Delegating RAM Computations

Yael Kalai[1(✉)] and Omer Paneth[2]

[1] Microsoft Research, Redmond, USA
yael@microsoft.com
[2] Boston University, Boston, USA

Abstract. In the setting of cloud computing a user wishes to delegate its data, as well as computations over this data, to a cloud provider. Each computation may read and modify the data, and these modifications should persist between computations. Minding the computational resources of the cloud, delegated computations are modeled as RAM programs. In particular, the delegated computations' running time may be sub-linear, or even exponentially smaller than the memory size.

We construct a two-message protocol for delegating RAM computations to an untrusted cloud. In our protocol, the user saves a short digest of the delegated data. For every delegated computation, the cloud returns, in addition to the computation's output, the digest of the modified data, and a proof that the output and digest were computed correctly. When delegating a T-time RAM computation M with security parameter k, the cloud runs in time $\mathrm{poly}(\mathsf{T}, k)$ and the user in time $\mathrm{poly}(|M|, \log \mathsf{T}, k)$.

Our protocol is secure assuming super-polynomial hardness of the Learning with Error (LWE) assumption. Security holds even when the delegated computations are chosen adaptively as a function of the data and output of previous computations.

We note that RAM delegation schemes are an improved variant of memory delegation schemes [Chung et al. CRYPTO 2011]. In memory delegation, computations are modeled as Turing machines, and therefore, the cloud's work always grows with the size of the delegated data.

1 Introduction

In recent years, with the growing popularity of cloud computing platforms, more and more users store data and run computations on the cloud. This raises many concerns. As cryptographers, our first concern is that of secrecy: users may wish to hide their confidential data and computations from the cloud. But perhaps a more fundamental concern is that of *integrity*: ensuring that the cloud is doing what it is supposed to do. In this paper we focus on the latter.

We ask the following question: how can a cloud provider convince a user that a delegated computation was performed correctly? We believe that the adoption of

O. Paneth—Supported by the Simons award for graduate students in theoretical computer science and an NSF Algorithmic foundations grant 1218461.

M. Hirt and A. Smith (Eds.): TCC 2016-B, Part II, LNCS 9986, pp. 91–118, 2016.
DOI: 10.1007/978-3-662-53644-5_4

cloud computing services depends on the existence of such mechanisms. Indeed, even if not every computation is explicitly checked, the mere ability to check computations may be desirable.

RAM Delegation. We model the above problem as follows. Initially the user owns some memory D containing the data it wishes to delegate. In order to verify the correctness of future computations over this memory, the user must save some short digest of the memory D. We therefore allow the user to pre-process the memory once, before delegating it, and compute a digest d. We also allow the cloud to pre-process the memory before storing it. During this pre-processing the cloud can compute auxiliary information that will be stored together with the memory and used to construct proofs efficiently.

To compute on the memory, the user specifies a program M and sends its description to the cloud. We model the program M as a RAM program. We believe that this is the most realistic choice when the outsourced memory is very large and the computation may not access it all.[1] The cloud sends back to the user the output y of the program M when executed on the memory D. The user can delegate multiple computations sequentially where each computation may modify the memory. We require that the state of the memory persists between computations. Therefore, after every computation, the cloud sends back to the user, together with the output y, the new digest $\mathsf{d}_{\mathsf{new}}$ corresponding to the new digest of the memory.

The cloud also provides a proof that the output y and the new digest $\mathsf{d}_{\mathsf{new}}$ are correct with respect to the program M and the digest d of the original memory. We require that this proof proceeds in two messages, namely, together with the program M, the user sends a challange ch, and together with y and $\mathsf{d}_{\mathsf{new}}$, the cloud sends a proof pf. Thus, the proof of correctness does not require additional rounds of interaction. We refer to such a protocol as a *two-message delegation scheme for RAM computations.*

1.1 Our Results

We construct a two-message delegation scheme for RAM computations based on the Learning with Errors (LWE) assumption.

Efficiency. For security parameter k and for initial memory of size n such that $n < 2^k$, the user's and the cloud's pre-processing time is $n \cdot \mathrm{poly}(k)$, and the digest is of size $\mathrm{poly}(k)$. If the running time of the delegated RAM program is T (we assume that $\mathsf{T} < 2^k$), then the running time of the cloud is $\mathsf{T}^3 \cdot \mathrm{poly}(k)$. The communication complexity of the proof, and the time it takes the user to generate a challenge and verify a proof are $\mathrm{poly}(k)$, and are independent of the computation time.

[1] For example, consider the setting where the user wishes to simply retrieve an element from the outsourced database D. We would like the runtime of the user in this case to be proportional to $\log |D|$, as opposed to proportional to D.

Adaptive Soundness. The soundness of our scheme holds even if the adversary (acting as the cloud) can choose the program to be delegated adaptively depending on the memory and on the outcome of previously delegated computations. This feature is especially important in applications where the pre-processing step is performed once and then used and reused to delegate many computations over time. We emphasize that our protocol may not be sound if the adversary chooses the program adaptively depending on the user's challenge ch.

Public Pre-processing. In a two-message delegation scheme for RAM computations the user must pre-process the memory before delegating it. In our scheme the pre-processing step is public – it does not require any secret randomness. In particular, the user is not required to keep any secret state between computations. This feature also allows a single execution of the pre-processing step to serve multiple users, as long as they all trust the generated memory digest.[2]

Security with Adversarial Digest. We prove that our scheme is sound even in the setting where the pre-processing step is executed by an untrusted party. In this setting honest users cannot be sure that the digest they hold corresponds to some "correct" memory, or even that it is indeed the digest of any memory string. The soundness we require is that an adversary cannot prove that the same computation with the same digest leads to two different outcomes. We note that soundness for digests that are honestly computed follows from this stronger formulation.

Efficient Pre-processing. Another feature of our scheme is that the efficiency of the pre-processing step only depends on the *initial* memory size and does not depend on the amount of memory required to execute future computations. In particular, if there is no initial memory to delegate, the pre-processing step can be skipped.[3]

Informal Theorem 1.1. *There exists a two-message delegation scheme for RAM computations, with efficiency, adaptive soundness and public pre-processing, as described above, assuming the existence of a collision resistant hash family that is sub-exponentially secure and assuming that the LWE problem (with security parameter k) is hard to break in time quasi-polynomial in T, where T is an upper bound on the running time of the delegated computations.*

We note that the existence of a sub-exponentially secure collision resistant hash family follows from the sub-exponential hardness of the LWE problem.

[2] When different users delegate different computations that may *change* the memory, there should be an external mechanism to synchronize these computations, and make sure that every computation is verified with respect to the most recent digest of the memory.

[3] In fact, we can always replace the pre-processing step with an initial delegation round where the user delegates a program that initializes the memory.

On the Necessity of Cryptographic Assumptions. Since the user does not store its memory locally, and only stores a short digest, we cannot hope to get information-theoretic soundness. An all powerful malicious cloud can always cheat by finding a fake memory D' with the same digest as the original memory, and perform computations using the fake memory. Therefore, the soundness of our scheme must rely on some hardness assumption (such as the hardness of finding digest collisions).

On Delegation with Secrecy. Our delegation protocol does not achieve secrecy. That is, it does not hide the user's data and computations from the cloud. One method for achieving secrecy is to execute the entire delegation protocol under fully-homomorphic encryption. However, this method is not applicable when delegating RAM computations, since it increases the cloud's running time proportionally to the size of the entire memory.

1.2 Previous Work

We compare our result with previous results on delegating computation in various models based on various computational assumptions.

Delegating Non-deterministic Computations. Previous works constructed delegation schemes for non-deterministic computations in the random oracle model or based on strong "knowledge" assumptions. As we observe in this work (see Sect. 1.3), any delegation scheme for non-deterministic computations, combined with a collision-resistant hash function, can be used to delegate RAM computations.

The Random Oracle Model. Based on the interactive arguments of Kilian [Kil92], Micali [Mic94] gave the first construction of a non-interactive delegation scheme in the random oracle model. Micali's scheme supports non-deterministic computations and can therefore be used to delegate RAM computations assuming also the existence of a collision-resistant hash family.[4] The main advantage of Micali's scheme over the scheme presented in this work is that it is completely non-interactive (it requires one message rather than two). In particular, Micali's scheme is also *publicly verifiable*. However, our scheme can be proven secure in the standard model based on standard cryptographic assumptions.

Knowledge Assumptions. In a sequence of recent works, non-interactive (one message) delegation schemes in the *common reference string* (CRS) model, were constructs based on strong and non-standard "knowledge" assumptions such as variants of the Knowledge of Exponent assumption [Gro10, Lip12, DFH12, GGPR13,

[4] The solution described in Sect. 1.3 makes non-black-box use of the collision-resistant hash function, and therefore we cannot replace the hash function with the random oracle.

BCI+13, BCCT13, BCC+14]. These schemes support non-deterministic computations and can therefore be used to delegate RAM computations. Some of the above schemes are also publicly verifiable (the user does not need any secret trapdoor on the CRS). The main advantage of our scheme is that it can be based on standard cryptographic assumptions.

Indistinguishability Obfuscation. Several recent results construct non-interactive (one message) delegation schemes for RAM computations in the CRS model based on indistinguishability obfuscation [GHRW14, BGL+15, CHJV15, CH15, CCC+15]. Next we compare our scheme to the obfuscation based schemes.

The advantage of their schemes is that they achieve secrecy. In fact, they construct stronger objects such as garbling and obfuscation schemes for RAM computations. In addition, their schemes are publicly verifiable. The advantages of our scheme, compared to the obfuscation based schemes, are the following:

Assumptions. Our scheme is based on the hardness of the LWE problem – a standard and well studied cryptographic assumption. In particular, the LWE problem is known to be as hard as certain worst-case lattice problems.

Adaptivity. In our scheme security holds even against an adaptive adversary that chooses the delegated computations as a function of the delegated memory. In contrast, the obfuscation based schemes only have static security. That is, in the analysis all future delegated computations must be fixed before the memory is delegated. We note that using complexity leveraging and sub-exponential hardness assumptions it is possible to prove that obfuscation based schemes are secure against a *bounded* number of adaptively chosen computations, where the bound on the number of computations depends on the size of the CRS.

Security with adversarial digest. In our scheme the pre-processing step is public and soundness holds even in the setting where the pre-processing step is executed by an untrusted party. In the obfuscation based schemes however, the pre-processing step requires private randomness and if it is not carried out honestly the cloud may be able to prove arbitrary statements.

Following our work, Canetti et al. [CCHR15] and Ananth et al. [ACC+15] gave a delegation scheme for RAM computations from indistinguishability obfuscation that satisfies the same notion of adaptivity as our scheme. These constructions do not have public digest and they are not secure with adversarial digest.

Learning with Errors. We review existing delegation protocols based on the hardness of the LWE problem. These protocols are less efficient than our delegation protocols for RAM computations.

Deterministic Turing Machine delegation. The work of [KRR14] gives a two-message delegation scheme for deterministic Turing machine computations based on the quasi-polynomial hardness of the LWE problem. The main differences between delegation of RAM computations and delegation of deterministic Turing machine computations are as follows:

1. In deterministic Turing machine delegation, the user needs to save the entire memory (thought of as the input to the computation), while in RAM delegation, the user only needs to save a short digest of the memory.
2. In deterministic Turing machine delegation, the cloud's running time depends on the running time of the computation when described as a Turing machine, rather than a RAM program. In particular, the cloud's running time always grows with the memory size, even if the delegated computation does not access the entire memory.

We mention that our scheme has better asymptotic efficiency than the scheme of [KRR14] even for Turing machine computations. For delegated computations running in time T and space S the cloud's running time in our scheme is $\mathsf{T}^3 \cdot \mathrm{poly}(k)$ instead of $(\mathsf{T} \cdot S)^3 \cdot \mathrm{poly}(k)$ as in [KRR14].

Memory Delegation. As mentioned in [KRR14], the techniques of Chung et al. [CKLR11] can be used to convert the [KRR14] scheme into a memory delegation scheme that overcomes the first difference above, but not the second one.

Fully-Homomorphic Signatures. The work of Gorbunov et al. [GVW15] on fully-homomorphic signatures gives a non-interactive, publicly verifiable protocol in the CRS model, overcoming both differences above. However, while their protocol has small communication, the user's work is still proportional to computation's running time. Additionally, their protocol does not support computations that write to the memory.

Proofs of Proximity. Finally, we mention a recent line of works on proofs of proximity [RVW13,GR15,KR15,GGR15]. These proofs can be verified much faster than the size of the memory, however, unlike in RAM delegation, in their model the user does not get to pre-process the memory. Instead the user has oracle access to the memory during proof verification. In proofs of proximity the user is only convinced that the computation output is consistent with some memory that is close to the real memory. Additionally, in proofs of proximity the verification takes time at least $\Omega(\sqrt{n})$ where n is the memory size [KR15].

1.3 Technical Overview

We start with a high level description of our scheme.

Pre-processing. In the pre-processing step, the user computes a hash-tree [Mer87] over the memory D and saves the root of this tree as the digest d. The cloud also pre-processes the delegated memory D by computing the same hash-tree and stores the entire tree. The hash-tree allows the cloud to efficiently access the memory in an "authenticated" way. Specifically, the cloud performs the following operations:

1. Read a bit from memory.
2. Write a bit to memory, update the hash tree, and obtain a new digest.

The cloud can then compute a short certificate (in the form of an authenticated path), authenticating the value of the bit read or the value of the updated digest. The time required to access the memory and compute the certificate depends only logarithmically on the memory size.

Emulated Computations and their Transcript. When the user delegates a computation given by a RAM program M, the cloud starts by emulating the execution of M on the memory D as described in [BEG+91]: whenever M accesses the memory, the cloud performs an authenticated memory access via the hash tree. When the emulation of M terminates, the cloud obtains the program output y and the updated memory digest $\mathsf{d}_{\mathsf{new}}$. The cloud also compiles a *transcript* of the memory accessed during the computation. This transcript contains an ordered list of M's memory accesses. For every memory access, the transcript contains the memory location, the bit that was read or written, the new memory digest (in case the memory changed), and the certificate of authenticity. This transcript allows to "re-execute" the computation of the program M and obtain y and $\mathsf{d}_{\mathsf{new}}$, without accessing the memory D directly. Moreover, it is computationally hard to find a valid transcript (containing only valid certificates) that yields the wrong output or digest $(y', \mathsf{d}'_{\mathsf{new}}) \neq (y, \mathsf{d}_{\mathsf{new}})$. For security parameter k and a RAM program M executing in time $\mathsf{T} \leq 2^k$, the time to generate the transcript and to re-execute the program based on the transcript is $\mathsf{T} \cdot \mathrm{poly}(k)$.

Proof of Correctness. After emulating the execution of M, the cloud sends the output y and the new digest $\mathsf{d}_{\mathsf{new}}$ to the user. The cloud also proves to the user that it knows a valid computation transcript which is consistent with y and $\mathsf{d}_{\mathsf{new}}$. More formally, we consider a non-deterministic Turing machine TVer that accepts an input tuple $(M, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}})$ if and only if there exists a valid transcript Trans with respect to d such that the emulation of the program M with Trans produces the output y and the digest $\mathsf{d}_{\mathsf{new}}$.

Proving knowledge of a witness Trans that makes TVer accept $(M, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}})$ requires a delegation scheme supporting *non-deterministic computations*. The problem with this approach is that currently, two-message delegation schemes for non-deterministic computations are only known in the random oracle model or based on strong knowledge assumptions (see Sect. 1.2). However, it turns out that for the specific computation TVer, we can construct a two-message delegation scheme based on standard cryptographic assumptions.

Re-purposing the KRR Proof System. Our solution is based on the delegation scheme of Kalai et al. [KRR14]. While in general, their proof system only supports deterministic computations, we extend their security proof so it also applies to non-deterministic computations of a certain form.

We start with a brief overview of the [KRR14] proof system and explain why it does not support general non-deterministic computations. Then we describe the extended security proof and the type of non-deterministic computations it does support.

The [KRR14] proof system can be used to prove that a deterministic Turing machine M is accepting. The soundness proof of [KRR14] has two steps. In the first step M is translated into a 3-SAT formula ϕ that is satisfiable if and only if M is accepting. The analysis of [KRR14] shows that if the cloud convinces the user to accept, then the formula ϕ satisfies a relaxed notion of satisfiability called *local satisfiability* (See [KRR14, Lemma 7.29]). In the second step, the specific structure of the formula ϕ is exploited to prove that if ϕ is locally satisfiable it must also be satisfiable.

The work of Paneth and Rothblum [PR14] further abstracts the notion of local satisfiability, redefining it in a way that is independent of the protocol of [KRR14]. Based on this abstraction, they separate the *construction* of [KRR14] into two steps. In the first step, the main part of the [KRR14] proof system is converted into a protocol for proving local satisfiability of formulas. In the second step, the cloud uses this protocol to convince the user that the formula ϕ is locally satisfiable. As before, the structure of the formula ϕ is exploited to prove that ϕ is satisfiable.

Local Satisfiability. Unlike full-fledged satisfiability, the notion of local satisfiability only considers assignments to ℓ variables at a time, where ℓ is a locality parameter that may be much smaller than the total number of variables in the formula. Formally, we say that a 3-SAT formula ϕ is ℓ-locally satisfiable if for every set Q of ℓ variables there exists a distribution D_Q over assignments to the variables in Q such that the following conditions are satisfied:

Everywhere local consistency. For every set Q of ℓ variables, a random assignment in D_Q satisfies all local constraints in ϕ over the variables in Q with high probability.

No-signaling. For every set Q of ℓ variables and for every subset $Q' \subseteq Q$, the distribution of an assignment sampled from D_Q restricted to the variables in Q' is independent of the other variables in $Q \setminus Q'$.

From Local Satisfiability to Full-Fledged Satisfiability. In the [KRR14] proof system, ℓ is a fixed polynomial in the security parameter, independent of the size of the formula ϕ (the communication complexity of the proof grows with ℓ). In this setting, local satisfiability does not generally imply full-fledged satisfiability. However, the analysis of [KRR14] exploits the specific structure of ϕ to go from local satisfiability to full-fledged satisfiability. The proof of this step crucially relies on the fact that the formula ϕ describes a deterministic computation. We show how to extend this proof for non-deterministic computations of a specific form.

Roughly, we require that (computationally) there exists a unique "correct" witness that can be verified *locally*. Namely, for any proposed witness (that can be found efficiently) and any bit of this proposed witness, it is possible to verify that the value of this bit agrees with the correct witness in time that is independent of the running time of the entire computation.

More on the Analysis of KRR. We describe the argument of [KRR14] and explain why it fails for non-deterministic computations. To go from local satisfiability to full-fledged satisfiability, the proof of [KRR14] relies on the fact that the formula ϕ describing an accepting deterministic computation has a unique satisfying assignment. We call this the *correct* assignment to ϕ. The rest of the proof uses the fact that the variables of ϕ can be partitioned into "layers" such that variables in the i-th layer correspond to the computation's state immediately before the i-th computation step. The proof proceeds by induction over the layers. In the inductive step we assume that local assignments to any ℓ variables in the i-th layer are correct (agrees with the correct assignment) with high probability and prove that the same holds for the $(i+1)$-st layer. Indeed, if the local assignment to some set of ℓ variables in the $(i+1)$-st layer is correct with a significantly lower probability, the special structure of ϕ and the no-signaling property of the assignments can be used to argue that there must exist a set of ℓ variables whose assignment violates ϕ's local constraints with some significant probability.

Non-deterministic Computations. The above argument does not extend to non-deterministic computations, since the notion of a "correct" assignment is not well defined in this setting. Moreover, even if there is a unique witness that makes the computation accept, and we consider the correct assignment defined by this witness, the above argument still fails. The issue is that even if every local assignment to any set of variables in the i-th layer is correct, there could still be more than one assignment to variables in the $(i+1)$-st layer satisfying all of ϕ's local constraints.

We show how to overcome this problem for non-deterministic computations where (computationally) there exists a unique "correct" witness that can be verified *locally*, as described above. Consider for example the computation of the Turing machine TVer on input $(M, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}})$ where d is the digest of the initial memory D. The (computationally) unique witness for this computation is a transcript of the program execution that can be verified locally – one step at a time.

In more details, let Trans be the correct transcript defined by the execution of M on memory D. Let ϕ be the formula describing the computation of $\mathsf{TVer}(M, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}})$. We prove that any accepting local assignment to variables of ϕ must agree with the global correct assignment to ϕ defined by the execution of TVer with the (well defined) transcript Trans. As in the case of deterministic computations, we partition ϕ's variables into layers. In the i-th inductive step we assume that local assignments to any ℓ variables in the i-th layer are correct with high probability. If the local assignment to some set of ℓ variables in the $(i+1)$-th layer is correct with a significantly lower probability then we prove that the assignment must describe an incorrect transcript. Since both the correct transcript and the incorrect one contain valid certificates, we can use these certificates to break the security of the hash tree.

Multi-prover Arguments. The presentation of the construction in [KRR14], as well as the presentation in the body of this work, goes through the intermediate

step of constructing a no-signaling multi-prover proof-system. In more details, [KRR14] first construct a no-signaling multi-prover interactive proof for local-satisfiability. They then leverage local-satisfiability to prove full-fledged satisfiability, resulting in a no-signaling multi-prover interactive proof (with unconditional soundness) for deterministic computations. Finally, they transform any no-signaling multi-prover interactive proof into a delegation scheme assuming fully-homomorphic encryption.

Our construction follows the same blueprint. We first construct a no-signaling multi-prover interactive argument for RAM computations, and then transform it into a delegation scheme. (Due to space limitations, we do not describe the transformation from a multi-prover interactive argument into a delegation scheme which can be found in the full version of this work [KP15].) Unlike in [KRR14], the soundness of our multi-prover arguments is conditional on the existence of collision-resistent hashing. We note that for RAM delegation, computational assumptions are necessary even in the multi-prover model.

2 Tools and Definitions

2.1 Notation

For sets B, S, we denote by B^S the set of vectors of elements in B indexed by the elements of S. That is, every vector $\mathbf{a} \in B^S$ is of the form $\mathbf{a} = (\mathsf{a}_i \in B)_{i \in S}$. For a vector $\mathbf{a} \in B^S$ and a subset $Q \subseteq S$, we denote by $\mathbf{a}[Q] \in B^Q$ the vector that contains only the elements in $\mathbf{a}$ with indices in Q, that is, $\mathbf{a}[Q] = (\mathsf{a}_i)_{i \in Q}$.

2.2 RAM Computation

We consider the standard model of RAM computation where a program M can access an initial memory string $D \in \{0,1\}^n$. For an input x, we denoted by $M^D(x)$ an execution of the program M with input x and initial memory D. For a bit $y \in \{0,1\}$ and for a string $D_{\mathsf{new}} \in \{0,1\}^n$ we also use the notation $y \leftarrow M^{(D \to D_{\mathsf{new}})}(x)$ to denote that y is the output of the program M on input x and initial memory D, and D_{new} is the final memory string after the execution. For simplicity we think only of RAM programs that output a single bit.[5] The computation of M is carried out one step at a time by a CPU algorithm STEP. STEP is a polynomial-time algorithm that takes as input a description of a program M, an input x, a state of size $O(\log n)$, and a bit that was supposedly read from memory, and it outputs a quadruple

$$(\mathsf{state}_{\mathsf{new}}, i^{\mathsf{r}}, i^{\mathsf{w}}, b^{\mathsf{w}}) \leftarrow \mathsf{STEP}(M, x, \mathsf{state}, b^{\mathsf{r}}),$$

where $\mathsf{state}_{\mathsf{new}}$ is the updated state, i^{r} denotes the location in memory to be read next, the location i^{w} denotes the location in memory to write to next,

[5] A program that outputs multiple bits can be simulated by executing several programs in parallel, or by writing the output directly to the memory.

and the bit b^{w} denotes the bit to be written in location i^{w}. The execution $M^D(x)$ proceeds as follows. The program starts with some empty initial state state_1. By convention we set the first memory location read by the program to be $i_1^{\mathsf{r}} = 1$. Starting from $j = 1$, the j-th execution step proceeds as follows:

1. Read from memory the bit $b_j^{\mathsf{r}} \leftarrow D[i_j^{\mathsf{r}}]$.
2. Compute $(\mathsf{state}_{j+1}, i_{j+1}^{\mathsf{r}}, i_{j+1}^{\mathsf{w}}, b_{j+1}^{\mathsf{w}}) \leftarrow \mathsf{STEP}(M, x, \mathsf{state}_j, b_j^{\mathsf{r}})$.
3. Write a bit to memory $D[i_{j+1}^{\mathsf{w}}] \leftarrow b_{j+1}^{\mathsf{w}}$. (If $i_{j+1}^{\mathsf{w}} = \perp$ no writing is performed in this step.)

The execution terminates when the program STEP outputs a special terminating state. We assume that the terminating state includes the value of the output bit y. Note that after the last step was executed and an output has been produced, the memory is written to one last time. We say that a machine M is *read only*, if for every $(x, \mathsf{state}, b^{\mathsf{r}})$, $\mathsf{STEP}(M, x, \mathsf{state}, b^{\mathsf{r}})$ outputs $(\mathsf{state}_{\mathsf{new}}, i^{\mathsf{r}}, i^{\mathsf{w}}, b^{\mathsf{w}})$ where $i^{\mathsf{w}} = \perp$.

Remark 2.1 (Space complexity of STEP*).* We assume without loss of generality that the RAM program M reads the input x once and copies it to memory. Therefore the space complexity of the algorithm STEP is polylog(n).

2.3 Hash Tree

Let $D \in \{0,1\}^n$ be a string. Let k be a security parameter such that $n < 2^k$.

A hash-tree scheme consists of algorithms:

$$(\mathsf{HT.Gen}, \mathsf{HT.Hash}, \mathsf{HT.Read}, \mathsf{HT.Write}, \mathsf{HT.VerRead}, \mathsf{HT.VerWrite}),$$

with the following syntax and efficiency:

- $\mathsf{HT.Gen}(1^k) \rightarrow \mathsf{key}$:
  A randomized polynomial-time algorithm that outputs a hash key, denoted by key.
- $\mathsf{HT.Hash}(\mathsf{key}, D) \rightarrow (\mathsf{tree}, \mathsf{rt})$:
  A deterministic polynomial-time algorithm that outputs a hash tree denoted by tree, and a hash root rt of size poly(k) (we assume that both strings tree and rt include key).
- $\mathsf{HT.Read}^{\mathsf{tree}}(i^{\mathsf{r}}) \rightarrow (b^{\mathsf{r}}, \mathsf{pf})$:
  A deterministic read-only RAM program that accesses the initial memory string tree, runs in time poly(k), and outputs a bit, denoted by b^{r}, and a proof, denoted by pf.
- $\mathsf{HT.Write}^{\mathsf{tree}}(i^{\mathsf{w}}, b^{\mathsf{w}}) \rightarrow (\mathsf{rt}_{\mathsf{new}}, \mathsf{pf})$:
  A deterministic RAM program that accesses the initial memory string tree, runs in time poly(k), and outputs a new hash root, denoted by $\mathsf{rt}_{\mathsf{new}}$, and a proof, denoted by pf.
- $\mathsf{HT.VerRead}(\mathsf{rt}, i^{\mathsf{r}}, b^{\mathsf{r}}, \mathsf{pf}) \rightarrow b$:
  A deterministic polynomial-time algorithm that outputs an acceptance bit b.

– $\mathsf{HT.VerWrite}(\mathsf{rt}, i^{\mathsf{w}}, b^{\mathsf{w}}, \mathsf{rt}_{\mathsf{new}}, \mathsf{pf}) \to b$:
A deterministic polynomial-time algorithm that outputs an acceptance bit b.

Definition 2.1 (Hash-Tree). *A hash-tree scheme*

$$(\mathsf{HT.Gen}, \mathsf{HT.Hash}, \mathsf{HT.Read}, \mathsf{HT.Write}, \mathsf{HT.VerRead}, \mathsf{HT.VerWrite}),$$

satisfies the following properties.

– *Completeness of Read. For every $k \in \mathbb{N}$ and for every $D \in \{0,1\}^n$ such that $n \leq 2^k$, and for every $i^{\mathsf{r}} \in [n]$*

$$\Pr\left[\begin{array}{l} 1 = \mathsf{HT.VerRead}(\mathsf{rt}, i^{\mathsf{r}}, b^{\mathsf{r}}, \mathsf{pf}) \\ D[i^{\mathsf{r}}] = b^{\mathsf{r}} \end{array} \middle| \begin{array}{l} \mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k) \\ (\mathsf{tree}, \mathsf{rt}) \leftarrow \mathsf{HT.Hash}(\mathsf{key}, D) \\ (b^{\mathsf{r}}, \mathsf{pf}) \leftarrow \mathsf{HT.Read}^{\mathsf{tree}}(i^{\mathsf{r}}) \end{array}\right] = 1.$$

– *Completeness of Write. For every $k \in \mathbb{N}$ and for every $D \in \{0,1\}^n$ such that $n \leq 2^k$, for every $i^{\mathsf{w}} \in [n], b^{\mathsf{w}} \in \{0,1\}$, and for $D_{\mathsf{new}} \in \{0,1\}^n$ that is equal to the string D except that $D_{\mathsf{new}}[i^{\mathsf{w}}] = b^{\mathsf{w}}$*

$$\Pr\left[\begin{array}{l} 1 = \mathsf{HT.VerWrite}(\mathsf{rt}, i^{\mathsf{w}}, b^{\mathsf{w}}, \mathsf{rt}'_{\mathsf{new}}, \mathsf{pf}) \\ \mathsf{rt}'_{\mathsf{new}} = \mathsf{rt}_{\mathsf{new}} \end{array} \middle| \begin{array}{l} \mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k) \\ (\mathsf{tree}, \mathsf{rt}) \leftarrow \mathsf{HT.Hash}(\mathsf{key}, D) \\ (\mathsf{tree}_{\mathsf{new}}, \mathsf{rt}_{\mathsf{new}}) \leftarrow \mathsf{HT.Hash}(\mathsf{key}, D_{\mathsf{new}}) \\ (\mathsf{rt}'_{\mathsf{new}}, \mathsf{pf}) \leftarrow \mathsf{HT.Write}^{\mathsf{tree}}(i^{\mathsf{w}}, b^{\mathsf{w}}) \end{array}\right] = 1.$$

– *Soundness of Read. For every polynomial size adversary Adv there exists a negligible function μ such that for every $k \in \mathbb{N}$*

$$\Pr\left[\begin{array}{l} (b', \mathsf{pf}') \neq (b, \mathsf{pf}) \\ 1 = \mathsf{HT.VerRead}(\mathsf{rt}, i, b, \mathsf{pf}) \\ 1 = \mathsf{HT.VerRead}(\mathsf{rt}, i, b', \mathsf{pf}') \end{array} \middle| \begin{array}{l} \mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k) \\ (\mathsf{rt}, i, b, \mathsf{pf}, b', \mathsf{pf}') \leftarrow \mathsf{Adv}(\mathsf{key}) \end{array}\right] \leq \mu(k).$$

– *Soundness of Write. For every poly-size adversary Adv there exists a negligible function μ such that for every $k \in \mathbb{N}$*

$$\Pr\left[\begin{array}{l} (\mathsf{rt}'_{\mathsf{new}}, \mathsf{pf}') \neq (\mathsf{rt}_{\mathsf{new}}, \mathsf{pf}) \\ 1 = \mathsf{HT.VerWrite}(\mathsf{rt}, i, b, \mathsf{rt}_{\mathsf{new}}, \mathsf{pf}) \\ 1 = \mathsf{HT.VerWrite}(\mathsf{rt}, i, b, \mathsf{rt}'_{\mathsf{new}}, \mathsf{pf}') \end{array} \middle| \begin{array}{l} \mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k) \\ (\mathsf{rt}, i, b, \mathsf{rt}_{\mathsf{new}}, \mathsf{pf}, \mathsf{rt}'_{\mathsf{new}}, \mathsf{pf}') \leftarrow \mathsf{Adv}(\mathsf{key}) \end{array}\right] \leq \mu(k).$$

We say that the hash-tree scheme is (S, ϵ)-secure, for a function $S(k)$ and a negligible function $\epsilon(k)$, if for every constant $c > 0$, the soundness of read and soundness of write properties hold for every adversary of size $S(k)^c$ with probability at most $\epsilon(k)^c$. We say that the hash-tree scheme has sub-exponential security if it is $(2^{k^\delta}, 2^{-k^\delta})$-secure for some constant $\delta > 0$.

Remark 2.2 (Unique proofs in Definition 2.1). In the soundness properties of Definition 2.1 we make the strong requirement that it is hard to find two different proofs for any statement (even a correct one). This strong requirement simplifies the proof of Theorem 4.1, however the proof can be modified to rely on a weaker soundness requirement.

Theorem 2.1 ([Mer87]). *A hash-tree scheme satisfying Definition 2.1 can be constructed from any family of collision-resistant hash functions. Moreover, the hash-tree scheme is sub-exponentially secure if the underlying collision-resistant hash family is sub-exponentially secure.*

2.4 Delegation for RAM Computations

Let M be a T-time RAM program, let $x \in \{0,1\}^m$ be an input to the program, and let $D \in \{0,1\}^n$ be some initial memory string. Let k be a security parameter such that $|M|, \mathsf{T}(m), n < 2^k$. A two-message delegation scheme for RAM computations consists of algorithms:

$$(\mathsf{ParamGen}, \mathsf{MemGen}, \mathsf{QueryGen}, \mathsf{Output}, \mathsf{Prover}, \mathsf{Verifier}),$$

with the following syntax and efficiency:

- $\mathsf{ParamGen}(1^k) \to \mathsf{pp}$:
  A randomized polynomial-time algorithm that outputs public parameters pp.
- $\mathsf{MemGen}(\mathsf{pp}, D) \to (\mathsf{dt}, \mathsf{d})$:
  A deterministic polynomial-time algorithm that outputs the processed memory dt, and a digest of the memory d of size $\mathrm{poly}(k)$.
- $\mathsf{QueryGen}(1^k) \to (\mathsf{q}, \mathsf{st})$:
  A randomized polynomial-time algorithm that outputs a query q and a secret state st.
- $\mathsf{Output}^{\mathsf{dt}}(1^{\mathsf{T}(m)}, n, M, x) \to (y, \mathsf{d}_{\mathsf{new}}, \mathsf{Trans})$:
  A deterministic RAM program running in time $\mathsf{T}(m) \cdot \mathrm{poly}(k)$ that accesses the processed memory dt, and outputs the output bit y, and a new digest $\mathsf{d}_{\mathsf{new}}$ of size $\mathrm{poly}(k)$ and a computation transcript Trans.
- $\mathsf{Prover}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{Trans}, \mathsf{q}) \to \mathsf{pf}$:
  A deterministic algorithm running in time $\mathrm{poly}(\mathsf{T}(m), k)$ that outputs a proof pf of size $\mathrm{poly}(k)$.
- $\mathsf{Verifier}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, \mathsf{pf}) \to b$:
  A deterministic algorithm running in time $m \cdot \mathrm{poly}(k)$ that outputs an acceptance bit b.

Remark 2.3 (Statement-independent queries). In the above, the queries generated by the algorithm $\mathsf{QueryGen}$ are independent of the program, the input and the memory digest. We could consider a more liberal definition that allows such a dependency, however, in our construction this is not needed.

Remark 2.4 (Verifier efficiency). We note that the dependence of the verification time on the input length m can be improved. In particular, in our construction, given oracle access to a *low-degree extension* encoding of the input x, the verifier's running time is $\mathrm{poly}(k)$.

Remark 2.5 (The Output *algorithm).* In the above interface we separated the prover computation into two algorithms. The first algorithm, Output, accesses the memory, carries out the computation, and produces the output as well as a *transcript* of the computation. This transcript may include all the memory accessed during the RAM computation or any other information. We only restrict the size of the transcript to be related to the running time of the RAM computation. The second algorithm, Prover, is given the transcript and the challenge query and outputs the proof. This separation ensures that the memory locations accessed by the prover are independent of the challenge query. This property is used in the transformation from no-signaling multi-prover arguments to delegationin [KP15].

Definition 2.2 (Two-Message Argument for RAM computations). *A two-message delegation scheme* $(\mathsf{ParamGen}, \mathsf{MemGen}, \mathsf{QueryGen}, \mathsf{Prover}, \mathsf{Verifier})$ *for RAM computations satisfies the following properties.*

– *Completeness.* *For every security parameter $k \in \mathbb{N}$, every T-time RAM program M, every input $x \in \{0,1\}^m$, every $D \in \{0,1\}^n$, and every (y, D_{new}) such that $\mathsf{T}(m), n \leq 2^k$ and $y \leftarrow M^{(D \to D_{\mathsf{new}})}(x)$*

$$\Pr\left[\begin{array}{l|l} 1 = \mathsf{Verifier}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, \mathsf{pf}) & \\ (\mathsf{dt}_{\mathsf{new}}, \mathsf{d}_{\mathsf{new}}) = \mathsf{MemGen}(\mathsf{pp}, D_{\mathsf{new}}) & \\ \mathsf{pp} \leftarrow \mathsf{ParamGen}(1^k) & \\ (\mathsf{dt}, \mathsf{d}) \leftarrow \mathsf{MemGen}(\mathsf{pp}, D) & \\ (\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{QueryGen}(1^k) & \\ (y, \mathsf{d}_{\mathsf{new}}, \mathsf{Trans}) \leftarrow \mathsf{Output}^{\mathsf{dt} \to \mathsf{dt}_{\mathsf{new}}}(1^{\mathsf{T}(m)}, n, M, x) & \\ \mathsf{pf} \leftarrow \mathsf{Prover}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{Trans}, \mathsf{q}) & \end{array}\right] = 1.$$

– *Soundness.* *For every pair of polynomial-size adversaries $(\mathsf{Adv}_1, \mathsf{Adv}_2)$ there exists a negligible function μ such that for every $k \in \mathbb{N}$*

$$\Pr\left[\begin{array}{l|l} (y, \mathsf{d}_{\mathsf{new}}) \neq (y', \mathsf{d}'_{\mathsf{new}}) & \\ 1 = \mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, \mathsf{pf}) & \\ 1 = \mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y', \mathsf{d}'_{\mathsf{new}}), \mathsf{st}, \mathsf{pf}') & \\ \mathsf{pp} \leftarrow \mathsf{ParamGen}(1^k) & \\ (M, x, 1^{\mathsf{T}}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}, y', \mathsf{d}'_{\mathsf{new}}) \leftarrow \mathsf{Adv}_1(1^k, \mathsf{pp}) & \\ (\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{QueryGen}(1^k) & \\ (\mathsf{pf}, \mathsf{pf}') \leftarrow \mathsf{Adv}_2(1^k, \mathsf{pp}, \mathsf{q}) & \end{array}\right] \leq \mu(k).$$

We say that the delegation scheme is (S, ϵ)-secure, for a function $S(k)$ and a negligible function $\epsilon(k)$, if for every constant $c > 0$, the soundness property holds for every pair of adversaries of size $S(k)^c$ with probability at most $\epsilon(k)^c$.

2.5 Multi-prover Arguments for RAM Computations

Let ℓ be a polynomial, M be a T-time RAM program, let $x \in \{0,1\}^m$ be an input to the program, and let $D \in \{0,1\}^n$ be some initial memory string. Let k be a security parameter such that $|M|, \mathsf{T}(m), n < 2^k$. An *$\ell$-prover argument* for RAM computations consists of algorithms:

$$(\mathsf{ParamGen}, \mathsf{MemGen}, \mathsf{QueryGen}, \mathsf{Output}, \mathsf{Prover}, \mathsf{Verifier}),$$

with the following syntax and efficiency:

- $\mathsf{ParamGen}(1^k) \to \mathsf{pp}$:
  A randomized polynomial-time algorithm that outputs public parameters pp.
- $\mathsf{MemGen}(\mathsf{pp}, D) \to (\mathsf{dt}, \mathsf{d})$:
  A deterministic polynomial-time algorithm that outputs the processed memory dt and a digest of the memory d of size $\mathrm{poly}(k)$.
- $\mathsf{QueryGen}(1^k) \to ((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st})$:
  A randomized polynomial-time algorithm that outputs a set of $\ell = \ell(k)$ queries $(\mathsf{q}_1, \ldots, \mathsf{q}_\ell)$, and a secret state st.
- $\mathsf{Output}^{\mathsf{dt}}(1^{\mathsf{T}(m)}, n, M, x) \to (y, \mathsf{d}_{\mathsf{new}}, \mathsf{Trans})$:
  A deterministic RAM program running in time $\mathsf{T}(m) \cdot \mathrm{poly}(k)$ that accesses the processed memory dt, and outputs the output bit y, a new digest $\mathsf{d}_{\mathsf{new}}$ of size $\mathrm{poly}(k)$, and a computation transcript Trans.
- $\mathsf{Prover}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{Trans}, \mathsf{q}) \to \mathsf{a}$:
  A deterministic algorithm running in time $\mathrm{poly}(\mathsf{T}(m), k)$ that outputs an answer a of size $\mathrm{poly}(k)$ to a single query q.
- $\mathsf{Verifier}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) \to b$:
  A deterministic algorithm running in time $m \cdot \mathrm{poly}(k)$ that outputs an acceptance bit b.

Remark 2.6 (Statement-independent queries). In the above, the queries generated by the algorithm $\mathsf{QueryGen}$ are independent of the program, the input and the memory digest. We could consider a more liberal definition that allows such a dependency, however, in our construction this is not needed.

Remark 2.7 (Verification efficiency). We note that the dependence of the verification time on the input length m can be improved. In particular, in our construction, given oracle access to a *low-degree extension* encoding of the input x, the verifier's running time is $\mathrm{poly}(k)$.

Remark 2.8 (The Output algorithm). In the above interface we separated the prover computation into two algorithms. The first algorithm, Output, accesses the memory, carries out the computation, and produces the output as well as a *transcript* of the computation. This transcript may include all the memory accessed during the RAM computation or any other information. We only restrict the size of the transcript to be related to the running time of the RAM computation. The second algorithm, Prover, is given the transcript and a challenge query and outputs an answer. This separation ensures that the memory locations

accessed by the prover are independent of the challenge queries. This property is used in the transformation from no-signaling multi-prover arguments to delegationin [KP15].

Definition 2.3 (Multi-Prover Argument for RAM computations). *Let $\ell = \ell(k)$ be a polynomial in the security parameter. An ℓ-prover argument system* $(\mathsf{ParamGen}, \mathsf{MemGen}, \mathsf{QueryGen}, \mathsf{Output}, \mathsf{Prover}, \mathsf{Verifier})$ *for RAM computations satisfies the following properties.*

- *Completeness.* *For every security parameter $k \in \mathbb{N}$, every T-time RAM program M, every input $x \in \{0,1\}^m$, every $D \in \{0,1\}^n$, and every (y, D_{new}), such that $\mathsf{T}(m), n \leq 2^k$ and $y \leftarrow M^{(D \to D_{\mathsf{new}})}(x)$*

$$\Pr\left[\begin{array}{l|l} 1 = \mathsf{Verifier}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) & \\ (\mathsf{dt}_{\mathsf{new}}, \mathsf{d}_{\mathsf{new}}) = \mathsf{MemGen}(\mathsf{pp}, D_{\mathsf{new}}) & \\ \mathsf{pp} \leftarrow \mathsf{ParamGen}(1^k) & \\ (\mathsf{dt}, \mathsf{d}) \leftarrow \mathsf{MemGen}(\mathsf{pp}, D) & \\ ((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st}) \leftarrow \mathsf{QueryGen}(1^k) & \\ (y, \mathsf{d}_{\mathsf{new}}, \mathsf{Trans}) \leftarrow \mathsf{Output}^{\mathsf{dt} \to \mathsf{dt}_{\mathsf{new}}}(1^{\mathsf{T}(m)}, n, M, x) & \\ \forall i \in [\ell] : \mathsf{a}_i \leftarrow \mathsf{Prover}((M, x, \mathsf{T}(m), \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{Trans}, \mathsf{q}_i) & \end{array}\right] = 1.$$

- *Soundness.* *For every pair of polynomial-size adversaries $(\mathsf{Adv}_1, \mathsf{Adv}_2)$ there exists a negligible function μ such that for every $k \in \mathbb{N}$ and for $\ell = \ell(k)$*

$$\Pr\left[\begin{array}{l|l} (y, \mathsf{d}_{\mathsf{new}}) \neq (y', \mathsf{d}'_{\mathsf{new}}) & \\ 1 = \mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) & \\ 1 = \mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y', \mathsf{d}'_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}'_1, \ldots, \mathsf{a}'_\ell)) & \\ \mathsf{pp} \leftarrow \mathsf{ParamGen}(1^k) & \\ (M, x, 1^{\mathsf{T}}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}, y', \mathsf{d}'_{\mathsf{new}}) \leftarrow \mathsf{Adv}_1(1^k, \mathsf{pp}) & \\ ((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st}) \leftarrow \mathsf{QueryGen}(1^k) & \\ \forall i \in [\ell] : (\mathsf{a}_i, \mathsf{a}'_i) \leftarrow \mathsf{Adv}_2(1^k, \mathsf{pp}, \mathsf{q}_i) & \end{array}\right] \leq \mu(k).$$

We say that the argument system is (S, ϵ)-secure, for a function $S(k)$ and a negligible function $\epsilon(k)$, if for every constant $c > 0$, the soundness property holds for every pair of adversaries of size $S(k)^c$ with probability at most $\epsilon(k)^c$.

2.6 No-Signaling Multi-prover Arguments for RAM Computations

No signaling multi-prover arguments are multi-prover arguments, where the cheating provers are given extra power. In multi-prover arguments (or proofs), each prover answers its own query *locally*, without knowing anything about the queries that were sent to the other provers.

In the no-signaling model we allow the *malicious* provers' answers to depend on all the queries, as long as for any subset $Q \subset [\ell]$ and for every two query vectors $\mathbf{q}^1 = (\mathsf{q}^1_1, \ldots, \mathsf{q}^1_\ell)$ and $\mathbf{q}^2 = (\mathsf{q}^2_1, \ldots, \mathsf{q}^2_\ell)$, such that $\mathbf{q}^1[Q] = \mathbf{q}^2[Q]$, the corresponding vectors of answers $\mathbf{a}^1, \mathbf{a}^2$ (as random variables) satisfy that $\mathbf{a}^1[Q]$

and $\mathbf{a}^2[Q]$ are identically distributed. Intuitively, this means that the answers of the provers in the set Q do not contain information about the queries to the provers outside Q, except for the information that is already found in the queries to the provers in Q.

Definition 2.4. *For a set B and for $\ell \in \mathbb{N}$, we say that a pair of vectors of correlated random variables*

$$\mathbf{q} = (\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathbf{a} = (\mathsf{a}_1, \ldots, \mathsf{a}_\ell) \in B^{[\ell]}.$$

is no-signaling *if for every subset $Q \subset [\ell]$ and every two vectors $\mathbf{q}^1, \mathbf{q}^2$ in the support of $\mathbf{q}$ such that $\mathbf{q}^1[Q] = \mathbf{q}^2[Q]$, the random variables $\mathbf{a}[Q]$ conditioned on $\mathbf{q} = \mathbf{q}^1$ and $\boldsymbol{a}[Q]$ conditioned on $\mathbf{q} = \mathbf{q}^2$ are identically distributed.*

If these random are not identical, but rather, the statistical distance between them is at most δ, we say that the pair $(\mathbf{q}, \mathbf{a})$ is δ-no-signaling.

Definition 2.5. *An ℓ-prover argument system* $(\mathsf{ParamGen}, \mathsf{MemGen}, \mathsf{QueryGen}, \mathsf{Output}, \mathsf{Prover}, \mathsf{Verifier})$ *for RAM computations is said to be sound against δ-no-signaling strategies (or provers) if the following (more general) soundness property is satisfied:*

For every pair of polynomial-size adversaries $(\mathsf{Adv}_1, \mathsf{Adv}_2)$ *satisfying a δ-no-signaling condition (specified below), there exists a negligible function μ such that for every $k \in \mathbb{N}$ and for $\ell = \ell(k)$:*

$$\Pr\left[\begin{array}{l} (y, \mathsf{d}_{\mathsf{new}}) \neq (y', \mathsf{d}'_{\mathsf{new}}) \\ 1 = \mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) \\ 1 = \mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y', \mathsf{d}'_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}'_1, \ldots, \mathsf{a}'_\ell)) \end{array} \middle| \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{ParamGen}(1^k) \\ (M, x, 1^{\mathsf{T}}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}, y', \mathsf{d}'_{\mathsf{new}}) \leftarrow \mathsf{Adv}_1(1^k, \mathsf{pp}) \\ ((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st}) \leftarrow \mathsf{QueryGen}(1^k) \\ ((\mathsf{a}_1, \mathsf{a}'_1), \ldots, (\mathsf{a}_\ell, \mathsf{a}'_\ell)) \leftarrow \mathsf{Adv}_2(1^k, \mathsf{pp}, (\mathsf{q}_1, \ldots, \mathsf{q}_\ell)) \end{array}\right] \leq \mu(k).$$

where $(\mathsf{Adv}_1, \mathsf{Adv}_2)$ *satisfy the δ-no-signaling condition if the random variables* $(\mathsf{q}_1, \ldots, \mathsf{q}_\ell)$ *and* $((\mathsf{a}_1, \mathsf{a}'_1), \ldots, (\mathsf{a}_\ell, \mathsf{a}'_\ell))$ *are δ-no-signaling.*

We say that the argument system is (S, ϵ)-secure against δ-no-signaling strategies, for a function $S(k)$ and a negligible function $\epsilon(k)$, if for every constant $c > 0$, the soundness property holds with probability at most $\epsilon(k)^c$ for every pair of adversaries of size $S(k)^c$ satisfying the δ-no-signaling condition.

3 Local Satisfiability

In this section we introduce the notion of *local satisfiability* for formulas, and state a result of [KRR14] providing a no-signaling multi-prover argument for the *local* satisfiability of any *non-deterministic* Turing machine computation. This presentation is based on an abstraction of [PR14].

We start by describing, for every non-deterministic Turing machine M and input x, a formula $\varphi_{M,x}$ of a specific structure that is satisfiable if and only if M accepts x. Then we define the notion of local satisfiability for formulas. Finally we state a result of [KRR14] providing a no-signaling multi-prover argument for the local satisfiability of formulas of the form $\varphi_{M,x}$.

3.1 A Formula Describing Non-Deterministic Computations

The machine M. Let M be a T-time S-space non-deterministic Turing machine. We can think of M as a two-input machine, such that M accepts the input x if and only if there exists a witness w such that $M(x, w)$ accepts. In what follows, we consider a machine M and an input x such that $|x|$ is smaller than the machine's space S. Therefore, we can assume that M copies the entire input x to its work tape. However, the witness w we consider may be such that $|w|$ is much larger than S and therefore w must be given on a separate read-only read-once witness tape.

The Machine's State. For $i \in [\mathsf{T}]$ let $\mathsf{st}_i \in \{0,1\}^{O(\mathsf{S})}$ denote the state of the computation $M(x, w)$ immediately before the i-th step. The state st_i includes:

- the machine's state.
- the entire content of the work tape, including the reading head's location.
- the reading head's location j on the witness tape, and the witness bit w_j.

Note that st_i does not include the entire content of the witness tape which may be much longer than S.

The following theorem states that the decision of whether a non-deterministic Turing machine M accepts an inputs x can be converted into a 3-CNF formula $\varphi_{M,x}$ of a specific structure. Loosely speaking, the variables of $\varphi_{M,x}$ correspond to the entire tableau of the computation of $M(x, w)$, and the formula verifies the consistency of all the states of this computation. Thus, $\varphi_{M,x}$ can be separated into sub-formulas, where each sub-formula verifies the consistency of two adjacent states of the computation. This intuition is formalized in the following theorem.

Theorem 3.1. *For any T-time S-space non-deterministic Turing machine M and any input x there exists a 3-CNF Boolean formula $\varphi_{M,x}$ of size $O(\mathsf{T} \cdot \mathsf{S})$ such that the following holds:*

1. *$\varphi_{M,x}$ is satisfiable if and only if M accepts x. Moreover, given a witness for the fact that M accepts x there is an efficient way to find a satisfying assignment to $\varphi_{M,x}$.*
2. *The formula $\varphi_{M,x}$ can be written as*

$$\varphi_{M,x} = \bigwedge_{i \in [\mathsf{T}-1]} \varphi^i_{M,x},$$

and the set of the input variables of $\varphi_{M,x}$, denoted by V, can be divided into subsets

$$V = \bigcup_{i \in [\mathsf{T}]} V_i,$$

such that each formula $\varphi^i_{M,x}$ is over the variables $V_i \cup V_{i+1}$, and each $V_i \subseteq V$ is of size $\mathsf{S}' = O(\mathsf{S})$.

3. *There exists an efficient algorithm* State *such that given an assignment to the variables V_i, outputs a state st_i of the computation of $M(x)$ immediately before the i-th step,*

$$\mathsf{st}_i = \mathsf{State}(\mathsf{a}[V_i]).$$

The algorithm State *satisfies the following properties:*

- *For every $i \in [\mathsf{T} - 1]$ and for every assignment $\mathsf{a} \in \{0,1\}^{V_i \cup V_{i+1}}$, if $\varphi^i_{M,x}(\mathsf{a}) = 1$ then the states*

$$\mathsf{st}_i = \mathsf{State}(\mathsf{a}[V_i]), \quad \mathsf{st}_{i+1} = \mathsf{State}(\mathsf{a}[V_{i+1}])$$

  are consistent with the program M.
- *For every assignment $\mathsf{a} \in \{0,1\}^{V_1 \cup V_2}$, if $\varphi^1_{M,x}(\mathsf{a}) = 1$ then the state*

$$\mathsf{st}_1 = \mathsf{State}(\mathsf{a}[V_1])$$

  is the initial state of the machine M with the input x.
- *For every assignment $\mathsf{a} \in \{0,1\}^{V_{\mathsf{T}-1} \cup V_{\mathsf{T}}}$, if $\varphi^{\mathsf{T}-1}_{M,x}(\mathsf{a}) = 1$ then the state*

$$\mathsf{st}_{\mathsf{T}} = \mathsf{State}(\mathsf{a}[V_{\mathsf{T}}])$$

  is an accepting state.

Remark 3.1 (On the formula size). It is well known that there exists a formula of size only $\tilde{O}(\mathsf{T})$ (independent of S) that is satisfiable if and only if M accepts x. Such a formula can be obtained by first making the machine M oblivious [PF79]. However such a formula will not have the desired structure described in Theorem 3.1.

3.2 Definition of Local Satisfiability

In this section we define the notion of local satisfiability for formulas.

Definition 3.1 (Local Assignment Generator *[PR14]*). *Let φ be a 3-CNF formula over a set of variables V. An (ℓ, ϵ, δ)-local assignment generator* Assign *for φ is a probabilistic algorithm running in time* $\mathrm{poly}(|V|)$ *that takes as input a set of at most ℓ queries $Q \subseteq V, |Q| \leq \ell$, and outputs an assignment $\mathbf{a} \in \{0,1\}^Q$, such that the following two properties hold.*

- **Everywhere Local Consistency.** *For every set $Q \subseteq V, |Q| \leq \ell$, with probability $1 - \epsilon$ over a draw*

$$\mathbf{a} \leftarrow \mathsf{Assign}(Q),$$

  the assignment is locally consistent with the formula φ. That is, for every variables $q_1, q_2, q_3 \in Q$, every clause in φ over the variables q_1, q_2, q_3 is satisfied by the assignment $\mathbf{a}[\{q_1, q_2, q_3\}]$.

– **No-signaling.** *For every (all powerful) distinguisher D and every pair of sets Q, Q' such that $Q' \subseteq Q \subseteq V, |Q| \leq \ell$:*

$$\left| \Pr_{\mathbf{a} \leftarrow \mathsf{Assign}(Q)} [D(\mathbf{a}[Q']) = 1] - \Pr_{\mathbf{a}' \leftarrow \mathsf{Assign}(Q')} [D(\mathbf{a}') = 1] \right| \leq \delta.$$

Remark 3.2 (On ordered queries). In [PR14], the notion of local satisfiability is formalized using an ordered vector of queries. In Definition 3.1 however, the queries are given as an unordered set. We note that these formulations are equivalent.

3.3 No-Signaling Multi-prover Arguments for Local Satisfiability

To obtain our results we use a multi-prover proof system satisfying a no-signaling local soundness property (see Theorem 3.2 below). Such a proof system was constructed in [KRR14].

Let k be the security parameter and let $\ell = \ell(k)$ be a polynomial. Let M be a non-deterministic Turing machine running in time T and space S, let $x \in \{0,1\}^m$ be an input to M such that $\mathsf{T}(m) < 2^k$ and let w be a witness. We consider an ℓ-prover proof system $(\mathsf{LS.QueryGen}, \mathsf{LS.Prover}, \mathsf{LS.Verifier})$ with the following syntax and efficiency:

– $\mathsf{LS.QueryGen}(1^k) \to ((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st})$:
A randomized polynomial-time algorithm that outputs a set of $\ell = \ell(k)$ queries $(\mathsf{q}_1, \ldots, \mathsf{q}_\ell)$, and a secret state st.
– $\mathsf{LS.Prover}(1^{\mathsf{T}(m)}, M, x, w, \mathsf{q}) \to \mathsf{a}$:
A deterministic algorithm running in time $(\mathsf{T}(m) \cdot \mathsf{S}(m))^3 \cdot \mathrm{poly}(k)$ that outputs an answer a to a single query q where $|\mathsf{a}| = O(\log(k))$.
– $\mathsf{LS.Verifier}(M, x, \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) \to b$:
A deterministic algorithm running in time $m \cdot \mathrm{poly}(k)$, that outputs an acceptance bit b.

The completeness and no-signaling local soundness properties of the above proof system are given by Theorem 3.2 proved in [KRR14].[6]

Theorem 3.2 (*[KRR14]*). *There exists a polynomial ℓ_0, such that for every polynomial ℓ' and for $\ell = \ell_0 \cdot \ell'$ there exists an ℓ-prover proof system* $(\mathsf{LS.QueryGen}, \mathsf{LS.Prover}, \mathsf{LS.Verifier})$ *that satisfies the following properties.*

– *Completeness. For every T-time (two input) Turing machine M, every input $x \in \{0,1\}^m$ and witness w such that $M(x,w) = 1$, every $k \in \mathbb{N}$ such that $\mathsf{T}(m) < 2^k$, and for $\ell = \ell(k)$,*

$$\Pr\left[1 = \mathsf{LS.Verifier}(M, x, \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) \;\middle|\; \begin{array}{l} ((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st}) \leftarrow \mathsf{LS.QueryGen}(1^k) \\ \forall i \in [\ell] : \mathsf{a}_i \leftarrow \mathsf{LS.Prover}(1^{\mathsf{T}(m)}, M, x, w, \mathsf{q}_i) \end{array}\right] = 1.$$

[6] The proof of Theorem 3.2 follows by combining Lemmas 14.1, 6.1 and 7.29 in [KRR14] together with the fact that all the claims and lemmas in Sects. 7.1–7.5 hold for arbitrary setting of parameters, and in particular for any ϵ and δ.

– *No-Signaling Local Soundness.* *There exists a probabilistic polynomial-time oracle machine* Assign *such that the following holds. For every* T*-time (two input) Turing machine* M*, every input* $x \in \{0,1\}^m$*, every security parameter* $k \in \mathbb{N}$ *such that* $\mathsf{T}(m) < 2^k$ *and* $\ell = \ell(k)$*, every* $\epsilon = \epsilon(k)$*, every* $\delta = \delta(k)$*, and every* δ*-no-signaling cheating prover* Prover^* *such that*

$$\Pr\left[1 = \mathsf{LS.Verifier}(M, x, \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)) \;\middle|\; \begin{array}{l}((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st}) \leftarrow \mathsf{LS.QueryGen}(1^k) \\ (\mathsf{a}_1, \ldots, \mathsf{a}_\ell) \leftarrow \mathsf{Prover}^*(\mathsf{q}_1, \ldots, \mathsf{q}_\ell)\end{array}\right] \geq \epsilon,$$

$\mathsf{Assign}^{\mathsf{Prover}^*}$ *is an* $(\ell', \delta', \epsilon')$*-local assignment generator for the 3-CNF formula* $\varphi_{M,x}$ *given by Theorem 3.1, with*

$$\delta' = \frac{\delta \cdot 2^{k \cdot \mathrm{polylog}(\mathsf{T}(m))}}{\epsilon}, \quad \epsilon' = \frac{\delta \cdot \mathrm{polylog}(\mathsf{T}(m))}{\epsilon}.$$

As before, we say that Prover^* *is* δ*-no-signaling if the random variables* $(\mathsf{q}_1, \ldots, \mathsf{q}_\ell)$ *and* $(\mathsf{a}_1, \ldots, \mathsf{a}_\ell)$ *are* δ*-no-signaling.*

Remark 3.3. The oracle machine Assign constructed in [KRR14] has a super-polynomial runtime.[7] However, by carefully observing the proof, it is easy to see that this super-polynomial blowup is unnecessary. This was formally proved in a followup work of [BHK16].

4 No-Signaling Multi-prover Arguments for RAM Computations

4.1 Verifying RAM Computations via Local Satisfiability

In this section we translate any RAM computation into a non-deterministic Turing machine such that the RAM computation is correct if and only if the Turing machine's computation is locally satisfiable. Consider an execution of a RAM program M that on input x and initial memory string D outputs y and results in memory D_{new} within time T. Consider also a hash-tree of the initial memory D rooted at rt and a hash-tree of the final memory D_{new} rooted at $\mathsf{rt}_{\mathsf{new}}$.

We describe a Turing machine TVer that takes as input tuples of the form $(M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}})$, together with a corresponding witness, which is a *transcript* of the RAM computation. We start by describing the algorithm TGen which generates the transcript. Roughly, the transcript contains a hash-tree proof of consistency for every memory access made by M (the precise structure of the transcript is described below). We then describe the algorithm TVer. The running time of TVer and TGen is proportional to the running time of the RAM computation (up to polynomial factors in the security parameter) and is independent of the size of the memory. In terms of soundness we argue that for any $(M, x, \mathsf{T}, \mathsf{rt})$ (even if rt is not computed honestly as the hash-tree root of some memory) and for every $(y', \mathsf{rt}'_{\mathsf{new}}) \neq (y, \mathsf{rt}_{\mathsf{new}})$, any cheating prover that passes

[7] This blowup is due to the soundness amplification lemma of [KRR14].

the no-signaling local soundness criterion for the computation of TVer with both the input $(M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}})$ and the input $(M, x, \mathsf{T}, \mathsf{rt}, y', \mathsf{rt}'_{\mathsf{new}})$ can be used to break the soundness of the hash tree.

Let M be a RAM program, $x \in \{0,1\}^m$ be an input, and $D \in \{0,1\}^n$ be an initial memory string. Let

$$(\mathsf{HT.Gen}, \mathsf{HT.Hash}, \mathsf{HT.Read}, \mathsf{HT.Write}, \mathsf{HT.VerRead}, \mathsf{HT.VerWrite})$$

be a hash-tree scheme and let

$$\begin{aligned} \mathsf{key} &\leftarrow \mathsf{HT.Gen}(1^k), \\ (\mathsf{tree}, \mathsf{rt}) &\leftarrow \mathsf{HT.Hash}(\mathsf{key}, D). \end{aligned}$$

The Transcript Generation Program TGen. We start by describing a program TGen that creates the transcript of the computation $M^D(x)$. Let

$$\mathsf{TGen}^{(\mathsf{tree} \to \mathsf{tree}_{\mathsf{new}})}(1^k, 1^{\mathsf{T}}, n, M, x) \to (y, \mathsf{rt}_{\mathsf{new}}, \mathsf{Trans})$$

be the following RAM program. TGen emulates the execution of $M^D(x)$ step by step as described in Sect. 2.2. The emulation begins with the initial memory containing the hash tree $\mathsf{tree}_1 = \mathsf{tree}$ with the initial root $\mathsf{rt}_1 = \mathsf{rt}$, the empty initial state state_1 and the read location $i^{\mathsf{r}}_1 = 1$. Starting from $j = 1$, the j-th emulation step proceeds as follows:

1. Read from the hash tree the bit:

$$(b^{\mathsf{r}}_j, \mathsf{pf}^{\mathsf{r}}_j) \leftarrow \mathsf{HT.Read}^{\mathsf{tree}_j}(i^{\mathsf{r}}_j).$$

2. Compute $(\mathsf{state}_{j+1}, i^{\mathsf{r}}_{j+1}, i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}) \leftarrow \mathsf{STEP}(M, x, \mathsf{state}_j, b^{\mathsf{r}}_j)$.
3. If $i^{\mathsf{w}}_{j+1} \neq \bot$, write a bit to the hash tree:

$$(\mathsf{rt}_{j+1}, \mathsf{pf}^{\mathsf{w}}_{j+1}) \leftarrow \mathsf{HT.Write}^{(\mathsf{tree}_j \to \mathsf{tree}_{j+1})}(i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}).$$

The program M terminates after T emulation steps were completed with the terminating state $\mathsf{state}_{\mathsf{T}+1}$, which contains the output bit y. TGen then outputs y, $\mathsf{rt}_{\mathsf{new}} = \mathsf{rt}_{\mathsf{T}+1}$ and the transcript:

$$\mathsf{Trans} = \left(\left(i^{\mathsf{r}}_j, b^{\mathsf{r}}_j, \mathsf{pf}^{\mathsf{r}}_j\right), \left(i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}, \mathsf{rt}_{j+1}, \mathsf{pf}^{\mathsf{w}}_{j+1}\right) \right)_{j \in [\mathsf{T}]}.$$

The running time of the program TGen is $\mathsf{T} \cdot \mathrm{poly}(k)$.
The transcript verification program TVer. Let

$$\mathsf{TVer}((M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}}), \mathsf{Trans}) \to b$$

be the following Turing machine. TVer verifies the emulation of $M^D(x)$ based on the transcript:

$$\mathsf{Trans} = \left(\left(i^{\mathsf{r}}_j, b^{\mathsf{r}}_j, \mathsf{pf}^{\mathsf{r}}_j\right), \left(i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}, \mathsf{rt}_{j+1}, \mathsf{pf}^{\mathsf{w}}_{j+1}\right) \right)_{j \in [\mathsf{T}']},$$

produced by TGen. The program first verifies that $\mathsf{T}' = \mathsf{T}$ Then, starting from the initial root $\widetilde{\mathsf{rt}_1} = \mathsf{rt}$, the empty initial state state_1, the read location $\widetilde{i^{\mathsf{r}}_1} = 1$, and from $j = 1$, the j-th verification step proceeds as follows:

1. Verify that $\widetilde{i^{\mathsf{r}}_j} = i^{\mathsf{r}}_j$ and that

$$1 = \mathsf{HT.VerRead}(\widetilde{\mathsf{rt}_j}, i^{\mathsf{r}}_j, b^{\mathsf{r}}_j, \mathsf{pf}^{\mathsf{r}}_j).$$

2. Compute $(\mathsf{state}_{j+1}, \widetilde{i^{\mathsf{r}}_{j+1}}, \widetilde{i^{\mathsf{w}}_{j+1}}, \widetilde{b^{\mathsf{w}}_{j+1}}) \leftarrow \mathsf{STEP}(M, x, \mathsf{state}_j, b^{\mathsf{r}}_j)$.
3. Verify that $(\widetilde{i^{\mathsf{w}}_{j+1}}, \widetilde{b^{\mathsf{w}}_{j+1}}) = (i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1})$.
4. If $i^{\mathsf{w}}_{j+1} = \perp$ then verify that $\widetilde{\mathsf{rt}_j} = \mathsf{rt}_{j+1}$. Else, verify that

$$1 = \mathsf{HT.VerWrite}(\widetilde{\mathsf{rt}_j}, i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}, \mathsf{rt}_{j+1}, \mathsf{pf}^{\mathsf{w}}_{j+1}).$$

5. If $j = \mathsf{T}$ verify that $\mathsf{rt}_{\mathsf{T}+1} = \mathsf{rt}_{\mathsf{new}}$ and that $\mathsf{state}_{\mathsf{T}+1}$ is terminating and includes the output y.
6. $\widetilde{\mathsf{rt}_{j+1}} \leftarrow \mathsf{rt}_{j+1}$.

The program outputs 1 if and only if all the verifications were successful. The running time of the program TVer is $\mathsf{T} \cdot \mathrm{poly}(k)$ and its space complexity is $\mathrm{poly}(k) \cdot \mathrm{polylog}(n) = \mathrm{poly}(k)$ (see Remark 2.1).

Additional structure of TVer. In order to prove Theorem 4.1 below, we make additional assumptions on the structure of the Turing machine TVer. We start by introducing some notation.

Verification Blocks. We assume that the execution of the machine can be divided into *blocks* where the computation in the j-th block is executing the j-th verification step. This assumption is satisfied by some "natural" implementation of TVer.

Formally, let $b = b(k) \leq \mathrm{poly}(k)$ be the block size. For every input $\tilde{x} = (M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}})$ and for every transcript

$$\mathsf{Trans} = \left(\left(i^{\mathsf{r}}_j, b^{\mathsf{r}}_j, \mathsf{pf}^{\mathsf{r}}_j \right), \left(i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}, \mathsf{rt}_{j+1}, \mathsf{pf}^{\mathsf{w}}_{j+1} \right) \right)_{j \in [\mathsf{T}]},$$

(not neccessarily such that $\mathsf{TVer}(\tilde{x}, \mathsf{Trans})$ accepts) let $\mathsf{T}' = \mathsf{T} \cdot b$ be the running time of $\mathsf{TVer}(\tilde{x}, \mathsf{Trans})$. For $i \in [\mathsf{T}']$ let st_i be the state of the computation $\mathsf{TVer}(\tilde{x}, \mathsf{Trans})$ immediately before the i-th step, and let $\mathsf{st}_{\mathsf{T}'+1}$ be the final state of the computation. The variables st_i describe the states of the computation of the program TVer, as defined by Theorem 3.1. (Note that these states are different from the local variables state_j used by the program TVer to emulate the RAM computation M.) For $j \in [\mathsf{T}]$, let B_j be the set of states in the j-th computation block.

$$B_j = \{\mathsf{st}_i : (j-1) \cdot b < i \leq j \cdot b\}.$$

For notational convenience, we also define the block $B_{\mathsf{T}+1} = \{\mathsf{st}_{\mathsf{T}'+1}\}$ which describes the state of the computation after the final verification stap.

Additional requirements on the structure of TVer. Using the notion of blocks we formulate some additional requirements on the structure of TVer.

1. For every $j \in [\mathsf{T}]$, the bits of the transcript read in the j-th computation block contain the j-th entry of the transcript. Formally, there exists an efficient algorithm $\mathsf{TVer.Transcript}$ such that given the set of states B_j, outputs the j-th entry of the transcript

$$\left(i^{\mathsf{r}}_j, b^{\mathsf{r}}_j, \mathsf{pf}^{\mathsf{r}}_j\right), \left(i^{\mathsf{w}}_{j+1}, b^{\mathsf{w}}_{j+1}, \mathsf{rt}_{j+1}, \mathsf{pf}^{\mathsf{w}}_{j+1}\right) = \mathsf{TVer.Transcript}(B_j).$$

   We also require that $\perp = \mathsf{TVer.Transcript}(B_{\mathsf{T}+1})$.
2. For every $j \in [\mathsf{T}]$, the j-th computation block contains the j-th state in the emulation of M. Formally, there exists an efficient algorithm $\mathsf{TVer.State}$ such that given the set of states B_j, outputs the state of M, the location of the next read and the root of the hash-tree before the j-th step of the emulation

$$\left(\mathsf{state}_j, \widetilde{i^{\mathsf{r}}_j}, \widetilde{\mathsf{rt}_j}\right) = \mathsf{TVer.State}(B_j).$$

   On the final block $B_{\mathsf{T}+1}$, $\mathsf{TVer.State}$ outputs the terminating state of M, the last read location (TVer never reads the bit in this location), and the root of the final memory state.

$$\left(\mathsf{state}_{\mathsf{T}+1}, \widetilde{i^{\mathsf{r}}_{\mathsf{T}+1}}, \widetilde{\mathsf{rt}_{\mathsf{T}+1}}\right) = \mathsf{TVer.State}(B_{\mathsf{T}+1}).$$

3. When one of the tests performed by TVer fails, the machine transitions into a "rejecting state". Once TVer is in a rejecting state, we require that all its future states are rejecting and TVer rejects. Formally, there exists an efficiently computable predicate Reject such that
   (a) If in the j-th verification step test 1, 3 or 4 fails, or if $j = \mathsf{T}$ and test 5 fails, then $\mathsf{Reject}(B_j) = 1$.
   (b) For every $j \in [\mathsf{T}]$ if $\mathsf{Reject}(B_j) = 1$ then $\mathsf{Reject}(B_{j+1}) = 1$.
   (c) The computation $\mathsf{TVer}(\tilde{x}, \mathsf{Trans})$ rejects if and only if $\mathsf{Reject}(B_{\mathsf{T}+1}) = 1$.

Theorem 4.1. *The machines* TGen *and* TVer *satisfy the following properties:*

– *Completeness. For every* $k \in \mathbb{N}$*, every* T*-time RAM program* M*, every input* $x \in \{0,1\}^m$*, every initial memory* $D \in \{0,1\}^n$ *and every* (y, D_{new}) *such that* $\mathsf{T}(m), n \leq 2^k$ *and* $y \leftarrow M^{(D \to D_{\mathsf{new}})}(x)$

$$\Pr\left[\begin{array}{l|l} 1 = \mathsf{TVer}((M, x, \mathsf{T}(m), \mathsf{rt}, y', \mathsf{rt}'_{\mathsf{new}}), \mathsf{Trans}) & \\ (y', \mathsf{rt}'_{\mathsf{new}}) = (y, \mathsf{rt}_{\mathsf{new}}) & \\ \mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k) & \\ (\mathsf{tree}, \mathsf{rt}) \leftarrow \mathsf{HT.Hash}(\mathsf{key}, D) & \\ (\mathsf{tree}_{\mathsf{new}}, \mathsf{rt}_{\mathsf{new}}) \leftarrow \mathsf{HT.Hash}(\mathsf{key}, D_{\mathsf{new}}) & \\ (y', \mathsf{rt}'_{\mathsf{new}}, \mathsf{Trans}) \leftarrow \mathsf{TGen}^{(\mathsf{tree} \to \mathsf{tree}_{\mathsf{new}})}(1^k, 1^{\mathsf{T}(m)}, n, M, x) & \end{array}\right] = 1.$$

– *Soundness*
Assume HT *is an* (S, ϵ)*-secure hash-tree scheme for a function* $S(k)$ *and a negligible function* $\epsilon(k)$*. There exists a polynomial* ℓ' *such that for every constant* $c > 0$ *and every triplet of adversaries* $(\mathsf{Adv}_1, \mathsf{Adv}_2, \mathsf{Adv}_3)$ *of size* $S(k)^c$*, there exist constants* $c_1, c_2 > 0$ *such that for every large enough* $k \in \mathbb{N}$

$$\Pr\left[\begin{matrix}(y, \mathsf{rt}_{\mathsf{new}}) \neq (y', \mathsf{rt}'_{\mathsf{new}}) \\ \mathsf{CHEAT}\end{matrix} \;\middle|\; \begin{matrix}\mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k) \\ (M, x, 1^{\mathsf{T}}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}}, y', \mathsf{rt}'_{\mathsf{new}}) \leftarrow \mathsf{Adv}_1(1^k, \mathsf{key})\end{matrix}\right] \leq \epsilon(k)^{c_2},$$

where CHEAT *is the event that:*

- $\mathsf{Adv}_2(\mathsf{key}, \cdot)$ *is an* $(\ell'(k), S(k)^{-c_1}, S(k)^{-c_1})$*-local assignment generator for the 3-CNF formula* $\varphi_{\mathsf{TVer}, \tilde{x}_2}$ *where* $\tilde{x}_2 = (M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}})$ *and* $\varphi_{\mathsf{TVer}, \tilde{x}}$ *is as defined in Theorem 3.1.*
- $\mathsf{Adv}_3(\mathsf{key}, \cdot)$ *is an* $(\ell'(k), S(k)^{-c_1}, S(k)^{-c_1})$*-local assignment generator for the 3-CNF formula* $\varphi_{\mathsf{TVer}, \tilde{x}_3}$ *where* $\tilde{x}_3 = (M, x, \mathsf{T}, \mathsf{rt}, y', \mathsf{rt}'_{\mathsf{new}})$ *and* $\varphi_{\mathsf{TVer}, \tilde{x}'}$ *is as defined in Theorem 3.1.*

The proof of Theorem 4.1 can be found in the full version of this work [KP15].

4.2 The Protocol

In this section we describe our no-signaling multi-prover argument for RAM computations. The construction uses the following components.

– A hash-tree scheme $(\mathsf{HT.Gen}, \mathsf{HT.Hash}, \mathsf{HT.Read}, \mathsf{HT.Write}, \mathsf{HT.VerRead}, \mathsf{HT.VerWrite})$, given by Theorem 2.1.
– The ℓ-prover proof system $(\mathsf{LS.QueryGen}, \mathsf{LS.Prover}, \mathsf{LS.Verifier})$ for local satisfiability given by Theorem 3.2 in Sect. 3.3, where $\ell = \ell' \cdot \ell_0$, and ℓ' is the polynomial given by Theorem 4.1 and ℓ_0 is the polynomial given by Theorem 3.2.
– The transcript generation and verification programs $\mathsf{TGen}, \mathsf{TVer}$ described in Sect. 4.1. We only rely on the following facts
  - The programs $\mathsf{TGen}, \mathsf{TVer}$ satisfy Theorem 4.1.
  - For security parameter k and for a T-time computation, the running time of the transcript generation program TGen is $\mathsf{T} \cdot \mathrm{poly}(k)$. The running time of the transcript verification program TVer (on the transcript generated by TGen) is $\mathsf{T} \cdot \mathrm{poly}(k)$ and its space complexity is $\mathrm{poly}(k)$.

The multi-prover argument is given by the following procedures:

– $\mathsf{ParamGen}(1^k)$ generates a key for the hash-tree:

$$\mathsf{key} \leftarrow \mathsf{HT.Gen}(1^k),$$

and outputs $\mathsf{pp} = \mathsf{key}$.
– $\mathsf{MemGen}(\mathsf{pp}, D)$, given $\mathsf{pp} = \mathsf{key}$, computes a hash-tree for the memory D:

$$(\mathsf{tree}, \mathsf{rt}) \leftarrow \mathsf{HT.Hash}(\mathsf{key}, D),$$

and outputs $(\mathsf{dt}, \mathsf{d}) = (\mathsf{tree}, \mathsf{rt})$.

– $\mathsf{QueryGen}(1^k)$ executes the query generation algorithm of the local-satisfiability proof system:

$$((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st}) \leftarrow \mathsf{LS.QueryGen}(1^k),$$

and outputs $((\mathsf{q}_1, \ldots, \mathsf{q}_\ell), \mathsf{st})$.
– $\mathsf{Output}^{\mathsf{dt}}(1^{\mathsf{T}}, n, M, x)$, given access to the memory $\mathsf{dt} = \mathsf{tree}$, executes the transcript generation program:

$$(y, \mathsf{rt}_{\mathsf{new}}, \mathsf{Trans}) \leftarrow \mathsf{TGen}^{(\mathsf{tree} \to \mathsf{tree}_{\mathsf{new}})}(1^k, 1^{\mathsf{T}}, n, M, x),$$

and outputs $(y, \mathsf{d}_{\mathsf{new}}, \mathsf{Trans}) = (y, \mathsf{rt}_{\mathsf{new}}, \mathsf{Trans})$.
– $\mathsf{Prover}((M, x, \mathsf{T}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{Trans}, \mathsf{q})$, where $(\mathsf{d}, \mathsf{d}_{\mathsf{new}}) = (\mathsf{rt}, \mathsf{rt}_{\mathsf{new}})$, does the following:
  1. Let $\mathsf{T}' = \mathsf{T} \cdot \mathrm{poly}(k)$ and $\mathsf{S}' = \mathrm{poly}(k)$ be the time and space complexity of the computation
  $$\mathsf{TVer}((M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}}), \mathsf{Trans}).$$
  2. Execute the local-satisfiability prover for the above computation:
  $$\mathsf{a} \leftarrow \mathsf{LS.Prover}(1^{\mathsf{T}'}, \mathsf{TVer}, (M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}}), \mathsf{Trans}, \mathsf{q}).$$
  3. Output a.
– $\mathsf{Verifier}((M, x, \mathsf{T}, \mathsf{d}, y, \mathsf{d}_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell))$, where $(\mathsf{d}, \mathsf{d}_{\mathsf{new}}) = (\mathsf{rt}, \mathsf{rt}_{\mathsf{new}})$, executes the local-satisfiability verifier:

$$b \leftarrow \mathsf{LS.Verifier}(\mathsf{TVer}, (M, x, \mathsf{T}, \mathsf{rt}, y, \mathsf{rt}_{\mathsf{new}}), \mathsf{st}, (\mathsf{a}_1, \ldots, \mathsf{a}_\ell)),$$

and outputs b.

Theorem 4.2. *Assume* HT *is an* (S, ϵ)*-secure hash-tree scheme for a function* $S(k)$ *and a negligible function* $\epsilon(k)$*. Then* $(\mathsf{ParamGen}, \mathsf{MemGen}, \mathsf{QueryGen}, \mathsf{Output}, \mathsf{Prover}, \mathsf{Verifier})$ *is an* ℓ*-prover argument system for RAM computations that is* (S, ϵ)*-secure against* δ*-no-signaling provers for* $\delta(k) = 2^{-k \cdot \mathrm{polylog}(S(k))}$.

The proof of Theorem 4.2 follows by combining Theorems 3.2 and 4.1 and can be found in the full version of this work [KP15].

References

[ACC+15] Ananth, P., Chen, Y.-C., Chung, K.-M., Lin, H., Lin, W.-K.: Delegating RAM computations with adaptive soundness and privacy. IACR Cryptology ePrint Archive, 2015:1082 (2015)

[BCC+14] Bitansky, N., Canetti, R., Chiesa, A., Goldwasser, S., Lin, H., Rubinstein, A., Tromer, E.: The hunting of the SNARK. IACR Cryptology ePrint Archive, 2014:580 (2014)

[BCCT13] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for snarks and proof-carrying data. In: STOC, pp. 111–120 (2013)

[BCI+13] Bitansky, N., Chiesa, A., Ishai, Y., Paneth, O., Ostrovsky, R.: Succinct non-interactive arguments via linear interactive proofs. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 315–333. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36594-2_18

[BEG+91] Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: 32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1–4 October 1991, pp. 90–99 (1991)

[BGL+15] Bitansky, N., Garg, S., Lin, H., Pass, R., Telang, S.: Succinct randomized encodings and their applications. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14–17, 2015, pp. 439–448 (2015)

[BHK16] Brakerski, Z., Holmgren, J., Kalai, Y.T.: Non-interactive RAM and batch NP delegation from any PIR. Electron. Colloquium Comput. Complex. (ECCC) **23**, 77 (2016)

[CCC+15] Chen, Y.-C., Chow, S.S.M., Chung, K.-M., Lai, R.W.F., Lin, W.-K., Zhou, H.-S.: Computation-trace indistinguishability obfuscation and its applications. IACR Cryptology ePrint Archive,2015:406 (2015)

[CCHR15] Canetti, R., Chen, Y., Holmgren, J., Raykova, M.: Succinct adaptive garbled ram. Cryptology ePrint Archive, Report 2015/1074 (2015). http://eprint.iacr.org/

[CH15] Canetti, R., Holmgren, J.: Fully succinct garbled RAM. IACR Cryptology ePrint Archive, 2015:388 (2015)

[CHJV15] Canetti, R., Holmgren, J., Jain, A., Vaikuntanathan, V.: Succinct garbling and indistinguishability obfuscation for RAM programs. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14–17, 2015, pp. 429–437 (2015)

[CKLR11] Chung, K.-M., Kalai, Y.T., Liu, F.-H., Raz, R.: Memory delegation. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 151–168. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22792-9_9

[DFH12] Damgård, I., Faust, S., Hazay, C.: Secure two-party computation with low communication. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 54–74. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28914-9_4

[GGPR13] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38348-9_37

[GGR15] Goldreich, O., Gur, T., Rothblum, R.: Proofs of proximity for context-free languages and read-once branching programs. Electron. Colloquium Comput. Complex. (ECCC) **22**, 24 (2015)

[GHRW14] Gentry, C., Halevi, S., Raykova, M., Wichs, D.: Outsourcing private RAM computation. In: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18–21, 2014, pp. 404–413 (2014)

[GR15] Gur, T., Rothblum, R.D.: Non-interactive proofs of proximity. In: Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11–13, 2015, pp. 133–142 (2015)

[Gro10] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17373-8_19

[GVW15] Gorbunov, S., Vaikuntanathan, V., Wichs, D.: Leveled fully homomorphic signatures from standard lattices. In: Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14–17, 2015, pp. 469–477 (2015)

[Kil92] Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: Proceedings of the 24th Annual ACM Symposium on Theory of Computing, pp. 723–732 (1992)

[KP15] Kalai, Y.T., Paneth, O.: Delegating RAM computations. IACR Cryptology ePrint Archive, 2015:957 (2015)

[KR15] Kalai, Y.T., Rothblum, R.D.: Arguments of proximity (extended abstract). In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 422–442. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_21

[KRR14] Kalai, Y.T., Raz, R., Rothblum, R.D.: How to delegate computations: the power of no-signaling proofs. In: Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014, pp. 485–494 (2014)

[Lip12] Lipmaa, H.: Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. In: Cramer, R. (ed.) TCC 2012. LNCS, vol. 7194, pp. 169–189. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28914-9_10

[Mer87] Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) CRYPTO 1987. LNCS, vol. 293, pp. 369–378. Springer, Heidelberg (1988). doi:10.1007/3-540-48184-2_32

[Mic94] Micali, S.: CS proofs (extended abstracts). In': 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20–22 November 1994, pp. 436–453 (1994)

[PF79] Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM **26**(2), 361–381 (1979)

[PR14] Paneth, O., Rothblum, G.N.: Publicly verifiable non-interactive arguments for delegating computation. Cryptology ePrint Archive, Report 2014/981 (2014). http://eprint.iacr.org/

[RVW13] Rothblum, G.N., Vadhan, S.P., Wigderson, A.: Interactive proofs of proximity: delegating computation in sublinear time. In: Symposium on Theory of Computing Conference, STOC 2013, Palo Alto, CA, USA, June 1–4, 2013, pp. 793–802 (2013)