Parallelizable Delegation from LWE

Cody Freitag[*] Rafael Pass[†] Naomi Sirkin[‡]

Abstract

We present the first non-interactive delegation scheme for P with *time-tight parallel prover efficiency* based on standard hardness assumptions. More precisely, in a time-tight delegation scheme—which we refer to as a SPARG (succinct parallelizable argument)—the prover's parallel running time is $t + \text{polylog}(t)$, while using only $\text{polylog}(t)$ processors and where t is the length of the computation. (In other words, the proof is computed essentially in parallel with the computation, with only some minimal additive overhead in terms of time).

Our main results show the existence of a publicly-verifiable, non-interactive, SPARG for P assuming polynomial hardness of LWE. Our SPARG construction relies on the elegant recent delegation construction of Choudhuri, Jain, and Jin (FOCS'21) and combines it with techniques from Ephraim et al (EuroCrypt'20).

We next demonstrate how to make our SPARG *time-independent*—where the prover and verifier do not need to known the running-time t in advance; as far as we know, this yields the first construction of a time-tight delegation scheme with time-independence based on any hardness assumption.

We finally present applications of SPARGs to the constructions of VDFs (Boneh et al, Crypto'18), resulting in the first VDF construction from standard polynomial hardness assumptions (namely LWE and the minimal assumption of a sequentially hard function).

[*]Cornell Tech, cfreitag@cs.cornell.edu

[†]Cornell Tech and Tel-Aviv University, rafael@cs.cornell.edu

[‡]Cornell Tech, nephraim@cs.cornell.edu

Contents

1 Introduction

In an interactive proof system, a prover interacts with a verifier in order to prove the validity of a computational statement, with the guarantee that the verifier will be convinced if and only if the statement is true. Since their introduction by Goldwasser, Micali, and Rackoff [GMR89], proof systems have become one of the most fundamental concepts in cryptography and more generally in theoretical computer science.

In this work, we focus on the application of proof systems to computational delegation, where a weak verifier outsources a potentially expensive computation to a powerful yet untrusted prover, who performs the computation and returns the output as well as a proof certifying its validity. We focus on delegating deterministic polynomial-time computation with the non-trivial requirement that the proof system is *succinct* [Kil92, Mic00], meaning that the verifier's running time and the length of the communication between the prover and verifier is essentially independent of the running time of the delegated computation.

Interest in succinct delegation has exploded in recent years due to its many applications in internet-scale, distributed protocols like blockchains and cryptocurrencies. Two key features for enabling these applications are non-interactivity and public verifiability. Non-interactivity stipulates that a proof consists of just a single message to the verifier, and public verifiability means that any third party can trust the validity of the proof. Such delegation schemes are known as publicly-verifiable SNARGs (succinct, non-interactive, arguments), and have seen immense effort in recent years from both the applied and theoretical communities in cryptography (see, e.g., [PHGR13, BCG+14, BCC+17, BBHR19, CJJ21]).

On the theory side, constructing publicly verifiable SNARGs from standard assumptions was previously elusive for many years, partially because of inherent bottlenecks for constructing SNARGs for all of NP from falsifiable assumptions [GW11]. However, the beautiful recent works of Kalai, Paneth, and Yang [KPY19] and Choudhuri, Jain, Jin [CJJ21] have shown that when restricting to languages in P, SNARGs can be constructed from falsifiable assumptions, including most recently from the polynomial hardness of LWE [CJJ21].

On Parallel Prover Efficiency. Aside from improving the underlying assumptions, a major bottleneck for the adoption of SNARGs has been prover efficiency. There have been many works (e.g., [BC12, CFH+15, WZC+18, HR18, BHR+20, BHR+21] to name a few) focused on improving the asymptotic efficiency of the prover as much as possible under various assumptions. In the setting of delegation, this means that the running time of the prover should ideally be as close as possible to the time t of the delegated computation, which is inherent for the prover to even compute the output itself. To date, the best asymptotic constructions achieve quasi-linear overhead by the prover, with running time $t \cdot \mathrm{poly}(\lambda, \log t)$ where λ is the security parameter.

Recently, the work of Ephraim, Freitag, Komargodski, and Pass [EFKP20b] showed how to construct parallelizable delegation schemes (which they call SPARKs) where the prover has *parallel* running time $t + \mathrm{poly}(\lambda, \log t)$ (i.e., with only *additive overhead* and no multiplicative overhead) using only a modest number, $\mathrm{poly}(\lambda, \log t)$, of processors. Their protocols even work for NP, but at the cost of either assuming SNARKs (succinct non-interactive arguments of knowledge) for NP—that are only known to exist from non-standard and non-falsifiable assumptions—or only achieving an *interactive* protocol (assuming just standard collision-resistant hash functions). Thus, the state-of-the art leaves open the question of whether we can get a *non-interactive* delegation scheme, even just for P, with tight prover efficiency from standard (falsifiable) assumptions:

Can we construct publicly verifiable, succinct, parallelizable *delegation schemes for* P

from standard, falsifiable, assumptions?

We refer to such publicly verifiable, succinct, parallelizable delegation schemes as SPARGs (succinct parallelizable arguments) for P, following the notation of SPARKs from [EFKP20b].

In this work, we resolve the above-mentioned problem, constructing the first non-interactive delegation schemes where the prover has $t + \text{poly}(\lambda, \log t)$ parallel running time using $\text{poly}(\lambda, \log t)$ processors based on standard assumptions. More precisely, our construction only relies on the polynomial hardness of the LWE assumption.

Theorem 1.1 (SPARGs for P from LWE; Informal (see Corollary 5.2))**.** *Assuming hardness of LWE, there exists a non-interactive SPARG for* P*.*

We additionally present strengthenings of the above theorem—including a SPARG for computations that are themselves parallelized, and obtaining so-called *time-independent* SPARGs, where the prover and verifier need not know the length t of the computation in advance—and present corollaries of these results, including the first construction of a Verifiable Delay Function (VDF) [BBBF18] from standard (polynomial) hardness assumptions.

1.1 Our Results in More Detail

Let us present our results in more detail. As a starting point for our work, we observe that SPARGs for P can be constructed based on the notion of RAM delegation, following the framework of the SPARK construction due to [EFKP20b], so long as the RAM delegation scheme satisfies *quasi-linear prover efficiency*. RAM delegation is known under various assumptions, and most recently was shown secure under LWE [CJJ21]. Unfortunately, known RAM delegation schemes do not satisfy the quasi-linear prover efficiency that we desire. Therefore, our main result is to show how to adapt existing schemes to satisfy a notion of efficiency that will suffice for our construction.

Updatable RAM Delegation. We start by defining the notion of an *updatable* RAM delegation scheme with quasi-linear efficiency. From an efficiency perspective, this is weaker than a (non-updatable) RAM delegation scheme satisfying quasi-linear efficiency. Nevertheless, we show that it suffices for our purposes, and can be constructed by relying on the RAM delegation scheme of [CJJ21].

At a high level, an updatable RAM delegation scheme is a delegation scheme for RAM computations that allows for incremental updates and proofs for intermediate pieces of the overall computation. Specifically, a prover can perform part of a computation and obtain the resulting state as well as some additional auxiliary information aux corresponding to this section of the computation. Given aux, it can then continue to update the computation to a new state, producing a new piece of auxiliary information aux'. The auxiliary information aux for any sub-computation can be used as a "witness" to *efficiently* compute a proof for the corresponding piece of the computation. (We note that the proof is for a deterministic computation, but the auxiliary input/ witness is provided for efficiency purposes.) This enables a large computation to be updated and proved in different pieces, and in particular allows for taking advantage of the prover's knowledge of aux, from running the computation, in order to generate a proof with significantly less overhead.

In more detail, we require an updatable delegation scheme with the following efficiency properties:

- **Efficiency of computing aux:** Given a RAM configuration cf, auxiliary information $\mathsf{aux}_{\mathsf{cf}}$, and time t, the new configuration cf' and its associated auxiliary information $\mathsf{aux}_{\mathsf{cf}'}$ that results after t steps of computation starting from cf can be computed in time $t + \text{poly}(\lambda)$ using $\text{poly}(\lambda)$ parallel processors.

- **Efficiency of generating proofs given aux:** Given auxiliary information aux corresponding to a t step transition from initial configuration cf to final configuration cf', a proof of correctness for this transition can be generated in time $t \cdot \text{poly}(\lambda, \log t)$. For an updatable scheme, we refer to this as *quasi-linear prover efficiency.* (Note that this is a stronger efficiency requirement than the one used in [CJJ21], where the prover running-time would grow with $|\mathsf{cf}|$.)

Let us highlight that any RAM delegation scheme is also an updatable one (by simply letting aux be empty), but does not necessarily satisfy quasi-linear overhead when generating proofs given aux. Using the auxiliary information, aux, is helpful for us in achieving this prover efficiency. In particular, we show how to combine the ideas behind the SNARG construction of [CJJ21] with the updatable hash tree from [EFKP20b] to get an updatable RAM delegation from LWE with the desired efficiency.

Theorem 1.2 (Efficient Updatable RAM Delegation; Informal (see Theorem 4.4))**.** *Assuming hardness of LWE, there exists a succinct, publicly verifiable, updatable RAM delegation scheme with quasi-linear prover efficiency.*

SPARGs from updatable RAM delegation. Next, we show how to adapt the SPARK construction of [EFKP20b] to rely on any updatable RAM delegation scheme with quasi-linear prover efficiency, rather than relying on SNARKs with quasi-linear prover efficiency. We highlight that the construction in [EFKP20b] relied on the proof of knowledge property of the underlying delegation scheme (i.e., the SNARK in use) and it is not known how to replace it with just a SNARG. This is why we resort to using the more complicated object of an updatable RAM delegation scheme with quasi-linear prover efficiency.

Theorem 1.3 (SPARGs from Updatable RAM Delegation; Informal (see Theorem 5.1))**.** *Assume the existence of a succinct, publicly verifiable, updatable RAM delegation scheme with quasi-linear efficiency. Then there exists a non-interactive SPARG for* P.

Theorem 1.1 then follows as a direct corollary of Theorems 1.2 and 1.3.

We also extend this result to the setting of *parallel* computations. Specifically, given a computation that can be done in time t with p processors, we show a SPARG that preserves depth by running in time $t + \text{poly}(\lambda, \log(t \cdot p))$, while only using $p \cdot \text{poly}(\lambda, \log(t \cdot p))$ processors. This is in contrast to the naive approach of using the above SPARG for sequential computations, which would naively result in parallel time that depends on the total work $t \cdot p$ rather than the depth t. We obtain this result by extending the updatable RAM delegation scheme above to be depth preserving for parallel computations—that is, *both* the parallel time and processors used by the delegation scheme scale quasi-linearly with that of the computation.

Theorem 1.4 (SPARGs for Parallel Computations; Informal (see Theorem 6.9))**.** *Assume the existence of a succinct, publicly verifiable, updatable RAM delegation scheme for parallel computations that is depth-preserving. Then there exists a non-interactive SPARG for polynomial-time, parallel computations.*

Time-independent SPARGs. SPARKs [EFKP20b] were initially defined such that in order to prove a t-time computation, the prover was provided the time bound t as input. It is perhaps natural to assume that this might be necessary in order to "fit the computation of the proof" in during the computation itself. However, in many scenarios, the time bound t may not be a priori known. To circumvent this issue, we define the notion of a *time-independent* SPARG, which satisfies the same properties as a SPARG except that the prover and verifier no longer get t as input. We additionally show how to extend the above construction to achieve a time-independent SPARG from LWE:

Theorem 1.5 (Time-independent SPARGs from LWE; Informal (see Corollary 7.4)). *Assuming hardness of LWE, there exists a non-interactive, time-independent SPARG for* P*.*

As far as we know, this yields the first construction of a SPARG with time-independence based on any hardness assumption (that is, a similar result was not known from the stronger notion of SPARKs).

To prove Theorem 1.5, we define the notion of a *time-tight*, updatable RAM delegation. Essentially, this is a RAM delegation as above, but with the prover efficiency properties of a SPARG, where the final configuration is not known at the start of proof generation. We emphasize that the prover for such a scheme is given the time bound t as input in order to compute the proof in time $t + \mathrm{poly}(\lambda, \log(t))$. We then give a generic transformation that starts with any time-tight, updatable RAM delegation scheme (that is given the time bound t as input) and constructs a non-interactive, time-independent SPARG.

Theorem 1.6 (Time-independent SPARG transformation; Informal (see Theorem 7.3)). *Given any time-tight, updatable RAM delegation scheme, there exists a non-interactive, time-independent SPARG for* P*.*

Furthermore, a minor adaptation of our construction of a SPARG for P from LWE (Theorem 1.1 above) satisfies the notion of a time-tight, updatable RAM delegation scheme, which gives Theorem 1.5 above.

Applications: Verifiable Delay Functions from Standard Assumptions. Finally, we observe that one of the main applications of non-interactive SPARKs for P from [EFKP20b] was to constructing verifiable delay functions [BBBF18]. Roughly speaking, a VDF is *publicly-verifiable* function that can be computed in time t, but cannot be noticeably sped up with $\mathrm{poly}(t)$ processors. VDFs have important applications in generating trusted randomness in distributed applications (see [BBBF18, Chi, Eth] for more details).

[EFKP20b] showed that any function f can be made verifiable essentially "for free", by computing the output of the f and a proof certifying its correctness using a SPARK for f, and that a VDF can be obtained by simply computing any *sequential function*—that is, a function that can be computed in time t, but cannot be noticeably sped up with $\mathrm{poly}(t)$ processors—with a SPARK. But given that non-interactive SPARKs are only known based on non-falsifiable assumptions, this only gave new VDF constructions assuming non-falsifiable assumptions (namely, the existence of SNARKs for NP).

We note, however, that the transformation in [EFKP20b] actually does not rely on the argument of knowledge property of the underlying SPARK and a SPARG for parallel P computations suffices. Consequently, we can achieve the same results but replacing the SNARK assumptions from [EFKP20b] with just polynomial hardness of LWE.

Theorem 1.7 (VDFs from LWE and any sequential function; Informal (see Corollary 8.2))**.** *Assuming the (polynomial) hardness of LWE and the existence of a sequential function, there exists a verifiable delay function.*

Let us highlight that the assumption that sequential functions exist is *necessary* for the construction of a VDF—any VDF trivially is a sequential function. On top of this minimal assumption, our construction only assume the hardness of LWE. As far as we know, before our work, it was not known how to get VDF (in the plain model, without random oracles) based on any standard polynomial hardness + the assumption that sequential functions exist. In particular, previously, VDFs were known based on either (a) *iteratively-sequential functions*[1] and SNARGs [BBBF18], (b) sequential functions and SNARKs for NP [EFKP20b], or (c) sub-exponential LWE assumption and the sequentiality of repeated squaring in a group of unknown order [LV20], or various construction in the random oracle model [Pie19, Wes19, EFKP20a]. We emphasize that in terms of practical efficiency, our construction does not compete with constructions in the ROM (such as [Pie19, Wes19]), but our goal here is simply to show that VDFs as a primitive can be based on standard hardness assumptions.

As pointed out in [EFKP20b], since a SPARG makes any deterministic computation verifiable, our transformation applies to sequential functions that may satisfy other properties like memory-hardness. We note that memory-hardness is useful for ASIC-resistance in VDFs, making so attackers cannot easily invest in special-purpose hardware and gain an advantage in computing the VDF quicker. Informally, a *memory-hard sequential function* is a sequential function that additionally requires a large memory footprint throughout the computation (for a more formal treatment, see, e.g., [DGN03, DNW05, AS15, ACK$^+$16, ABP17, ABP18, DLP18] for examples of different definitions and constructions of candidate memory-hard functions). It follows that our techniques can be used to achieve a memory-hard VDF based on the hardness of LWE and the existence of any memory-hard sequential function (and our result is not tailored to any specific definition of memory-hardness). Previously, the only known construction of a memory-hard VDF was the construction in [EFKP20b] which relied on the existence of a memory-hard sequential function and SNARKs for NP.

1.2 Related Work

We first focus on the computational assumptions needed for SNARGs and RAM delegation. In the setting of information-theoretic security, the celebrated protocols of Goldwasser, Kalai, and Rothblum [GKR15] and Reingold, Rothblum, and Rothblum [RRR21] first showed how to construct *interactive* delegation protocols for bounded depth and bounded space computations, respectively. Shifting our attention to simple 2-round protocols or non-interactive protocols in the CRS model with only computational security, Kalai, Raz, and Rothblum [KRR14] construct *privately verifiable* delegation for any time and space Turing machines based on the quasi-polynomial hardness of LWE. Kalai and Paneth [KP16] extend this to the setting of privately verifiable RAM delegation, and it was shown how to implement this approach based on polynomial-hardness assumptions by Brakerski, Holmgren, and Kalai [BHK17]. Holmgren and Rothblum [HR18] show how to implement the approach of [KRR14] for RAM delegation with a specific no-signaling MIP with quasi-linear overhead in both time and space, based on the subexponential hardness of LWE. Kalai, Paneth, and Yang [KPY19] achieved the first *publicly verifiable* RAM delegation scheme based on a new falsfiable decisional assumption on groups with bilinear maps. Jawale, Kalai, Khurana, and Zhang [JKKZ21]

[1]An iteratively sequential function f has the property that the t-wise composition $f^{(t)}$ of f cannot be computed faster than computing f sequentially t times, even with $\mathrm{poly}(t)$ processors.

show how to achieve publicly verifiable delegation for bounded depth computation from subexponential hardness of LWE. Finally, Choudhuri, Jain, and Jin [CJJ21] construct publicly veriable RAM delegation from polynomial hardness of LWE.

We note that implicit in the works of [KP16, BHK17, KPY19, CJJ21], building off the techniques of [KRR14], is the notion of a quasi-argument for a class of restricted NP statements. This is an argument system that has a special "no-signaling" extractor for certain NP languages that is used to prove soundness of RAM delegation statements relative to an associated hash tree.

Efficient PCPs. We note that many SNARGs and delegation protocols are based on probabilistically checkable proofs (PCPs) building off the protocols of Kilian [Kil92] (in the interactive setting) and Micali [Mic00] (in the random oracle model using the Fiat-Shamir heuristic [FS86]). Originally PCP constructions required polynomial length and prover running time [BFLS91, ALM+98]. Ben-Sasson and Sudan [BS08] gave the first construction of a PCP with quasi-linear overhead, meaning that a PCP for a t-time (possibly non-deterministic) computation had overall size $t \cdot \mathrm{polylog}(t)$. Subsequent work by [BCGT13] give a highly parallelizable PCP that can be computed in parallel time $\mathrm{polylog}(t)$ with t processors, *after computing the computation tableau.* Interactive oracle proofs (IOPs) are a multi-round generalization of PCPs, introduced in [RRR21] and [BCS16], that are also useful for delegation protocols. There is a fruitful line of work [BCG+17, BCG+19, RR20] resulting in linear-size IOPs useful for delegation, although the prover still runs in at least quasi-linear time.

Parallelism in proofs. The works of [BBBF18] and [DGMV20] first introduced the technique of computing a proof in parallel to a computation in order to improve the prover's parallel efficiency. They first applied this technique to iteratively sequential functions, which necessarily have low space, in the context of verifiable delay functions. The work of [EFKP20b] shows how to apply this technique generically to any, not necessarily space bounded, computation. However, their generic transformation requires interaction or relies on SNARKs in the non-interactive setting.

2 Techniques

In this section, we give an overview of our SPARG constructions. Our constructions will be for RAM computations, so we start with a brief overview of our model. Recall that a RAM machine M is an algorithm with random access to a (possibly long) string D in memory, and keeps a small local state state. At each step of computation, M reads or writes to a location in memory and updates its local state. We say that $M(x)$ outputs y in t steps if, when the initial memory of M contains x, after t steps the local state has a special halting symbol and y is written to memory. The *configuration* cf of a RAM machine at any step of the computation consists of its memory and local state, and hence fully describes the computation at that point.

2.1 SPARGs from LWE

In this section, we overview our construction of SPARGs for P. Our starting point is the non-interactive SPARK construction for NP due to [EFKP20b]. Recall that to construct SPARGs, we are only concerned with proving *soundness* for deterministic, polynomial-time computations, whereas the SPARK construction is an argument of knowledge, which is a stronger notion that in turn relies on assumptions that are too strong for our setting. We start by giving an overview of the SPARK construction, and then discuss how we modify it to achieve SPARGs from weaker assumptions.

SPARK construction. We start by overviewing the SPARK construction of [EFKP20b], henceforth the EFKP construction, which relies on a SNARK for NP. To prove that a $M(x) = y$ in t steps, recall that the goal is for the SPARK prover to run in time at most $t+\mathrm{polylog}\, t$. The high-level approach of EFKP is to split the computation into sub-computations, and give a SNARK proof for each sub-computation in parallel to computing and proving subsequent steps of the computation.

To illustrate this, suppose that the underlying SNARK requires time $2k$ to prove k steps of RAM computation. Then, the largest portion of computation that can be computed and proven by time t is $k = t/3$, as one can spend time $t/3$ computing these steps of the computation, and then spend time $2t/3$ proving that it was done correctly, thus obtaining a proof π_1 of the first $t/3$ steps by time t. The observation of EFKP (following prior works [BBBF18, DGMV20]) is that this idea can be applied recursively. Specifically, while π_1 is being proven, they continue by computing and proving $1/3$ of the remaining computation *in parallel to proving* π_1. Overall, they show that this results in roughly $O(\log t)$ "threads", where each thread computes $1/3$ of the remaining computation, and then begins a SNARK proof while the next parallel thread starts computing. Thus, the full SPARK proof consists of $O(\log t)$ SNARK proofs, all completing by time t. More generally, if the underlying SNARK could prove k steps of computation in time $\alpha^\star \cdot k$, then this would result in having roughly $\alpha^\star \cdot \log t$ proofs (and parallel processors).

While this approach seems promising, it only gives a SPARK for computations with bounded memory size. In particular, it requires giving proofs about intermediate states of the RAM computation. Since the intermediate state of a RAM computation is its configuration cf, the above approach requires using the SNARK to prove statements of the form $(M, \mathsf{cf}, \mathsf{cf}', k)$ stating the M transitions from configuration cf to configuration cf' in time k. However, the size of each configuration scales with the memory size of M, and thus giving SNARK proofs for these statements will depend on the memory size as well.

To remedy this, rather than proving that M transitions from cf to cf' in k steps, EFKP show that the prover can maintain an updatable digest rt to the configuration at any given time step, and prove that there exists a sequence of k updates to rt, according to M, that result in rt'. At a high level, the digest corresponds to a Merkle tree of the memory at each time step based on a collision-resistant hash function, and each time M reads or writes to memory, the corresponding update is done to the Merkle tree. At the end of the computation, the prover can simply open the bits of the output y with respect to the final digest, which the verifier can then check efficiently.

Crucially, each update to the digest can be certified with a very short proof (corresponding to its authentication path in the Merkle tree). Therefore, they rely on a SNARK for the NP language $\mathcal{L}_{\mathsf{upd}}$ that where an instance $(M, \mathsf{rt}, \mathsf{rt}', k)$ has a witness consisting of the k updates to the Merkle tree. The relation for this language has complexity roughly $k \cdot \mathrm{poly}(\lambda)$, as it only requires running M for k steps and checking that each update was done correctly. It is therefore feasible to have a SNARK where the prover overhead for proving $\mathcal{L}_{\mathsf{upd}}$ statements is independent of t. Specifically, EFKP instantiate this framework with a SNARK with *quasilinear overhead*, where an instance corresponding to k updates can be proven in time roughly $k \cdot \mathrm{poly}(\lambda, \log k)$.

Relaxing SPARKs to SPARGs. Given that the EFKP construction relies on an underlying argument of knowledge, a natural approach to constructing a SPARG is to replace the underlying SNARK with a SNARG, and try to prove soundness for computations in P.

Consider the following straightforward attempt to prove soundness with this approach. Suppose for contradiction that there exists an adversary $\mathcal{A}$ who succeeds at convincing the verifier of a false statement (M, x, t, y) where $M(x) \neq y$. Following the EFKP construction, this means that $\mathcal{A}$ outputs sub-proofs $\pi_1, \ldots, \pi_m$, where the ith sub-proof certifies that M transitions from digest

rt_{i-1} to digest rt_i in some number of steps. Ideally, we would like to say that if the statement itself is false, then there must be a sub-proof corresponding to a false statement, hence breaking soundness of the underlying SNARG. However, we cannot claim that this is the case—all the sub-proofs could correspond to true statements if one of them contains a collision in the hash function.

Specifically, it could be the case that for some i, the sequence of updates used by $\mathcal{A}$ to prove that rt_{i-1} transitions to rt_i corresponds to a "divergent" path of computation, and in reality M makes a different sequence of updates after the step corresponding to rt_{i-1}.

The proof of [EFKP20b] relied on the extractability of the SNARK to show that if all sub-statements were true, then $\mathcal{A}$ must be able to produce a hash collision at the point where the computation diverged, in contradiction. However, if we are only relying on a SNARG, we have no way to extract the collision and reach a contradiction.

Nevertheless, we have one advantage over the EFKP approach which we have not yet used—we are only trying to prove soundness for deterministic computations, whereas their proof had to hold even for non-deterministic ones. In particular, this means that given M, x, we can actually compute the true sequence of updates in polynomial time, and thus determine exactly in which sub-proof the computation diverged.

This does not quite solve the problem, because we still have no way to extract a collision between rt_{i-1} and rt_i. However, it does capture an important soundness property, which will turn out to be the key component of our construction. Observe that the above proof of soundness would succeed if the underlying SNARG satisfied the following:

> No PPT adversary $\mathcal{A}$ can produce a proof π, a transcript of the computation of M as well as digests $\mathsf{rt}, \mathsf{rt}'$ and some number of steps k such that (a) the verifier accepts π as a proof for $(M, \mathsf{rt}, \mathsf{rt}', k)$, (b) rt is the correct digest at the beginning of the computation, but (c) rt' is not the correct resulting digest after k steps.

This definition morally captures the fact that $\mathcal{A}$ should not be able to find a collision in the hash function, but does not require extractability to actually produce that collision. In particular, it can be viewed as a notion of soundness relative to a CRH, where the verifier only sees a digest of the statement, yet cannot be convinced on digests of false statements.

From RAM Delegation to SPARGs. We observe that this property stated above is in fact the notion of soundness for RAM delegation schemes. In particular, prior work (such as [KP16, KPY19, CJJ21]) adopted this as a meaningful notion of soundness for RAM delegation to capture the setting where a weak verifier, who may have pre-computed a digest of a large database, delegates a computation on that database and can verify the updated digest after the computation to enable future outsourcing on the updated database.

Putting everything together, to prove soundness of the EFKP construction for deterministic computations, it suffices to rely on a RAM delegation scheme with the above soundness notion, rather than a SNARK. By relying on the recent RAM delegation scheme due to [CJJ21], we obtain a sound scheme based only on LWE.

Updatable Delegation. There is one remaining caveat to the construction, which is that by replacing the SNARK with a delegation scheme, we have to ensure that each sub-proof computed using the delegation scheme can be done with low prover overhead so that the resulting construction satisfies the tight efficiency requirements of a SPARG.

Looking at the delegation scheme due to [CJJ21], in order to delegate the computation of M starting at configuration cf, the scheme first computes a Merkle tree of cf (analogously to the Merkle tree approach in [EFKP20b]), and then proceeds to compute the updates to the Merkle, and prove their correctness using underlying building blocks. We observe that other than computing this initial Merkle tree, the delegation prover has quasilinear overhead. Specifically, we show that when delegating a statement corresponding to k steps of computation, everything other than computing the initial Merkle tree can be done in time $k \cdot \text{poly}(\lambda, \log k)$.

To put this into context in our scheme, recall that we will be breaking up the computation of M into sub-computations, indexed by configurations $\mathsf{cf}_0, \mathsf{cf}_1, \ldots, \mathsf{cf}_m$, for which we will then use the delegation scheme to prove that cf_{i-1} transitions to cf_i for each sub-computation i. However, if the delegation prover then hashes down each cf_i at the beginning of each sub-proof, the running time of our SPARG will then rely on the memory size, which as mentioned above, does not suffices for us.

We resolve this by using another piece of the EFKP construction, specifically their Merkle tree instantiation. Recall that they gave a construction, termed a *concurrently updatable hash function*, which enabled updating the Merkle tree in parallel to the computation with very little overhead. We observe that if the Merkle tree in the RAM delegation scheme is instantiated with a concurrently updatable hash function, then when computing each configuration cf_i, we can compute in parallel the Merkle tree digest of cf_i, and give this to the delegation prover as auxiliary input.

At a high level, this captures a notion which we call *updatability* for RAM delegation schemes, since while running the computation from computing a proof that cf_{i-1} transitions to cf_i, the Merkle tree for cf_i computed during the proof can be given to the next prover.

We show that the [CJJ21] scheme satisfies this notion of upatability when instantiated with the hash tree due to [EFKP20b], and that this notion of updatability suffices to achieve the required prover efficiency from the delegation scheme in order to instantiate the EFKP framework and obtain a SPARG for P.

2.2 SPARGs for Parallel Computations

The above framework gives a SPARG for *sequential* computations—namely, a proof system that runs in time $t + \text{poly}(\lambda, \log t)$ for t-time computations. However, it is very natural to consider the setting where the computation itself can be parallelized. In this setting, we show that our SPARG construction can be extended to prove parallel computations while preserving the depth of the computation. Specifically, for computations that take time t with p processors, our SPARG will run in time $t + \text{poly}(\lambda, \log(t \cdot p))$ with $p \cdot \text{poly}(\lambda, \log(t \cdot p))$ processors.

To achieve this, recall that the prover in our SPARG construction above splits the computation into many sub-computations. For each sub-computation, the prover runs the computation in parallel to updating a hash tree to its memory. It then uses an updatable RAM delegation scheme to prove correctness of this sub-computation. Efficiency of the resulting construction relies on the fact that (1) computing $M(x)$ and updating the hash tree can be done in parallel in time essentially t, and (2) the delegation scheme has quasi-linear overhead, so proving any sequence of k steps takes time $k \cdot \text{poly}(\lambda, \log k)$.

To extend this to the setting of parallel computations, we observe that the prover can run the computation in time t with p processors. Moreover, the hash tree due to [EFKP20b] allows for concurrent updates, and so the updates can be done in parallel to the computation. However, a challenge arises when using the updatable RAM delegation scheme in this setting, as we have to prove correctness of *concurrent* updates. Specifically, for a sub-computation corresponding to k steps, the concurrent updates to the hash tree result in k updates each to p locations in memory (as opposed to a single location each, as in the sequential case). The efficiency of our updatable RAM delegation scheme depends, in particular, polynomially on the time to verify a single update, which is $\mathrm{poly}(p)$ when considering concurrent updates. Therefore, this would not result in a delegation scheme with quasilinear prover efficiency—instead, the prover time would depend polynomially on p, which is undesirable when p is large.

The dependence on the time to verify a single update is inherent to our updatable RAM delegation construction, and in particular stems from the underlying building blocks used in [CJJ21]. Therefore, it is not immediately clear how to move forward—in order to avoid any delays with running the main computation, we have to perform concurrent updates, but the delegation scheme is incompatible with these updates.

To solve this, we observe that we can transform *concurrent* updates to *sequential* ones—namely, k concurrent updates on p locations each can be turned into $k \cdot p$ updates, each to a single location. We call a hash tree with this property *sequentializable*. At a high level, we do so by taking advantage of the Merkle tree structure, and the fact that an authentication path for an individual location ℓ can be derived from the updates to a set of locations containing ℓ. We form the authentication paths corresponding to the sequential updates level by level, resulting in time $\mathrm{poly}(\lambda, \log p)$ to sequentialize a concurrent update when using p processors. Therefore, for a sub-computation with k steps, we can sequentialize the updates in parallel time $k \cdot \mathrm{poly}(\lambda, \log p)$. Crucially, sequentializing the updates does not delay the main computation of $M(x)$—instead, the sequentialization can be seen as part of the "proof" phase, before calling the RAM delegation prover.

After sequentializing the updates, a k-time sub-computation results in $k \cdot p$ individual updates. We are not quite done, because applying our updatable RAM delegation scheme to prove correctness of these updates would result in time quasilinear in the total work $k \cdot p$, rather than simply k. As the final step in our construction, we observe that the computation of the RAM delegation proof can be parallelized as well. Specifically, recall that our RAM delegation scheme is given the updates as a witness to the computation, and is only required to compute the proof. When given $T = t \cdot p$ sequentialized updates, it runs in quasilinear time $T \cdot \mathrm{poly}(\lambda, \log T)$. As a final observation, we show that for any number of processors p, the RAM delegation prover can be made to run in time $T/p \cdot \mathrm{poly}(\lambda, \log T)$ with p processors, when given these updates. At a high level, this follows due to the fact that the underlying updatable delegation scheme treats the T updates as a batch of T individual statements for which it proves correctness. In particular, we show that the proofs of these statements (and the information tying them together) can be computed in parallel, thus giving the desired efficiency.

Putting everything together, the combination of sequentializing the updates and running the parallelized delegation prover gives the desired quasilinear efficiency for our RAM delegation scheme, which in turn suffices to get a SPARG for parallel computations.

2.3 Time-Independent SPARGs

We consider the application of SPARGs to *time-tight* RAM delegation, where by time-tight we mean a delegation protocol that satisfies the same efficiency properties as a SPARG. So far, we have assumed that the time bound t for the computation is provided as input. This seems like the a natural requirement as we have to compute the proof of the computation *completely during* the computation itself. We show that this is actually not necessary, at least in the case of non-interactive delegation. In particular, we show how to construct a non-interactive SPARG for any t-time computation $M(x)$ where t does not need to be provided as input—we refer to this as a *time-independent* SPARG—given a non-interactive SPARG that does take as input the time bound t. (In fact, we actually use a time-tight RAM delegation scheme in order to break up the computation into different parts, which we will discuss more below.)

As a first attempt, what if the prover computed a SPARG for all possible time bounds T? The

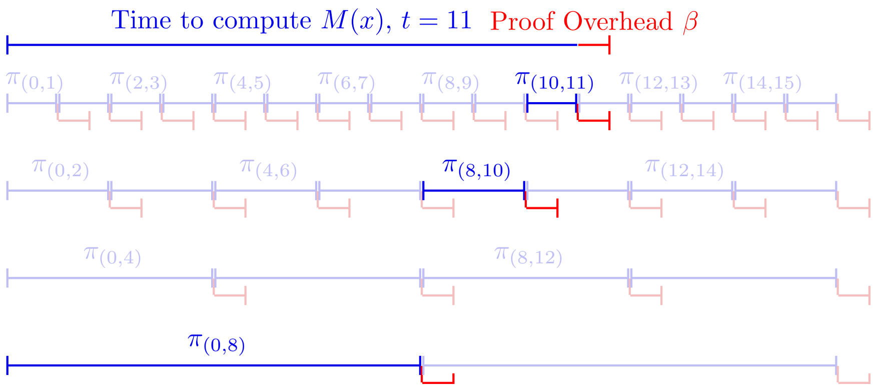

Figure 1: An example of the time-independent SPARG prover for a computation $M(x)$ that takes $t = 11$ steps. A proof $\pi_{(a,b)}$ corresponds to a RAM delegation proof that M on input x starts at configuration cf_a and ends at configuration cf_b. The horizontal axis represents parallel time, and the prover is computing all proofs along a given vertical slice in parallel. Each separate thread corresponds to a memory block that is being updated and outputting proofs for the corresponding intervals at the same time. The additive overhead per interval is indicated in red, but can be computed separately while the subsequent update continues. The final proof output by the prover consists of the sub-proofs corresponding to the 1's in the binary representation of the actual time t. For $t = 11$ shown in the picture, this corresponds to the 8, 2, and 1 digits, so the prover eventually outputs the proofs $\pi_{(0,8)}$, $\pi_{(8,10)}$, and $\pi_{(10,11)}$. All other proofs are discarded, and are thus greyed out in the picture above.

prover could run the computation on the side, see when it halts, and use the proof corresponding to the actual time bound t, ignoring all other proofs. If we compute all the SPARG proofs in parallel, then the prover will compute a proof in the desired parallel time, but this requires using more than t processors! Even worse, we don't necessarily know a priori a bound on what the running time will be, so we would even need to use potentially super-polynomially many processors to handle all polynomial-time computations. Instead, we want to compute the time-independent SPARG using only a modest, say fixed polynomial $\text{poly}(\lambda)$ in the security parameter, overhead in the number of processors required.

In an effort to reduce the number of processors used, the prover could instead compute proofs only for the time bounds $T = 2^0, 2^1, 2^2, 2^4, \ldots, 2^\lambda$, assuming that the polynomial time bound t is at most 2^λ for large enough security parameters λ. Now we only have a $\lambda + 1$ overhead in the number of processors required. However, if in computing $M(x)$ we find out that the true time bound t is not close to a power of 2, then we may have a factor of 2 over head in the time to compute the next largest proof that encapsulates the full computation. Even a small multiplicative overhead is not allowed for SPARGs, so this approach unfortunately does not achieve what we want.

In order to maintain optimal parallel time with only a small overhead in the number of processors used, we leverage the techniques described in Section 2.1 to break down the proof of the entire computation into proofs of various sub-computations while still guaranteeing soundness. This is why we actually need to start with RAM delegation for our underlying scheme so that breaking the proofs into many parts does not scale with the space of the underlying computation.

The idea of the full construction is to compute proofs for the time bounds $T = 2^0, 2^1, \ldots, 2^\lambda$, but after each proof of size 2^i finishes, to continue to compute proofs in regular intervals of size 2^i that continue that computation (using the same associated memory). So, for every size 2^i and every $j \geq 1$, we will have a proof corresponding to the interval of the computation between steps $(j-1) \cdot 2^i$ and $j \cdot 2^i$. For any such starting point a and ending point b, we let $\pi_{(a,b)}$ denote the associated proof. Ignoring efficiency for now, this means that after the machine $M(x)$ halts at time t, we can simply collect m proofs $\pi_{(0,a_1)}$, $\pi_{(a_1,a_2)}$, $\ldots$, $\pi_{(a_{m-1},t)}$ that cover the entire interval from 0 to t via intervals of powers of 2. These intervals will then simply correspond to the binary representation of t, so there will be $m \leq \lambda$ proofs in total.

In order to make this approach work, we need a specific, extremely efficient, underlying RAM delegation scheme. Concretely, we need it to be the case that we can have a thread of computation that computes proofs for all size 1 intervals $(0,1)$, $(1,2)$, $(2,3)$, $\ldots$, $(t-1,t)$ without blowing up the complexity of the protocol. Fortunately, our main SPARG construction actually gives us an updatable delegation scheme that is also *time-tight*. Essentially, this is an updatable delegation scheme where an update of any sequence of k steps also outputs a proof of correctness for those k steps. Furthermore, these updates and proofs can be pipelined together efficiently, ensuring that computing a proof for all size 1 intervals in a row as above does not blow up the overall complexity nor delay the output of later proofs in the sequence (namely the proof for the interval $(i, i+1)$ still finishes at time $i+1+\text{poly}(\lambda)$, where the delay is independent of i or t).

We finish by arguing why the protocol is succinct and satisfies the optimal parallel time requirement of a SPARG while using only a fixed $\text{poly}(\lambda)$ number of processors. For succinctness, recall the number of proofs that the prover needs to output is simply the number of 1s in the binary representation of the actual time bound t. Assuming $t < 2^\lambda$, this implies that the number of delegation proofs m that need to be sent at most λ, so there is at most a λ overhead in the size of the proofs for the time-independent SPARG over the underlying RAM delegation scheme.

Analyzing the running time of the prover, we note that by assumption, the underlying updatable delegation scheme has only an additive overhead of some polynomial $\beta(\lambda)$ to compute proofs with its updates, using at most $\beta(\lambda)$ processors for each update procedure. All of the required proofs finish by time $t+\beta(\lambda)$, so the prover satisfies the required runtime efficiency, we just need to bound the number of processors used. As each update/ proof computation uses $\beta(\lambda)$ processors, we just need to bound the number of update procedures happening at any given time. To do so, consider any T steps into the computation. All proofs $\pi_{(a,b)}$ for a final configuration cf_b where $b < T - \lambda \cdot \beta(\lambda)$ have already been completed, as described above, so there are most $\lambda \cdot \beta(\lambda)$ proofs in progress for ending configurations at or before cf_T. Also, for each size 2^i, there is at most one proof of size 2^i that could have been started and ends after cf_T. This implies that are at most $\lambda + \lambda \cdot \beta(\lambda)$ updates computed at any given time, so the prover requires only a $\text{poly}(\lambda)$ number of processors in total.

We emphasize that this transformation fundamentally relies on the fact that the underlying delegation scheme is "time-tight" like a SPARG. Otherwise the overlap among all of the proofs would be too great, and the protocol would require too many processors.

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 $\mathcal{X}$, we denote by $x \leftarrow \mathcal{X}$ the process of sampling a value x from the uniform distribution on $\mathcal{X}$. $\text{Supp}(X)$ 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 $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ there exists a negligible function negl such that $\left|\Pr\left[\mathcal{A}_\lambda(1^\lambda, X_\lambda) = 1\right] - \Pr\left[\mathcal{A}_\lambda(1^\lambda, Y_\lambda) = 1\right]\right| \leq \mathsf{negl}(\lambda)$ 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.

3.1 RAM Model

RAM computation consists of a machine M which keeps some local state state and has read/write access to memory $D \in (\{0,1\}^\lambda)^*$ (equivalent to the tape of a Turing machine). Here, λ is the security parameter and length of a word, and we let $n \leq 2^\lambda$ be the number of words in memory required to run M (see below). When we write $M(x)$ to denote running M on input x, this means that M expects its initial memory D to consist of x followed by zeros. The computation of $M(x)$ is defined in steps, where at each step the machine either reads or writes to a location in memory and updates its local state. We assume that when M writes to a memory location ℓ, it receives the word previously at ℓ. Without loss of generality, we assume that the state can hold $O(\log n)$ bits, or a constant number of words, and that the local state at each time step includes the word read in the previous step. We also assume that n words in memory can be allocated and initialized to zeros for free.

The computation halts when the local state consists of a special halting value with the output y of $M(x)$ written at the start of the memory. We define the running time of a RAM machine M as the number of accesses it makes to its working memory, which corresponds to the number of steps.

We define the *configuration* cf at any step of the computation to include the local state and full memory at that step. This representation allows us to refer to RAM machines that transition from a configuration cf to configuration cf' in some number of steps, as the configuration has all information required to perform a step.

In order to measure the complexity of RAM computation, we note that on a fixed CPU architecture, RAM computation can be modeled where the program M and input x are both given in memory and executed using a fixed machine U. We therefore fix any universal RAM machine U and define the complexity of running $M(x)$ to be the number of steps required to run $U(M, x)$. As all of our RAM computation will be in this model, for simplicity we say that $M(x)$ requires access to n words of memory if $U(M, x)$ uses a total of n words in memory to write M, x, and all the memory used by the computation. Henceforth, we say that $M(x)$ halts in time t if running U on memory $M||x||0^{n-|M,x|}$ for t steps results in a halting state.

Parallel RAM Computation. We will also consider computations 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. We assume that all processes in a PRAM computation have local registers that can be used to communicate the results of each step.

Unless otherwise stated, we consider RAM machines to be in the exclusive-read exclusive-write (EREW) model. This is the most restrictive PRAM model, where if some process accesses a location (either a read or a write) in memory while another process accesses the same location (either a read or a write), there are no guarantees for the resulting effect. For some of our results, we give constructions in the concurrent-read exclusive-write (CREW) model, where processes that read concurrently from a single location both receive the value at that location, but there are no guarantees on concurrent writes.

To model the complexity of a (P)RAM machine M, we consider two complexity measures: work and depth. Specifically, the work done by $M(x)$ consists of the total amount of computation done by all processors measured in steps (or equivalently memory accesses). The depth of $M(x)$ is the number of sequential steps until M halts, where steps that occur in parallel are counted as one step.

3.2 Universal Languages

In this section we define a universal language for deterministic RAM computation with long output, following the universal relation introduced by [BG08].

Definition 3.1. *The universal language $\mathcal{L}^{\mathcal{U}}$ is the set of instances (M, x, y, L, t) where M is a deterministic RAM machine such that $M(x)$ outputs y within t steps, and additionally $|y| \leq L$. We extend this to parallel computations by letting $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ denote the set of instances (M, x, y, L, t, p) where M is instead a PRAM machine that outputs in t parallel steps using p processors.*

Additionally, we will be considering intermediate portions of RAM computation, where the universal RAM machine U (see Section 3.1) transitions from configuration cf to cf' in t steps.

Definition 3.2. *The universal RAM delegation language $\mathcal{L}^{\mathsf{del}}$ is the set of instances $(\mathsf{cf}, \mathsf{cf}', t)$ such that the universal RAM machine U transitions from configuration cf to configuration cf' in t steps. We extend this to PRAM delegation by letting $\mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ denote the set of instances $(\mathsf{cf}, \mathsf{cf}', t, p)$ such that U transitions from cf to cf' in t parallel steps with p processors.*

3.3 Verifiable Delay Functions

In this section, we recall the definition of Verifiable Delay Function [BBBF18]. This definition is adapted from [EFKP20b].

Definition 3.3. *A* Verifiable Delay Function *(VDF) is a tuple of algorithms* $(\mathsf{Gen}, \mathsf{Sample}, \mathsf{Eval}, \mathsf{Verify})$ *with the following syntax:*

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda)$**:** *A PPT algorithm that on input a security parameter λ in unary, outputs public parameters* pp.
- $x \leftarrow \mathsf{Sample}(1^\lambda, \mathsf{pp})$**:** *A PPT algorithm that on input a security parameter λ in unary, and public parameters* pp*, outputs an value x.*
- $(y, \pi) \leftarrow \mathsf{Eval}(1^\lambda, \mathsf{pp}, x, t)$**:** *An algorithm that on input a security parameter λ in unary, public parameters* pp*, and a value x and integer t, outputs a value y and a proof π. We let* $\mathsf{Eval}_1, \mathsf{Eval}_2$ *denote the functions that compute only the first or second output of* Eval*, respectively, and require that* Eval_1 *is deterministic.*
- $b \leftarrow \mathsf{Verify}(1^\lambda, \mathsf{pp}, x, t, (y, \pi))$**:** *A PPT algorithm that on input a security parameter λ in unary, public parameters* pp*, value x, integer t, value y and proof π, outputs a bit b indicating whether to accept or reject.*

We require the following properties.

- **Completeness.** *For every $\lambda, t \in \mathbb{N}$,* pp *in the support of* $\mathsf{Gen}(1^\lambda)$*, $x \in \{0,1\}^*$,*

$$\mathsf{Verify}(1^\lambda, \mathsf{pp}, x, t, \mathsf{Eval}(1^\lambda, \mathsf{pp}, x, t)) = 1.$$

- **Soundness.** *For every non-uniform PPT algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$ and polynomial T, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) \\ (x, y', \pi') \leftarrow \mathcal{A}_\lambda(\mathsf{pp}) \\ y = \mathsf{Eval}_1(1^\lambda, \mathsf{pp}, x, T(\lambda)) \end{array} : \begin{array}{l} \mathsf{Verify}(1^\lambda, \mathsf{pp}, x, T(\lambda), (y', \pi')) = 1 \\ \wedge\ y \neq y' \end{array}\right] \leq \mathsf{negl}(\lambda).$$

- **Honest Evaluation.** *There exist polynomials p, q such that for any $\lambda \in \mathbb{N}$,* pp *in the support of* $\mathsf{Gen}(1^\lambda)$*, and x in the support of* $\mathsf{Sample}(1^\lambda, \mathsf{pp})$*, it holds that* $\mathsf{Eval}(1^\lambda, \mathsf{pp}, x, t)$ *can be computed in time $t + p(\lambda, \log t)$ with $q(\lambda, \log t)$ processors.*

- **ϵ-Sequentiality.** *For all non-uniform PPT algorithms $\mathcal{A}_0 = \{\mathcal{A}_{0,\lambda}\}_{\lambda\in\mathbb{N}}$, there exists a negligible function* negl *such that for all $\lambda \in \mathbb{N}$,*

$$\Pr\left[\begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda) \\ \mathcal{A}_1 \leftarrow \mathcal{A}_0(\mathsf{pp}) \\ x \leftarrow \mathsf{Sample}(1^\lambda, \mathsf{pp}) \\ (t, y) \leftarrow \mathcal{A}_1(x) \end{array} : \begin{array}{l} \mathsf{Eval}_1(1^\lambda, \mathsf{pp}, x, t) = y \\ \wedge\ \mathsf{depth}(\mathcal{A}_1) \leq (1 - \epsilon) \cdot t \end{array}\right] \leq \mathsf{negl}(\lambda).$$

We note that the main differences between various definitions of VDFs is the running time of the honest evaluator. For example, the definition of [BBBF18] requires that VDF.Eval can be computed in time *exactly* t, whereas other works (e.g., [Pie19, EFKP20a, EFKP20b]) relax this. We allow for additive slack in this evaluation time. Regardless of the specific running time of VDF.Eval, we emphasize that the hardness of a VDF can be measured by considering the "gap" between honest evaluation and sequentiality, which intuitively measures how much of a speedup an adversary can gain over an honest evaluator. We also note that our definition of soundness is slightly weaker than, e.g., [EFKP20a], as we only require soundness on a priori fixed polynomials T.

We additionally define a sequential function.

Definition 3.4. *A* Sequential Function *is a tuple of algorithms* (Gen, Sample, Eval) *with the syntax of the corresponding algorithms* (VDF.Gen, VDF.Sample, $\mathsf{VDF.Eval}_1$) *in a VDF, that satisfy honest evaluation and sequentiality, and for which there exists a polynomial p such that for any $\lambda, t \in \mathbb{N}$,* pp *in the support of* $\mathsf{Gen}(1^\lambda)$*, and $x \in \{0,1\}^\star$, the output length of* $\mathsf{Eval}(1^\lambda, \mathsf{pp}, x, t)$ *is at most $p(\lambda, \log t)$.*

We remark that the requirement on the output length of a sequential function is necessary for the function to be non-trivial. In particular, without this requirement, the sequential function may simply output 1^t. We also note that a VDF implies a sequential function. This follows by considering the first output of the VDF evaluation function to be the output of the sequential function. The resulting sequential function has bounded output due to the efficiency of verification in a VDF.

3.4 RAM Delegation

In this section, we define RAM delegation, which will be the main building block for our SPARG construction. Following [BHK17, KP16, KPY19, CJJ21], we define RAM delegation to capture the following scenario: A verifier wishes to delegate a RAM computation M with some initial configuration cf, such that running M for t steps starting with cf results in configuration cf'. As M may potentially use a large amount of memory, these configurations could be very long, and thus the approach in recent works has been to consider a verifier that only receives digests $\mathsf{rt}, \mathsf{rt}'$ of the configurations $\mathsf{cf}, \mathsf{cf}'$.

Recently, [KPY19, CJJ21] showed delegation schemes for RAM where soundness holds when the verifier only receives these digests, and moreover suffice to delegate general computation with Turing machines. We adopt this notion for this work. As discussed in Section 3.1, we will assume that the machine M is already part of the memory in cf and thus give a definition for a fixed universal RAM computation with the universal machine U.

Definition 3.5 (RAM Delegation)**.** *A publicly verifiable, succinct RAM delegation scheme for $\mathcal{L}^{\mathsf{del}}$ is a tuple of probabilistic algorithms* $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V})$ *with the following syntax:*

- $(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda)$*: A PPT algorithm that on input a security parameter λ outputs a common reference string* crs *and a digest key* dk*. We assume without loss of generality that* crs *contains* dk*.*

- $\mathsf{rt} = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$*: A deterministic algorithm that on input a digest key* dk *and a RAM configuration* cf *outputs a digest* rt*.*

- $\pi \leftarrow \mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t))$*: A probabilistic algorithm that on input a common reference string* crs*, and a statement* $(\mathsf{cf}, \mathsf{cf}', t)$*, outputs a proof π.*

- $b \leftarrow \mathsf{Del.V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t), \pi)$*: A PPT algorithm that on input a a common reference string* crs*, common reference string* crs*, statement* $(\mathsf{rt}, \mathsf{rt}', t)$*, 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 $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{L}^{\mathsf{del}}$ with $t, n \leq 2^\lambda$ where n is the memory size of the configurations, it holds*

$$\Pr\left[\begin{array}{l} (\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda) \\ \mathsf{rt} = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}) \\ \mathsf{rt}' = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}') \\ \pi \leftarrow \mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t)) \end{array} : \mathcal{V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t), \pi) = 1 \right] = 1.$$

- **Soundness:** *For any non-uniform polynomial-time algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$, polynomial-time computable function T, and polynomial $\overline{T}$ such that $T(\lambda) \leq \overline{T}(\lambda)$ for all $\lambda \in \mathbb{N}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l} (\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda) \\ (\mathsf{cf}, \mathsf{cf}', \mathsf{rt}, \mathsf{rt}', \pi) \leftarrow \mathcal{A}_\lambda(\mathsf{crs}, \mathsf{dk}) \end{array} : \begin{array}{l} \mathcal{V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t), \pi) = 1 \\ \wedge\ (\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{L}^{\mathsf{del}} \\ \wedge\ \mathsf{rt} = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}) \\ \wedge\ \mathsf{rt}' \neq \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}') \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where $t = T(\lambda)$.

- **Collision resistance:** *For any non-uniform polynomial-time algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l} (\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda) \\ (\mathsf{cf}, \mathsf{cf}') \leftarrow \mathcal{A}_\lambda(\mathsf{crs}, \mathsf{dk}) \end{array} : \begin{array}{l} \mathsf{cf} \neq \mathsf{cf}' \\ \wedge\ \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}') \end{array}\right] \leq \mathsf{negl}(\lambda).$$

- **Succinctness:** *There exist polynomials q_1, q_2, q_3 such that for any $\lambda \in \mathbb{N}$, $(\mathsf{crs}, \mathsf{dk})$ in the support of $\mathsf{Del.S}(1^\lambda)$, $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{L}^{\mathsf{del}}$, and proof π in the support of $\mathcal{P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t))$, it holds that*
    - $|\mathsf{Del.V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t), \pi)| \leq q_1(\lambda, \log t)$ *and*
    - $|\pi| \leq q_2(\lambda, \log t)$.
    - $\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$ *is computable in time* $|\mathsf{cf}| \cdot q_3(\lambda)$ *and has output length length* λ.

3.5 SPARGs

In this section, we define SPARGs for P based on the notion of SPARKs introduced in [EFKP20b]. We note that while they do not restrict to computations with $t \leq 2^\lambda$ steps, we require this as it is standard in related notions (e.g., RAM delegation) and required for our construction.

Definition 3.6 (Non-interactive SPARGs for P)**.** *A* Non-interactive Succinct Parallelizable Argument *for a language $\mathcal{L} \subseteq \mathcal{L}^{\mathcal{U}}$ is a tuple of probabilistic algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ with the following syntax:*

- $\mathsf{crs} \leftarrow \mathcal{G}(1^\lambda)$: *A PPT algorithm that on input a security parameter λ outputs a common reference string* crs.
- $(y, \pi) \leftarrow \mathcal{P}(\mathsf{crs}, (M, x, L, t))$: *A probabilistic algorithm that on input a common reference string* crs*, and a statement (M, x, L, t), outputs a value y and a proof π.*
- $b \leftarrow \mathcal{V}(\mathsf{crs}, (M, x, y, L, t), \pi)$: *A PPT algorithm that on input a common reference string* crs*, 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 $(M, x, y, L, t) \in \mathcal{L}$ where M has access to $n \leq 2^\lambda$ words in memory and $t \leq 2^\lambda$,*

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

- **Soundness for P:** *For all non-uniform polynomial-time provers $\mathcal{P}^\star = \{\mathcal{P}^\star_\lambda\}_{\lambda \in \mathbb{N}}$ and every polynomial T, there is a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{G}(1^\lambda) \\ ((M, x, y, L), \pi) \leftarrow \mathcal{P}^\star_\lambda(\mathsf{crs}) \end{array} : \begin{array}{l} \mathcal{V}(\mathsf{crs}, (M, x, y, L, t), \pi) = 1 \\ \wedge\ (M, x, y, L, t) \notin \mathcal{L} \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where $t = T(\lambda)$.

- **Succinctness:** *There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$,* crs *in the support of $\mathcal{G}(1^\lambda)$, $(M, x, L, t) \in \mathcal{L}$ where M uses $n \leq 2^\lambda$ words in memory, $t \leq 2^\lambda$, and (y, π) in the support of $\mathcal{P}(\mathsf{crs}, (M, x, L, t))$, it holds that*
    - $\mathsf{work}_{\mathcal{V}}(\mathsf{crs}, (M, x, y, L, t), \pi) \leq q_1(\lambda, |(M, x)|, L, \log t)$,
    - $|y| \leq L$, *and*
    - $|\pi| \leq q_2(\lambda, L, \log t)$.

- **Optimal prover depth:** *There exists polynomials q_1 and q_2 such that for all $\lambda \in \mathbb{N}$ and $(M, x, t, L, y) \in \mathcal{L}$ where M has access to $n \leq 2^\lambda$ words in memory and $t \leq 2^\lambda$, it holds that*
$$\mathsf{depth}_{\mathcal{P}}(\mathsf{crs}, (M, x, L, t)) = t + q_1(\lambda, |(M, x)|, L, \log t)$$
and the total number of processors used by $\mathcal{P}$ is in $q_2(\lambda, \log t)$.

If the above holds for $\mathcal{L} = \mathcal{L}^{\mathcal{U}}$, we say that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a non-interactive SPARG for polynomial-time RAM computation*.*

4 Updatable RAM Delegation

In this section, we discuss the main building block for our construction—updatable RAM delegation with quasilinear overhead and local opening. Our starting point will be the recent delegation scheme due to Choudhuri, Jain, and Jin [CJJ21], henceforth referred to as the CJJ construction. In Section 4.1, we start by giving an overview of their construction, and show that it satisfies our definition of RAM delegation. Then, in Section 4.2, we analyze the CJJ prover overhead, which, as we show, depends on the memory size of the computation and therefore is not quasi-linear. To remedy this, we introduce the notion of updatable delegation, and show that the CJJ scheme can be instantiated as an updatable delegation scheme with quasi-linear prover overhead. Finally, in Section 4.3, we extend this notion to satisfy a local opening property, which will be necessary for our construction.

4.1 The CJJ Delegation Scheme

We start by giving an overview of the CJJ delegation scheme. We note that they present their construction for a specific RAM machine M, but we simply treat this as the universal RAM machine U.

The CJJ construction relies on the following building blocks:

- A hash tree that supports local reads and writes. This can be instantiated from collision-resistant hash functions.
- A no-signalling somewhere-extractable commitment scheme, with a locality parameter ℓ corresponding to the size of extracted sets, which in particular determines the efficiency of the commitment.
- A non-interactive batch argument (BARG) for NP. This is an argument where k instances of a language can certified with a proof that only depends sub-linearly on k.

At a high level, their construction follows an approach in recent works (see, e.g., [KP16, KPY19, EFKP20b]) which uses a locally updatable hash tree (based on Merkle trees) to succinctly prove that each step of RAM computation was done correctly. Specifically, to prove that a RAM machine transitions from configuration cf to configuration cf' in t steps, they run the computation while simultaneously maintaining a hash tree of the memory at each step. Each step can then be verified succinctly (in particular in time independent of $|\mathsf{cf}|$) by verifying succinct local openings to the hash tree. To turn this approach into a full-fledged delegation scheme, previous works have employed a combination of succinct proof systems with various extractability properties to show soundness.

In the CJJ construction, they follow this framework. After running the computation along with computing a short opening to the hash tree at each step, they give a no-signalling commitment

c to the sequence of t updates to the hash tree. They then prove, using a BARG, that each step of the computation was done correctly and consistently. Specifically, the BARG is for the relation computed by the circuit C_{step} that on input an index i and openings to c corresponding to the ith step of computation, checks that (1) these openings are consistent with c, (2) correspond to a valid step of computation, and (3) are valid openings to the hash tree. To show that this construction is sound, they rely on a combination of BARG soundness, the no-signalling extraction of the commitment scheme, and collision resistance. They show that this results in a scheme for (deterministic) RAM delegation which can be based solely on LWE.

Next, we discuss the differences between the notion of RAM delegation satisfied by this construction, and Definition 3.5.

Dependence on t. The CJJ scheme gives delegation for a fixed time bound t, and in particular the setup algorithm depends on t. We observe that it suffices for the setup algorithm to assume a fixed upper bound of 2^λ on t. Regarding efficiency, the crs size, proof length, and running time of the verifier are all bounded by a fixed polynomial $\mathrm{poly}(\lambda, \log t)$. Thus, from an efficiency standpoint, replacing t with 2^λ in the setup phase still results in a succinct argument.

Regarding correctness, t is used in two places in the setup algorithm. First, it is used to sample the key for the no-signalling commitment, because it determines the length of messages that the commitment scheme is required to support. The construction of the commitment scheme is based on [HW14], and is essentially a Merkle tree composed with fully homomorphic encryption (FHE) scheme, where FHE evaluation is done on each pair of sibling nodes to determine the value of their parent node. To do so, a separate FHE key is required for each level. It thus suffices to simply generate λ such keys, as there will be at most $\log t \leq \lambda$ levels. It follows that the functionality of the commitment scheme is preserved even without prior knowledge of the message length.

The second place where t is used by the setup algorithm is when sampling public parameters for BARG. The BARG construction, also given in [CJJ21], relies on a BARG for 2^i statements for $i = 1, \ldots, \log t$, and uses a priori knowledge of t to generates the CRS for each scheme. Similar to the case of the commitment scheme, it suffices to generate λ keys for this upfront, where the ith CRS will be for a BARG for 2^i statements.

Putting everything together, it follows that the setup for the CJJ construction can be made independent of t. In order to support this change, we give t to the prover and verifier in the clear, as it will no longer be implicit in the CRS.

Soundness for polynomially-bounded T. In the CJJ scheme, the soundness property requires that for every PPT adversary $\mathcal{A}_\lambda$ and polynomial T, the probability of $\mathcal{A}_\lambda$ successfully proving a $T(\lambda)$-step false statement (that is, a statement $(\mathsf{cf}, \mathsf{cf}', T(\lambda))$ where running the machine starting at configuration cf for $T(\lambda)$ steps does not result in configuration cf') is negligible. For technical reasons, we will require soundness to hold even when T is not a polynomial, as long as it is both polynomial-time computable and polynomially bounded. The CJJ scheme directly extends to hold in this case.

Delegation for arbitrary computation. The final difference between the CJJ construction and our definition is that the construction gives a delegation scheme for an a priori fixed RAM program M. This is because the setup algorithm for their scheme depends on $|M|$. As mentioned above, we will be using this for the universal machine U with an a priori fixed size. We therefore omit the size of the RAM machine to the setup algorithm.

Putting everything together, it follows that the CJJ scheme satisfies our notion of RAM delegation. As their scheme is based on LWE, the following holds.

Theorem 4.1 ([CJJ21])**.** *Assuming the hardness of LWE, there exists a publicly verifiable, succinct RAM delegation scheme for $\mathcal{L}^{\mathsf{del}}$.*

4.2 Updatable Delegation with Quasilinear Overhead

For our SPARG construction, we will be concerned with delegation schemes with tight prover efficiency. In this section, we analyze the prover efficiency of the CJJ construction, and then show that it can be made quasilinear in t when the prover is additionally given a *witness* for the RAM computation. Along the way, we introduce the notion of Updatable Delegation, which enables the desired prover efficiency and may be of independent interest.

We start by looking at the efficiency of each building block in the CJJ scheme individually.

- Hash tree: The hash tree used in [CJJ21] is effectively a Merkle tree based on a collision resistant hash function. Computing the hash tree of a given configuration cf can be done in time $|\mathsf{cf}| \cdot \mathrm{poly}(\lambda)$, but when given the hash tree already in memory, updating a word in the tree can be done in time logarithmic in the size of the memory of the RAM program, and so can be done in time $\mathrm{poly}(\lambda)$.

- BARG: Recall that the BARG enables proving k instances of an NP relation computable by a circuit C. At a high level, the BARG prover in the construction due to [CJJ21] does the following:

  1. For each $i \in [k]$, it first computes a PCP π_i for the i'th statement. This takes time $k \cdot \mathrm{poly}(\lambda, |C|)$. Let $L \in \mathrm{poly}(\lambda, |C|)$ denote the length of a single PCP.
  2. It then commits columnwise to the PCPs. Creating L commitments to k bits each takes time $L \cdot k \cdot \mathrm{poly}(\lambda)$ (similar to below, the commitment is a variation on a Merkle tree, where committing can be done in time linear in the committed message).
  3. It then applies a correlation-intractable hash to the circuit C and commitment. As shown in [CJJ21], the hash can be evaluated in time $\mathrm{poly}(\lambda, \log k, |C|)$.
  4. Next, it samples PCP queries for a single PCP using randomness derived from the correlation-intractable hash. They use a PCP requiring $\mathrm{poly}(\lambda, \log |C|)$ queries that can be sampled in time $\mathrm{poly}(\lambda, |C|)$.
  5. For each PCP, it then opens the query locations in the commitments. For each PCP, this corresponds to opening a bit in $\mathrm{poly}(\lambda, \log |C|)$ commitments. As each value can be opened in time $\mathrm{poly}(\lambda, \log k)$ due to the Merkle-tree structure of the commitment, putting everything together this takes time $k \cdot \mathrm{poly}(\lambda, \log k, \log |C|)$.
  6. Finally, it recurses by running a BARG for $k/2$ instances, where they show that the circuit for the smaller BARG has size $\mathrm{poly}(\lambda, \log k, \log |C|)$. Overall, there are $\log k$ recursions.

  Putting everything together, the BARG prover runs in time $k \cdot \mathrm{poly}(\lambda, |C|, \log k)$.

- No-signalling somewhere-extractable commitment: The no-signalling commitment construction is parameterized by an integer ℓ, which determines the number of bits extractable from the commitment scheme. For a fixed parameter ℓ, the construction consists of ℓ independent Merkle trees. Each Merkle tree consists of an FHE encryption of the committed message at the leaves, and uses FHE evaluation to compute the value of each node based on the values of its child nodes. Thus, computing the commitment to a message of length N can be done in time $\ell \cdot N \cdot \mathrm{poly}(\lambda)$, because it requires computing ℓ Merkle trees, which each require encrypting N bits and performing N FHE evaluations. Moreover, local openings to a single bit in this commitment can be computed and verified in time $\ell \cdot \mathrm{poly}(\lambda, \log N)$, as openings consist of an authentication path in each of the ℓ Merkle trees.

Putting everything together, to delegate a t-time computation using the CJJ scheme, the prover (a) creates a hash tree of the starting configuration, (b) runs the computation while simultaneously updating the hash tree, (c) commits to the sequence of updates to the hash tree, where each update additionally contains some efficiently computable auxiliary information, (d) creates local openings in the commitment as a witness to each step of computation, and (e) proves that the computation is correct using a BARG for the circuit C_{step}. From the above analysis, (a) takes the time to run $\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$ when cf is the starting configuration, (b) takes time $t \cdot \mathrm{poly}(\lambda)$, (c) takes time $\ell \cdot N \cdot \mathrm{poly}(\lambda)$ where ℓ is the length of a single update and N is the length of the committed message, (d) takes time $(t \cdot \ell) \cdot \ell \cdot \mathrm{poly}(\lambda, \log N)$ to open ℓ bits for each of the t steps, and (e) takes time $t \cdot \mathrm{poly}(\lambda, |C_{\mathsf{step}}|, \log t)$. It remains to discuss the specific values $|C_{\mathsf{step}}|$, ℓ, and N used in the protocol. The parameter ℓ corresponds to the length of the values needed verify a single step of computation, by computing that step and verifying the openings in the hash tree, and so $\ell \in \mathrm{poly}(\lambda)$ (for a fixed polynomial that depends on the size of the universal RAM machine U). The committed message consists of these values for each of the t steps, and thus $N = t \cdot \ell$. Finally, the circuit C_{step} consists of computing a single step of the RAM program and verifying the openings to the hash tree and commitment, which together takes time $\ell \cdot \mathrm{poly}(\lambda, \log N) \in \mathrm{poly}(\lambda, \log t)$. All together, this shows that the prover runs in time

$$\begin{aligned}
&\mathsf{Time}\left(\mathsf{Del.P}(1^\lambda, (\mathsf{cf}, \mathsf{cf}', t))\right) \\
&\leq \mathsf{Time}\left(\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})\right) + t \cdot \mathrm{poly}(\lambda) + \ell \cdot N \cdot \mathrm{poly}(\lambda) + t \cdot \ell^2 \cdot \mathrm{poly}(\lambda, \log N) + t \cdot \mathrm{poly}(\lambda, |C_{\mathsf{step}}|, \log t) \\
&\leq |\mathsf{cf}| \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda, \log t) + t \cdot \mathrm{poly}(\lambda, \log t) \\
&\in |\mathsf{cf}| \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda, \log t).
\end{aligned}$$

Achieving quasilinear efficiency. For our SPARG construction, it will be crucial that the running time of the delegation prover $\mathsf{Del.P}$ does not depend on n, the memory size of the RAM program. Therefore, the CJJ prover efficiency does not suffice for us, since the running time of the prover on $(\mathsf{cf}, \mathsf{cf}', t)$ depends linearly on $|\mathsf{cf}|$.

We observe that this dependence on $|\mathsf{cf}|$ is due to the fact that the prover is given an arbitrary starting configuration cf, and must compute a Merkle tree on the memory given in cf. For our SPARG construction, we are not concerned with RAM computation from an arbitrary starting point cf. Instead, we will start from an initial (short) configuration cf_0, for which we can afford to run in time proportional to $|\mathsf{cf}_0|$ to generating the initial hash tree.

However, this does not entirely solve the problem, because rather than proving that cf_0 results in the final configuration cf' after t steps of computation, we will instead determine "midpoints"—namely, configurations $\mathsf{cf}_1, \ldots, \mathsf{cf}_m$, where $\mathsf{cf}_m = \mathsf{cf}'$. We will then rely on the delegation scheme to prove statements of the form $(\mathsf{cf}_0, \mathsf{cf}_1, k_1)$, $(\mathsf{cf}_1, \mathsf{cf}_2, k_2)$, …, $(\mathsf{cf}_{m-1}, \mathsf{cf}_m, k_m)$, that is, that starting at cf_{i-1} and running for some number of steps k_i results in configuration cf_i. The main idea below is that when we prove each statement $(\mathsf{cf}_{i-1}, \mathsf{cf}_i, k_i)$, we will already have information about cf_{i-1} from proving the previous statement. In particular, we will show that we can already have the Merkle tree for cf_{i-1} in memory when we start the ith statement, rather than creating it from scratch.

This exact setting was addressed in [EFKP20b], where they showed that the hash tree can be instantiated with collision-resistant hash functions to achieve the following guarantees:

1. Computing the hash tree for the initial configuration can be done in time $|\mathsf{cf}_0| \cdot \mathrm{poly}(\lambda)$.

2. Given a hash tree in memory corresponding to *any* configuration cf, it holds that the computation can be run for any number of steps k while updating the hash tree with only $\mathrm{poly}(\lambda)$ additive overhead. This implies that if cf results in cf' after k steps of computation, and we have already computed a hash tree for cf, then we can compute the hash tree for cf' in time $k + \mathrm{poly}(\lambda)$.

The requirements for the hash tree of [CJJ21] (which is based on [KP16]) are satisfied by that of [EFKP20b] (see Section A.1 for the definition, and [EFKP20b] for a more in-depth discussion and comparison between various definitions). Therefore, we observe that the CJJ construction satisfies the following notion.

Definition 4.2. *Consider a RAM delegation scheme* $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V})$ *with the following syntax modifications and additional algorithm* $\mathsf{Del.Update}$*:*

- $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$*: The digest algorithm additionally outputs a value* tree.

- $(\mathsf{rt}', \mathsf{tree}', w) = \mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$*: The update algorithm takes as input a digest key* dk*, integer* t*, and a value* tree*, and outputs a digest* rt'*, a value* tree' *and a witness* w*.*

- $\pi \leftarrow \mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t), w)$*: The prover additionally takes as input a witness* w*. We require that completeness is preserved when* $\mathsf{Del.P}$ *receives the witness* w *computed by* $\mathsf{Del.Update}$.

We note that tree *and* w *can be communicated as pointers to memory. In particular, this implies that* $\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$ *still runs in time* $|\mathsf{cf}| \cdot \mathrm{poly}(\lambda)$*.*

We say that the scheme is β-updatable *if for any* $\lambda \in \mathbb{N}$*, statement* $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{L}^{\mathsf{del}}$*, keys* $(\mathsf{crs}, \mathsf{dk})$ *in the support of* $\mathsf{Del.S}(1^\lambda)$*,* $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$*, and* $(\mathsf{rt}', \mathsf{tree}', w) = \mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$*,*

$$(\mathsf{rt}', \mathsf{tree}') = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}')$$

and $\mathsf{Del.Update}$ *runs in* $t + \beta(\lambda)$ *steps with* $\beta(\lambda)$ *processors. Furthermore, for any two consecutive updates of length* t_1 *and* t_2 *starting at initial state* $(\mathsf{rt}_0, \mathsf{tree}_0)$*, let* $(\mathsf{rt}_1, \mathsf{tree}_1, w_1) = \mathsf{Del.Update}(\mathsf{dk}, t_1, \mathsf{tree}_0)$ *and* $(\mathsf{rt}_2, \mathsf{tree}_2, w_2) = \mathsf{Del.Update}(\mathsf{dk}, t_2, \mathsf{tree}_1)$*. Then, the output* $(\mathsf{rt}_2, \mathsf{tree}_2, w_2)$ *can be computed in time* $t_1 + t_2 + \beta(\lambda)$*. When* $\beta(\lambda) \in \mathrm{poly}(\lambda)$*, we say the scheme is updatable.*

We emphasize that $\mathsf{Del.P}$ no longer has access to the hash tree in memory, as this would create memory conflicts between $\mathsf{Del.P}$ and $\mathsf{Del.Update}$. Instead, we can view $\mathsf{Del.Update}$ as the algorithm that runs the computation on the hash tree, and collects all of the information needed to prove correctness—namely, the hash tree updates, which make up the witness w. The prover $\mathsf{Del.P}$ can then use this witness to form the proof. In the following definition, we quantify the prover efficiency in an updatable delegation scheme.

Definition 4.3. *An updatable RAM delegation scheme satisfies* α-prover efficiency *if for all* $\lambda \in \mathbb{N}$*,* $(\mathsf{crs}, \mathsf{dk})$ *in the support of* $\mathsf{Del.S}(1^\lambda)$*, statement* $(\mathsf{cf}, \mathsf{cf}', t) \in \mathcal{L}^{\mathsf{del}}$ *using* $n \leq 2^\lambda$ *memory with* $t \leq 2^\lambda$*,* $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$*, and* $(\mathsf{rt}', \mathsf{tree}', w) = \mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$*, it holds that*

$$\mathsf{Time}\left(\mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t), w)\right) = \alpha(\lambda, t).$$

Based on the above discussion, the CJJ scheme can be made to satisfy updatability and quasi-linear prover efficiency. Specifically, we will instantiate the hash tree in the CJJ construction with that of [EFKP20b], and modify the delegation scheme as follows:

- $\mathsf{Del.D}(1^\lambda, \mathsf{cf})$ will output rt as before, as the root of the hash tree, and set tree to be the full hash tree.
- $\mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$ will start with the hash tree in tree, run the computation for t steps while updating the hash tree, and then output $(\mathsf{rt}', \mathsf{tree}', w)$ where rt' is the resulting root, tree' is the updated tree, and w is the list of all authentication paths for the t updates.
- $\mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t), w)$ will use the updates in w to run the prover algorithm, rather than computing them from scratch.

By combining the above discussion with Theorem 4.1, we get the following.

Theorem 4.4. *Assuming the hardness of LWE, there exists a publicly verifiable, succinct, and updatable RAM delegation scheme* $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V}, \mathsf{Del.Update})$ *for* $\mathcal{L}^{\mathsf{del}}$ *with* α*-prover efficiency for* $\alpha(\lambda, t) \leq t \cdot \mathrm{poly}(\lambda, \log t)$.

Proof sketch. For completeness, we note that the CJJ prover only requires the hash tree updates to form the values that make up the proof. Thus, completeness follows when the prover receives the witness w computed by $\mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$ where $(\mathsf{rt}, \mathsf{tree}) \leftarrow \mathsf{Del.D}(1^\lambda, \mathsf{cf})$. For the prover efficiency, it follows from the discussion and analysis of the CJJ scheme above that $\mathsf{Del.P}$ runs in quasilinear time, given a witness consisting of updates to the computation.

For the updatability property of the resulting scheme, the efficiency of $\mathsf{Del.Update}$ follows from that of the [EFKP20b] hash tree construction. For the correctness of $\mathsf{Del.Update}$, consider any configuration cf, and let cf' be the configuration resulting from t steps of computation. We want to show that for $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$, it holds that $(\mathsf{rt}', \mathsf{tree}')$ given by $\mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$ is equal to $\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}')$.

This follows from the updatability property of the hash tree in [EFKP20b]. We note that the property we require here is slightly stronger than what they show, yet nonetheless follows from their construction. The updatability property in [EFKP20b] (called update completeness), required that after creating an initial hash tree for cf and doing the sequence of t updates to get to cf', the resulting tree could be locally opened to any value consistent with the configuration cf'. Here, we require that after creating an initial hash tree to cf and performing the updates, the resulting tree is *identical* to the one that would have been generated had we created a hash tree for cf'.

Nevertheless, the [EFKP20b] construction satisfies this property. Specifically, the algorithm for hashing cf first does an initialization step to hash an empty configuration, and then does an update to all non-$\perp$ positions in cf. Let us denote these by S. The update algorithm modifies all of the ancestors of nodes in S, denoted $\mathsf{ancestors}(S)$, in the hash tree.

Let us denote the positions changed when going from cf to cf' by S'. Hashing cf and then updating to cf' therefore results in generating the initial (empty) tree, doing an update to positions in S by writing to $\mathsf{ancestors}(S)$, and then doing an update to positions in S' by writing to $\mathsf{ancestors}(S')$. Conversely, hashing cf' corresponds to generating the initial (empty) tree, and then doing an update to positions in $S \cup S'$ by writing to $\mathsf{ancestors}(S \cup S')$. Since $\mathsf{ancestors}(S) \cup \mathsf{ancestors}(S') = \mathsf{ancestors}(S \cup S')$, and the values written are deterministic from the child values, one can show that this results in identical trees. □

4.3 Local Opening

Given a RAM delegation scheme in which the verifier receives *digests* of the full configuration, we will also require a scheme with a very natural *local opening* property: a set of locations can be locally opened with respect to a digest, providing a short proof of the opening. As most RAM delegation schemes employ an underlying Merkle tree, these are amenable to efficient local openings whenever the Merkle tree is already in memory. We define the local opening property as follows.

Definition 4.5. *An updatable RAM delegation scheme* $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V}, \mathsf{Del.Update})$ *satisfies* local opening *with algorithms* $(\mathsf{Del.Open}, \mathsf{Del.VerOpen})$ *with the following syntax:*

- $(V, \mathsf{st}, \pi) = \mathsf{Del.Open}(\mathsf{dk}, \mathsf{tree}, S)$*: On input a key* dk*, a tree* tree*, and a set of locations* S*, output a set* V *of* $|S|$ *words, a state* st*, and a proof* π*.*
- $b = \mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}, S, V, \mathsf{st}, \pi)$*: On input a key* dk*, a digest* rt*, a set of locations* S*, a set of words* V*, a state* st*, and a proof* π*, output a bit* b *indicating whether to accept or reject.*

As above, we note that tree *can be given as a pointer to memory. We require the following properties:*

- **Local Completeness.** *Let* dk *be in the support of* $\mathsf{Del.S}(1^\lambda)$*,* cf *be a configuration,* S *be an ordered set of locations. Let* $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$ *and* $(V, \mathsf{st}, \pi) = \mathsf{Del.Open}(\mathsf{dk}, \mathsf{tree}, S)$*. Then,*
$$\mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}, S, V, \mathsf{st}, \pi) = 1.$$
- **Local Soundness.** *For all non-uniform PPT adversaries* $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$*, there exists a negligible function* negl *such that for all* $\lambda \in \mathbb{N}$*, it holds that*
$$\Pr\left[\begin{array}{l}(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda) \\ (\mathsf{rt}, S, (V, \mathsf{st}, \pi), (V', \mathsf{st}', \pi') \leftarrow \mathcal{A}_\lambda((\mathsf{crs}, \mathsf{dk}))\end{array} : \begin{array}{l}\mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}, S, V, \mathsf{st}, \pi) = 1 \wedge \\ \mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}, S, V', \mathsf{st}', \pi') = 1 \wedge \\ (\mathsf{st}, V) \neq (\mathsf{st}', V')\end{array}\right]$$
$$\leq \mathsf{negl}(\lambda).$$

Since the [EFKP20b] hash tree satisfies local opening, this property also extends to the delegation scheme. We note that the syntactical difference between the above definition and the local opening property in [EFKP20b] is that we include the local state st in an opening. This is due to the nature of a RAM delegation scheme, which is concerned with a hash tree corresponding to a RAM computation. Specifically, a digest rt in the delegation scheme above consists of a local state st along with a hash tree digest digest. We therefore include the state here so as to make local opening a natural extension of the collision-resistance property of RAM delegation, which states that one cannot open a digest to two different configurations (that is, to $\mathsf{cf} = (\mathsf{st}, D)$ and $\mathsf{cf}' = (\mathsf{st}', D')$ where $(\mathsf{st}, D) \neq (\mathsf{st}', D')$). The above definition extends this to the setting of local opening. We can realize this definition from [EFKP20b] by defining the algorithm $\mathsf{Del.Open}$, when given a digest $\mathsf{rt} = (\mathsf{st}, \mathsf{digest})$, to output the state st and the hash tree opening with respect to digest. We can similarly define $\mathsf{Del.VerOpen}$ to accept the opening (V, st, π) with respect to rt on locations S if and only if st matches the one given in rt and (V, π) is a valid hash tree opening for digest on locations S. Putting everything together, we get the following corollary to Theorem 4.4.

Corollary 4.6. *Assuming the hardness of LWE, there exists a publicly verifiable, succinct, and updatable RAM delegation scheme for* $\mathcal{L}^{\mathsf{del}}$ *with local opening and* α*-updatable prover efficiency for* $\alpha(\lambda, t) \leq t \cdot \mathrm{poly}(\lambda, \log t)$*.*

5 SPARGs for P

In this section, we give our construction of SPARGs for (sequential) RAM computations. Our construction relies on a β-updatable RAM delegation scheme $\mathsf{Del} = (\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V}, \mathsf{Del.Update}, \mathsf{Del.Open}, \mathsf{Del.VerOpen})$ for $\mathcal{L}^{\mathsf{del}}$ with local opening and α-prover efficiency (see Section 3.4). We use the following parameters when proving a statement (M, x, L, t).

- $n \leq 2^\lambda$ is the memory used by M.
- α is the function denoting the prover efficiency of Del. We let $\alpha^\star \triangleq \alpha(\lambda, t)/t$ be the multiplicative overhead, with respect to t, of running $\mathsf{Del.P}$.
- β is the function denoting the efficiency of $\mathsf{Del.Update}$.
- $\gamma \triangleq \alpha^\star + 1$ is the fraction of remaining steps done in each chunk of the computation.

Theorem 5.1. *Let* Del *be a publicly verifiable, succinct, and updatable delegation scheme for* $\mathcal{L}^{\mathsf{del}}$ *with local opening and α-prover efficiency. Then, $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, given in Figure 5, is a SPARG for $\mathcal{L}^{\mathcal{U}}$. Specifically, for all $\lambda \in \mathbb{N}$ and $(M, x, y, L, t) \in \mathcal{L}^{\mathcal{U}}$ where M has access to $n \leq 2^\lambda$ words in memory and $t \leq 2^\lambda$, the following hold. Let $\alpha^\star$ be the multiplicative overhead of* $\mathsf{Del.P}$ *with respect to the number of steps of computation. Then:*

- *The depth of the prover is bounded by $t + L + (\alpha^\star)^2 \cdot \mathrm{poly}(\lambda, |M, x|, \log t)$ when using $\mathrm{poly}(\lambda) + \alpha^\star \log t$ processors.*
- *The proof size is bounded by $\alpha^\star \cdot \mathrm{poly}(\lambda, \log t)$.*
- *The work of the verifier is bounded by $\alpha^\star \cdot L \cdot \mathrm{poly}(\lambda, |M, x|, \log t)$.*

By Corollary 4.6, it holds that there exists an updatable RAM delegation scheme with local opening based on LWE where $\alpha^\star \in \mathrm{poly}(\lambda, \log t)$. Therefore, by combining Theorem 5.1 with Corollary 4.6, we get the following corollary.

Corollary 5.2. *Assuming the hardness of LWE, there exists a SPARG for $\mathcal{L}^{\mathcal{U}}$.*

We prove Theorem 5.1 by showing completeness in Lemma 5.3, soundness in Lemma 5.4, prover efficiency in Lemma 5.8, and succinctness in Lemma 5.9.

Lemma 5.3 (Completeness)**.** *For every $\lambda \in \mathbb{N}$ and $(M, x, y, L, t) \in \mathcal{L}^{\mathcal{U}}$ where M has access to $n \leq 2^\lambda$ words in memory and $t \leq 2^\lambda$, it holds that*

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

Proof. $\mathcal{V}$ accepts if and only if conditions 3a, 3b, 3c, and 3d hold. Conditions 3a and 3c hold by the completeness, updatability, and local completeness properties of the RAM delegation scheme. Conditions 3b and 3d follow by inspection. ∎

Lemma 5.4 (Soundness)**.** *For any non-uniform PPT algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$ and polynomial T, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l}\mathsf{pp} \leftarrow \mathcal{G}(1^\lambda) \\ ((M, x, y, L), \pi) \leftarrow \mathcal{A}_\lambda(\mathsf{pp})\end{array} : \begin{array}{l}\mathcal{V}(\mathsf{crs}, (M, x, y, L, t), \pi) = 1 \\ \wedge\ (M, x, y, L, t) \notin \mathcal{L}^{\mathcal{U}}\end{array}\right] \leq \mathsf{negl}(\lambda)$$

where $t = T(\lambda)$.

SPARG $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for $\mathcal{L}^{\mathcal{U}}$ given an updatable delegation scheme with local opening $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V}, \mathsf{Del.Update}, \mathsf{Del.Open}, \mathsf{Del.VerOpen})$:

- $\mathcal{G}(1^\lambda)$:
    1. $(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda)$.
    2. Output $\mathsf{pp} = (\mathsf{crs}, \mathsf{dk})$.
- $\mathcal{P}(1^\lambda, \mathsf{pp}, (M, x, L, t))$:
    1. Let cf_0 be the initial configuration for $M(x)$, which includes the (empty) local state and M, x. Let $(\mathsf{rt}_0, \mathsf{tree}_0) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}_0)$.
    2. Compute γ as in the parameters paragraph. Initialize $T := t$ to be the number of steps remaining in the computation.
    3. For $i = 1, 2, \dots$, repeat the following until $T = 0$:
        (a) Calculate the number of steps k_i to compute in this iteration. If $T > \gamma \log T$, set $k_i = \lfloor T/\gamma \rfloor$, and otherwise set $k_i = T$.
        (b) Compute k_i steps of M starting with configuration cf_{i-1}. Let cf_i be the resulting configuration.
        (c) In parallel to Step 3b, compute $(\mathsf{rt}_i, \mathsf{tree}_i, w_i) \leftarrow \mathsf{Del.Update}(\mathsf{dk}, k_i, \mathsf{tree}_{i-1})$.
        (d) Without waiting for Step 3c to halt (but after Step 3b), spawn a process that continues to the next iteration with $T = T - k_i$.
        (e) After Steps 3b and 3c complete, spawn a parallel thread to compute $\tau_i \leftarrow \mathsf{Del.P}(\mathsf{crs}, (M, \mathsf{cf}_{i-1}, \mathsf{cf}_i, k_i), w_i)$.
    4. Let $(y, \mathsf{st}, \pi_y) = \mathsf{Del.Open}(\mathsf{dk}, [L], \mathsf{tree}_m)$, where m is the number of iterations of the loop above.
    5. Let $\vec{\mathsf{rt}} = (\mathsf{rt}_1, \dots, \mathsf{rt}_m)$, $\vec{\tau} = (\tau_1, \dots, \tau_m)$, and $\vec{k} = (k_1, \dots, k_m)$. Output (y, π) where $\pi = (\vec{\mathsf{rt}}, \vec{\tau}, \vec{k}, \mathsf{st}, \pi_y)$.
- $\mathcal{V}(1^\lambda, \mathsf{pp}, (M, x, y, L, t), \pi)$:
    1. Parse $\pi = (\vec{\mathsf{rt}}, \vec{\tau}, \vec{k}, \mathsf{st}, \pi_y)$.
    2. Let cf_0 be the initial configuration of $M(x)$ and compute rt_0 as $\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}_0)$.
    3. Output 1 if and only if the following hold, and 0 otherwise:
        (a) $\mathsf{Del.V}(\mathsf{crs}, (\mathsf{rt}_{i-1}, \mathsf{rt}_i, k_i), \tau_i)$ accepts for all $i \in [m]$.
        (b) k_i is as defined above for each $i \in [m]$, and $t \leq 2^\lambda$.
        (c) $\mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}_m, [L], y, \mathsf{st}, \pi_y) = 1$.
        (d) st is a halting state, and $|y| \leq L$.

Figure 2: SPARG for $\mathcal{L}^{\mathcal{U}}$.

Proof. Assume for contradiction that there exists a non-uniform PPT adversary $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda \in \mathbb{N}}$, polynomial T, and polynomial q such that $\mathcal{A}_\lambda$ breaks the soundness of the SPARG construction for a $T(\lambda)$-time computation with probability $1/q(\lambda)$ for infinitely many $\lambda \in \mathbb{N}$, that is

$$\Pr\left[\begin{array}{l}(\mathsf{crs},\mathsf{dk}) \leftarrow \mathcal{G}(1^\lambda)\\ ((M,x,y,L),\pi) \leftarrow \mathcal{A}_\lambda((\mathsf{crs},\mathsf{dk}))\end{array} : \begin{array}{l}\mathcal{V}(\mathsf{pp},(M,x,y,L,t),\pi)=1\\ \wedge\ (M,x,y,L,t)\notin \mathcal{L}^{\mathcal{U}}\end{array}\right] \geq \frac{1}{q(\lambda)}. \tag{5.1}$$

For every such $\lambda \in \mathbb{N}$, we will construct an adversary $\mathcal{B}_\lambda$ either against the local soundness or soundness of the underlying RAM delegation scheme. On input $(\mathsf{crs},\mathsf{dk})$, the algorithm $\mathcal{B}_\lambda$ does the following.

1. Run $\mathcal{A}_\lambda((\mathsf{crs},\mathsf{dk}))$ to get $((M,x,y,L),\pi)$ and let $t = T(\lambda)$. Parse $\pi = (\vec{\mathsf{rt}},\vec{\tau},\vec{k},\mathsf{st},\pi_y)$. Let m be the number of values in each of the vectors given by $\mathcal{A}_\lambda$, and let rt_0 be the digest given by $\mathsf{Del.D}$ for the initial configuration of $M(x)$.

2. Compute $M(x)$ for t steps by starting at the initial configuration $\overline{\mathsf{cf}}_0$ for $M(x)$. Let $\overline{\mathsf{cf}}_1,\dots,\overline{\mathsf{cf}}_m$ be the configurations computed at the indices corresponding to $\vec{k}$, that is, $\overline{\mathsf{cf}}_i$ corresponds to the configuration after $\sum_{j=1}^{i} k_j$ steps.

3. Compute the corresponding digests by letting $(\overline{\mathsf{rt}}_0,\overline{\mathsf{tree}}_0) = \mathsf{Del.D}(\mathsf{dk},\overline{\mathsf{cf}}_0)$ and $(\overline{\mathsf{rt}}_i,\overline{\mathsf{tree}}_i,*) = \mathsf{Del.Update}(\mathsf{dk},k_i,\overline{\mathsf{tree}}_{i-1})$ for $i \in [m]$.

4. If $\mathsf{rt}_m = \overline{\mathsf{rt}}_m$, then output $(\mathsf{rt}_m,[L],(y,\mathsf{st},\pi_y),(\overline{y},\overline{\mathsf{st}},\overline{\pi_y}))$ against the local soundness of the delegation scheme, where $(\overline{y},\overline{\mathsf{st}},\overline{\pi_y}) = \mathsf{Del.Open}(\mathsf{dk},\overline{\mathsf{tree}}_m,[L])$.

5. Otherwise, let $i^\star \in [m]$ be the smallest index where $\mathsf{rt}_{i^\star} \neq \overline{\mathsf{rt}}_{i^\star}$. Output $(\overline{\mathsf{cf}}_{i^\star-1},\overline{\mathsf{cf}}_{i^\star},\overline{\mathsf{rt}}_{i^\star-1},\mathsf{rt}_{i^\star},\tau_{i^\star})$ against the soundness of the delegation scheme.

To analyze the success of $\mathcal{B}_\lambda$, we will show that whenever $\mathcal{A}_\lambda$ succeeds, the values output by $\mathcal{B}_\lambda$ contradict either local soundness or the soundness of the underlying delegation scheme. Specifically, when $\mathsf{rt}_m = \overline{\mathsf{rt}}_m$, we show that $\mathcal{B}_\lambda$ breaks local soundness by outputting in Step 4, and when $\mathsf{rt}_m \neq \overline{\mathsf{rt}}_m$, then there exists an $i^\star \in [m]$ which causes $\mathcal{B}_\lambda$ to output in 5 and break soundness.

More formally, we complete the proof through a sequence of claims below. We first show in Claim 5.5 that $\mathcal{B}_\lambda$ runs in polynomial time, and thus it suffices to show that the values output by $\mathcal{B}_\lambda$ give a contradiction. To show this, let $\mathsf{win}_{\mathcal{A}}$ denote the event in Equation 5.1. By Equation 5.1,

$$\frac{1}{q(\lambda)} \leq \Pr\left[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m = \overline{\mathsf{rt}}_m\right] + \Pr\left[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m \neq \overline{\mathsf{rt}}_m\right].$$

We then show in Claim 5.6 that $\Pr[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m = \overline{\mathsf{rt}}_m]$ is bounded above by the probability that $\mathcal{B}_\lambda$ breaks local soundness. This is therefore bounded by a negligible function negl, and so

$$\Pr\left[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m \neq \overline{\mathsf{rt}}_m\right] \geq \frac{1}{q(\lambda)} - \mathsf{negl}(\lambda) \geq \frac{1}{2q(\lambda)}$$

for infinitely many $\lambda \in \mathbb{N}$. We then show in Claim 5.7 that the probability that $\mathcal{B}_\lambda$ breaks soundness by outputting in Step 5 is at least $1/p(\lambda) \cdot \Pr[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m \neq \overline{\mathsf{rt}}_m]$ for a polynomial p depending on $\mathcal{A}_\lambda$. Combining this with the above, this implies that $\mathcal{B}_\lambda$ breaks soundness with probability at least $1/(2p(\lambda)q(\lambda))$ which gives a contradiction. We proceed to show the claims.

Claim 5.5. *There exists a polynomial $p_{\mathcal{B}}$ such that $\mathcal{B}_\lambda$ runs in time $p_{\mathcal{B}}(\lambda)$ for all $\lambda \in \mathbb{N}$.*

Proof. We show that $\mathcal{B}_\lambda$ can be implemented as a polynomial-time RAM machine. We have that $\mathcal{B}_\lambda$ (1) runs $\mathcal{A}_\lambda$, (2) computes $M(x)$ for t steps, (3) compute the m digests using $\mathsf{Del.Update}$, and then (4) computes its output. We note that m and $|M, x|$ are polynomially bounded as the output length of $\mathcal{A}_\lambda$ is linear in these values, and t is polynomial by assumption.

We have that (1) runs in polynomial time and (2) takes time t. For (3), computing the initial digest and then the sequential calls to $\mathsf{Del.Update}$ can be done in time $|M, x| \cdot \mathrm{poly}(\lambda) + t \cdot \mathrm{poly}(\lambda)$ by the updatability of the delegation scheme. For (4), computing the output requires looking through m digests and outputting the corresponding configurations. For $\mathcal{B}_\lambda$ to be able to output the configurations, we note that even though they may be as large as $n = 2^\lambda$, the machine M can only access t positions in memory, and so the non-accessed positions remain as 0's. Thus, the intermediate configurations can be represented with $O(t \cdot \lambda)$ bits rather than relying on n. Putting everything together, we have that $\mathcal{B}_\lambda$ runs in polynomial time. ■

Claim 5.6. *When $\mathcal{A}_\lambda$ succeeds and $\mathsf{rt}_m = \overline{\mathsf{rt}}_m$, then $\mathcal{B}_\lambda$ outputs values in Step 4 that violate the local soundness of the delegation scheme. Formally,*

$$\Pr\left[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m = \overline{\mathsf{rt}}_m\right]$$
$$\leq \Pr\left[\begin{array}{l} (\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda) \\ (\mathsf{rt}_m, [L], (y, \mathsf{st}, \pi_y), (\overline{y}, \overline{\mathsf{st}}, \overline{\pi_y})) \leftarrow \mathcal{B}_\lambda(\mathsf{crs}, \mathsf{dk}) \end{array} : \begin{array}{l} \mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}_m, [L], y, \mathsf{st}, \pi) = 1 \\ \mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}_m, [L], \overline{y}, \overline{\mathsf{st}}, \overline{\pi}) = 1 \\ (\mathsf{st}, y) \neq (\overline{\mathsf{st}}, \overline{y}) \end{array}\right]$$

where the first probability is over the randomness of $\mathsf{Del.S}$ *and* $\mathcal{A}_\lambda$.

Proof. Suppose $\mathcal{A}_\lambda$ succeeds and $\mathsf{rt}_m = \overline{\mathsf{rt}}_m$. In this case, $\mathcal{B}_\lambda$ outputs $(\mathsf{rt}_m, [L], (y, \mathsf{st}, \pi_y), (\overline{y}, \overline{\mathsf{st}}, \overline{\pi_y}))$ in Step 4. Since $\mathcal{V}$ accepts, then $\mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}_m, [L], y, \mathsf{st}, \pi_y) = 1$.

We claim that $\mathsf{Del.VerOpen}(\mathsf{dk}, \overline{\mathsf{rt}}_m, [L], \overline{y}, \overline{\mathsf{st}}, \overline{\pi_y}) = 1$ as well. To see this, note that $(\overline{\mathsf{rt}}_m, \overline{\mathsf{tree}}_m) = \mathsf{Del.D}(\mathsf{dk}, \overline{\mathsf{cf}}_m)$ by updatability, and therefore $(\overline{y}, \overline{\mathsf{st}}, \overline{\pi_y}) = \mathsf{Del.Open}(\mathsf{dk}, \overline{\mathsf{tree}}_m, [L])$ by local completeness, which gives the above. Lastly, as $\mathsf{rt}_m = \overline{\mathsf{rt}}_m$, both openings are accepting for the same digest.

It remains to show that $(\mathsf{st}, y) \neq (\overline{\mathsf{st}}, \overline{y})$, i.e., the openings correspond to two different values. It suffices to show that when $\mathsf{st} = \overline{\mathsf{st}}$ then $y \neq \overline{y}$. When $\mathsf{st} = \overline{\mathsf{st}}$ and $\mathcal{A}_\lambda$ succeeds, we have that (1) st is a halting state, since this is checked by $\mathcal{V}$, and (2) $(M, x, y, L, t) \notin \mathcal{L}^{\mathcal{U}}$. Together, these imply that $M(x)$ indeed halts in t steps with output $\overline{y} \neq y$, which gives a contradiction. Therefore, whenever $\mathcal{A}_\lambda$ succeeds and $\mathcal{B}_\lambda$ outputs in Step 4, the values output by $\mathcal{B}_\lambda$ contradict the local soundness of the delegation scheme. ■

Claim 5.7. *There exists a polynomial-time computable function K with $K(\lambda) \leq T(\lambda)$ for all λ, and a polynomial p, such that for $k = K(\lambda)$,*

$$\frac{1}{p(\lambda)} \cdot \Pr\left[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m \neq \overline{\mathsf{rt}}_m\right]$$
$$\leq \Pr\left[\begin{array}{l} (\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda) \\ (\overline{\mathsf{cf}}_{i^\star-1}, \overline{\mathsf{cf}}_{i^\star}, \overline{\mathsf{rt}}_{i^\star-1}, \mathsf{rt}_{i^\star}, \tau_{i^\star}) \leftarrow \mathcal{B}_\lambda(\mathsf{crs}, \mathsf{dk}) \end{array} : \begin{array}{l} \mathsf{Del.V}(\mathsf{crs}, (\overline{\mathsf{rt}}_{i^\star-1}, \mathsf{rt}_{i^\star}, k), \tau_i) = 1 \\ (\overline{\mathsf{cf}}_{i^\star-1}, \overline{\mathsf{cf}}_{i^\star}, k) \in \mathcal{L}^{\mathcal{U}} \\ (\overline{\mathsf{rt}}_{i^\star-1}, *) = \mathsf{Del.D}(\mathsf{dk}, \overline{\mathsf{cf}}_{i^\star-1}) \\ (\mathsf{rt}_{i^\star}, *) \neq \mathsf{Del.D}(\mathsf{dk}, \overline{\mathsf{cf}}_{i^\star}) \end{array}\right].$$

Proof. Let $i^\star \in [m]$ be the smallest index with $\mathsf{rt}_{i^\star} \neq \overline{\mathsf{rt}}_{i^\star}$. We start by showing that the values output by $\mathcal{B}_\lambda$ violate soundness for a $k_{i^\star}$-time computation, where we recall that $k_{i^\star}$ is the number of steps in statement $i^\star$ given by $\mathcal{A}_\lambda$. Relative to the output $(\overline{\mathsf{cf}}_{i^\star-1}, \overline{\mathsf{cf}}_{i^\star}, \overline{\mathsf{rt}}_{i^\star-1}, \mathsf{rt}_{i^\star}, \tau_{i^\star})$ given by $\mathcal{B}_\lambda$, it holds that:

1. $\mathsf{Del.V}(\mathsf{crs}, (\overline{\mathsf{rt}}_{i^\star-1}, \mathsf{rt}_{i^\star}, k_{i^\star}), \tau_i)$ because $\mathsf{rt}_{i^\star-1} = \overline{\mathsf{rt}}_{i^\star-1}$ by definition of $i^\star$, and $\mathsf{Del.V}$ accepts the ith proof given by $\mathcal{A}_\lambda$ when $\mathcal{A}_\lambda$ succeeds.
2. $(\overline{\mathsf{cf}}_{i^\star-1}, \overline{\mathsf{cf}}_{i^\star}, k_{i^\star}) \in \mathcal{L}^{\mathcal{U}}$, because this corresponds to the true computation of $M(x)$ computed by $\mathcal{B}_\lambda$.
3. $(\overline{\mathsf{rt}}_{i^\star-1}, *) = \mathsf{Del.D}(\mathsf{dk}, \overline{\mathsf{cf}}_{i^\star-1})$ by the updatability of the delegation scheme.
4. $(\mathsf{rt}_{i^\star}, *) \neq \mathsf{Del.D}(\mathsf{dk}, \overline{\mathsf{cf}}_{i^\star})$ by definition of $i^\star$ and the updatability of the delegation scheme.

Therefore, the values given by $\mathcal{B}_\lambda$ contradict the soundness of the delegation scheme for a $k_{i^\star}$-time computation. To break the soundness of the delegation scheme, recall that we need to show the existence of an adversary $\mathcal{B}_\lambda$ and a polynomial-time computable, polynomially-bounded function $K(\lambda)$ such that $\mathcal{B}_\lambda$ breaks the soundness for a computation with $k = K(\lambda)$ steps, whereas above we showed that $\mathcal{B}_\lambda$ succeeds by choosing k adaptively as $k_{i^\star}$. We note that for any λ, there are at most m possible values of $k_{i^\star}$, because $\mathcal{V}$ checks that each $k_{i^\star}$ is set correctly. Since the output of $\mathcal{A}_\lambda$ is linear in m, then m is bounded by a fixed polynomial $p(\lambda)$ corresponding to the runtime of $\mathcal{A}_\lambda$. It follows that there exists an $i \in [m]$ such that $\mathcal{B}_\lambda$ succeeds more than a $1/m \geq 1/p(\lambda)$ fraction of the time on k_i, that is, $\mathcal{B}_\lambda$ breaks soundness for a k_i-time computation with probability at least $1/p(\lambda) \cdot \Pr[\mathsf{win}_{\mathcal{A}} \wedge \mathsf{rt}_m \neq \overline{\mathsf{rt}}_m]$. This completes the claim by defining $K(\lambda)$ to calculate the value of k_i based on λ and $T(\lambda)$, and noting that $K(\lambda)$ is polynomially bounded by $T(\lambda)$. ■

This completes the proof of Lemma 5.4. ■

Lemma 5.8 (Prover Efficiency)**.** *There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$ and $(M, x, L, t) \in \mathcal{L}^{\mathcal{U}}$ where M has access to $n \leq 2^\lambda$ words in memory, and $t \leq 2^\lambda$, it holds that*

$$\mathsf{depth}_{\mathcal{P}}(1^\lambda, (M, x, L, t)) \leq t + L + (\alpha^\star)^2 \cdot q(\lambda, |M, x|, \log t)$$

with $q_2(\lambda) + \alpha^\star \log t$ processors.

Proof. Given a statement (M, x, L, t), the work of the prover consists of (1) initializing values for the computation, (2) computing and proving each sub-computation, and (3) computing its output.

For initialization, this requires computing γ as well as the initial digest rt_0. The initial configuration consists of the empty initial state of M as well as the input x, and so by the efficiency of the underlying delegation scheme, $(\mathsf{rt}_0, \mathsf{tree}_0)$ can be computed in time $|M, x| \cdot \mathrm{poly}(\lambda)$. Computing the parameter $\gamma = \alpha^\star + 1$ requires evaluating $\alpha(\lambda, t)$ which can be computed in time polynomial in its input length. Putting these together, initialization takes time $\mathrm{poly}(\lambda, |M, x|, \log t)$. For computing its output, we have that the output y is of length L and so together, everything other than computing and proving $M(x)$ takes time $L + \mathrm{poly}(\lambda, |M, x|, \log t)$.

For computing and proving each sub-computation, we note that the number of proofs m computed by $\mathcal{P}$ can be bounded by $m \leq \gamma \log t$, which follows directly from [EFKP20b, Claim 6.14]. Moreover, by the efficiency of $\mathsf{Del.Update}$ and the quasilinear overhead of $\mathsf{Del.P}$, all proofs finish within depth $t + \gamma^2 \cdot (\log t + 1) + \beta(\lambda)$ with $3\beta(\lambda) + m$ processors by [EFKP20b, Claim 6.15]. Since $\gamma = \alpha^\star + 1$, $\beta(\lambda) \in \mathrm{poly}(\lambda)$, and $m \leq \gamma \log t$, it follows that all sub-proofs complete within $t + (\alpha^\star)^2 \cdot \mathrm{poly}(\lambda, \log t)$ with $\mathrm{poly}(\lambda) + \alpha^\star \log t$ processors. ■

Lemma 5.9 (Succinctness)**.** *There exist polynomials q_1, q_2 such that for any $\lambda \in \mathbb{N}$, and pp in the support of $\mathcal{G}(1^\lambda)$, the following hold. For $(M, x, L, t) \in \mathcal{L}^{\mathcal{U}}$ where M has access to $n \leq 2^\lambda$ words in memory, $t \leq 2^\lambda$, and (y, π) in the support of $\mathcal{P}(1^\lambda, (M, x, L, t))$, it holds that the proof length is bounded by*

$$|\pi| \leq \alpha^\star \cdot q_1(\lambda, \log t).$$

For any $(M, x, y, L, t), \pi \in \{0,1\}^*$ where M has access to $n \leq 2^\lambda$ words in memory and $t \leq 2^\lambda$, it holds that

$$\mathsf{work}_{\mathcal{V}}(1^\lambda, (M, x, y, L, t), \pi) \leq \alpha^\star \cdot L \cdot q_2(\lambda, |M, x|, \log t).$$

Proof. We start with bounding the proof length. The proof π contains $(\vec{\mathsf{rt}}, \vec{\tau}, \vec{k}, \mathsf{st})$. By the efficiency of Del, we have that each digest in rt has length λ and each proof in τ has size $\mathrm{poly}(\lambda, \log t)$. Moreover, each integer in $\vec{k}$ is at most t, and so requires $\log t$ bits. As there are $m \leq \gamma \log t = (\alpha^\star + 1) \cdot \log t$ of each of these in the proof, everything besides st requires $\alpha^\star \cdot \mathrm{poly}(\lambda, \log t)$. As st is a local RAM state that contains a constant number of words, it follows that the proof length is at most $\alpha^\star \cdot \mathrm{poly}(\lambda, \log t)$.

Next, to bound the verifier efficiency, we have that the verifier first parses the proof, which has length $\alpha^\star \cdot \mathrm{poly}(\lambda, |M|, \log t)$ as discussed above. It then computes cf_0 and rt_0, which can be computed in time $\mathrm{poly}(\lambda, |M, x|)$. Then, it runs $\mathsf{Del.V}$ for each of the m sub-proofs, which can be done in time $\mathrm{poly}(\lambda, |M|, \log t)$ by the succinctness of Del. It also checks $\mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}_m, [L], y, \pi_y)$, which runs in polynomial time in its inputs, and hence in time $\mathrm{poly}(\lambda, L)$. Lastly, it does consistency checks, which take time $|L| \cdot \mathrm{poly}(\lambda, \log t)$ to check st, $|y|$, and t. It also checks that each k_i is correct, which requires computing $\gamma = \alpha(\lambda, t) + 1$, and then, starting with $T = t$, iteratively checking for each i if the number of steps remaining is greater than $\gamma \log T$, and if so calculating $\lfloor T/\gamma \rfloor$. These can all be computed in polynomial time in their input length, adding a total of $m \cdot \mathrm{poly}(\log \lambda, \log t)$ work. Putting everything together and using the fact that $m \leq \gamma \log t$ and $\gamma = \alpha^\star + 1$, $\mathcal{V}$ runs in time $\alpha^\star \cdot L \cdot \mathrm{poly}(\lambda, |M, x|, \log t)$. ∎

6 SPARGs for Parallel Computations

In this section, we give a SPARG construction for parallel RAM computations. Specifically, for a computation that takes time t with p processors, we show a SPARG where the prover runs in time proportional to the parallel time t when using p processors, rather than running in time proportional to the total work $t \cdot p$ (see Definition A.7 for a formal definition).

At a high level, we will be modifying the SPARG from Section 5 as follows. Recall that the SPARG prover runs the computation of $M(x)$ by viewing it as a sequence of m consecutive sub-computations. For each sub-computation, it runs $\mathsf{Del.Update}$ to form a witness w_i that $\mathsf{Del.P}$ can use to prove correctness of that sub-computation. Recall that when instantiating the updatable delegation scheme via Section 4, the witness w_i simply consists of updates to a Merkle tree corresponding to the computation, and the prover $\mathsf{Del.P}$ uses these updates to form the proof, without needing access to the tree or computation in memory.

To extend this to the parallel setting, we first note that the hash tree of [EFKP20b] (which we are already using to instantiate our construction) allows for *parallel* updates, and so $\mathsf{Del.Update}$ can perform p updates to the hash tree simultaneously with minimal overhead, and then set w_i to be the hash tree openings to the parallel updates. However, a challenge arises in that if we modify $\mathsf{Del.P}$ to instead prove the correctness of t parallel updates, rather than proving correctness of $t \cdot p$ sequential updates, the running time of $\mathsf{Del.P}$ grows polynomially in p, which we want to avoid.

This is due to the fact that the underlying scheme depends on the time to verify a single step of computation, which requires checking all p parallel updates at each step.

To fix this, we will show that it suffices for $\mathsf{Del.P}$ to first "sequentialize" the updates given in w_i—that is, transform t parallel updates into $t \cdot p$ sequential ones. We formalize this as a property of the hash tree in Section 6.1, and in particular show that it can be done in time $\mathrm{poly}(\lambda, \log p)$ with p processors. We then show in Section 6.2 that when instantiating our updatable delegation scheme with a hash tree that allows for sequentializing updates, it can be made to satisfy quasi-linear parallel efficiency, meaning that the prover uses p processors and runs in time $t + \mathrm{poly}(\lambda, \log(t \cdot p))$. Finally, in Section 6.3, we show that when instantiating our SPARG construction with this updatable delegation scheme, it gives a SPARG for parallel computations.

6.1 Sequentializable Hash Trees

In this section, we show that parallel updates in the hash tree construction due to [EFKP20b] can be transformed into sequential ones, with overhead essentially independent of the parallelism. We start by defining this property.

Definition 6.1 (Sequentializable Hash Trees)**.** *A concurrently updatable hash function* ($\mathsf{H.Gen}$, $\mathsf{H.Hash}$, $\mathsf{H.Open}$, $\mathsf{H.Update}$, $\mathsf{H.VerOpen}$, $\mathsf{H.VerUpd}$) *with β-parallel efficiency is* sequentializable *using an algorithm* $\mathsf{H.Sequentialize}$ *with the following syntax:*

- $(\vec{\mathsf{rt}}, \vec{\pi}) = \mathsf{H.Sequentialize}(\mathsf{pp}, \mathsf{tree}, S, V, \tau)$*: A deterministic algorithm that takes as input public parameters* pp*, a hash tree* tree*, an ordered set $S \subseteq [n]$, a tuple V of words in $\{0,1\}^\lambda$, and a proof τ, and outputs a vector* $\vec{\mathsf{rt}}$ *of $|S|$ digests and a vector $\vec{\pi}$ of $|S|$ proofs.*

We require the following properties.

- **Correctness.** *For any $\lambda, n \in \mathbb{N}$, string $D \in \{0,1,\perp\}^n$,* pp *in the support of* $\mathsf{H.Gen}(1^\lambda, n)$*, ordered set $S \subseteq [n]$, and tuple V of words in $\{0,1\}^\lambda$, compute*

  1. $(\mathsf{tree}, \mathsf{rt}_0) = \mathsf{H.Hash}(\mathsf{pp}, D)$
  2. $(\mathsf{rt}, \tau) = \mathsf{H.Update}(\mathsf{pp}, \mathsf{tree}, S, V)$
  3. $(\vec{\mathsf{rt}}, \vec{\pi}) = \mathsf{H.Sequentialize}(\mathsf{pp}, \mathsf{tree}, S, V, \tau)$

  Then, it holds that

  $$\mathsf{rt}_{|S|} = \mathsf{rt} \quad \textit{and} \quad \mathsf{H.VerUpd}(\mathsf{pp}, \mathsf{rt}_{i-1}, \ell_i, v_i, \mathsf{rt}_i, \pi_i) = 1$$

  for each $i \in [|S|]$, where ℓ_i, v_i, rt_i*, and π_i are the ith values in S, V,* $\vec{\mathsf{rt}}$*, and $\vec{\pi}$, respectively.*

- **Efficiency.** $\mathsf{H.Sequentialize}(\mathsf{pp}, \mathsf{tree}, S, V, \tau)$ *can be computed in time $\beta(\lambda) \cdot \mathrm{poly}(\log |S|, \log n)$ with $|S|$ processors.*

We note that the inputs to $\mathsf{H.Sequentialize}$ *can be given as pointers to memory to enable the above efficiency.*

We achieve the above definition by showing that the concurrently updatable hash function construction due to [EFKP20b] can be made sequentializable. Henceforth, we refer to this as the EFKP hash function. We prove the following lemma.

Lemma 6.2. *Assuming the existence of collision-resistant hash function families, there exists a concurrently updatable hash function that is sequentializable.*

To prove the lemma, we start with some preliminaries, following [EFKP20b].

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

Each node in a binary tree is denoted with a label, defined as follows. The root is denoted by the empty string ϵ. All other nodes are labeled recursively. For a node whose parent has label ℓ, we label its left child as $\ell||0$ and its right child as $\ell||1$. We note that with this notation, leaves are labeled with strings in $\{0,1\}^n$.

Definition 6.3 (Ancestor and Dangling Nodes)**.** *For a complete binary tree and a set of leaves S, we define the following sets:*

- $\mathsf{ancestors}(S)$ *is the set of nodes that are ancestors of any node in S, including those in S.*
- $\mathsf{dangling}(S)$ *is the set of nodes that are siblings of nodes in* $\mathsf{ancestors}(S)$*, but themselves are not in* $\mathsf{ancestors}(S)$*.*

For a single node ℓ, we simply write $\mathsf{ancestors}(\ell)$ *and* $\mathsf{dangling}(\ell)$ *to denote the corresponding sets relative to $\{\ell\}$.*

We are now ready to prove the lemma.

Proof of Lemma 6.2. Let $\mathsf{H} = (\mathsf{H.Gen}, \mathsf{H.Hash}, \mathsf{H.Open}, \mathsf{H.Update}, \mathsf{H.VerOpen}, \mathsf{H.VerUpd})$ be the hash tree construction given by EFKP. Fix a hash tree tree in memory, a set of nodes S, and tuple V of words, and let (rt, τ) be the update given by $\mathsf{H.Update}(\mathsf{pp}, \mathsf{tree}, S, V)$. Recall that τ consists of a value for each node in S before the update, as well as values for each node in $\mathsf{dangling}(S)$.

Let $|S| = p$. The algorithm $\mathsf{H.Sequentialize}(\mathsf{pp}, S, V, \tau)$ will use p processors $\rho_1, \ldots, \rho_p$, each with $2\log n$ words of allocated memory. At a high level, each processor ρ_i will compute an authentication path corresponding to updating only the ith node ℓ_i in S, while pretending that the $i-1$ first nodes were already updated. It will do so by forming lists A_i and B_i, level by level, of values for the ancestor and dangling nodes for ℓ_i after the ith update. For each level of the tree, all processors will find the necessary values in parallel. This will result in an algorithm that takes parallel time proportional to the height of the tree, but not the number of processors. Next, we give the formal algorithm.

$\mathsf{H.Sequentialize}(\mathsf{pp}, S, V, \tau)$:

1. Use τ to form a partial Merkle tree T in memory corresponding to the values of all nodes in $\mathsf{ancestors}(S) \cup \mathsf{dangling}(S)$ before any updates. Specifically, using the values given in τ for $S \cup \mathsf{dangling}(S)$, iteratively hash each pair of siblings to form the partial Merkle tree.
2. Sort S in lexicographic order and let the sorted set be $S = (\ell_1, \ldots, \ell_p)$. Sort V and the same way, so that for each i, the ith value corresponds to ℓ_i.
3. Each processor ρ_i initializes empty lists A_i and B_i, and writes the value of ℓ_i from V to A_i.
4. For each level j of the tree, starting from the leaves, each processor ρ_i does the following:
   (a) **Find step.** Let $\ell_{\mathsf{sib},i}$ be the node in $\mathsf{dangling}(\ell_i)$ at level j.
      i. Check to see if $\ell_{\mathsf{sib},i}$ would have already been updated before the ith update (meaning, if it is an ancestor of some $\ell_k \in S$ with $k < i$). Specifically, if $\ell_{\mathsf{sib},i}$ is a left child, find the rightmost node ℓ_k in S with $\ell_{\mathsf{sib},i} \in \mathsf{ancestors}(\ell_k)$.
      ii. If such a node ℓ_k is found, let $v_{\mathsf{sib},i}$ be the last value in the list A_k.

iii. Otherwise, let $v_{\mathsf{sib},i}$ be the value of $\ell_{\mathsf{sib},i}$ in T.

iv. Write $v_{\mathsf{sib},i}$ to B_i.

(b) **Hash step.** Hash the most recent values in A_i and B_i together (with the left node's value first) and add the resulting value to A_i.

5. Each processor then sets its output π_i to B_i, and rt_i to the last value in A_i.

To complete the proof, it remains to analyze the correctness and efficiency of $\mathsf{H.Sequentialize}$. We show correctness in Claim 6.4 and efficiency in Claim 6.5 below.

Claim 6.4. *The algorithm* $\mathsf{H.Sequentialize}$ *satisfies correctness. That is, for* $(\vec{\mathsf{rt}}, \vec{\pi}) \leftarrow \mathsf{H.Sequentialize}(\mathsf{pp}, \mathsf{tree}, S, V, \tau)$*, it holds that*

$$\mathsf{rt}_{|S|} = \mathsf{rt} \quad \textit{and} \quad \mathsf{H.VerUpd}(\mathsf{pp}, \mathsf{rt}_{i-1}, \ell_i, v_i, \mathsf{rt}_i, \pi_i)$$

for each $i \in [|S|]$*, where* v_i *is the value in* V *corresponding to node* ℓ_i*.*

Proof. Let A_i and B_i be the list of values for nodes in $\mathsf{ancestors}(\ell_i)$ and $\mathsf{dangling}(\ell_i)$, respectively, formed by processor ρ_i during the algorithm. We will show that if we had performed the p updates sequentially using p calls to $\mathsf{H.Update}$, rather than updating all of S together, then the values in A_i are exactly those that would have been written to $\mathsf{ancestors}(\ell_i)$ by the ith update, and the values in B_i are identical to those that would have been computed for the proof of the ith update. By update completeness, this will give the claim.

For any level $j \in [\log n - 1]$, let $A_i[j]$ and $B_i[j]$ denote the jth value in A_i and B_i, respectively. We will say that a value $A_i[j]$ or $B_i[j]$ is *correct* if it is equal to the value of the corresponding node at level j after the jth update, when performing the updates sequentially as described above. We will show by induction on j that after the jth iteration of the loop, the most recent values in A_i and B_i are correct for all processors i.

For the base cases, it suffices to show that A_i and the partial tree T are initialized correctly. For initializing A_i, the first value in A_i (corresponding to the value of ℓ_i after the update) is set to be the value given by V, which by definition is the updated value of ℓ_i. For the partial tree T, recall that T corresponds to the tree defined only on nodes in $\mathsf{ancestors}(S) \cup \mathsf{dangling}(S)$, and is formed by starting with the values for nodes in $S \cup \mathsf{dangling}(S)$ before the updates, and then using these to compute the values for the rest of the nodes. We observe that for any node in this tree not in $S \cup \mathsf{dangling}(S)$, both of its children must also be in T, and thus the values for $S \cup \mathsf{dangling}(S)$ suffice to fill in the values for the remaining nodes. Since the remaining nodes' values are calculated by hashing their children's values, the tree T contains the correct values for $\mathsf{ancestors}(S) \cup \mathsf{dangling}(S)$ before the update.

Next, suppose that the values added to A_i and B_i in iteration $j-1$ of the loop are correct for all i. We will show that the values added to A_i and B_i in the jth iteration of the loop are correct. It will be helpful to note that after iteration $j-1$, A_i contains j values (one for each ancestor in the first j levels), and B_i contains $j-1$ values, so we want to show that $A_i[j+1]$ and $B_i[j]$ are correct. For A_i, we have that $A_i[j+1]$ is set to the hash of $A_i[j]$ and $B_i[j]$. Thus, it suffices to show that $B_i[j]$ is correct.

For B_i, recall that at this point, we would like to add the value for the node at level j in $\mathsf{dangling}(\ell_i)$, denoted $\ell_{\mathsf{sib},i}$. We would like to find the value for this node just prior to the ith update, as it is not changed by the ith update. Since $\ell_{\mathsf{sib},i} \in \mathsf{dangling}(\ell_i)$, it is either in $\mathsf{dangling}(S)$, or is an ancestor of some node in S, and these two sets are mutually exclusive. We therefore have the following cases:

- **Case 1: $\ell_{\mathsf{sib},i} \in \mathsf{dangling}(S)$.** In this case, we take its value from T, which corresponds to its value before any update by the base case. Note that nodes in $\mathsf{dangling}(S)$ are not updated when updating nodes in S, and thus this is the desired value.

- **Case 2: $\ell_{\mathsf{sib},i} \in \mathsf{ancestors}(S)$ and is a right child.** In this case, *every* node in S for which $\ell_{\mathsf{sib},i}$ is an ancestor is to the right of ℓ_i—this is because $\ell_{\mathsf{sib},i}$ is not an ancestor of ℓ_i itself. Therefore, the value of $\ell_{\mathsf{sib},i}$ would not have been changed by any update prior to the ith update, so the correct value is the value of $\ell_{\mathsf{sib},i}$ before any update. The algorithm finds this value in T, which is correct by our base case.

- **Case 3: $\ell_{\mathsf{sib},i} \in \mathsf{ancestors}(S)$ and is a left child.** Finally, in this case, the value of $\ell_{\mathsf{sib},i}$ is updated by some update prior to the ith update. It is possible that it is updated more than once. Therefore, the correct value for $\ell_{\mathsf{sib},i}$ is its value after the last time it was updated, which corresponds to the rightmost node in S that has $\ell_{\mathsf{sib},i}$ as an ancestor. The algorithm finds this node ℓ_k, and then sets the value for $\ell_{\mathsf{sib},i}$ to be the most recent value in A_k. Since all processors work in sync level by level, this is the value for the ancestor of ℓ_k at level j after the kth update, which is therefore correct.

Therefore, $B_i[j]$ and hence $A_i[j+1]$ are set correctly in the jth iteration, which gives the claim. ■

Claim 6.5. *Suppose that a single hash can be computed in time $\beta(\lambda)$. Then,* $\mathsf{H.Sequentialize}(\mathsf{pp}, S, V, \tau)$ *is an algorithm in the CREW model that runs in time $\beta(\lambda) \cdot \mathrm{poly}(\log p, \log n)$ with p processors.*

Proof. We analyze the efficiency of $\mathsf{H.Sequentialize}$ step by step, starting with forming the partial Merkle tree T in Step 1 on nodes in $\mathsf{ancestors}(S) \cup \mathsf{dangling}(S)$. Since this tree is on $\mathsf{ancestors}(S) \cup \mathsf{dangling}(S)$, it contains at most $2p$ nodes at each level—$\mathsf{ancestors}(S)$ contains at most p nodes at each level, and each of those can have at most one sibling in $\mathsf{dangling}(S)$. Therefore, this step can be done using p processors first by copying the $O(p \log n)$ values for $S \cup \mathsf{dangling}(S)$ to the right place in memory in time $O(\log n)$, and then, at each level, using the processors to do the (at most) p hashes concurrently. This can be done in the CREW model by assigning processors to hashes based on the processors' indices. Letting $\beta = \beta(\lambda)$ denote the time to perform a single hash, it follows that this step can be done in time $O(\beta \cdot \log n)$ with p processors. For the rest of the algorithm, we assume that we can access a value in this tree in time $O(\log n)$ given the node's label (by using the label to "traverse" the tree).

Next, in Step 2, we sort S (and V). This can be done in time $O(\log |S|) = O(\log p)$ with p processors using a parallel sorting algorithm in the CREW model (such as that of [Col88]). Initializing the lists A_i and B_i in Step 3 can be then done in constant time (as V is sorted, so the ith value can be directly accessed). We note that for the remainder of the algorithm, processor ρ_i only writes to A_i and B_i, and so no processors attempt to write concurrently to the same location.

Next, we look at the loop in Step 4. At each level of the tree, ρ_i finds a value for $\ell_{\mathsf{sib},i}$ in Step 4a, and then performs a hash in step 4b. The hash takes time β, so it remains to bound the runtime of the find step. First, ρ_i searches S for a descendant of $\ell_{\mathsf{sib},i}$. Since S is sorted, this can be done using a binary search for the leftmost and rightmost descendants of $\ell_{\mathsf{sib},i}$, and returning the rightmost node found in this range. Therefore, this takes time $O(\log |S|) = O(\log p)$. If no such node is found, ρ_i retrieves the value for that node from T, which takes time $O(\log n)$ by the discussion above. Therefore, the find step can be done in time $O(\log p + \log n)$. Since each processor does the find step in parallel, and there are a total of $\log n$ levels, overall the loop takes time $O(\log n \cdot (\beta + \log p + \log n))$ with p processors.

Since each processor computes its output throughout the algorithm, Step 5 takes constant time. Therefore, putting everything together, the algorithm $\mathsf{H.Sequentialize}$ runs in time $\beta \cdot \text{poly}(\log p, \log n)$ with p processors, giving the claim. ■

This completes the proof of Lemma 6.2. □

6.2 Parallelizable Delegation

In this section, we show that the delegation scheme from Corollary 4.6 can be made depth preserving for parallel computations. Specifically, consider delegating a computation that can be run in parallel time t when using p processors. We show that when instantiating the updatable delegation scheme with a sequentializable hash tree, we obtain an updatable scheme where the prover runs in parallel time $t \cdot \text{poly}(\lambda, \log(t \cdot p))$ when using p processors.

Syntactically, when considering parallel computations, we note that the number of processors p is included in the statement. Completeness, collision resistance, and local opening are unchanged from the sequential case. Soundness and succinctness naturally extend to consider computations with total work $t \cdot p$ rather than t. The formal definition is given in Definition A.6. Next, we formalize the required prover efficiency.

Definition 6.6 (Parallel Prover Efficiency for Delegation)**.** *For functions α and ρ, an updatable PRAM delegation scheme for $\mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ satisfies (α, ρ)-*prover efficiency *if for all $\lambda \in \mathbb{N}$, keys $(\mathsf{crs}, \mathsf{dk})$ in the support of $\mathsf{Del.S}(1^\lambda)$, statement $(\mathsf{cf}, \mathsf{cf}', t, p) \in \mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ using $n \leq 2^\lambda$ memory with $t, p \leq 2^\lambda$, $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$, and $(\mathsf{rt}', \mathsf{tree}', w) = \mathsf{Del.Update}(\mathsf{dk}, t, p, \mathsf{tree})$, it holds that $\mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t, p), w)$ runs in time $\alpha(\lambda, t, p)$ with $\rho(\lambda, t, p)$ processors.*

When $\alpha(\lambda, t, p) \in t \cdot \text{poly}(\lambda, \log(t \cdot p))$ and $\rho(\lambda, t, p) \in p \cdot \text{poly}(\lambda, \log(t \cdot p))$, we say that the delegation scheme is depth-preserving.

We also extend the efficiency of $\mathsf{Del.Update}$ to take into account parallel updates.

Definition 6.7 (Parallel Update Efficiency)**.** *A β-updatable PRAM delegation scheme for $\mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ has the following efficiency for $\mathsf{Del.Update}$. For any $\lambda \in \mathbb{N}$, statement $(\mathsf{cf}, \mathsf{cf}', t, p) \in \mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$, keys $(\mathsf{crs}, \mathsf{dk})$ in the support of $\mathsf{Del.S}(1^\lambda)$, $(\mathsf{rt}, \mathsf{tree}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$, it holds that $\mathsf{Del.Update}(\mathsf{dk}, t, p, \mathsf{tree})$ runs in time $t + \beta(\lambda)$ with $p \cdot \beta(\lambda)$ processors. When $\beta(\lambda) \in \text{poly}(\lambda)$, we simply say the scheme is* updatable.

Using these definitions, we can state the main result of this section.

Theorem 6.8. *Assuming the hardness of LWE, there exists a publicly verifiable, succinct, and updatable PRAM delegation scheme for $\mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ with local opening that is depth-preserving. The construction is in the CREW model.*

We prove this result in two steps. First, we analyze the parallel efficiency of the prover $\mathsf{Del.P}$ from the delegation scheme in Corollary 4.6, and show that $\mathsf{Del.P}$ can prove *sequential* computations on T steps in parallel time $(T/p) \cdot \text{poly}(\lambda, \log T)$ with p processors for any p, when given a witness consisting of the hash tree updates corresponding to the computation. Then, we show that by instantiating the above delegation scheme with a sequantializable hash tree, we can obtain a scheme for parallel computations. Specifically, consider computations with parallel time t on p processors. After running these computations to obtain t hash tree updates each to p words in memory, we will use the sequentializability of the hash tree to transform these into $T = t \cdot p$ updates, each to a single word. We will then run $\mathsf{Del.P}$ above using p processors, and show that this results in the desired efficiency.

Parallel efficiency of $\mathsf{Del.P}$. We start by analyzing the *parallel* efficiency of the delegation scheme given in Corollary 4.6, when proving *sequential* computations. Recall that to prove a statement $(\mathsf{cf}, \mathsf{cf}', T)$, the prover $\mathsf{Del.P}$ takes as input a witness w consisting of T hash tree updates, each updating a single word in memory. It then uses these to run the prover algorithm from the CJJ construction. We recall from Section 4.2 that the prover does the following:

1. Commit to the sequence of T updates, each of length $\ell \in \text{poly}(\lambda)$. The specific commitment computes ℓ copies of a Merkle-tree style commitment, each in time $T \cdot \ell \cdot \text{poly}(\lambda)$ to a string of length $T \cdot \ell$.

2. Create local openings in the commitment for each of the T updates. This requires opening ℓ bits for each of the T updates. Each opening (to a single bit) takes time $\ell \cdot \text{poly}(\lambda, \log(T \cdot \ell)) \in \text{poly}(\lambda, \log T)$.

3. Use a BARG for the circuit C_{step} to prove the correctness of the T updates. Internally, the BARG construction does the following:

   (a) Computes a PCP for each of the T statements, each taking time $\text{poly}(\lambda, |C_{\mathsf{step}}|)$.

   (b) Commits columnwise to the PCPs, creating $\text{poly}(\lambda, |C_{\mathsf{step}}|)$ commitments to T bits each.

   (c) Applies a correlation-intractable hash to C and the commitment, and then samples PCP queries. Together, these can be done in time $\text{poly}(\lambda, \log T, |C_{\mathsf{step}}|)$.

   (d) Opens the query locations in the corresponding commitments. This requires opening T values in $\text{poly}(\lambda, \log |C_{\mathsf{step}}|)$ commitments, each of which can be done in time $\text{poly}(\lambda, \log T)$ time. Together, this takes time $T \cdot \text{poly}(\lambda, \log T, \log |C_{\mathsf{step}}|)$.

   (e) Recurses on a BARG for $T/2$ instances with a circuit of size $\text{poly}(\lambda, \log T, \log |C_{\mathsf{step}}|)$, for a total of $\log T$ recursions.

We observe that this can be improved by using parallelism. Specifically, suppose $\mathsf{Del.P}$ has p processors. Then, we can obtain the following efficiency:

1. Each of the ℓ commitments to $T \cdot \ell$ bits can be computed in time $(T \cdot \ell / p) \cdot \text{poly}(\lambda)$. Since $\ell \in \text{poly}(\lambda)$, this results in parallel time $(T/p) \cdot \text{poly}(\lambda)$ with p processors for the commitment.

2. The local openings in the commitment can be similarly parallelized, so that we can open the required $T \cdot \ell$ bits in time $(T \cdot \ell / p) \cdot \text{poly}(\lambda, \log T) \in (T/p) \cdot \text{poly}(\lambda, \log T)$.

3. We can parallelize the BARG as follows:

   (a) The PCPs can be computed in parallel, thereby taking time $(T/p) \cdot \text{poly}(\lambda, |C_{\mathsf{step}}|)$.

   (b) For each column of T bits, the commitment can be computed in time $(T/p) \cdot \text{poly}(\lambda)$. Therefore, committing to the columns will take $(T/p) \cdot \text{poly}(\lambda, |C_{\mathsf{step}}|)$ time.

   (c) Computing the correlation-intractable hash and sampling PCP queries can be done in $\text{poly}(\lambda, \log T, |C_{\mathsf{step}}|)$ time as before.

   (d) The commitment openings can be computed in parallel, since each bit can be opened independently. Therefore, this can be done in time $(T/p) \cdot \text{poly}(\lambda, \log T, \log |C_{\mathsf{step}}|)$.

   (e) We can then recurse on $T/2$ instances, on a circuit of size $\text{poly}(\lambda, \log T, \log |C_{\mathsf{step}}|)$, similarly parallelizing the recursive steps.

   Therefore, the BARG can be computed in time $(T/p) \cdot \text{poly}(\lambda, \log T, |C_{\mathsf{step}}|)$ with p processors.

Finally, we note that when proving sequential computations, C_{step} is the circuit that verifies a single step of computation, in which case $|C_{\mathsf{step}}| \in \mathrm{poly}(\lambda, \log T)$. Putting everything together, it follows that $\mathsf{Del.P}$ tuns in time

$$(T/p) \cdot \mathrm{poly}(\lambda, \log T)$$

with p processors.

Depth-preserving delegation. Let Del be the delegation scheme from Corollary 4.6 and let H be a sequentializable, concurrently updatable hash tree (which, by definition, allows for parallel updates). We will show that by instantiating Del with H and modifying the $\mathsf{Del.Update}$ and $\mathsf{Del.P}$ algorithms, this gives a delegation scheme for parallel computations. The modified algorithms are as follows:

- $\mathsf{Del.Update}_{\mathsf{par}}(\mathsf{dk}, t, p, \mathsf{tree})$: Run $\mathsf{Del.Update}(\mathsf{dk}, t, \mathsf{tree})$ to update the hash tree for t steps, but use p processors to perform concurrent updates in parallel. The resulting witness w thereby consists of t updates each to at most p words in memory.
- $\mathsf{Del.P}_{\mathsf{par}}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t, p), w)$:
    1. Parse w as a sequence of t parallel updates. For each update, use $\mathsf{H.Sequentialize}$ to create at most p sequential updates. Let w' be the resulting sequence of $T \leq t \cdot p$ updates.
    2. Output $\pi \leftarrow \mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', T), w')$.

We will show that this results in a scheme satisfying the statement of Theorem 6.8.

Proof of Theorem 6.8. It suffices to show completeness, soundness, succinctness, local opening, updatability, and prover efficiency. Soundness and local opening follow directly from the corresponding properties of Del. Completeness follows from the sequentializability of H and completeness of Del. Succinctness follows from that of Del, and in particular gives a scheme where proof length and verifier efficiency depend polynomially on $\log(t \cdot p)$.

For updatability, we note that the functionality of $\mathsf{Del.Update}_{\mathsf{par}}$ follows from that of H (just as in the case of $\mathsf{Del.Update}$). For the efficiency of $\mathsf{Del.Update}_{\mathsf{par}}$, we have that $\mathsf{Del.Update}_{\mathsf{par}}(\mathsf{dk}, t, p, \mathsf{tree})$ runs in time $t + \mathrm{poly}(\lambda)$ with $p \cdot \mathrm{poly}(\lambda)$ processors, by the parallel efficiency of H, as required. Moreover, concurrent updates can be pipelined to get the desired efficiency, by the parallel efficiency of H.

To show that the prover is depth preserving, we recall that by the sequentializability of H, it holds that $\mathsf{H.Sequentialize}$ runs in time $\beta(\lambda) \cdot \mathrm{poly}(\log p, \log n)$ with p processors. As $\beta(\lambda) \in \mathrm{poly}(\lambda)$, $n \leq 2^{\lambda}$, and $\mathsf{Del.P}_{\mathsf{par}}$ runs $\mathsf{H.Sequentialize}$ t times, this takes time $t \cdot \mathrm{poly}(\lambda, \log p)$. Then, by the discussion above, running $\mathsf{Del.P}$ on a T-time statement with $T \leq t \cdot p$ can be done in time $(T/p) \cdot \mathrm{poly}(\lambda, \log T) \leq t \cdot \mathrm{poly}(\lambda, \log(t \cdot p))$ with p processors. Putting everything together, this shows that $\mathsf{Del.P}_{\mathsf{par}}$ is depth-preserving, which gives Theorem 6.8. □

6.3 SPARG Construction for Parallel Computations

In this section, we give our construction of SPARGs for parallel RAM computations.

Theorem 6.9. *Let Del be a publicly verifiable, succinct, and updatable delegation scheme for $\mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ with local opening that is depth preserving. Then, there exists a SPARG for $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$. Specifically, for all $\lambda \in \mathbb{N}$ and $(M, x, y, L, t, p) \in \mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ where M has access to $n \leq 2^{\lambda}$ words in memory and $t \leq 2^{\lambda}$, the following hold. Let (α, ρ) be the prover efficiency of $\mathsf{Del.P}$, and let $\alpha^\star = \alpha(\lambda, t, p)/t$*

be the multiplicative overhead of Del.P *with respect to the number of depth of computation and* $\rho^\star = \rho(\lambda, t, p)$*. Then:*

- *The depth of the prover is bounded by* $t + L + (\alpha^\star)^2 \cdot \text{poly}(\lambda, |M, x|, \log(t \cdot p))$ *when using* $(p + \rho^\star) \cdot \alpha^\star \cdot \text{poly}(\lambda, \log t)$ *processors.*
- *The proof size is bounded by* $\alpha^\star \cdot \text{poly}(\lambda, \log(t \cdot p))$*.*
- *The work of the verifier is bounded by* $\alpha^\star \cdot L \cdot \text{poly}(\lambda, |M, x|, \log(t \cdot p))$*.*

The construction is in the CREW model.

By combining Theorem 6.9 with Theorem 6.8, we obtain the following corollary.

Corollary 6.10. *Assuming the hardness of LWE, there exists a SPARG for* $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$*.*

Proof of Theorem 6.9. The theorem follows by instantiating the SPARG $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ in Figure 5 with a delegation scheme Del.P for PRAM that is depth preserving. Completeness, soundness, and succinctness follow similarly to the sequential case, given by Theorem 5.1. It suffices to show prover efficiency.

Let (α, ρ) be the prover efficiency of Del.P and let $\alpha^\star = \alpha(\lambda, t, p)/t$ and $\rho^\star = \rho(\lambda, t, p)$. We showed in Lemma 5.8 that the prover depth can be split into initializing values for the computation, computing and proving each sub-computation, and computing its output. It follows from our analysis that everything other than computing and proving each sub-computation takes time $L + \text{poly}(\lambda, |M, x|, \log(t \cdot p))$ with a single processor.

For computing and proving the output, it follows directly from [EFKP20b, Claims 6.14 and 6.15] that $\mathcal{P}$ computes $m \leq \gamma \log t$ proofs, and all proofs complete within depth $t + \gamma^2(\log t + 1) + \beta(\lambda)$. Since $\gamma = \alpha^\star + 1$ and $\beta(\lambda) \in \text{poly}(\lambda)$, it follows that all sub-proofs complete within depth $t + (\alpha^\star)^2 \cdot \text{poly}(\lambda, \log t)$.

For the number of processors, we have that $\mathcal{P}$ runs the computation using p processors. In parallel to running the computation, $\mathcal{P}$ runs Del.Update. We note that by the parallel efficiency of the hash tree, all of the calls to Del.Update can be done using a total of $p \cdot \beta(\lambda)$ processors (exactly as done in [EFKP20b]). Finally, for each sub-computation, $\mathcal{P}$ runs Del.P, and therefore require $\rho^\star$ processors for each of the m sub-proofs. Putting everything together, $\mathcal{P}$ requires $p + p \cdot \beta(\lambda) + \rho^\star \cdot m$ processors. As $\beta \in \text{poly}(\lambda)$ and there are a total of $m \leq \gamma \log t = (\alpha^\star + 1) \cdot \log t$ sub-computations, it follows that the prover uses $(p + \rho^\star) \cdot \alpha^\star \cdot \text{poly}(\lambda, \log t)$ processors. ☐

7 Time-Independent SPARGs

In this section, we show how to construct non-interactive SPARGs that don't depend on an a priori given time bound t. We refer to such a SPARG as being *time-independent*. The main distinction is that the prover doesn't take as input the time t of the computation $M(x)$, which we provide the syntax for below.

Definition 7.1 (Time-Independent SPARG)**.** *We say that a SPARG given by the algorithms* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is* time-independent *if the prover* $\mathcal{P}$ *has the following modified syntax, and all properties hold with respect to such a prover:*

- $(y, t, \pi) \leftarrow \mathcal{P}(\mathsf{crs}, (M, x, L))$*: A probabilistic algorithm that on input a common reference string* crs*, a statement* (M, x, L)*, and a witness* w*, outputs a value* y*, a time* t*, and a proof* π*.*

Our construction is black-box from any *time-tight*, updatable RAM delegation scheme (with local opening) that takes as input the time bound t for the computation. At a high level, a time-tight RAM delegation scheme is one where the update algorithm also output the proof π, while still only incurring an additive $\mathrm{poly}(\lambda, \log t)$ overhead while using at most $\mathrm{poly}(\lambda, \log t)$ processors. We note that our construction of a non-interactive SPARG in Section 5 actually implies a time-tight updatable RAM delegation scheme.

We use this as a building block in contrast to just a non-interactive SPARG in order to prove statements corresponding to intermediate configurations of the RAM machine (as we do in our main SPARG construction), which may be large, so we instead prove things relative to the corresponding digests. We formalize a time-tight updatable RAM delegation scheme in Definition 7.2.

Definition 7.2 (Time-tight Updatable RAM Delegation). *A time-tight updatable RAM delegation scheme with local opening consists of algorithms* $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.UpdateP}, \mathsf{Del.V}, \mathsf{Del.Open}, \mathsf{Del.VerOpen})$*, where* $\mathsf{Del.S}$*,* $\mathsf{Del.D}$*,* $\mathsf{Del.V}$*,* $\mathsf{Del.Open}$*,* $\mathsf{Del.VerOpen}$ *have the same syntax as an updatable RAM delegation scheme with local opening, and* $\mathsf{Del.UpdateP}$ *has the following syntax*

- $(\mathsf{rt}', \mathsf{tree}', \pi) = \mathsf{Del.UpdateP}(\mathsf{crs}, t, \mathsf{tree})$*: The update algorithm takes as input a common reference string* crs*, an integer* t*, and a value* tree*, and outputs a digest* rt'*, a value* tree' *and a proof* π*.*

We require that all properties of an updatable RAM delegation scheme with local opening hold where the outputs of $\mathsf{Del.Update}$ *and* $\mathsf{Del.P}$ *are replaced by* $\mathsf{Del.UpdateP}$*, and* $\mathsf{Del.UpdateP}$ *satisfies the efficiency and updatability property of* $\mathsf{Del.Update}$ *for* $t \leq 2^\lambda$*.*

We next describe our construction of a time-independent, non-interactive SPARG $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ given a time-tight updatable RAM delegation scheme $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.UpdateP}, \mathsf{Del.V}, \mathsf{Del.Open}, \mathsf{Del.VerOpen})$. Let M be a RAM machine with input x. We assume that the computation time t of $M(x)$ is bounded by 2^λ where λ is the security parameter. For any $\lambda \in \mathbb{N}$, let crs, dk be any output of a RAM delegation setup algorithm $\mathsf{Del.S}(1^\lambda)$. For any $i \in \mathbb{N}$, we denote by cf_i the RAM configuration for the computation $M(x)$ after i steps, and $(\mathsf{rt}_i, \mathsf{tree}_i) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}_i)$. For any $a < b \in [0, 2^\lambda]$, we use $\pi_{(a,b)}$ to refer to a non-interactive proof of the statement $(\mathsf{cf}_a, \mathsf{cf}_b, (b-a))$ with respect to crs (where the prover only knows cf_a via the corresponding memory tree_a at the start), which will be verified given crs and $(\mathsf{rt}_a, \mathsf{rt}_b, (b-a))$.

We next provide a high level overview of our construction, with a formal description provided in Figure 3. Suppose that $M(x)$ halts with an output after t steps. We will construct a sequence of $m \leq \lambda$ valid proofs $\pi_{(0,a_1)}, \pi_{(a_1,a_2)}, \ldots, \pi_{(a_{m-1},t)}$ for $a_1 < a_2 < \ldots < a_{m-1}$ without knowing the running time t of $M(x)$ in advance.

Initially, the prover prepares λ distinct blocks of memory each starting with configuration cf_0. With each block of memory, the prover starts a fresh updatable time-tight delegation protocol using $\mathsf{Del.UpdateP}$ to compute $\pi_{(0,2^i)}$ for all $i \in [0, \lambda-1]$, all in parallel. Additionally, we continue performing updates of the form $\pi_{((j-1)\cdot 2^i, j \cdot 2^i)}$ in sequence for all $i \in [0, \lambda - 1]$ and $j \geq 2$ using the ith block of memory. This ensures we have a proof $\pi_{(a,b)}$ for every interval of size 2^i starting at any time step which is a multiple of 2^i. So, we define the intervals $a_0 = 0, a_1, \ldots, a_{m-1}, t$ to correspond to the binary representation of t, so that each interval (a_{i-1}, a_i) that we need will be computed.

Once the computation halts at time t, the prover will wait for a proof corresponding to $\pi_{(a_{m-1},t)}$ to finish for some initial step $a_{m-1} \in \mathbb{N}$, and the prover can halt all other currently running update threads at this time. This final proof output by the prover is the sequence of m proofs $\pi_{(0,a_1)}, \pi_{(a_1,a_2)}, \ldots, \pi_{(a_{m-1},t)}$ that eventually finished at configuration cf_t corresponding to the output of the computation. These proofs correspond to the ones in the binary representation of the time bound t, so the total number of proofs m is at most λ.

The $\mathcal{P}$ and $\mathcal{V}$ algorithm for our non-interactive SPARG for $\mathcal{L}^{\mathcal{U}}$ given a time-tight updatable RAM delegation protocol $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.UpdateP}, \mathsf{Del.V}, \mathsf{Del.Open}, \mathsf{Del.VerOpen})$.

- $\mathcal{P}(1^\lambda, \mathsf{pp}, (M, x, L))$:
  1. Let cf_0 be the initial configuration and start computing $M(x)$ in λ parallel threads with different blocks of memory. Label each block of memory by a unique index $i \in [0, \lambda - 1]$. Let cf_i be the RAM configuration after i steps and $(\mathsf{rt}_i, \mathsf{tree}_i) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}_i)$.
  2. In parallel for each $i \in [0, \lambda - 1]$ and for each $j \geq 1$ such that $j \cdot 2^i \leq 2^\lambda$, compute $(\mathsf{rt}_{j \cdot 2^i}, \mathsf{tree}_{j \cdot 2^i}, \pi_{((j-1) \cdot 2^i, j \cdot 2^i)}) \leftarrow \mathsf{Del.UpdateP}(\mathsf{crs}, 2^i, \mathsf{tree}_0)$ in the ith block of memory.
  3. Once $M(x)$ halts, set t to be the number of steps that it took to halt. Let $s = (s_{\lambda-1}, \ldots, s_0) \in \{0, 1\}^\lambda$ be the binary representation of t such that $s_i = 1$ corresponds to the 2^i component of t for $i \in [0, \lambda - 1]$. For $i \in [\lambda]$, define t_i to be the sum of the i most significant bits of t, so $t_i = \sum_{j=\lambda-i}^{\lambda-1} s_j \cdot 2^j$ and note that $t_\lambda = t$. Let $a_0 = 0$ and $a_1, \ldots, a_m$ be the distinct, positive t_i values in increasing order, so $m \leq \lambda$ and $a_m = t$. Set $\tau_i = \pi_{(a_{i-1}, a_i)}$ for $i \in [m]$ once they have all been computed, and halt all other proofs and computation.
  4. Let y be the output of M and let st be the local state of M both given in the final configuration cf_t. Let $\vec{\mathsf{rt}} = (\mathsf{rt}_{a_1}, \ldots, \mathsf{rt}_{a_m})$ and $\vec{\tau} = (\tau_1, \ldots, \tau_m)$. Output (y, t, π) where $\pi = (\vec{\mathsf{rt}}, \vec{\tau}, \mathsf{st})$.
- $\mathcal{V}(1^\lambda, \mathsf{pp}, (M, x, y, L, t), \pi)$:
  1. Parse $\pi = ((\mathsf{rt}_1, \ldots, \mathsf{rt}_m), (\tau_1, \ldots, \tau_m), \mathsf{st})$ and compute $\mathsf{rt}_0 = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}_0)$ where cf_0 is the initial configuration of M. Given t, compute $a_0, \ldots, a_m$ as $\mathcal{P}$ does and set $k_i = a_i - a_{i-1}$ for all $i \in [m]$.
  2. Output 1 if and only if the following hold, and 0 otherwise:
     (a) $\mathsf{Del.V}(\mathsf{crs}, (\mathsf{rt}_{i-1}, \mathsf{rt}_i, k_i), \tau_i)$ accepts for all $i \in [m]$.
     (b) $\mathsf{Del.VerOpen}(\mathsf{dk}, \mathsf{rt}_m, [L], y, \mathsf{st}, \pi_y) = 1$.
     (c) st is a halting state, and $|y| \leq L$.

Figure 3: SPARG for $\mathcal{L}^{\mathcal{U}}$.

Theorem 7.3. *Suppose there exists a time-tight, updatable RAM delegation scheme. Then, there exists a time-independent, non-interactive SPARG for $\mathcal{L}^{\mathcal{U}}$.*

Before proving the theorem, we note that our construction of a non-interactive SPARG in Section 5 actually gives a time-tight, updatable RAM delegation scheme. So, we get the following corollary based on Corollary 5.2.

Corollary 7.4. *Assuming the hardness of LWE, there exists a time-independent, non-interactive SPARG.*

We proceed to prove the main theorem of this section.

Proof of Theorem 7.3. We will prove that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ of Figure 3 is a time-independent, non-interactive SPARG for $\mathcal{L}^{\mathcal{U}}$ assuming $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.UpdateP}, \mathsf{Del.V}, \mathsf{Del.Open}, \mathsf{Del.VerOpen})$ is a time-tight, updatable RAM delegation scheme. Completeness is straightforward by construction, and soundness follows from the proof of Lemma 5.4. For succinctness, the verifier needs to check m RAM delegation proofs, where m corresponds to the number of ones in the binary representation of the time bound t. As $t \leq 2^\lambda$, this implies that $m \leq \lambda$. Thus, succinctness follows from the proof of Lemma 5.9.

Next, we argue that the prover satisfies optimal prover depth. Let $\beta \in \text{poly}(\lambda)$ be the polynomial guaranteed to exist by the updatability property of the underlying time-tight RAM delegation scheme. So, for any update consisting of T sequential steps, the underlying delegation scheme requires depth $T + \beta(\lambda)$ while using $\beta(\lambda)$ processors to compute the updates as well as the proof π. Additionally, for any two back to back updates of $t_1 + t_2$ steps, the second update finishes at time $t_1 + t_2 + \beta$.

We first show that the depth of $\mathcal{P}$ is bounded by $t + q_1(\lambda)$ for a polynomial q_1. After t time steps, $M(x)$ will halt by definition. Let $a_0, a_1, \ldots, a_m$ be the intervals specified for the prover $\mathcal{P}$ in Figure 3 where the gaps between them are $k_i = a_i - a_{i-1}$ for all $i \in [m]$. After t steps have completed, M will halt and then the prover needs to prepare its output and identify the correct set of proofs. The main bottleneck for the prover is waiting for the RAM delegation proofs $\tau_i = \pi_{(a_{i-1}, a_i)}$ to finish for all $i \in [m]$. By the updatability property of the underlying RAM delegation, each of the proofs τ_i will finish computing their output by time $a_i + \beta(\lambda)$ in the protocol, so they will all finish by time $t + \beta(\lambda)$. Computing all other parts of the output can be done in $\text{poly}(\lambda)$ time, independent of t, so the total depth of the prover is $t + q_1(\lambda)$ for some fixed polynomial q_1.

We now argue that there exists a polynomial q_2 such that at any point in time during $\mathcal{P}$'s computation, at most $q_2(\lambda)$ processors are being used. Let $T < t + q_1(\lambda)$ be an arbitrary number of steps into the prover's computation. We will bound the number of processors being used during step T. Note that each delegation proof uses $\beta(\lambda)$ processors by assumption, so it suffices to bound the number of proofs being computed at any given time. We split the proofs into two groups: (1) proofs $\pi_{(a,b)}$ where $b \leq T$ and (2) proofs $\pi_{(a,b)}$ where $b > T$.

- For group (1), we show that there are at most $\lambda \cdot \beta(\lambda)$ proofs. Above, we argued that all proofs $\pi_{(a,b)}$ finish by time $b + \beta(\lambda)$. This implies that all proofs $\pi_{(a,b)}$ where $b < T - \beta(\lambda)$ have already been completed. However, there are at most λ proofs for each $b \in \mathbb{N}$, so there are at most $\lambda \cdot \beta(\lambda)$ proofs running from group (1), as claimed.

- For group (2), we claim there are at most λ proofs. Each block of memory $i \in [0, \lambda - 1]$ consists of proof covering the intervals $(j-1) \cdot 2^i$ to $j \cdot 2^i$ for $j \geq 1$. So at time T, the only group (2) proof currently being computed will be the interval (a, b) that contains T. Any earlier interval is not a group (2) proof by definition, and no later proofs will have been started at time T. So there are at most λ group (2) proofs, as required.

Putting the above observations together, we conclude that there are at most $\lambda \cdot \beta(\lambda) + \lambda$ proofs being computed at any point in time, which implies that $\mathcal{P}$ uses at most $q_2(\lambda) = (\lambda \cdot \beta(\lambda) + \lambda) \cdot \beta(\lambda)$ processors, as required.

□

Time-Independent Non-interactive and Interactive SPARKs. We conclude this section by noting that the exact same construction works to construct non-interactive SPARKs for NP (satisfying an argument of knowledge property instead of simply soundness) when starting with a delegation scheme for NP satisfying an appropriate argument of knowledge property. By the work of [EFKP20b], this would imply time-independent non-interactive SPARKs for NP from any SNARK for NP.

In contrast to non-interactive SPARKs, we do know how to construct interactive SPARKs for NP simply from collision-resistant hash functions. However, the time-independent construction does not work generically if the underlying delegation scheme is interactive. The interactivity would cause our construction to have at least t rounds of interaction, so it would not be succinct. We leave it as an open question to construct time-independent SPARKs for NP from standard assumptions (namely, without starting from SNARKs for NP).

8 Application to Verifiable Delay Functions

In this section, we show that SPARGs for P and any sequential function imply a VDF. We note that a sequential function is a minimal assumption as VDFs directly imply sequential functions. We use the following building blocks and parameters.

- A sequential function $\mathsf{SF} = (\mathsf{SF.Gen}, \mathsf{SF.Sample}, \mathsf{SF.Eval})$. Let $p_{\mathsf{SF}}, q_{\mathsf{SF}}$ be the polynomials from the honest evaluation property of SF such that $\mathsf{SF.Eval}(1^\lambda, \cdot, \cdot, t)$ runs in time $t + p_{\mathsf{SF}}(\lambda, \log t)$ with $q_{\mathsf{SF}}(\lambda, \log t)$ processors. Let ℓ_{SF} be the polynomial such that the output length is bounded by $\ell_{\mathsf{SF}}(\lambda, \log t)$.
- A SPARG $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ for any $\mathcal{L} \in \mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ containing $\mathsf{SF.Eval}$.

Construction. Our VDF construction $\mathsf{VDF} = (\mathsf{VDF.Gen}, \mathsf{VDF.Sample}, \mathsf{VDF.Eval}, \mathsf{VDF.Verify})$ is as follows.

- $\mathsf{pp} \leftarrow \mathsf{VDF.Gen}(1^\lambda)$**:**
  1. Sample $\mathsf{crs} \leftarrow \mathcal{G}(1^\lambda)$ and $k \leftarrow \mathsf{SF.Gen}(1^\lambda)$.
  2. Output $\mathsf{pp} = (\mathsf{crs}, k)$.
- $x \leftarrow \mathsf{VDF.Sample}(1^\lambda, \mathsf{pp})$**:**
  1. Sample and output $x \leftarrow \mathsf{SF.Sample}(1^\lambda, k)$.
- $(y, \pi) \leftarrow \mathsf{VDF.Eval}(1^\lambda, \mathsf{pp}, x, t)$**:**
  1. Recall that $p_{\mathsf{SF}}, q_{\mathsf{SF}}, \ell_{\mathsf{SF}}$ are the polynomials denoting the efficiency of $\mathsf{VDF.Eval}$. Let $\mathsf{statement} = (\mathsf{SF.Eval}, (1^\lambda, k, x, t), \ell_{\mathsf{SF}}(\lambda, \log t), t + p_{\mathsf{SF}}(\lambda, \log t), q_{\mathsf{SF}}(\lambda, \log t))$.
  2. Compute and output $(y, \pi) \leftarrow \mathcal{P}(1^\lambda, \mathsf{crs}, \mathsf{statement})$.
- $b \leftarrow \mathsf{VDF.Verify}(1^\lambda, \mathsf{pp}, x, t, (y, \pi))$**:**
  1. Let $\mathsf{statement}' = (\mathsf{SF.Eval}, (1^\lambda, k, x, t), y, \ell_{\mathsf{SF}}(\lambda, \log t), t + p_{\mathsf{SF}}(\lambda, \log t), q_{\mathsf{SF}}(\lambda, \log t))$ (note that $\mathsf{statement}'$ differs from $\mathsf{statement}$ used by $\mathsf{VDF.Eval}$ as it contains the output y).
  2. Output $b \leftarrow \mathcal{V}(1^\lambda, \mathsf{crs}, \mathsf{statement}', \pi)$.

Theorem 8.1. *Assuming the existence of a SPARG for $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ and a sequential function, there exists a VDF.*

By combining this with Corollary 6.10, we obtain the following.

Corollary 8.2. *Assuming the hardness of LWE and a sequential function, there exists a VDF.*

Proof of Theorem 8.1. We prove completeness, sequentiality, honest evaluation, soundness, and discuss the efficiency of our algorithms. Completeness follows directly from the completeness of the SPARG. Sequentiality follows directly from the sequentiality of SF.

Before proving honest evaluation and soundness, we note that the first output of $\mathsf{VDF.Eval}$ can be computed as $\mathsf{VDF.Eval}_1(1^\lambda, \mathsf{pp}, x, t) = \mathsf{SF.Eval}(1^\lambda, k, x, t)$. This follows from the completeness of $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, and will be useful below.

Honest Evaluation. Fix any $\lambda, t \in \mathbb{N}$, $\mathsf{pp} = (\mathsf{crs}, k)$ in the support of $\mathsf{VDF.Gen}(1^\lambda)$, and x in the support of $\mathsf{VDF.Sample}(1^\lambda, \mathsf{pp})$. By the honest evaluation property of SF, it holds that $\mathsf{SF.Eval}(1^\lambda, k, x, t)$ can be computed in time $t + p_{\mathsf{SF}}(\lambda, \log t)$ with $q_{\mathsf{SF}}(\lambda, \log t)$ processors. Moreover, the output length is bounded by $\ell_{\mathsf{SF}}(\lambda, \log t)$. Putting these together, it follows that $\mathcal{P}$ on input $\mathsf{statement} = (\mathsf{SF.Eval}, (1^\lambda, k, x, t), \ell_{\mathsf{SF}}(\lambda, \log t), t + p_{\mathsf{SF}}(\lambda, \log t), q_{\mathsf{SF}}(\lambda, \log t))$ runs in depth

$$\begin{aligned}&t + p_{\mathsf{SF}}(\lambda, \log t) + \mathrm{poly}(\lambda, |\mathsf{SF.Eval}, 1^\lambda, k, x, t)|, \lambda, \log((t + p_{\mathsf{SF}}(\lambda, \log t)) \cdot q_{\mathsf{SF}}(\lambda, \log t)))\\ &\in t + \mathrm{poly}(\lambda, |(M_{\lambda,t}, k, x)|, \log t)\end{aligned}$$

with $\mathrm{poly}(\lambda, \log((t + p_{\mathsf{SF}}(\lambda, \log t)) \cdot q_{\mathsf{SF}}(\lambda, \log t))) \in \mathrm{poly}(\lambda, \log t)$ processors. Since k and x are sampled by polynomial-time algorithms, it holds that $|k, x| \in \mathrm{poly}(\lambda)$. Moreover, we can assume that $\mathsf{SF.Eval}$ is represented as a Turing machine with constant size. It follows that $\mathsf{VDF.Eval}$ can be computed in depth $t + p'(\lambda, \log t)$ with $q'(\lambda, \log t)$ processors for fixed polynomials p', q'.

Soundness. Suppose for contradiction that there exists a non-uniform PPT algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}$ and polynomials T, q such that for infinitely many $\lambda \in \mathbb{N}$, it holds that

$$\Pr\left[\begin{array}{l}\mathsf{pp} \leftarrow \mathsf{VDF.Gen}(1^\lambda)\\ (x, y', \pi') \leftarrow \mathcal{A}_\lambda(\mathsf{pp})\\ y = \mathsf{VDF.Eval}_1(1^\lambda, \mathsf{pp}, x, t)\end{array} : \begin{array}{l}\mathsf{VDF.Verify}(1^\lambda, \mathsf{pp}, x, t, (y', \pi')) = 1\\ \wedge\ y \neq y'\end{array}\right] \geq 1/q(\lambda)$$

where $t = T(\lambda)$.

We construct an adversary $\mathcal{P}^\star = \{\mathcal{P}^\star_\lambda\}_{\lambda\in\mathbb{N}}$ against the soundness of the SPARG. For each $\lambda \in \mathbb{N}$, the algorithm $\mathcal{P}^\star_\lambda(\mathsf{crs})$ has $\mathcal{A}_\lambda$ and $p_{\mathsf{SF}}, q_{\mathsf{SF}}, \ell_{\mathsf{SF}}, T$ hardcoded and does the following:

1. Sample $k \leftarrow \mathsf{SF.Gen}(1^\lambda)$ and let $\mathsf{pp} = (\mathsf{crs}, k)$.
2. Run $(x, y', \pi') \leftarrow \mathcal{A}_\lambda(\mathsf{pp})$.
3. Output $(\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell'), \pi')$, where $\ell' = \ell_{\mathsf{SF}}(\lambda, \log t')$ for $t' = t + p_{\mathsf{SF}}(\lambda, \log t)$ and $t = T(\lambda)$.

Note that since $\mathsf{SF.Gen}$ and $\mathcal{A}_\lambda$ are PPT algorithms, $\mathcal{P}^\star_\lambda$ runs in time $\mathrm{poly}(\lambda)$.

To analyze the success of $\mathcal{P}^\star_\lambda$, we will show that $\mathcal{P}^\star_\lambda$ breaks the soundness of $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ on a statement that takes time $T'(\lambda)$ with $P'(\lambda)$ processors, where $T'(\lambda) = T(\lambda) + p_{\mathsf{SF}}(\lambda, \log T(\lambda))$ and $P(\lambda) = q_{\mathsf{SF}}(\lambda, \log T(\lambda))$. We start by expanding the probability above based on our construction. By definition of $\mathcal{P}^\star$, the fact that $\mathsf{VDF.Verify}$ simply runs $\mathcal{V}$, and because $\mathsf{VDF.Eval}_1$ evaluates to $\mathsf{SF.Eval}$, the above inequality implies that

$$\Pr\left[\begin{array}{l}\mathsf{crs} \leftarrow \mathcal{G}(1^\lambda)\\ (\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell'), \pi') \leftarrow \mathcal{P}^\star(\mathsf{crs})\\ y = \mathsf{SF.Eval}(1^\lambda, k, x, t)\end{array} : \begin{array}{l}\mathcal{V}(1^\lambda, \mathsf{crs}, (\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell', t', p'), \pi') = 1\\ \wedge\ y \neq y'\end{array}\right]$$
$$\geq 1/p(\lambda),$$

where $t' = T'(\lambda)$, $p' = P'(\lambda)$, and $\ell' = \ell_{\mathsf{SF}}(\lambda, \log t')$. Finally, as $\mathsf{SF.Eval}$ is deterministic, the output y is unique, and so $y' \neq y$ implies that $(\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell', t', p') \notin \mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$. Therefore, the above implies that

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{G}(1^\lambda) \\ (\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell'), \pi') \leftarrow \mathcal{P}^\star(\mathsf{crs}) \\ y = \mathsf{SF.Eval}(1^\lambda, k, x, t) \end{array} : \begin{array}{l} \mathcal{V}(1^\lambda, \mathsf{crs}, (\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell', t', p'), \pi') = 1 \\ \wedge\ (\mathsf{SF.Eval}, (1^\lambda, k, x, t), y', \ell', t', p') \notin \mathcal{L}^{\mathcal{U}}_{\mathsf{par}} \end{array}\right]$$
$$\geq 1/p(\lambda)$$

in contradiction to the soundness of the SPARG.

Efficiency. Lastly, we discuss the efficiency of our construction. We have that $\mathsf{VDF.Gen}$ and $\mathsf{VDF.Sample}$ are polynomial-time algorithms, since $\mathcal{G}$, $\mathsf{SF.Gen}$, and $\mathsf{SF.Sample}$ can be run in polynomial time. The efficiency of $\mathsf{VDF.Eval}$ was discussed above. For $\mathsf{VDF.Verify}$, the running time depends on the time to form $\mathsf{statement}'$ and run $\mathcal{V}$. By the succinctness of the SPARG, $\mathcal{V}$ runs in time $\mathrm{poly}(\lambda, |\mathsf{SF.Eval}, 1^\lambda, k, x, t|, \ell_{\mathsf{SF}}(\lambda, \log t), \log((t + p_{\mathsf{SF}}(\lambda, \log t)) \cdot (q_{\mathsf{SF}}(\lambda, \log t))))$. Since $|\mathsf{SF.Eval}, 1^\lambda, k, x, t|$ is polynomially bounded, overall this takes time $\mathrm{poly}(\lambda, \log t)$, which is polynomial in its input length, as desired. $\square$

Acknowledgements

This work was supported in part by NSF Award SATC-1704788, NSF Award RI-1703846, AFOSR Award FA9550-18-1-0267, DARPA Award HR00110C0086, and a JP Morgan Faculty Award. Rafael Pass's work was done partially while visiting Tel-Aviv University. Cody Freitag's work was done partially during an internship at NTT Research, and he is also supported in part by the National Science Foundation Graduate Research Fellowship under Grant No. DGE–2139899. Naomi Sirkin was also supported in part by a JP Morgan AI Research PhD Fellowship. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of NSF, DARPA 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

[ABP17] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-robust graphs and their cumulative memory complexity. In *EUROCRYPT (3)*, volume 10212 of *Lecture Notes in Computer Science*, pages 3–32, 2017.

[ABP18] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 99–130. Springer, 2018.

[ACK+16] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. On the complexity of scrypt and proofs of space in the parallel random oracle model. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 358–387. Springer, 2016.

[ALM$^+$98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In *STOC*, pages 595–603. ACM, 2015.

[BBBF18] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *CRYPTO (1)*, volume 10991 of *Lecture Notes in Computer Science*, pages 757–788. Springer, 2018.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *CRYPTO (3)*, volume 11694 of *Lecture Notes in Computer Science*, pages 701–732. Springer, 2019.

[BC12] Nir Bitansky and Alessandro Chiesa. Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 255–272. Springer, 2012.

[BCC$^+$17] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. The hunting of the SNARK. *J. Cryptol.*, 30(4):989–1066, 2017.

[BCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014.

[BCG$^+$17] Eli Ben-Sasson, Alessandro Chiesa, Ariel Gabizon, Michael Riabzev, and Nicholas Spooner. Interactive oracle proofs with constant rate and query complexity. In *ICALP*, volume 80 of *LIPIcs*, pages 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[BCG$^+$19] Eli Ben-Sasson, Alessandro Chiesa, Lior Goldberg, Tom Gur, Michael Riabzev, and Nicholas Spooner. Linear-size constant-query iops for delegating computation. In *TCC (2)*, volume 11892 of *Lecture Notes in Computer Science*, pages 494–521. Springer, 2019.

[BCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete efficiency of probabilistically-checkable proofs. In *STOC*, pages 585–594. ACM, 2013.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *TCC (B2)*, volume 9986 of *Lecture Notes in Computer Science*, pages 31–60, 2016.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC*, pages 21–31. ACM, 1991.

[BG08] Boaz Barak and Oded Goldreich. Universal arguments and their applications. *SIAM J. Comput.*, 38(5):1661–1694, 2008.

[BHK17] Zvika Brakerski, Justin Holmgren, and Yael Tauman Kalai. Non-interactive delegation and batch NP verification from standard computational assumptions. In *STOC*, pages 474–482. ACM, 2017.

[BHR$^+$20] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In *TCC (2)*, volume 12551 of *Lecture Notes in Computer Science*, pages 168–197. Springer, 2020.

[BHR$^+$21] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In *CRYPTO (4)*, volume 12828 of *Lecture Notes in Computer Science*, pages 123–152. Springer, 2021.

[BS08] Eli Ben-Sasson and Madhu Sudan. Short pcps with polylog query complexity. *SIAM J. Comput.*, 38(2):551–607, 2008.

[CFH$^+$15] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 253–270. IEEE Computer Society, 2015.

[Chi] Chia network. https://chia.net/. Accessed: 2019-05-17.

[CJJ21] Arka Rai Choudhuri, Abhishek Jain, and Zhengzhong Jin. Snargs for $\mathcal{P}$ from LWE. In *FOCS*, pages 68–79. IEEE, 2021.

[Col88] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[DGMV20] Nico Döttling, Sanjam Garg, Giulio Malavolta, and Prashant Nalini Vasudevan. Tight verifiable delay functions. In *SCN*, volume 12238 of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2020.

[DGN03] Cynthia Dwork, Andrew V. Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.

[DLP18] Thaddeus Dryja, Quanquan C. Liu, and Sunoo Park. Static-memory-hard functions, and modeling the cost of space vs. time. In *TCC (1)*, volume 11239 of *Lecture Notes in Computer Science*, pages 33–66. Springer, 2018.

[DNW05] Cynthia Dwork, Moni Naor, and Hoeteck Wee. Pebbling and proofs of work. In *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2005.

[EFKP20a] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Continuous verifiable delay functions. In *EUROCRYPT (3)*, volume 12107 of *Lecture Notes in Computer Science*, pages 125–154. Springer, 2020.

[EFKP20b] Naomi Ephraim, Cody Freitag, Ilan Komargodski, and Rafael Pass. Sparks: Succinct parallelizable arguments of knowledge. In *EUROCRYPT (1)*, volume 12105 of *Lecture Notes in Computer Science*, pages 707–737. Springer, 2020.

[Eth] Ethereum foundation. https://www.ethereum.org/. Accessed: 2019-05-17.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, 2015.

[GMR89] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[GW11] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, pages 99–108. ACM, 2011.

[HR18] Justin Holmgren and Ron Rothblum. Delegating computations with (almost) minimal time and space overhead. In *FOCS*, pages 124–135. IEEE Computer Society, 2018.

[HW14] Pavel Hubácek and Daniel Wichs. On the communication complexity of secure function evaluation with long output. *IACR Cryptol. ePrint Arch.*, page 669, 2014.

[JKKZ21] Ruta Jawale, Yael Tauman Kalai, Dakshita Khurana, and Rachel Zhang. Snargs for bounded depth computations and PPAD hardness from sub-exponential LWE. In *STOC*, pages 708–721. ACM, 2021.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, pages 723–732. ACM, 1992.

[KP16] Yael Tauman Kalai and Omer Paneth. Delegating RAM computations. In *TCC (B2)*, pages 91–118, 2016.

[KPY19] Yael Tauman Kalai, Omer Paneth, and Lisa Yang. How to delegate computations publicly. In *STOC*, pages 1115–1124. ACM, 2019.

[KRR14] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. How to delegate computations: the power of no-signaling proofs. In *STOC*, pages 485–494. ACM, 2014.

[LV20] Alex Lombardi and Vinod Vaikuntanathan. Fiat-shamir for repeated squaring with applications to ppad-hardness and vdfs. In *CRYPTO (3)*, volume 12172 of *Lecture Notes in Computer Science*, pages 632–651. Springer, 2020.

[Mic00] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society, 2013.

[Pie19] Krzysztof Pietrzak. Simple verifiable delay functions. In *ITCS*, volume 124 of *LIPIcs*, pages 60:1–60:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[RR20] Noga Ron-Zewi and Ron D. Rothblum. Local proofs approaching the witness length [extended abstract]. In *FOCS*, pages 846–857. IEEE, 2020.

[RRR21] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *SIAM J. Comput.*, 50(3), 2021.

[Wes19] Benjamin Wesolowski. Efficient verifiable delay functions. In *EUROCRYPT (3)*, volume 11478 of *Lecture Notes in Computer Science*, pages 379–407. Springer, 2019.

[WZC+18] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In *USENIX Security Symposium*, pages 675–692. USENIX Association, 2018.

A Additional Preliminaries

A.1 Concurrently Updatable Hash Functions

In this section, we recall the definition of concurrently updatable hash functions due to [EFKP20b]. Their construction is defined for the following natural notion.

Definition A.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.*

A *concurrently updatable hash function* is a tuple of algorithms $(\mathsf{H.Gen}, \mathsf{H.Hash}, \mathsf{H.Open}, \mathsf{H.Update}, \mathsf{H.VerOpen}, \mathsf{H.VerUpd})$ with the following syntax.

- $\mathsf{pp} \leftarrow \mathsf{H.Gen}(1^\lambda, n)$**:** A PPT algorithm that on input the security parameter λ in unary and an integer n, outputs public parameters pp.
- $(\mathsf{tree}, \mathsf{digest}) = \mathsf{H.Hash}(\mathsf{pp}, D)$**:** A deterministic algorithm that on input public parameters pp and a partial string D, outputs a pointer tree to a location in memory and a string digest.
- $(V, \pi) = \mathsf{H.Open}(\mathsf{pp}, \mathsf{tree}, S)$**:** A read-only deterministic algorithm that on input public parameters pp, a pointer tree, 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 $v_i \in \{0,1\}^\lambda \cup \{\perp\}$, and a proof π.
- $(\mathsf{digest}, \tau) = \mathsf{H.Update}(\mathsf{pp}, \mathsf{tree}, S, V)$**:** A deterministic algorithm that on input public parameters pp, a pointer tree, 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.
- $b = \mathsf{H.VerOpen}(\mathsf{pp}, \mathsf{digest}, S, V, \pi)$**:** 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 $v_i \in \{0,1\}^\lambda \cup \{\perp\}$, and a proof π, outputs a bit b.
- $b = \mathsf{H.VerUpd}(\mathsf{pp}, \mathsf{digest}, S, V, \mathsf{digest}', \tau)$**:** 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 note that when S is a single location ℓ and V is a single word v, to simplify notation we let $\mathsf{H.Open}$, $\mathsf{H.Update}$, $\mathsf{H.VerOpen}$, and $\mathsf{H.VerUpd}$ take ℓ and v as input rather than the singleton ordered set (ℓ) and tuple (v). We require the following completeness, soundness, and efficiency properties.

Definition A.2 (Completeness)**.** *Let $\lambda, n \in \mathbb{N}$ with $n \leq 2^\lambda$, pp be in the support of $\mathsf{H.Gen}(1^\lambda, n)$, $D = (n, I, A)$ be a partial string, and $m \geq 0$. 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* $(\mathsf{tree}, \mathsf{digest}^{(0)}) = \mathsf{H.Hash}(\mathsf{pp}, D)$.

2. For $i = 1, \dots, m$, compute $(\mathsf{digest}^{(i)}, \tau^{(i)}) = \mathsf{H.Update}(\mathsf{pp}, \mathsf{tree}, S^{(i)}, V^{(i)})$.

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

- **Open Completeness.** *Let $(V, \pi) = \mathsf{H.Open}(\mathsf{pp}, \mathsf{tree}, S)$ where $V = (v_1, \dots, v_p)$. Then,*
$$\mathsf{H.VerOpen}(\mathsf{pp}, \mathsf{digest}^{(m)}, S, V, \pi) = 1 \ \wedge \ D'_{\ell_i} = v_i \ \forall i \in [p].$$
- **Update Completeness.** *For any tuple $V \in (\{0,1\}^\lambda)^p$, let $(\mathsf{digest}, \tau) = \mathsf{H.Update}(\mathsf{pp}, \mathsf{tree}, S, V)$. It holds that*
$$\mathsf{H.VerUpd}(\mathsf{pp}, \mathsf{digest}^{(m)}, S, V, \mathsf{digest}, \tau) = 1.$$

Definition A.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 $n \leq 2^\lambda$,*

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

the probability is over the choice of $\mathsf{pp} \leftarrow \mathsf{H.Gen}(1^\lambda, n)$ *and* $\left(m, \left\{(\mathsf{digest}^{(i)}, S^{(i)}, V^{(i)}, \tau^{(i)})\right\}_{i \in [m]}, \mathsf{digest}^{(0)}, S^{(0)}, V^{(0)}, \pi^{(0)}, S, V, \pi\right) \leftarrow \mathcal{A}_\lambda(\mathsf{pp})$, *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 $i \in \{0, \dots, m\}$ is the largest index with $\ell \in S^{(i)}$.*
- *v^{final} is the value in V at the index of ℓ in S.*

Definition A.4 (Parallel Efficiency). *Let $\beta \colon \mathbb{N} \to \mathbb{N}$. We say that a concurrently updatable hash function satisfies β-*parallel efficiency *if the following hold for all $\lambda, n \in \mathbb{N}$ with $n \leq 2^\lambda$,* pp *in the support of* $\mathsf{H.Gen}(1^\lambda, n)$, *and ordered sets $S \subseteq [n]$:*

- *The algorithms* $\mathsf{H.Open}$, $\mathsf{H.Update}$, $\mathsf{H.VerOpen}$ *and* $\mathsf{H.VerUpd}$ *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* $\mathsf{H.Hash}(\mathsf{pp}, D)$ *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* tree, *and tuple $V \in (\{0,1\}^\lambda)^{|S|}$, define* $(V', \pi, \mathsf{digest}, \tau)$ *as follows:*
  - $(V', \pi) = \mathsf{H.Open}(\mathsf{pp}, \mathsf{tree}, S)$
  - $(\mathsf{digest}, \tau) = \mathsf{H.Update}(\mathsf{pp}, \mathsf{tree}, S, V)$

  There exists an algorithm $\mathsf{OpenUpdate}(\mathsf{pp}, \mathsf{tree}, S, V)$ *which outputs* $(V', \pi, \mathsf{digest}, \tau)$, *such that k sequential calls to* $\mathsf{OpenUpdate}$, *each on at most p_{max} locations, can be computed with $p_{\mathsf{max}} \cdot \beta(\lambda)$ work, which can be decoupled into depth $(k-1) + \beta(\lambda)$ using at most $p_{\mathsf{max}} \cdot \beta(\lambda)$ processors.*

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

We recall the following theorem from [EFKP20b] regarding the existence of a concurrently updatable hash function.

Theorem A.5 ([EFKP20b]). *Assuming the existence of collision-resistant hash function families, there exists a concurrently updatable hash function.*

A.2 Succinct Arguments for Parallel Computations

In this section, we extend the notions of RAM delegation (Definition 3.5) and SPARGs (Definition 3.6) to the setting of parallel computations.

Definition A.6 (RAM Delegation). *A publicly verifiable, succinct RAM Delegation Scheme for $\mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ is a tuple of probabilistic algorithms* $(\mathsf{Del.S}, \mathsf{Del.D}, \mathsf{Del.P}, \mathsf{Del.V})$ *with the following syntax:*

- $(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda)$*: A PPT algorithm that on input a security parameter λ outputs a common reference string* crs *and a digest key* dk*. We assume without loss of generality that* crs *contains* dk*.*

- $\mathsf{rt} = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$*: A deterministic algorithm that on input a digest key* dk *and a RAM configuration* cf *outputs a digest* rt*.*

- $\pi \leftarrow \mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t, p))$*: A probabilistic algorithm that on input a common reference string* crs*, and a statement* $(\mathsf{cf}, \mathsf{cf}', t, p)$*, outputs a proof π.*

- $b \leftarrow \mathsf{Del.V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t, p), \pi)$*: A PPT algorithm that on input a a common reference string* crs*, common reference string* crs*, statement* $(\mathsf{rt}, \mathsf{rt}', t, p)$*, 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 $(\mathsf{cf}, \mathsf{cf}', t, p) \in \mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$ with $t, n \leq 2^\lambda$ where n is the memory size of the configurations, it holds*

$$\Pr\left[\begin{array}{l}(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda)\\ \mathsf{rt} = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})\\ \mathsf{rt}' = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}')\\ \pi \leftarrow \mathsf{Del.P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t, p))\end{array} : \mathcal{V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t, p), \pi) = 1\right] = 1.$$

- **Soundness:** *For any non-uniform polynomial-time algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$, polynomial P, polynomial-time computable function T, and polynomial $\overline{T}$ such that $T(\lambda) \leq \overline{T}(\lambda)$ for all $\lambda \in \mathbb{N}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l}(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda)\\ (\mathsf{cf}, \mathsf{cf}', \mathsf{rt}, \mathsf{rt}', \pi) \leftarrow \mathcal{A}_\lambda(\mathsf{crs}, \mathsf{dk})\end{array} : \begin{array}{l}\mathcal{V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t, p), \pi) = 1\\ \wedge\ (\mathsf{cf}, \mathsf{cf}', t, p) \in \mathcal{L}^{\mathsf{del}}_{\mathsf{par}}\\ \wedge\ \mathsf{rt} = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})\\ \wedge\ \mathsf{rt}' \neq \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}')\end{array}\right] \leq \mathsf{negl}(\lambda),$$

  where $t = T(\lambda)$ and $p = P(\lambda)$.

- **Collision resistance:** *For any non-uniform polynomial-time algorithm $\mathcal{A} = \{\mathcal{A}_\lambda\}_{\lambda\in\mathbb{N}}$, there exists a negligible function* negl *such that for every $\lambda \in \mathbb{N}$, it holds that*

$$\Pr\left[\begin{array}{l}(\mathsf{crs}, \mathsf{dk}) \leftarrow \mathsf{Del.S}(1^\lambda)\\ (\mathsf{cf}, \mathsf{cf}') \leftarrow \mathcal{A}_\lambda(\mathsf{crs}, \mathsf{dk})\end{array} : \begin{array}{l}\mathsf{cf} \neq \mathsf{cf}'\\ \wedge\ \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}) = \mathsf{Del.D}(\mathsf{dk}, \mathsf{cf}')\end{array}\right] \leq \mathsf{negl}(\lambda).$$

- **Succinctness:** *There exist polynomials q_1, q_2, q_3 such that for any $\lambda \in \mathbb{N}$,* $(\mathsf{crs}, \mathsf{dk})$ *in the support of* $\mathsf{Del.S}(1^\lambda)$*, $(\mathsf{cf}, \mathsf{cf}', t, p) \in \mathcal{L}^{\mathsf{del}}_{\mathsf{par}}$, and proof π in the support of $\mathcal{P}(\mathsf{crs}, (\mathsf{cf}, \mathsf{cf}', t, p))$, it holds that*

- $|\mathsf{Del.V}(\mathsf{crs}, (\mathsf{rt}, \mathsf{rt}', t, p), \pi)| \leq q_1(\lambda, \log(t \cdot p))$ *and*
- $|\pi| \leq q_2(\lambda, \log(t \cdot p))$.
- $\mathsf{Del.D}(\mathsf{dk}, \mathsf{cf})$ *is computable in time* $|\mathsf{cf}| \cdot q_3(\lambda)$ *and has output length length* λ.

Definition A.7 (Non-interactive SPARGs for P)**.** *A* Non-interactive Succinct Parallelizable Argument *for a language* $\mathcal{L} \subseteq \mathcal{L}^{\mathcal{U}}$ *is a tuple of probabilistic algorithms* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *with the following syntax:*

- $\mathsf{crs} \leftarrow \mathcal{G}(1^\lambda)$*: A PPT algorithm that on input a security parameter* λ *outputs a common reference string* crs.

- $(y, \pi) \leftarrow \mathcal{P}(\mathsf{crs}, (M, x, L, t, p))$*: A probabilistic algorithm that on input a common reference string* crs*, and a statement* (M, x, L, t, p)*, outputs a value* y *and a proof* π.

- $b \leftarrow \mathcal{V}(\mathsf{crs}, (M, x, y, L, t, p), \pi)$*: A PPT algorithm that on input a common reference string* crs*, a statement* (M, x, y, L, t, p)*, 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* $(M, x, y, L, t, p) \in \mathcal{L}$ *where* M *has access to* $n \leq 2^\lambda$ *words in memory and* $t \leq 2^\lambda$,

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

- **Soundness for P:** *For all non-uniform polynomial-time provers* $\mathcal{P}^\star = \{\mathcal{P}^\star_\lambda\}_{\lambda \in \mathbb{N}}$ *and polynomials* T, P*, there is a negligible function* negl *such that for every* $\lambda \in \mathbb{N}$*, it holds that*

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{G}(1^\lambda) \\ ((M, x, y, L), \pi) \leftarrow \mathcal{P}^\star_\lambda(\mathsf{crs}) \end{array} : \begin{array}{l} \mathcal{V}(\mathsf{crs}, (M, x, y, L, t, p), \pi) = 1 \\ \wedge\ (M, x, y, L, t, p) \notin \mathcal{L} \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where $t = T(\lambda)$ *and* $p = P(\lambda)$.

- **Succinctness:** *There exist polynomials* q_1, q_2 *such that for any* $\lambda \in \mathbb{N}$, crs *in the support of* $\mathcal{G}(1^\lambda)$, $(M, x, L, t, p) \in \mathcal{L}$ *where* M *uses* $n \leq 2^\lambda$ *words in memory,* $t, p \leq 2^\lambda$*, and* (y, π) *in the support of* $\mathcal{P}(\mathsf{crs}, (M, x, L, t, p))$*, it holds that*

  - $\mathsf{work}_{\mathcal{V}}(\mathsf{crs}, (M, x, y, L, t, p), \pi) \leq q_1(\lambda, |(M, x)|, L, \log(t \cdot p))$,
  - $|y| \leq L$, *and*
  - $|\pi| \leq q_2(\lambda, L, \log(t \cdot p))$.

- **Optimal prover depth:** *There exists polynomials* q_1 *and* q_2 *such that for all* $\lambda \in \mathbb{N}$ *and* $(M, x, y, L, t, p) \in \mathcal{L}$ *where* M *has access to* $n \leq 2^\lambda$ *words in memory and* $t, p \leq 2^\lambda$*, it holds that*

$$\mathsf{depth}_{\mathcal{P}}(\mathsf{crs}, (M, x, L, t, p)) = t + q_1(\lambda, |(M, x)|, L, \log(t \cdot p))$$

and the total number of processors used by $\mathcal{P}$ *is in* $p \cdot q_2(\lambda, \log(t \cdot p))$.

If the above holds for $\mathcal{L} = \mathcal{L}^{\mathcal{U}}$*, we say that* $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ *is a* non-interactive SPARG for polynomial-time PRAM computation.