Parallelizable Delegation from LWE

Cody Freitag[1](✉), Rafael Pass[1,2], and Naomi Sirkin[1]

[1] Cornell Tech, New York, NY 10044, USA
{cfreitag,rafael,nephraim}@cs.cornell.edu
[2] Tel Aviv University, Tel Aviv, Israel

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).

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 [34], 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

E. Kiltz and V. Vaikuntanathan (Eds.): TCC 2022, LNCS 13748, pp. 623–652, 2022.
https://doi.org/10.1007/978-3-031-22365-5_22

proof system is *succinct* [40,42], 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., [8,10,15,22,43]).

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 [32]. However, the beautiful recent works of Kalai, Paneth, and Yang [38] and Choudhuri, Jain, Jin [22] 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 [22].

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., [16–18,23,35,48] 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 [29] 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 [29].

In this work, we resolve the above-mentioned problem, constructing the first non-interactive delegation schemes where the prover has $t + \mathrm{poly}(\lambda, \log t)$ parallel running time using $\mathrm{poly}(\lambda, \log t)$ 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 *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) [19] 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 [29], 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 [22]. 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 [22].

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 [22], 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 [22] with the updatable hash tree from [29] to get an updatable RAM delegation from LWE with the desired efficiency.

Theorem 1.2 (Efficient Updatable RAM Delegation; Informal (see Theorem *4.2*)). *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 [29] 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 [29] 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). *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 [29] 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). *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 $t + \text{poly}(\lambda, \log(t))$. 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). *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 [29] was to constructing verifiable delay functions [19]. 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 [19,21,30] for more details).

[29] 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 [29] 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 [29] with just polynomial hardness of LWE.

Theorem 1.7 (VDFs from LWE and any sequential function; Informal (see Corollary *3*)). *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 [19], (b) sequential functions and SNARKs for NP [29], or (c) sub-exponential LWE assumption and the sequentiality of repeated squaring in a group of unknown order [41], or various construction in the random oracle model [28,44,47]. We emphasize that in terms of practical efficiency, our construction does not compete with constructions in the ROM (such as [44,47]), but our goal here is simply to show that VDFs as a primitive can be based on standard hardness assumptions.

As pointed out in [29], 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

[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.

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., [1–4,25–27] 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 [29] 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 [33] and Reingold, Rothblum, and Rothblum [45] 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 [39] construct *privately verifiable* delegation for any time and space Turing machines based on the quasi-polynomial hardness of LWE. Kalai and Paneth [37] 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 [20]. Holmgren and Rothblum [35] show how to implement the approach of [39] 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 [38] 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 [36] show how to achieve publicly verifiable delegation for bounded depth computation from subexponential hardness of LWE. Finally, Choudhuri, Jain, and Jin [22] construct publicly verifable RAM delegation from polynomial hardness of LWE.

We note that implicit in the works of [20,22,37,38], building off the techniques of [39], 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 [40] (in the interactive setting) and Micali [42] (in the random oracle model using the Fiat-Shamir heuristic [31]). Originally PCP constructions required polynomial length and prover running time [5,6]. Ben-Sasson and Sudan [14] 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 $t \cdot \text{polylog}(t)$. Subsequent work by [11] give a highly parallelizable PCP that can be computed in parallel time polylog(t) with t processors, *after computing the computation tableau*. Interactive oracle proofs (IOPs) are a multi-round generalization of PCPs, introduced in [45] and [13], that are also useful for delegation protocols. There is a fruitful line of work [9,12,46] 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 [19] and [24] 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 [29] 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.

1.3 Organization

In Sect. 2, we give an overview of our SPARG constructions. Then, in Sect. 3, we give preliminaries. Next, in Sect. 4, we give our construction of updatable RAM delegation with quasilinear overhead from LWE. Then, in Sect. 5, we give our construction of SPARGs from updatable delegation. Our VDF construction is given in Sect. 6. Our construction of time-tight SPARGs, and that of SPARGs for parallel computations, are deferred to the full version.

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 [29]. 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 [29], 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 $t + \operatorname{polylog} t$. 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 [19,24]) 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 $k \cdot \text{poly}(\lambda)$, 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 $k \cdot \text{poly}(\lambda, \log k)$.

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 [29] 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 [22,37,38]) 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 [22], 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 [22], 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 [29]), 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 [22] scheme satisfies this notion of upatability when instantiated with the hash tree due to [29], 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 [29] 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 $\text{poly}(p)$ 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 [22]. 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 $\text{poly}(\lambda, \log p)$ 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 $k \cdot \text{poly}(\lambda, \log p)$. 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 $T \cdot \text{poly}(\lambda, \log T)$. As a final observation, we show that for any number of processors p, the RAM delegation prover can be made to run in time $T/p \cdot \text{poly}(\lambda, \log T)$ 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 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 Sect. 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 (Fig. 1).

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.

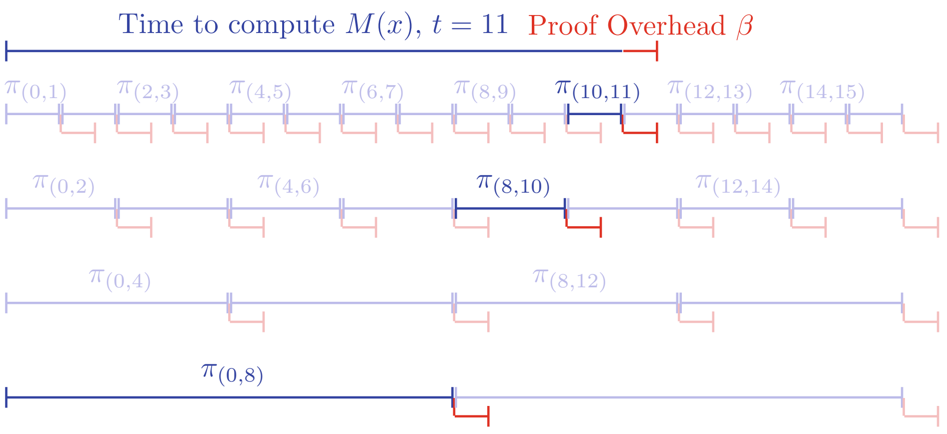

Fig. 1. An example of the time-independent SPARG prover for a computation $M(x)$ that takes $t = 11$ steps. A proof $\pi_{(a,b)}$ corresponds to a RAM delegation proof that M on input x starts at configuration cf_a and ends at configuration cf_b. The horizontal axis represents parallel time, and the 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.

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 $(0, 1)$, $(1, 2)$, $(2, 3)$, $\dots$, $(t-1, t)$ 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

In this section, we include the relevant preliminaries. Additional preliminaries, including definitions of verifiable delay functions, concurrently updatable hash functions, and succinct arguments for parallel computations are deferred to the full version.

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.

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 [7].

Definition 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 $|y| \le L$.*

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

Definition 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.*

3.3 RAM Delegation

In this section, we define RAM delegation, which will be the main building block for our SPARG construction. Following [20,22,37,38], 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, [22,38] 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 Sect. 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 (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*

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

  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* λ.

3.4 SPARGs

In this section, we define SPARGs for P based on the notion of SPARKs introduced in [29]. 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 4 (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* $t \leq 2^\lambda$,

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

- **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*

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

  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.

4.1 The CJJ Delegation Scheme

Our starting point will be the recent delegation scheme due to Choudhuri, Jain, and Jin [22], henceforth referred to as the CJJ construction. We start by giving an overview. 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 sublinearly on k.

At a high level, their construction follows an approach in recent works (see, e.g., [29,37,38]) 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.

In the full version, we discuss the differences between the notion of RAM delegation satisfied by this construction, and Definition 3, and show that the CJJ scheme satisfies our notion of RAM delegation. As their scheme is based on LWE, the following holds.

Theorem 4.1 [22]**.** *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 [22] 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 [22] 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 [22], 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 $\ell \cdot \text{poly}(\lambda, \log N) \in \text{poly}(\lambda, \log t)$. All together, this shows that the prover runs in time

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

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 $(\mathsf{cf}_0, \mathsf{cf}_1, k_1)$, $(\mathsf{cf}_1, \mathsf{cf}_2, k_2)$, $\ldots$, $(\mathsf{cf}_{m-1}, \mathsf{cf}_m, k_m)$, 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 [29], 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 $|\mathsf{cf}_0| \cdot \text{poly}(\lambda)$.
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 $\text{poly}(\lambda)$ 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 $k + \text{poly}(\lambda)$.

The requirements for the hash tree of [22] (which is based on [37]) are satisfied by that of [29] (see [29] for a more in-depth discussion and comparison between various definitions). Therefore, we observe that the CJJ construction satisfies the following notion.

Definition 5. *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 6. *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 [29], 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 4.2. *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* $\alpha(\lambda, t) \leq t \cdot \mathrm{poly}(\lambda, \log t)$.

The proof of Theorem 4.2 is deferred to the full version.

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. In this full version, we formally define the local opening property by giving additional algorithms $(\mathsf{Del.Open}, \mathsf{Del.VerOpen})$ to the updatable delegation scheme to capture this notion. We also show that our updatable RAM delegation scheme satisfies local opening (by relying on the local opening property of the [29] hash tree), and therefore get the following corollary to Theorem 4.2.

Corollary 1. *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 Sect. 3.3). 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.

```
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 $\mathsf{cf}_0$ 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 $\gamma$ as in the parameters paragraph. Initialize $T := t$ to be
       the number of steps remaining in the computation.
    3. For $i = 1, 2, \ldots$, 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 $\mathsf{cf}_{i-1}$. Let $\mathsf{cf}_i$
            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 com-
            pute $\tau_i \leftarrow \mathsf{Del.P}(\mathsf{crs}, (M, \mathsf{cf}_{i-1}, \mathsf{cf}_i, k_i), w_i)$.
    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 $\vec{\mathsf{rt}} = (\mathsf{rt}_1, \ldots, \mathsf{rt}_m)$, $\vec{\tau} = (\tau_1, \ldots, \tau_m)$, and $\vec{k} = (k_1, \ldots, k_m)$. Output
       $(y, \pi)$ 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 $\mathsf{cf}_0$ be the initial configuration of $M(x)$ and compute $\mathsf{rt}_0$ 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) $\mathsf{st}$ is a halting state, and $|y| \leq L$.
```

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

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 Fig. 2, 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* 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 1, 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 1, we get the following corollary.

Corollary 2. *Assuming the hardness of LWE, there exists a SPARG for* $\mathcal{L}^{\mathcal{U}}$.

The proof of Theorem 5.1 is deferred to the full version.

6 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 6.1. *Assuming the existence of a SPARG for* $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ *and a sequential function, there exists a VDF.*

Here, $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ is the notion of $\mathcal{L}^{\mathcal{U}}$ extended to parallel computations (this is defined formally in the full version). In the full version, we show that a SPARG for $\mathcal{L}^{\mathcal{U}}_{\mathsf{par}}$ can be based on LWE, which gives the following.

Corollary 3. *Assuming the hardness of LWE and a sequential function, there exists a VDF.*

The proof of Theorem 6.1 is deferred to the full version.

Acknowledgements. This work was supported in part by NSF CNS-2149305, CNS-2128519, 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

1. Alwen, J., Blocki, J., Pietrzak, K.: Depth-robust graphs and their cumulative memory complexity. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10212, pp. 3–32. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56617-7_1
2. Alwen, J., Blocki, J., Pietrzak, K.: Sustained space complexity. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 99–130. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_4
3. Alwen, J., Chen, B., Kamath, C., Kolmogorov, V., Pietrzak, K., Tessaro, S.: On the complexity of scrypt and proofs of space in the parallel random oracle model. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 358–387. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_13
4. Alwen, J., Serbinenko, V.: High parallel complexity graphs and memory-hard functions. In: STOC, pp. 595–603. ACM (2015)
5. Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. J. ACM **45**(3), 501–555 (1998)
6. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: STOC, pp. 21–31. ACM (1991)
7. Barak, B., Goldreich, O.: Universal arguments and their applications. SIAM J. Comput. **38**(5), 1661–1694 (2008)
8. Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable zero knowledge with no trusted setup. In: Boldyreva, A., Micciancio, D. (eds.) CRYPTO 2019. LNCS, vol. 11694, pp. 701–732. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-26954-8_23
9. Ben-Sasson, E., Chiesa, A., Gabizon, A., Riabzev, M., Spooner, N.: Interactive oracle proofs with constant rate and query complexity. In: ICALP. LIPIcs, vol. 80, pp. 40:1–40:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
10. Ben-Sasson, E., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society (2014)

11. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete efficiency of probabilistically-checkable proofs. In: STOC, pp. 585–594. ACM (2013)
12. Ben-Sasson, E., Chiesa, A., Goldberg, L., Gur, T., Riabzev, M., Spooner, N.: Linear-size constant-query IOPs for delegating computation. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019. LNCS, vol. 11892, pp. 494–521. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-36033-7_19
13. Ben-Sasson, E., Chiesa, A., Spooner, N.: Interactive oracle proofs. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9986, pp. 31–60. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53644-5_2
14. Ben-Sasson, E., Sudan, M.: Short PCPS with polylog query complexity. SIAM J. Comput. **38**(2), 551–607 (2008)
15. Bitansky, N., et al.: The hunting of the SNARK. J. Cryptol. **30**(4), 989–1066 (2017)
16. Bitansky, N., Chiesa, A.: Succinct arguments from multi-prover interactive proofs and their efficiency benefits. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 255–272. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_16
17. Block, A.R., Holmgren, J., Rosen, A., Rothblum, R.D., Soni, P.: Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12551, pp. 168–197. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64378-2_7
18. Block, A.R., Holmgren, J., Rosen, A., Rothblum, R.D., Soni, P.: Time- and space-efficient arguments from groups of unknown order. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12828, pp. 123–152. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84259-8_5
19. Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10991, pp. 757–788. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_25
20. Brakerski, Z., Holmgren, J., Kalai, Y.T.: Non-interactive delegation and batch NP verification from standard computational assumptions. In: STOC, pp. 474–482. ACM (2017)
21. Chia network. https://chia.net/. Accessed 17 May 2019
22. Choudhuri, A.R., Jain, A., Jin, Z.: Snargs for $\mathcal{P}$ from LWE. In: FOCS, pp. 68–79. IEEE (2021)
23. Costello, C., et al.: Geppetto: versatile verifiable computation. In: IEEE Symposium on Security and Privacy, pp. 253–270. IEEE Computer Society (2015)
24. Döttling, N., Garg, S., Malavolta, G., Vasudevan, P.N.: Tight verifiable delay functions. In: Galdi, C., Kolesnikov, V. (eds.) SCN 2020. LNCS, vol. 12238, pp. 65–84. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57990-6_4
25. Dryja, T., Liu, Q.C., Park, S.: Static-memory-hard functions, and modeling the cost of space vs. time. In: Beimel, A., Dziembowski, S. (eds.) TCC 2018. LNCS, vol. 11239, pp. 33–66. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03807-6_2
26. Dwork, C., Goldberg, A., Naor, M.: On memory-bound functions for fighting spam. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 426–444. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_25
27. Dwork, C., Naor, M., Wee, H.: Pebbling and proofs of work. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 37–54. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_3
28. Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: Continuous verifiable delay functions. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12107, pp. 125–154. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45727-3_5

29. Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: SPARKs: succinct parallelizable arguments of knowledge. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 707–737. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_25
30. Ethereum foundation. https://www.ethereum.org/. Accessed 17 May 2019
31. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
32. Gentry, C., Wichs, D.: Separating succinct non-interactive arguments from all falsifiable assumptions. In: STOC, pp. 99–108. ACM (2011)
33. Goldwasser, S., Kalai, Y.T., Rothblum, G.N.: Delegating computation: interactive proofs for muggles. J. ACM **62**(4), 27:1–27:64 (2015)
34. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM J. Comput. **18**(1), 186–208 (1989)
35. Holmgren, J., Rothblum, R.: Delegating computations with (almost) minimal time and space overhead. In: FOCS, pp. 124–135. IEEE Computer Society (2018)
36. Jawale, R., Kalai, Y.T., Khurana, D., Zhang, R.: Snargs for bounded depth computations and PPAD hardness from sub-exponential LWE. In: STOC, pp. 708–721. ACM (2021)
37. Kalai, Y., Paneth, O.: Delegating RAM computations. In: Hirt, M., Smith, A. (eds.) TCC 2016. LNCS, vol. 9986, pp. 91–118. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53644-5_4
38. Kalai, Y.T., Paneth, O., Yang, L.: How to delegate computations publicly. In: STOC, pp. 1115–1124. ACM (2019)
39. Kalai, Y.T., Raz, R., Rothblum, R.D.: How to delegate computations: the power of no-signaling proofs. In: STOC, pp. 485–494. ACM (2014)
40. Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: STOC, pp. 723–732. ACM (1992)
41. Lombardi, A., Vaikuntanathan, V.: Fiat-Shamir for repeated squaring with applications to PPAD-hardness and VDFs. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020. LNCS, vol. 12172, pp. 632–651. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-56877-1_22
42. Micali, S.: Computationally sound proofs. SIAM J. Comput. **30**(4), 1253–1298 (2000)
43. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society (2013)
44. Pietrzak, K.: Simple verifiable delay functions. In: ITCS. LIPIcs, vol. 124, pp. 60:1–60:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
45. Reingold, O., Rothblum, G.N., Rothblum, R.D.: Constant-round interactive proofs for delegating computation. SIAM J. Comput. **50**(3) (2021)
46. Ron-Zewi, N., Rothblum, R.D.: Local proofs approaching the witness length [extended abstract]. In: FOCS, pp. 846–857. IEEE (2020)
47. Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 379–407. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_13
48. Wu, H., Zheng, W., Chiesa, A., Popa, R.A., Stoica, I.: DIZK: a distributed zero knowledge proof system. In: USENIX Security Symposium, pp. 675–692. USENIX Association (2018)