists a T-verifiable delay function.*

Proof. Let $\epsilon \in (0,1)$ be the constant sequentiality gap that is guaranteed to exist for the given T-sequential function. By the definition of a sequential function, there exists a uniform algorithm A in $\mathsf{HonestP}^{T,\mathrm{polylog}}$ that computes the evaluation algorithm of the sequential function. It follows that there exists a polynomials p, q in $\mathrm{poly}(\lambda + \log T(\lambda))$ such that A is in $\mathsf{P}^{T,p,q} \subseteq \mathsf{HonestP}^{T,q}$. By setting $\mathsf{C}^{\mathsf{Adv}} = \mathsf{AdvP}^{(1-\epsilon)\cdot T}$ in Theorem 9.4, we get that there exists a verifiable $(\mathsf{HonestP}^{T,p}, \mathsf{AdvP}^{(1-\epsilon)\cdot T})$-hard function, which implies a hard function that can be computed by an algorithm in P^{T,p,q^0} for a function q^0 in $\mathrm{poly}(\lambda + \log(T(\lambda) \cdot p(\lambda))) \subseteq \mathrm{poly}(\lambda + \log(T(\lambda)))$ as p is in $\mathrm{poly}(\lambda + \log T(\lambda))$. Therefore, $\mathsf{P}^{T,p,q^0} \subseteq \mathsf{HonestP}^{T,\mathrm{polylog}}$, which gives the claim.

For the above corollary, we note that in the case where the sequential function takes as input a time bound T, the resulting verifiable delay function can also take in a time bound T. We note that similar to Corollary 9.5, we can instantiate the SPARKs in Corollary 9.8 based on collision-resistant hash functions and SNARKs for NP.

Candidate sequential functions. We briefly discuss existing candidate sequential functions that can be used in Corollary 9.8. We note that in all cases we discuss, it was already known how to construct VDFs, but we emphasize that our transformation is completely independent of the specific details of the underlying sequential function.

Any iterated sequential function is also a sequential function. An iterated sequential function has the additional structure that some small sequential component is repeated T times to obtain a sequential function with respect to any time bound T. The assumption is that any *a priori* unbounded number of iterations cannot be significantly sped up with parallelism. In other words, it is not possible to make shortcuts in the computation without computing all intermediate outputs in order. Boneh et al. [20] show how to construct VDFs from any iterated sequential function using any succinct non-interative argument for deterministic computations with quasi-linear prover overhead. Candidate iterated sequential functions include iterated hashing and repeated squaring in groups of unknown order [53]. For repeated squaring, more practically eficient VDF constructions are known that make use of the additional algebraic structure [51, 56].

Another approach for constructing sequential functions is using secure hardware. The construction, on input x, simply waits T steps and then outputs the evaluation y of a PRF on x. When implemented using secure hardware, the key for the PRF is kept hidden, so the only way to compute y is to use the hardware, which incurs the time T delay. This construction can be securely realized in software assuming indistinguishability obfuscation and the existence of a sequential *decision* problem (see, e.g., References [19, 47]). Furthermore, this construction can be turned into a VDF by making the secure function additionally output a signature on the pair (x, y). Soundness follows, since the only way to construct valid signatures is to compute the secure function.

It is an interesting open problem to construct new (non-iterated) sequential functions from simpler assumptions. Based on Corollary 9.8, any such construction immediately implies a VDF assuming publicly verifiable non-interactive SPARKs for deterministic computations with moderate parallelism.

Remark 10. We emphasize the importance that the underlying SPARK in the transformation can handle (deterministic) parallel computation that uses $\text{poly}(\lambda, \log T)$ processors. For most iterated functions, it is the case that each iteration can be sped up with parallelism, for example, by using ASICs. However, this amount of parallelism scales only polynomially with the input length, λ, and does not depend more than logarithmically on the total time bound T.

9.4 Applications to Memory-hard VDFs

We next show how publicly verifiable non-interactive SPARKs for deterministic computations can be used to construct memory-hard VDFs. A memory-hard VDF in turn implies publicly verifiable, non-interactive proofs of space [31]. There are various proposed definitions for memory-hardness. Alwen and Serbinenko [5] def ine*cumulative memory complexity*, which stipulates that the average memory usage for a function must be large. Alwen, Blocki, and Pietrzak [2] define a conceptually stronger notion of *sustained memory complexity* that stipulates a function must use large memory for many steps (rather than only on average).

We start by giving an overview of the definitions for cumulative and sustained memory complexity. For a parallel RAM machine M, an input x, and an index $i \in \mathbb{N}$, let $\text{Space}(M, x, i)$ be the number of non-zero words in memory during the ith (parallel) step of the computation of M on input x. The cumulative memory complexity of M is

$$\text{CMC}(M) = \max_x \sum_{i=1}^{\text{depth}_M(x)} \text{Space}(M, x, i),$$

where recall that $\text{depth}_M(x)$ is the parallel running time of M on input x. The s-sustained memory complexity of M is defined as

$$s\text{-SMC}(M) = \max_x \left| \left\{ i \in [\text{depth}_M(x)] : \text{Space}(M, x, i) > s \right\} \right| .$$

It was observed in Reference [2] that for any function f, there exists a machine M that implements f with $s\text{-SMC} \in O(T/\log T)$ where T is the depth required to compute f.

For any $S\colon \mathbb{N} \to \mathbb{N}$, we def ine$\mathrm{CMem}^S$ to be the class of algorithms $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ such that $\mathrm{CMC}(\mathsf{A}_\lambda) \leq S(\lambda) \cdot \mathrm{depth}_{\mathsf{A}_\lambda}$ for all $\lambda \in \mathbb{N}$. Similarly, we define SMem ^{S} be the class of algorithms $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ such that as a function of λ, $S(\lambda)$-$\mathrm{SMC}(\mathsf{A}_\lambda) \in o(\mathrm{depth}_{A_\lambda})$.

For simplicity of presentation, we define the following memory-hardness notions with respect to sustained memory complexity using SMem. However, we emphasize that we could analogously define the notion with respect to cumulative memory complexity using CMem or any other recently proposed memory-hardness definitions such as static memory-hardness [28]. We intuitively define an (S, T)-memory-hard sequential function is one that requires T parallel time to compute and cannot be computed using less than S memory for all but $o(T)$ steps. We formalize this as follows:

Definition 9.9 (Memory-hard Sequential Function). For any $S, T\colon \mathbb{N} \to \mathbb{N}$, the tuple (Gen, Sample, Eval) is a (S, T)-*memory-hard sequential function* if there exists an $\epsilon \in (0, 1)$ such that it is a $(\mathsf{HonestP}^{T,\mathrm{polylog}}, \mathsf{AdvP}^{(1-\epsilon)\cdot T} \cap \mathrm{SMem}^S)$-hard function.

A memory-hard VDF is simply a memory-hard sequential function that is also verifiable, formalized as follows:

Definition 9.10 (Memory-hard Verifiable Delay Function). Let $S, T\colon \mathbb{N} \to \mathbb{N}$. A (S, T)-*memory-hard verifiable delay function* is a tuple (Gen, Sample, EvalWithProof, Vf) such that (Gen, Sample, Eval) is a (S, T)-memory-hard sequential function and (Gen, EvalWithProof, Vf) is a verifiable function. In the case where $S \in \Omega(T/\log T)$ and each algorithm takes as input a time bound T, we say the tuple is simply a *memory-hard verifiable delay function* if it is a (S, T)-memory-hard verifiable delay function for any input T.

Similar to Corollary 9.8, it holds that memory-hardness is also preserved under the transformation of Theorem 9.4.

Corollary 9.11. *Let $S, T\colon \mathbb{N} \to \mathbb{N}$. Assuming the existence of publicly verifiable non-interactive SPARKs for deterministic computations with moderate parallelism, if there exists a (S, T)-memory-hard sequential function, then there exists a (S, T)-memory-hard verifiable delay function.*

Proof. Let $\epsilon \in (0, 1)$ be the constant guaranteed to exist for the given (S, T)-memory-hard sequential function. The corollary follows exactly as in the proof of Corollary 9.8, by setting $\mathsf{C}^{\mathsf{Adv}} = \mathsf{AdvP}^{(1-\epsilon)\cdot T} \cap \mathrm{SMem}^S$ in Theorem 9.4.

We note that by combining the above corollary with Theorem 8.1, we obtain memory-hard verifiable delay functions based on memory-hard sequential functions, collision-resistant hash functions, and SNARKs for NP.

Candidate memory-hard sequential functions. Most constructions of memory-hard sequential functions are proven secure in the (parallel) random oracle model and then instantiated with a suficient hash function $h\colon \{0,1\}^* \to \{0,1\}^\lambda$, which we will use in the remaining discussion. We emphasize that, once instantiated with a concrete hash function, the following candidates are only heuristically secure based on the random oracle methodology. As a result, our resulting transformations are secure under the same assumptions.

Percival [50] introduced the function Scrypt as a candidate memory-hard function. At a high level, Scrypt on input x first performs $T/2$ iterated hashes to generate a "database" D of size $T/2$, where $D[0] = x$ and $D[i] = h(D[i-1])$ for $i = 1, \ldots, T/2-1$. It then continues the hash chain while additionally indexing into this database. Specifically, $D[i] = h(D[i-1] \oplus D[D[i-1] \bmod T/2])$ for $i = T/2, \ldots, T$. The output of the function is defined to be $D[T]$. The honest evaluation of the function stores T words in memory. Intuitively, if an adversary stores much less than $T/2$ words, then if it encounters an index $D[i-1] \bmod T/2$ that is not stored, it will need to recompute this value from the closest stored position, which will take much more time. Indeed, Alwen et al. [4] show that Scrypt requires $\Omega(T^2)$ cumulative memory complexity. Furthermore, Scrypt is also sequential (in the random oracle model), as each subsequent query to h is uniformly random and hard to predict, so it behaves like an iterated random oracle. Using Scrypt, we can construct a VDF with high cumulative memory complexity assuming non-interactive SPARKs for deterministic computations. However, Scrypt does not have high sustained memory complexity, since for any S, it can be computed in time $O(T^2/S)$ using S memory.

A more general class of memory-hard function are based on labelings of directed acyclic graphs (DAGs). Let G_n be a DAG on n vertices $\{v_1, \dots, v_n\}$. The label of a node v_i, denoted ℓ_i, is recursively defined as $\ell_i = h(i, \ell_{p_1}, \dots, \ell_{p_d})$, where $p_1, \dots, p_d$ are the incoming edges to v_i. The function is defined by the graph G_n, the input is a seed to the hash function h, and the out-put is the label of the sink of the graph. The hash function is evaluated in the parallel random oracle model, so algorithms can query multiple points in parallel in one "round." For honest eval-uation, a parallel RAM algorithm can compute the graph labeling function with time complexity that scales with the depth of the graph and parallel complexity that scales with the width.[14] Mem-ory lower bounds in this model are proven via pebbling arguments on the underlying graphs (see, e.g., Reference [5] for more information). The depth of the graph also serves as a lower bound for the parallel time to compute such functions. Thus, non-interactive SPARKs for deterministic computations give a way to make such graph labeling functions verifiable. We emphasize that this implies that many works that give graph labeling constructions (e.g., References [2, 5]) that sat-isfy stricter memory-hardness requirements also are preserved under our framework. Specifically, Alwen et al. [2] construct a function that has s-SMC for $s \in \Omega(T/\log T)$ where T is the depth re-quired to compute the function. Using this function, Corollary 9.11 implies a memory-hard VDF assuming non-interactive SPARKs for deterministic computation.

Finally, as with sequential functions, another approach for constructing memory-hard sequential functions is via secure hardware. We assume that the secure hardware has some *a priori* bounded storage capacity of poly(λ) words, and any further required storage is stored externally to the secure enclave. As in the case of sequential functions, the secure hardware waits at least T time and then outputs a PRF evaluation on the input x. Additionally, the secure hardware can externally store a large randomly generated file and perform a simple "proof of storage" to make sure that it is stored for the entire duration of the execution. This can be implemented, for example, using a Merkle tree to verify that random locations of the file are being stored while only keeping the root of the Merkle tree within the secure enclave for authentication. At a high level, security follows, because the hardware only computes its output if enough time and memory have been used. As the PRF key is hidden, there is no other way to compute the output without running the secure hardware. As for the sequential function, this can be further made verifiable by outputting a signature on the PRF input and output.

We believe it is an interesting open question to construct memory-hard sequential functions in software without random oracles. Based on Corollary 9.11, this immediately gives a memory-hard VDF assuming publicly verifiable non-interactive SPARKs for deterministic computations.

APPENDICES

A WITNESS-EXTENDED EMULATION

In this section, we define the notion of witness-extended emulation for succinct arguments and show that this implies the argument of knowledge definition of Definition 3.2.

[14]We additionally need to account for the parallel time to compute the hash function, which increases the time and parallel complexity by at most a factor of poly(λ).

Recall that for a non-uniform prover $\mathsf{P}^{?} = \{\mathsf{P}^{?}_{\lambda}\}_{\lambda \square \mathsf{N}}$, we let $\mathsf{P}^{?}_{\lambda,z,s}$ denote the machine $\mathsf{P}^{?}_{\lambda}$ with auxiliary input z and randomness s fixed. Additionally, we let

$$View_{\mathsf{V}}^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t))$$

denote the distribution representing the view of V when interacting with $\mathsf{P}^{?}_{\lambda,z,s}$ on input 1^{λ} and (M,x,y,L,t). Additionally, we let $\mathsf{Acc_V}(view)$ be the predicate that outputs 1 if a view *view* is accepting for V and 0 otherwise. The definition below is based on the definition of Lindell [42] and extended to the case of arguments similar to Reference [37]. We additionally modify the definition to capture relations $R \square \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$ similar to Reference [8] as discussed in Section 3.3.

Definition A.1 (Witness-extended Emulation for NP *Arguments).* Let (P,V) be an interactive argument for a relation $R \square \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$. Let WE be a probabilistic machine that is given as input a security parameter 1^{λ}, a statement (M,x,y,L,t), and oracle access to a machine $\mathsf{P}^{?}_{\lambda,z,s}$. We let $\mathrm{WE}_1^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t))$ and $\mathrm{WE}_2^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t))$ denote the first and second outputs of the emulator, respectively.

We say that WE is a *witness-extended emulator for* (P,V) *and* R if there exists a polynomial q such that for every non-uniform probabilistic polynomial-time prover $\mathsf{P}^{?} = \{\mathsf{P}^{?}_{\lambda}\}_{\lambda \square \mathsf{N}}$ and every constant c, there exists a negligible function negl such that for every $\lambda \square \mathsf{N}$, (M,x,y,L,t) with $|(M,x,t,y)| \leq \lambda$, $L \leq \lambda$, and $t \leq \lambda^{c}$, and every $z,s \square \{0,1\}^{\square}$, the following hold:

(1) $\mathrm{WE}^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t))$ runs in expected polynomial time $q(\lambda,t)$.
(2) The view output by WE_1 is identically distributed to the view of V in a real interaction with $\mathsf{P}^{?}_{\lambda,z,s}$. That is, the corresponding distributions satisfy

$$\mathrm{WE}_1^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t)) \equiv View_{\mathsf{V}}^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t)).$$

(3) The probability that WE_1 outputs an accepting view for V, and yet WE_2 does not output a correct witness, is negligible. That is,

$$\Pr\left[\begin{array}{l} \mathsf{Acc_V}\left(\mathrm{WE}_1^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t))\right) = 1 \\ \square \left((M,x,y,L,t),\mathrm{WE}_2^{\mathsf{P}^{?}_{\lambda,z,s}}(1^{\lambda},(M,x,y,L,t))\right) < R \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We next show that the above definition of witness-extended emulation implies the argument of knowledge definition in Section 3.3 for NP relations.

Lemma A.2. *Let* (P,V) *be succinct argument for a relation* $R \square \mathsf{R}_{\mathsf{U}}^{\mathrm{TM}}$. *If there exists a witness-extended emulator* WE *for* (P,V) *and* R, *then* (P,V) *satisfies the argument of knowledge for* NP *condition in Definition 3.2.*

Proof. Using WE, we construct a probabilistic oracle machine E as required. Recall that both E and WE receive as input $(1^{\lambda},(M,x,y,L,t))$ and get oracle access to a prover $P^{?}_{\lambda,z,s}$, while E additionally gets oracle access to a verifier V_r with uniformly sampled randomness fixed to r. Let `(λ) denote the length of the randomness r used by $\mathsf{V}(1^{\lambda},\cdot)$. We define $\mathsf{E}^{\mathsf{P}^{?}_{\lambda,z,s},\mathsf{V}_r}$ as follows:

$\mathsf{E}^{\mathsf{P}^{?}_{\lambda,z,s},\mathsf{V}_r}(1^{\lambda},(M,x,y,L,t))$:

(1) Emulate the view between $P^{?}_{\lambda,z,s}$ and V_r on input $(1^{\lambda},(M,x,y,L,t))$. If V_r rejects in this view, then output $\square$.

(2) Sample $(view, w) \leftarrow \mathrm{WE}^{\mathsf{P}^{\star}_{\lambda,z,s}}(1^\lambda, (M,x,y,L,t))$ until $\mathrm{Acc}_{\mathsf{V}}(view) = 1$ or 2^{2^λ} iterations have passed.
- If $\mathrm{Acc}_{\mathsf{V}}(view) = 1$ at any point, then output the corresponding witness w.
- Otherwise, for all strings $w \in \{0,1\}^t$, output the first one such that $((M,x,y,L,t),w) \in R$ or $\bot$ if none exist.

It remains to prove that E satisfies the argument of knowledge for NP requirements of Definition 3.2. Specifically, let $\mathsf{P}^\star = \{\mathsf{P}^\star_\lambda\}_{\lambda\in\mathbb{N}}$ be a non-uniform probabilistic polynomial-time prover and c be any constant. We need to show that there exists a negligible function negl such that for every $\lambda \in \mathbb{N}$, $(M,x,y,L,t),z,s \in \{0,1\}^*$ with $|(M,x,y,t)| \le \lambda$, $L \le \lambda$, and $t \le |x|^c$, the following hold:

- Running time: $\mathsf{E}^{\mathsf{P}^\star_{\lambda,z,s},\mathsf{V}_r}(1^\lambda,(M,x,y,L,t))$ runs in expected time $q(\lambda,t)$ for some polynomial q (independent of $\mathsf{P}^\star$ and c), where the expectation is over a uniformly chosen $r \leftarrow \{0,1\}^{\ell(\lambda)}$ and the randomness of E.
- Correctness: It holds that

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{\ell(\lambda)} \\ w \leftarrow \mathsf{E}^{\mathsf{P}^\star_{\lambda,z,s},\mathsf{V}_r}(1^\lambda,(M,x,y,L,t)) \end{array} : \begin{array}{l} \langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r\rangle(1^\lambda,(M,x,y,L,t)) = 1 \\ \wedge\ ((M,x,y,L,t),w) \notin R \end{array}\right] \le \mathsf{negl}(\lambda).$$

We next focus on each of these conditions.

Running time. For the running time, we first note that by succinctness of (P,V), there exists a polynomial q_1 such that the number of messages and total communication between $\mathsf{P}^\star_{\lambda,z,s}$ and V_r is bounded by $q_1(\lambda,\log t)$. This also bounds the running time of emulating the interaction between $\mathsf{P}^\star_{\lambda,z,s}$ and V_r given oracle access to each machine. If V_r rejects, then we are done. Otherwise, we define the value

$$\epsilon = \Pr\left[r \leftarrow \{0,1\}^{\ell(\lambda)} : \langle \mathsf{P}^\star_{\lambda,z,s}, \mathsf{V}_r\rangle(1^\lambda,(M,x,y,L,t)) = 1\right],$$

which is greater than 0 in the case that V_r accepts for some choice of r. For the analysis of the expected running time, we note that we continue with probability ϵ where V_r accepts.

In this case, we first try running WE until its first output is an accepting view. By definition of witness-extended emulation, it holds that the first output of WE is identically distributed to the real interaction between $\mathsf{P}^\star_{\lambda,z,s}$ and V, so this means we will run WE at most $1/\epsilon$ times in expectation. By definition of WE, there exists a polynomial q_2 such that each run of WE takes expected $q_2(\lambda,t)$ time. So, this contributes at most $\epsilon\cdot(1/\epsilon)\cdot q_2(\lambda,t) = q_2(\lambda,t)$ to the expected running time.

We last consider the case where 2^{2^λ} independent iterations pass without finding an accepting view. This event occurs with probability $(1-\epsilon)^{2^{2^\lambda}}$ given that V_r initially accepted. In this case, we run in time $2^t\cdot\mathrm{poly}(\lambda,t)$ to emulate M on all choices of w of size at most t. Let B be this time to brute-force, which in particular is bounded by $2^{2^{\lambda/2}}$ for suficiently large λ, since $t \le |x|^c$. Thus, this case contributes a factor of at most $\epsilon\cdot(1-\epsilon)^{2^{2^\lambda}}\cdot B$ to the expected running time. We show that this is in fact bounded by 1, at least for suficiently large λ. In the case that $\epsilon < 1/B$, this clearly holds. When $\epsilon \ge 1/B$, we can bound $(1-\epsilon)^{2^{2^\lambda}} \le (1-1/B)^{B\cdot(2^{2^\lambda}/B)} \le (1/e)^{2^{2^{\lambda/2}}} \le 1/B$, using the fact that $B \le 2^{2^{\lambda/2}}$ for suficiently large λ. Thus, for any value of ϵ, it holds that $\epsilon\cdot(1-\epsilon)^{2^{2^\lambda}}\cdot B \le 1$ for suficiently large λ, so in particular is bounded by some polynomial $q_3(\lambda,t)$ (to account for small values of λ where this is not necessarily bounded by 1).

Putting it all together, we upper bound the expected running time of E by

$$q_1(\lambda,\log t) + q_2(\lambda,t) + q_3(\lambda,t) = q(\lambda,t),$$

for some polynomial q, independent of $\mathsf{P}^\star_{\lambda,z,s}$, as required.

Correctness. For correctness, suppose by way of contradiction that there exists a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} r \leftarrow \{0,1\}^{\ell(\lambda)} \\ w \leftarrow \mathsf{E}^{\mathsf{P}^{*}_{\lambda,z,s},\mathsf{V}_r}(1^\lambda,(M,x,y,L,t)) \end{array} : \begin{array}{l} \langle \mathsf{P}^{*}_{\lambda,z,s}, \mathsf{V}_r \rangle(1^\lambda,(M,x,y,L,t)) = 1 \\ \wedge\ ((M,x,y,L,t),w) \notin R \end{array}\right] > 1/p(\lambda).$$

We show this contradicts the correctness property for WE. For notational convenience, we first define the following events:

- We say WE$_1$ accepts when $\mathsf{Acc}_{\mathsf{V}}(\mathrm{WE}_1^{\mathsf{P}^{*}_{\lambda,z,s}}(1^\lambda,(M,x,y,L,t))) = 1$, and WE$_2$ is valid when $((M,x,y,L,t),\mathrm{WE}_2^{\mathsf{P}^{*}_{\lambda,z,s}}(1^\lambda,(M,x,y,L,t))) \in R$, where the probabilities are over the randomness of WE.
- We say V_r accepts when $\langle \mathsf{P}^{*}_{\lambda,z,s}, \mathsf{V}_r \rangle(1^\lambda,(M,x,y,L,t)) = 1$, and w is valid when $((M, x,y,L,t),w) \in R$, where the probabilities are over a random $r \leftarrow \{0,1\}^{\ell(\lambda)}$ and $w \leftarrow \mathsf{E}^{\mathsf{P}^{*}_{\lambda,z,s},\mathsf{V}_r}(1^\lambda,(M,x,y,L,t))$.

Towards a contradiction, we consider the event where the witness w output by E is valid given that V_r accepts. Let BF be the event that WE$_1$ fails to accept for 2^{2^λ} iterations, at which point E will always output a valid witness w (if one exists). We note that, since M is a Turing machine that runs in time at most t, it can only read the first t bits of its input string. Thus, if any valid witness exists, then there will exist a witness of length at most t, which will be found by brute force search. When BF does not occur, E samples a uniformly random output of WE$_2$ conditioned on WE$_1$ accepting. In the case where there exists any valid witness w for $((M,x,y,L,t),w) \in R$, this implies that

$$\begin{aligned} \Pr[w \text{ is valid} \mid \mathsf{V}_r \text{ accepts}] &= \Pr[\mathrm{BF}] \cdot 1 + (1 - \Pr[\mathrm{BF}]) \cdot \Pr[\mathrm{WE}_2 \text{ is valid} \mid \mathrm{WE}_1 \text{ accepts}] \\ &\geq \Pr[\mathrm{WE}_2 \text{ is valid} \mid \mathrm{WE}_1 \text{ accepts}]. \end{aligned}$$

For the case where there does not exist a valid witness, note that this inequality still holds, as both terms are simply zero. Considering the complement events, this implies that

$$\Pr[w \text{ is invalid} \mid \mathsf{V}_r \text{ accepts}] \leq \Pr[\mathrm{WE}_2 \text{ is invalid} \mid \mathrm{WE}_1 \text{ accepts}].$$

Because $\Pr[\mathrm{WE}_1 \text{ accepts}] = \Pr[\mathsf{V}_r \text{ accepts}]$, it follows that

$$\Pr\left[\begin{array}{l} \mathsf{V}_r \text{ accepts} \\ \wedge\ w \text{ is invalid} \end{array}\right] \leq \Pr\left[\begin{array}{l} \mathrm{WE}_1 \text{ accepts} \\ \wedge\ \mathrm{WE}_2 \text{ is invalid} \end{array}\right].$$

However, this implies that $\Pr[\mathrm{WE}_1 \text{ accepts} \wedge \mathrm{WE}_2 \text{ is invalid}] > 1/p(\lambda)$ for some polynomial p, in contradiction.

We conclude this section by relating our argument of knowledge definition for NP (Definition 3.2) to the standard definition given by Reference [9]. In the standard definition, the extractor E has oracle access to $\mathsf{P}^{*}_{\lambda,z,s}$, always extracts a witness, and runs in expected time $p(\lambda)/(\epsilon(\lambda) - \kappa(\lambda))$ for a polynomial p, where $\epsilon(\lambda)$ is the success probability of $\mathsf{P}^{*}_{\lambda,z,s}$ and $\kappa(\lambda)$ is the knowledge error (where these functions may additionally depend on the statement length).

Recall that Lindell showed that the standard definition for *proofs* of knowledge implies witness-extended emulation for proofs [42]. The difference between that definition of witness-extended emulation for proofs and ours for NP arguments (Definition A.1) is that, in addition to being for arguments rather than proofs, our requirements are for statements (M,x,y,L,t) with $|(M,x,t,y)| \leq \lambda$, $L \leq \lambda$, and $t \leq \lambda^c$. We also allow the emulator to run in time polynomial in λ, t (similar to universal arguments), rather than simply in λ.

We observe that upon making these same modifications to the standard argument of knowledge definition, it follows by Reference [42] that the resulting definition implies witness-extended emulation for arguments. By combining this with Lemma A.2, we conclude that Definition 3.2 is implied by a more standard definition.

B PROOFS FROM SECTION 6.4

Lemma B.1 (Adaptive Argument of Knowledge). $(\mathsf{G}_{\mathsf{ni}}, \mathsf{P}_{\mathsf{ni}}, \mathsf{V}_{\mathsf{ni}})$ *is an adaptive argument of knowledge for* NP.

Proof. Let C be a concurrently updatable hash function, let $\mathsf{H} = \{\mathsf{H}_\lambda\}_{\lambda \in \mathbb{N}}$ be a collision-resistant hash function family ensemble, and let $(\mathsf{G}_{\mathsf{snark}}, \mathsf{P}_{\mathsf{snark}}, \mathsf{V}_{\mathsf{snark}})$ be a SNARK for the language $\mathsf{L}_{\mathsf{upd}}$, given in Figure 2. Consider any non-uniform polynomial-time prover $\mathsf{P}^{\star} = \{\mathsf{P}^{\star}_\lambda\}_{\lambda \in \mathbb{N}}$ and security parameter $\lambda \in \mathbb{N}$. Let $\mathsf{P}^{\star}_{\lambda,z,s}$ denote $\mathsf{P}^{\star}_\lambda$ with auxiliary input z and hardcoded randomness s for any $z, s \in \{0,1\}^{*}$.

Let $(crs, st) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda)$. Recall that a proof in the non-interactive SPARK scheme consists of m sub-proofs, each corresponding to an instance of $\mathsf{L}_{\mathsf{upd}}$, as well as values certifying the output of the computation. As a subroutine to our full extractor, we first construct a probabilistic oracle machine $\mathsf{E}_{\mathsf{inner}}$ that uses extractors for the SNARK to extract witnesses for each sub-proof.

$\underline{\mathsf{E}_{\mathsf{inner}}(crs, z, s)}$:

(1) Compute $((M, x, y, L, t), \pi) = P^{\star}_{\lambda,z,s}(crs)$. Let m be the number of $\mathsf{L}_{\mathsf{upd}}$ statements specified by π, and for each $i \in [m]$, let $(statement_i, \pi_i)$ be ith $\mathsf{L}_{\mathsf{upd}}$ statement and proof, and let Y, π_{final} be the opening and proof certifying the output y, all given by π. Let k_i be the number of updates in each $\mathsf{L}_{\mathsf{upd}}$ statement for $i \in [m]$, and let M have access to n words in memory and p_M processors. Check that $\sum_{i=1}^{m} k_i = t$, $n \leq 2^\lambda$, and that Y consists of $\lceil L/\lambda \rceil$ words, and abort and output $\bot$ if any of these do not hold.
(2) Parse $crs = (crs_{\mathsf{snark}}, pp, h)$. For each $i \in [m]$, define $\mathsf{P}^{\star}_{i,z^0,s}(crs_{\mathsf{snark}})$ to be a SNARK prover with auxiliary input $z^0 = (pp, h, z)$ and randomness s hardcoded that runs $((M, x, y, L, t), \pi) \leftarrow \mathsf{P}^{\star}_{\lambda,z,s}((crs_{\mathsf{snark}}, pp, h))$ and outputs $(statement_i, \pi_i)$ given by π. Let $\mathsf{P}^{\star}_i$ denote this machine without its auxiliary input or randomness specified, and let $\mathsf{E}_{\mathsf{snark},i}$ be the SNARK extractor for $\mathsf{P}^{\star}_i$.
(3) For $i \in [m]$, compute $wit_i \leftarrow \mathsf{E}_{\mathsf{snark},i}(crs_{\mathsf{snark}}, z^0, s)$. Output $(wit_1, \ldots, wit_m, Y, \pi_{\mathsf{final}})$.

In the following claims, we show that (1) $\mathsf{E}_{\mathsf{inner}}$ runs in polynomial time (over the randomness of G_{ni} and its own random coins) and (2) with all but negligible probability (over randomness of G_{ni}, the coins of $\mathsf{E}_{\mathsf{snark}}$, and randomness for V_{ni}), either $\mathsf{P}^{\star}_{\lambda,z,s}$ fails to convinces V_{ni} or for all $i \in [m]$ the witness wit_i extracted by $\mathsf{E}_{\mathsf{snark},i}$ is valid for $statement_i$ with respect to R_{upd}:

Claim B.2. *There exists a polynomial q_{inner} such that for every $\lambda \in \mathbb{N}$ and $z, s \in \{0,1\}^{*}$, the running time of $\mathsf{E}_{\mathsf{inner}}(crs, z, s)$ is at most $q_{\mathsf{inner}}(\lambda, t \cdot p_M)$, where $(crs, st) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda)$, t is given by the statement output by $\mathsf{P}^{\star}_{\lambda,z,s}(crs)$ and p_M is the number of processors used by the machine M given in the statement.*

Proof. $\mathsf{E}_{\mathsf{inner}}$ runs $P^{\star}_{\lambda,z,s}$, does validity checks on its output, and runs $\mathsf{E}_{\mathsf{snark},i}$ for each $i \in [m]$. The running time of $\mathsf{P}^{\star}_{\lambda,z,s}$ is bounded by a polynomial $q^{\star}(\lambda)$ where $q^{\star}$ does not depend on λ, z, s. The checks on the output of $\mathsf{P}^{\star}_{\lambda,z,s}$ take time polynomial in its output length, which is therefore bounded by poly(λ). Note that if these checks pass, then it implies that for each $i \in [m]$, the number of updates k_i in the ith sub-statement is at most t and that $n \leq 2^\lambda$.

When the checks pass, $\mathsf{E}_{\mathsf{inner}}$ continues by running the SNARK extractors. For each $i \in [m]$, the running time of $\mathsf{E}_{\mathsf{snark},i}(crs_{\mathsf{snark}}, z^0, s)$ is a polynomial $q_i(\lambda, w_i)$ independent of λ, z, s, where w_i is the work to verify the ith $\mathsf{L}_{\mathsf{upd}}$ statement. As discussed in Section 6.2, this is bounded by $k_i \cdot \beta(\lambda, \log n) \cdot \mathrm{poly}(\lambda, |(M,x)|, p_M, \log t) \leq \mathrm{poly}(\lambda, |(M,x)|, p_M, t)$, since $k_i \leq t$ and $n \leq 2^\lambda$. As M, x are part of the output of $\mathsf{P}^\star_{\lambda,z,s}$, it follows that $|(M,x)|$ is bounded by $q^\star(\lambda)$, and so the work to verify the ith statement is bounded by a fixed polynomial $q^0(\lambda, p_M, t)$. Putting everything together, $\mathsf{E}_{\mathsf{inner}}$ runs in time

$$q_{\mathsf{inner}}(\lambda, t \cdot p_M) \leq q^\star(\lambda) + \mathrm{poly}(\lambda) + \sum_{i=1}^{m} q_i(\lambda, q^0(\lambda, p_M, t)).$$

Since the output length of the prover depends multiplicatively on m, then m is also bounded by $q^\star(\lambda)$, so it follows that $q_{\mathsf{inner}}(\lambda, t \cdot p_M)$ is polynomial in λ and $t \cdot p_M$.

Claim B.3. *There exists a negligible function* $\mathsf{negl}_{\mathsf{inner}}$ *such that for all* $\lambda \in \mathbb{N}$ *and* $z, s \in \{0,1\}^*$, *it holds that*

$$\Pr\left[\begin{array}{l} (crs, st) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda) \\ ((M,x,y,L,t),\pi) = \mathsf{P}^\star_{\lambda,z,s}(crs) \\ b \leftarrow \mathsf{V}_{\mathsf{ni}}(st, (M,x,y,L,t), \pi) \\ (wit_1, \ldots, wit_m, Y, \pi_{\mathsf{final}}) \leftarrow \mathsf{E}_{\mathsf{inner}}(crs, z, s) \end{array} : \begin{array}{l} b = 1 \wedge \\ \exists i \in [m] : (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right]$$
$$\leq \mathsf{negl}_{\mathsf{inner}}(\lambda),$$

where $statement_i$ *is defined to be the statement of the* i*th sub-proof.*

Proof. In all of the following probabilities, m is the number of $\mathsf{L}_{\mathsf{upd}}$ statements given in the proof π, and the statements are denoted $statement_1, \ldots, statement_m$. We start by applying a union bound to upper bound the probability in the statement of the claim by

$$\sum_{i \in [m]} \Pr\left[\begin{array}{l} (crs, st) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda) \\ ((M,x,y,L,t),\pi) \leftarrow \mathsf{P}^\star_{\lambda,z,s}(crs) \\ b \leftarrow \mathsf{V}_{\mathsf{ni}}(st, (M,x,y,L,t), \pi) \\ (wit_1, \ldots, wit_m, Y, \pi_{\mathsf{final}}) \leftarrow \mathsf{E}_{\mathsf{inner}}(crs, z, s) \end{array} : \begin{array}{l} b = 1 \wedge \\ (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right]. \tag{B.1}$$

We now upper bound the above for any particular $i \in [m]$.

By definition of $\mathsf{E}_{\mathsf{inner}}$, whenever the proof π satisfies $\sum_{i=1}^{m} k_i = t$, when M uses at most 2^λ words in memory, and when Y has the right length, then $\mathsf{E}_{\mathsf{inner}}$ runs a SNARK extractor for each i. Note that these are also requirements for V_{ni} to accept (regardless of the randomness for V_{ni}), and so in the event that $b = 1$, $\mathsf{E}_{\mathsf{inner}}$ attempts to extract a witness for each sub-proof. This implies that the above is equal to the probability where wit_i is sampled using the extractor $\mathsf{E}_{\mathsf{snark},i}(\lambda, z^0, s)$ for $\mathsf{P}^\star_i$ where $z^0 = (pp, z)$. Therefore, using the definitions of G_{ni} and $\mathsf{E}_{\mathsf{inner}}$, we can write the above probability as

$$\Pr\left[\begin{array}{l} (crs_{\mathsf{snark}}, st_{\mathsf{snark}}) \leftarrow \mathsf{G}_{\mathsf{snark}}(1^\lambda) \\ pp \leftarrow \mathrm{C.Gen}(1^\lambda, n) \\ h \leftarrow \mathsf{H}_\lambda \\ ((M,x,y,L,t),\pi) \leftarrow \mathsf{P}^\star_{\lambda,z,s}((crs_{\mathsf{snark}}, pp, h)) \\ b \leftarrow \mathsf{V}_{\mathsf{ni}}((st_{\mathsf{snark}}, pp, h), (M,x,y,L,t), \pi) \\ wit_i \leftarrow \mathsf{E}_{\mathsf{snark}}(crs_{\mathsf{snark}}, (pp, h, z), s) \end{array} : \begin{array}{l} b = 1 \wedge \\ (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right],$$

where $n = 2^\lambda$. Whenever $b = 1$, then V_{ni} accepts all sub-proofs, and therefore by definition of $\mathsf{P}^\star_i$, it follows that $\mathsf{V}_{\mathsf{snark}}$ accepts sub-proof i. We can therefore upper bound the above probability by

$$\Pr\left[\begin{array}{l} (crs_{\mathsf{snark}}, st_{\mathsf{snark}}) \leftarrow \mathsf{G}_{\mathsf{snark}}(1^\lambda) \\ pp \leftarrow \mathrm{C.Gen}(1^\lambda, n) \\ h \leftarrow \mathsf{H}_\lambda \\ (statement_i, \pi_i) \leftarrow \\ \quad \mathsf{P}^{*}_{i,(pp,h,z),s}(crs_{\mathsf{snark}}) \\ b_i \leftarrow \mathsf{V}_{\mathsf{snark}}(st_{\mathsf{snark}}, statement_i, \pi_i) \\ wit_i \leftarrow \mathsf{E}_{\mathsf{snark}}(crs_{\mathsf{snark}}, (pp,h,z), s) \end{array} : \begin{array}{l} b_i = 1 \wedge \\ (statement_i, wit_i) \notin R_{\mathsf{upd}} \end{array}\right]. \tag{B.2}$$

Next, for any fixed pp in the support of C.Gen and $h \in \mathsf{H}_\lambda$, the above probability is bounded by a negligible function negl_i that does not depend on λ, pp, h, z, s by the argument of knowledge property of the SNARK. It follows by the law of total probability (to sum over each choice of pp) that Equation (B.2) is bounded by negl_i.

Finally, by plugging this back into Equation (B.1), we obtain that the probability in the statement of the claim is upper bounded by $\sum_{i\in[m]} \mathsf{negl}_i(\lambda)$. Since m determines the length of the output of $\mathsf{P}^{*}_{\lambda,z,s}$, then $m \in \mathrm{poly}(\lambda)$, and so this is negligible as required.

Using $\mathsf{E}_{\mathsf{inner}}$ to extract the witnesses in the sub-protocols, we now define the full extractor E that outputs a witness w for (M, x, y, L, t).

$\underline{\mathsf{E}(crs, z, s)}$:

(1) Run $(wit_1, \dots, wit_m, Y, \pi_{\mathsf{final}}) \leftarrow \mathsf{E}_{\mathsf{inner}}(crs, z, s)$, and abort and output $\bot$ if the output of $\mathsf{E}_{\mathsf{inner}}$ is $\bot$. Let (M, x, y, L, t) be the statement output by $\mathsf{P}^{*}_{\lambda,z,s}$ when computed by $\mathsf{E}_{\mathsf{inner}}$.
(2) Parse each wit_i as a sequence of updates, which together yield an overall sequence of t updates $u_j = (digest_j, V_j^{\mathsf{prev}}, V_j^{\mathsf{rd}}, \pi_j, \tau_j)$ for $j \in [t]$ (abort if this is not the case). Specifically, let $(V_1^{\mathsf{rd}}, \dots, V_t^{\mathsf{rd}})$ be the tuples of values read from these updates.
(3) For $j = 1, \dots, t$, compute $(State_j, Op_j, S_j, V_j^{\mathsf{wt}}) = \text{parallel-step}(M, State_{j-1}, V_{j-1}^{\mathsf{rd}})$ where $State_0$ is the tuple containing the initial RAM state and $V_0^{\mathsf{rd}} = (\bot)$.
(4) Let $D^{\mathsf{Init}} \in \{0,1\}^{n\lambda}$ be the string where for each $\ell \in [n]$, the ℓth word is set to its value in V_i^{rd}, where i is the first iteration with $\ell \in S_i$, or the ℓth word in Y if ℓ is never accessed and $\ell \leq \lceil L/\lambda \rceil$, or 0^λ otherwise.
(5) Output w to be the string of length $n\lambda - |x|$ starting at position $|x|$ in D^{Init}.

We note that while D^{Init} and w above may be as large as $n \cdot \lambda$ bits, they can be specified while running M by using at most $\lambda + \log n$ bits for each non-zero value. Furthermore, they can have at most $t + \lceil L/\lambda \rceil$ non-zero values, since M makes at most t memory accesses, and at most $\lceil L/\lambda \rceil$ additional positions are accessed in specifying the output. Thus, D^{Init} and w can be computed with at most $\mathrm{poly}(\lambda, L, t, \log n)$ additive overhead in time and space.

Claim B.4. *There exists a polynomial q such that $\mathsf{E}(crs, z, s)$ runs in time at most $q(\lambda, t \cdot p_M)$.*

Proof. E first runs $\mathsf{E}_{\mathsf{inner}}$, which has running time bounded by a polynomial $q_{\mathsf{inner}}(\lambda, t \cdot p_M)$ by Claim B.2. Note that if $\mathsf{E}_{\mathsf{inner}}$ does not output $\bot$, then it implies in particular that the number n of words in memory used by M is at most 2^λ. It also implies L is bounded by a fixed polynomial in λ, since $\mathsf{E}_{\mathsf{inner}}$ checks that Y, which is part of the proof π, consists of $\lceil L/\lambda \rceil$ words and hence at least L bits. Using these, we bound the remaining running time of E by a polynomial in λ and $t \cdot p_M$, which completes the claim.

After running $\mathsf{E}_{\mathsf{inner}}$, E parses its output as a sequence of t updates, where each update has size at most $2\beta(\lambda, \log n) \cdot p_M \cdot \lambda \in \mathrm{poly}(\lambda)$ by the eficiency of the underlying hash function, which takes time $t \cdot p_M \cdot \text{poly}(\lambda)$. Using these updates to determine which values to read, E emulates M for t steps, which can be done in time $t \cdot p_M \cdot \text{poly}(\lambda, |M|)$. This can be bounded by $t \cdot p_M \cdot \text{poly}(\lambda)$, since M is part of the output of $\mathsf{P}^{\star}_{\lambda,z,s}$, so $|M|$ is bounded by a fixed polynomial in λ. Finally, E computes the initial memory D^{Init} to output a witness w, which, as discussed above, requires specifying at most $t + \lceil L/\lambda \rceil$ positions and therefore takes at most $\text{poly}(\lambda, L, t, \log n)$ time. This is bounded by $\text{poly}(\lambda, t)$, as $L, \log n$ are in $\text{poly}(\lambda)$. Altogether, E runs in time at most $q_{\mathsf{inner}}(\lambda, t \cdot p_M) + t \cdot p_M \cdot \text{poly}(\lambda) + t \cdot p_M \cdot \text{poly}(\lambda) + \text{poly}(\lambda, t)$ that can be bounded by a polynomial $q(\lambda, t \cdot p_M)$.

Claim B.5. *For every constant $c \in \mathbb{N}$, there exists a negligible function* negl *such that for all $\lambda \in \mathbb{N}$ and $z, s \in \{0,1\}^*$,*

$$\Pr\left[\begin{array}{l} (crs, st) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda) \\ ((M,x,y,L,t),\pi) \leftarrow \mathsf{P}^{\star}_{\lambda,z,s}(crs) \\ b \leftarrow \mathsf{V}_{\mathsf{ni}}(st,(M,x,y,L,t),\pi) \\ w \leftarrow \mathsf{E}(crs,z,s) \end{array} : \begin{array}{l} b = 1 \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ t \cdot p_M \le |x|^c \end{array}\right] \le \mathsf{negl}(\lambda),$$

where p_M is the number of processors used by M.

Proof. In the following, all probabilities are over $(crs, st) \leftarrow \mathsf{G}_{\mathsf{ni}}(1^\lambda)$, $((M,x,y,L,t),\pi) \leftarrow \mathsf{P}^{\star}_{\lambda,z,s}(crs)$, $b \leftarrow \mathsf{V}_{\mathsf{ni}}(st,(M,x,y,L,t),\pi)$, and $w \leftarrow \mathsf{E}(crs,z,s)$. We let $statement_i, \pi_i$ for all $i \in [m]$ be the statement and proof given $\mathsf{P}^{\star}_{\lambda,z,s}$ for the ith $\mathsf{L}_{\mathsf{upd}}$ instance, and we define p_M to be the number of processors used by M. Additionally, we let $wit_1, \ldots, wit_m, Y, \pi_{\mathsf{final}}$ be the output of $\mathsf{E}_{\mathsf{inner}}$ during the execution of E in each probability.

Suppose by way of contradiction that there exists a polynomial p such that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} b = 1 \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ t \cdot p_M \le |x|^c \end{array}\right] > \frac{1}{p(\lambda)}.$$

We can rewrite this probability as

$$\Pr\left[\begin{array}{l} b = 1 \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ t \cdot p_M \le |x|^c \wedge \\ \left(\forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}}\right) \end{array}\right] + \Pr\left[\begin{array}{l} b = 1 \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ t \cdot p_M \le |x|^c \wedge \\ \left(\exists i \in [m]\ (statement_i, wit_i) \notin R_{\mathsf{upd}}\right) \end{array}\right]$$

$$\le \Pr\left[\begin{array}{l} b = 1 \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ t \le |x|^c \wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \end{array}\right] + \mathsf{negl}_{\mathsf{inner}}(\lambda),$$

by Claim B.3 above. As $\mathsf{negl}_{\mathsf{inner}}(\lambda) < 1/(2p(\lambda))$ for infinitely many $\lambda \in \mathbb{N}$, this implies that for infinitely many $\lambda \in \mathbb{N}$,

$$\Pr\left[\begin{array}{l} b = 1 \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}_{\mathsf{U}}^{\mathrm{PRAM}} \wedge \\ t \cdot p_M \le |x|^c \wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \end{array}\right] > \frac{1}{2p(\lambda)}. \tag{B.3}$$

Given this, consider the following non-uniform adversary $\mathsf{A} = \{\mathsf{A}_\lambda\}_{\lambda \in \mathbb{N}}$ that can be used to break the soundness of either C or H, where A_λ has z, s and the description of $\mathsf{P}^{\star}_{\lambda}$ hardcoded:

$\underline{\mathsf{A}_\lambda(pp, h)}$:

(1) Sample $(crs_{\mathsf{snark}}, st_{\mathsf{snark}}) \leftarrow \mathsf{G}_{\mathsf{snark}}(1^\lambda)$. Let $crs = (crs_{\mathsf{snark}}, pp, h)$ and $st = (st_{\mathsf{snark}}, pp, h)$.

(2) Compute $((M, x, y, L, t), \pi) = \mathsf{P}^{\star}_{\lambda,z,s}(crs)$. Check that $t \cdot p_M \leq |x|^c$, where p_M is the number of processors used by M. If this does not hold, then abort and output $\bot$. Let $(State_{\mathsf{final}}, V^{\mathsf{rd}}_{\mathsf{final}})$ be the final states and words in π (corresponding to those sent in the final message).

(3) Run $w \leftarrow \mathsf{E}(crs, z, s)$. If E outputs $\bot$, then abort and output $\bot$. Otherwise, let $wit_1, \ldots, wit_m$ be the witnesses output by $\mathsf{E}_{\mathsf{inner}}$ for statements $statement_1, \ldots, statement_m$.

(4) Sample $b \leftarrow \mathsf{V}_{\mathsf{ni}}(st, (M, x, y, L, t), \pi)$. If $b = 0$, then abort and output $\bot$. If $b = 1$, then let $statement_i, \pi_i$ be the statement and proof for the ith $\mathsf{L}_{\mathsf{upd}}$ instance given by π for $i \in [m]$ and let Y, π_{final} be the opening for the final output given by π.

(5) If there exists a $j \in [m]$ such that $(statement_j, wit_j) \notin R_{\mathsf{upd}}$, then abort and output $\bot$. Otherwise, parse each witness wit_j as containing an initial set of states and words read $(State^{(j)}, V^{\mathsf{rd},(j)})$, as well as a sequence of updates. Let $u_1, \ldots, u_t$ be the sequence of t updates obtained across all m witnesses where $u_i = (digest_i, V^{\mathsf{prev}}_i, V^{\mathsf{rd}}_i, \pi_i, \tau_i)$ for all $i \in [t]$. Additionally, for each $i \in [t]$, let V_i be a tuple of $|S_i|$ values, where the jth value is that of V^{rd}_i or V^{wt}_i according to the corresponding operation given by Op_i.

Recall that E's emulation defined the starting values $(State_0, V^{\mathsf{rd}}_0)$ and values $(State_i, Op_i, S_i, V^{\mathsf{wt}}_i)$ for each RAM step. Last, let $digest_0$ be the initial digest computed by V.

(6) Check that E's emulation is consistent with the extracted updates. Specifically, let $K_0 = 0$ and let K_j be the number of updates in sub-statements 1 through j for each $j \in [m]$. If there exists a $j \in [m]$ such that $(State^{(j)}, V^{\mathsf{rd},(j)})$ is not equal to $(State_{K_{j-1}}, V^{\mathsf{rd}}_{K_{j-1}})$, then let j be the smallest such index and output $((State^{(j)}, V^{\mathsf{rd},(j)}), (State_{K_{j-1}}, V^{\mathsf{rd}}_{K_{j-1}}))$. Similarly, if $(State_{\mathsf{final}}, V^{\mathsf{rd}}_{\mathsf{final}}) \neq (State_t, V^{\mathsf{rd}}_t)$, then output these four values.

(7) Next, A_λ emulates the computation of $M(x, w)$. To avoid confusion with the values in the extracted update, we will use a superscript "?" to denote the values computed in this emulation. Let $State^?_0$ be a tuple containing the initial RAM state, $V^{\mathsf{rd}?}_0 = (\bot)$, and $D^? = x||w$ be the initial memory string for use by M.

For $i = 1, \ldots, t$, do the following:

(a) Compute $(State^?_i, Op^?_i, S^?_i, V^{\mathsf{wt}?}_i) = \text{parallel-step}(M, State^?_{i-1}, V^{\mathsf{rd}?}_{i-1})$.

(b) Read from and write to $D^?$ by running $V^{\mathsf{rd}?}_i = \mathsf{access}^{D^?}(Op^?_i, S^?_i, V^{\mathsf{wt}}_i)$.

Let $Y^?$ be the tuple containing the first $L' = \lceil L/\lambda \rceil$ words of $D^?$, and let $y^?$ be the concatenation of the first $outlen$ bits from $Y^?$, where $outlen$ is the output length specified by $State^0_m$.

(8) If there exists an index i such that $V^{\mathsf{rd}}_i \neq V^{\mathsf{rd}?}_i$, then let i be the smallest such index. Compute a digest of the empty partial string $(ptr^?, digest^?_0) = \mathsf{C.Hash}(pp, D_\bot)$ and then compute $(\bot, \pi^?) = \mathsf{C.Open}(pp, ptr^?, S_i)$. Output

$$\left(i-1, \left\{(digest_j, S_j, V_j, \tau_j)\right\}_{j \in [i-1]}, digest_0, S_i, (\bot)^{|S_i|}, \pi^?, V^{\mathsf{prev}}_i, \pi_i\right).$$

(9) If $Y \neq Y^?$, then compute a digest of the empty partial string $(ptr^?, digest^?_0) = \mathsf{C.Hash}(pp, D_\bot)$ and then compute $(\bot, \pi^?) = \mathsf{C.Open}(pp, ptr^?, [L'])$. Output

$$\left(t, \left\{(digest_j, S_j, V_j, \tau_j)\right\}_{j \in [t]}, digest_0, [L'], (\bot)^{L'}, \pi^?, Y, \pi_{\mathsf{final}}\right).$$

(10) Otherwise, abort and output $\bot$.

To analyze the success of A in breaking the soundness of C, below we argue that (1) A_λ runs in polynomial time; (2) if A_λ outputs in steps 8 or 9, then A_λ finds values that breaking the soundness of C; and (3) if A_λ reaches step 10, then it must be the case that $((M,x,y,L,t),w) \in \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}}$.

Given these claims, we can conclude the proof as follows. First, note that A_λ outputs in step 6, 8, 9, or 10 whenever $b = 1$, $(statement_i, wit_i) \in R_{\mathsf{upd}}$ for all $i \in [m]$, $t \cdot p_M \leq |x|^c$, and E does not output $\bot$. Note that if $(statement_i, wit_i) \in R_{\mathsf{upd}}$, then the output of E is not $\bot$. We can therefore break the event that A outputs in step 6, 8, 9, or 10 into two cases as

$$\Pr\left[\begin{array}{l} b = 1 \wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \wedge \\ t \cdot p_M \leq |x|^c \wedge \\ ((M,x,y,L,t),w) \in \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right] + \Pr\left[\begin{array}{l} b = 1 \wedge \\ \forall i \in [m]\ (statement_i, wit_i) \in R_{\mathsf{upd}} \wedge \\ t \cdot p_M \leq |x|^c \wedge \\ ((M,x,y,L,t),w) \notin \mathsf{R}^{\mathrm{PRAM}}_{\mathsf{U}} \end{array}\right].$$

We observe that the only difference between A_λ and the adversary given in the proof of Lemma 6.3 for the interactive case is in steps 1, 2, 3, and 4. After step 4, A_λ uses the witnesses it obtained identically to the adversary in the interactive case. It therefore follows by the same logic as in Subclaim 6.12 that the first term in the above probability is greater than the probability that A_λ outputs in step 10. The second term is greater than $1/(2p(\lambda))$ by Equation (B.3). Putting these together, we get that the probability that A_λ outputs in steps 6, 8, or 9 is greater than $1/(2p(\lambda))$. To obtain a contradiction, we show below that A_λ runs in polynomial time in Subclaim B.6. We observe that by Subclaim 6.9, if A_λ outputs in step 6, then it finds values that violate the soundness of H. Similarly, by Subclaim 6.11, if A_λ outputs in step 8 or step 9, then it finds values that violate the soundness of C. Therefore, A_λ can be used to break the soundness of C or of H with probability at least $1/(2p(\lambda))$, in contradiction.

Subclaim B.6. *There exists a polynomial q_{A} such that for every $pp \in \mathrm{Supp}\left(\mathsf{C.Gen}(1^\lambda, 2^\lambda)\right)$ and $h \in \mathsf{H}_\lambda$, the running time of $\mathsf{A}_\lambda(pp, h)$ is at most $q_{\mathsf{A}}(\lambda)$ for all $\lambda \in \mathbb{N}$.*

Proof. The running time of A_λ is bounded by the sum of (1) the time to run $\mathsf{G}_{\mathsf{snark}}$, (2) the time to run $\mathsf{P}^\star_{\lambda,z,s}$ and check its output, (3) the total amount of time A_λ spends running E, (4) the time to run V_{ni}, (5) the time to check that all $(statement_i, wit_i)$ pairs are in R_{upd}, (6) the time to check for and compute an output in step 6, (7) the time to emulate the execution of M, and (8) the time to check for and compute an output in steps 8 and steps 9. We separately argue that each of these runs in at most polynomial time in λ.

First, (1) is bounded by a polynomial in λ by the efficiency of $\mathsf{G}_{\mathsf{snark}}$ and C.Gen. (2) is bounded by a polynomial in λ, since $\mathsf{P}^\star_{\lambda,z,s}$ runs in fixed polynomial time $q^\star(\lambda)$ for any $z, s \in \{0,1\}^*$. Note that, since M, x are part of the prover's output, then $|(M,x)|$ is also bounded $q^\star(\lambda)$. When A_λ does not abort after running the prover, this implies that $t \cdot p_M \leq |x|^c \in \mathrm{poly}(\lambda)$. Next, (3) is bounded by a polynomial $q_{\mathsf{E}}(\lambda, t \cdot p_M)$ by Claim B.4. This is polynomial in λ by the bounds on $|x|, t$ above. Note that if A_λ does not abort after running E, then by definition of $\mathsf{E}_{\mathsf{inner}}$, this implies that $L \leq q^\star(\lambda)$ and $n \leq 2^\lambda$. For (4), by succinctness, the running time of V_{ni} is bounded by a fixed polynomial $\mathrm{poly}(\lambda, |(M,x)|, L, p_M, t) \in \mathrm{poly}(\lambda)$ by the aforementioned bounds. For (5), it requires checking that at most t updates are valid where each check requires a polynomial amount of work in λ, $|(M,x)|, \beta(\lambda, \log n), p_M, \log t$ by definition of $\mathsf{L}_{\mathsf{upd}}$ and the efficiency of C. By the above bounds, this is in $\mathrm{poly}(\lambda)$. Next, (6) requires checking equality of $(m+1) \cdot \mathrm{poly}(\lambda) \cdot p_M$ values, which is bounded by a polynomial in λ, since $p_M \leq |x|^c$ and because $m \in \mathrm{poly}(\lambda)$ as the output length of $\mathsf{P}^\star_{\lambda,z,s}$ depends on m. Next, (7) takes t steps of computation, each of which takes time bounded by a fixed polynomial in $\lambda, |M|, p_M$ by the definition of PRAM computation, which is polynomial in λ by the above. Last, (8) requires $(t + L + 1) \cdot p_M \cdot \lambda$ time to check equality of all corresponding values. Computing the initial hash and opening requires $2\beta(\lambda, \log n) \cdot p_M \in \mathrm{poly}(\lambda)$

by eficiency of C. Then, the full output has size at most $t \cdot p_M \cdot \text{poly}(\lambda)$ ⊠ poly(λ) and takes at most $t \cdot p_M \cdot \text{poly}(\lambda)$ ⊠ poly(λ) time to compute.

Therefore, the running time of A_λ is bounded by some polynomial $q_{\mathsf{A}}(\lambda)$ for all λ ⊠ $\mathbb{N}$.

This completes the proof of Claim B.5.

This completes the proof of Lemma B.1.

ACKNOWLEDGMENTS

We thank Krzysztof Pietrzak for a useful discussion regarding memory-hard functions.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the oficial policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

REFERENCES

[1] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. 2017. Depth-robust graphs and their cumulative memory complexity. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 10212)*. 3–32.
[2] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. 2018. Sustained space complexity. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 10821)*. Springer, 99–130.
[3] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. 2016. On the complexity of scrypt and proofs of space in the parallel random oracle model. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 9666)*. Springer, 358–387.
[4] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. 2017. Scrypt is maximally memory-hard. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 10212)*. 33–62.
[5] Joël Alwen and Vladimir Serbinenko. 2015. High parallel complexity graphs and memory-hard functions. In *STOC*. ACM, 595–603.
[6] Joël Alwen and Björn Tackmann. 2017. Moderately hard functions: Definition, instantiations, and applications. In *TCC (Lecture Notes in Computer Science, Vol. 10677)*. Springer, 493–526.
[7] Boaz Barak. 2001. How to go beyond the black-box simulation barrier. In *FOCS*. IEEE Computer Society, 106–115.
[8] Boaz Barak and Oded Goldreich. 2008. Universal arguments and their applications. *SIAM J. Comput.* 38, 5 (2008), 1661–1694.
[9] Mihir Bellare and Oded Goldreich. 1992. On defining proofs of knowledge. In *CRYPTO (Lecture Notes in Computer Science, Vol. 740)*. Springer, 390–420.
[10] Eli Ben-Sasson, Alessandro Chiesa, Ariel Gabizon, Michael Riabzev, and Nicholas Spooner. 2017. Interactive oracle proofs with constant rate and query complexity. In *ICALP (LIPIcs, Vol. 80)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 40:1–40:15.
[11] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. 2013. On the concrete eficiency of probabilistically-checkable proofs. In *STOC*. ACM, 585–594.
[12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (Lecture Notes in Computer Science, Vol. 8043)*. Springer, 90–108.
[13] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. 2016. Interactive oracle proofs. In *TCC (Lecture Notes in Computer Science, Vol. 9986)*. 31–60.
[14] Eli Ben-Sasson and Madhu Sudan. 2008. Short PCPs with polylog query complexity. *SIAM J. Comput.* 38, 2 (2008), 551–607.
[15] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The hunting of the SNARK. *J. Cryptol.* 30, 4 (2017), 989–1066.
[16] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2013. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*. ACM, 111–120.
[17] Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen. 2016. On the existence of extractable one-way functions. *SIAM J. Comput.* 45, 5 (2016), 1910–1952.
[18] Nir Bitansky and Alessandro Chiesa. 2012. Succinct arguments from multi-prover interactive proofs and their eficiency benefits. In *CRYPTO (Lecture Notes in Computer Science, Vol. 7417)*. Springer, 255–272.
[19] Nir Bitansky, Shafi Goldwasser, Abhishek Jain, Omer Paneth, Vinod Vaikuntanathan, and Brent Waters. 2016. Time-lock puzzles from randomized encodings. In *ITCS*. ACM, 345–356.

[20] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *CRYPTO (Lecture Notes in Computer Science, Vol. 10991)*. Springer, 757–788.

[21] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2018. A survey of two verifiable delay functions. *IACR Cryptol. ePrint Arch.* 2018 (2018), 712.

[22] Elette Boyle and Rafael Pass. 2015. Limits of extractability assumptions with distributional auxiliary input. In *ASIACRYPT (Lecture Notes in Computer Science, Vol. 9453)*. Springer, 236–261.

[23] Dario Catalano and Dario Fiore. 2013. Vector commitments and their applications. In *Public Key Cryptography (Lecture Notes in Computer Science, Vol. 7778)*. Springer, 55–72.

[24] Kai-Min Chung, Huijia Lin, and Rafael Pass. 2013. Constant-round concurrent zero knowledge from P-Certificates. In *FOCS*. IEEE Computer Society, 50–59.

[25] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile verifiable computation. In *S&P*. IEEE Computer Society, 253–270.

[26] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. 2019. Trapdoor hash functions and their applications. In *CRYPTO (Lecture Notes in Computer Science, Vol. 11694)*. Springer, 3–32.

[27] Nico Döttling, Sanjam Garg, Giulio Malavolta, and Prashant Nalini Vasudevan. 2020. Tight verifiable delay functions. In *SCN (Lecture Notes in Computer Science, Vol. 12238)*. Springer, 65–84.

[28] Thaddeus Dryja, Quanquan C. Liu, and Sunoo Park. 2018. Static-memory-hard functions, and modeling the cost of space vs. time. In *TCC (Lecture Notes in Computer Science, Vol. 11239)*. Springer, 33–66.

[29] Cynthia Dwork, Andrew V. Goldberg, and Moni Naor. 2003. On memory-bound functions for fighting spam. In *CRYPTO (Lecture Notes in Computer Science, Vol. 2729)*. Springer, 426–444.

[30] Cynthia Dwork, Moni Naor, and Hoeteck Wee. 2005. Pebbling and proofs of work. In *CRYPTO (Lecture Notes in Computer Science, Vol. 3621)*. Springer, 37–54.

[31] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. 2015. Proofs of space. In *CRYPTO (Lecture Notes in Computer Science, Vol. 9216)*. Springer, 585–605.

[32] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. 2020. Continuous verifiable delay functions. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 12107)*. Springer, 125–154.

[33] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. 2020. SPARKs: Succinct parallelizable arguments of knowledge. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 12105)*. Springer, 707–737.

[34] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO (Lecture Notes in Computer Science, Vol. 263)*. Springer, 186–194.

[35] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2015. Delegating computation: Interactive proofs for muggles. *J. ACM* 62, 4 (2015), 27:1–27:64.

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

[37] Jens Groth and Yuval Ishai. 2008. Sub-linear zero-knowledge argument for correctness of a shufle. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 4965)*. Springer, 379–396.

[38] Justin Holmgren and Ron Rothblum. 2018. Delegating computations with (almost) minimal time and space overhead. In *FOCS*. IEEE Computer Society, 124–135.

[39] Yael Tauman Kalai and Omer Paneth. 2016. Delegating RAM computations. In *TCC*. 91–118.

[40] Yael Tauman Kalai and Ran Raz. 2008. Interactive PCP. In *ICALP (Lecture Notes in Computer Science, Vol. 5126)*. Springer, 536–547.

[41] Joe Kilian. 1992. A note on eficient zero-knowledge proofs and arguments (extended abstract). In *STOC*. ACM, 723–732.

[42] Yehuda Lindell. 2003. Parallel coin-tossing and constant-round secure two-party computation. *J. Cryptol.* 16, 3 (2003), 143–184.

[43] Ralph C. Merkle. 1989. A certified digital signature. In *CRYPTO (Lecture Notes in Computer Science, Vol. 435)*. Springer, 218–238.

[44] Silvio Micali. 2000. Computationally sound proofs. *SIAM J. Comput.* 30, 4 (2000), 1253–1298.

[45] Silvio Micali and Rafael Pass. 2006. Local zero knowledge. In *STOC*. ACM, 306–315.

[46] Dalit Naor, Moni Naor, and Jeffery Lotspiech. 2001. Revocation and tracing schemes for stateless receivers. In *CRYPTO (Lecture Notes in Computer Science, Vol. 2139)*. Springer, 41–62.

[47] Omer Paneth. 2019. Alternative VDF Constructions. Retrieved from: https://dci.mit.edu/video-gallery/2019/5/29/alternate-vdf-constructions-by-omer-paneth-of-mit-vdf-day-2019.

[48] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 238–252.

[49] Rafael Pass and Alon Rosen. 2008. Concurrent nonmalleable commitments. *SIAM J. Comput.* 37, 6 (2008), 1891–1925.

[50] Colin Percival. 2009. Stronger key derivation via sequential memory-hard functions. In *BSDCan*.

[51] Krzysztof Pietrzak. 2019. Simple verifiable delay functions. In *ITCS (LIPIcs, Vol. 124)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 60:1–60:15.

[52] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. 2016. Constant-round interactive proofs for delegating computation. In *STOC*. ACM, 49–62.

[53] Ronald L. Rivest, Adi Shamir, and David A. Wagner. 1996. Time-lock Puzzles and Timed-release Crypto. Manuscript. https://people.csail.mit.edu/rivest/pubs/RSW96.pdf.

[54] Noga Ron-Zewi and Ron D. Rothblum. 2020. Local proofs approaching the witness length [extended abstract]. In *FOCS*. IEEE, 846–857.

[55] Paul Valiant. 2008. Incrementally verifiable computation or proofs of knowledge imply time/space eficiency. In *TCC (Lecture Notes in Computer Science, Vol. 4948)*. Springer, 1–18.

[56] Benjamin Wesolowski. 2019. Eficient verifiable delay functions. In *EUROCRYPT (Lecture Notes in Computer Science, Vol. 11478)*. Springer, 379–407.

[57] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A distributed zero knowledge proof system. In *USENIX Security Symposium*. USENIX Association, 675–692.

Received 15 June 2020; revised 24 May 2021; accepted 26 May 2022