Sublinear Zero-Knowledge Arguments for RAM Programs

Payman Mohassel[1], Mike Rosulek[2(✉)], and Alessandra Scafuro[3]

[1] Visa Research, Palo Alto, USA
pmohasse@visa.com
[2] Oregon State University, Corvallis, USA
rosulekm@eecs.oregonstate.edu
[3] North Carolina State University, Raleigh, USA
ascafur@ncsu.edu

Abstract. We describe a new succinct zero-knowledge argument protocol with the following properties. The prover commits to a large data-set M, and can thereafter prove many statements of the form $\exists w : \mathcal{R}_i(M, w) = 1$, where $\mathcal{R}_i$ is a public function. The protocol is *succinct* in the sense that the cost for the verifier (in computation & communication) does not depend on $|M|$, not even in any initialization phase In each proof, the computation/communication cost for *both* the prover and the verifier is proportional only to the running time of an oblivious RAM program implementing $\mathcal{R}_i$ (in particular, this can be sublinear in $|M|$). The only costs that scale with $|M|$ are the computational costs of the prover in a one-time initial commitment to M.

Known sublinear zero-knowledge proofs either require an initialization phase where the work of the verifier is proportional to $|M|$ and are therefore sublinear only in an amortized sense, or require that the computational cost for the prover is proportional to $|M|$ upon *each proof*.

Our protocol uses efficient crypto primitives in a black-box way and is UC-secure in the *global*, non-programmable random oracle, hence it does not rely on any trusted setup assumption.

1 Introduction

A zero-knowledge proof (or argument) allows a prover to convince a verifier that a statement $\exists w : R(w) = 1$ is true, without revealing anything about the witness w. In this work we study the problem of zero-knowledge proofs concerning large datasets. For example, suppose Alice holds a large collection of files, and wants to prove that there is a file in her collection whose SHA3-hash equals some public value.

Most techniques for zero-knowledge proofs are a poor fit for proving things about large data, since they scale at least linearly with the size of the witness. For realistically large data, it is necessary to adopt methods that have sublinear

M. Rosulek—Partially supported by NSF awards 1149647 & 1617197.

J.-S. Coron and J.B. Nielsen (Eds.): EUROCRYPT 2017, Part I, LNCS 10210, pp. 501–531, 2017.
DOI: 10.1007/978-3-319-56620-7_18

cost. There are several existing techniques for zero-knowledge proofs/arguments that have sublinear cost:

PCP Techniques: Kilian [27] and Micali [30] were the first to describe proof systems in which the verifier's cost is sublinear. The technique makes use of probabilistically checkable proofs (PCPs), which are proofs that can be verified by inspecting only a small (logarithmic) number of positions. Followup work has focused on improving the performance of the underlying PCP systems [4,6,9]. Besides the fact that constructing a PCP proof is still quite an inefficient procedure, the main drawback of the PCP approach is that if the prover wants to prove many statements about a single dataset M, he/she must expend effort proportional to $|M|$, *for each proof.*

SNARKs: Succinct non-interactive arguments of knowledge (SNARKs) [8,10,11,17] are the most succinct style of proof to-date. In the most efficient SNARKs, the verifier only processes a constant number of group elements. Born as a theoretically intriguing object that pushed the limit of proof length to the extreme, SNARKs have won the attention of the practical community [7,8,13,33] after an open-source library (libsnark [1]) was created, proving the concrete efficiency of such approach and resulting in its use in real-world applications such as Zerocash [5]. However, similar to the PCP approach, the main drawback of SNARKs is that each proof requires work for the prover that is proportional to the size of the dataset. Moreover, while SNARKs do require a trusted CRS, they are not directly compatible with the UC-framework due to their use of non black-box knowledge extraction (A recent work [28] put forward "snark-lifting" techniques to upgrade SNARKS into UC-secure NIZK. This transformation however results in zero-knowledge proofs whose sizes are linear in the witness instead of constant as in regular SNARKs).

Oblivious RAM: A recent trend in secure computation is to represent computations as RAM programs rather than boolean circuits [2,22,24]. This leads to protocols whose cost depends on the running time of the RAM program (which can be sublinear in the data size). Looking more closely, however, the RAM program must be an *oblivious* RAM. An inherent feature of oblivious RAM programs is that there must be an initialization phase in which every bit of memory is touched. In existing protocols, this initialization phase incurs linear cost for *all parties*. Therefore, RAM-based protocols are sublinear only in an *amortized* sense, as they incur an expensive setup phase with cost proportional to the data size.

Our Results. We construct a zero-knowledge argument based on RAM programs, with the following properties:

- A prover can commit to a large (private) dataset M, and then prove many statements of the form $\exists w_i : \mathcal{R}_i(M, w_i) = 1$, for public $\mathcal{R}_i$.
- The phase in which the prover commits to M has $|M|$ computation cost for the prover. This is the only phase in which the prover's effort is linear in M, but this effort can be reused for many proofs. Unlike prior ZK proofs

based on RAM programs [24], the cost to the verifier (in communication & computation) is constant in this initial phase. Unlike other approaches based on PCPs & SNARKs, the expensive step for the prover can be reused for many proofs about the same data.
– The communication/computation cost for both parties in each proof is proportional to the running time of a (oblivious) RAM program implementing $\mathcal{R}_i$. In particular, if $\mathcal{R}_i$ is sublinear in $|M|$, then the verifier's cost is sublinear. In succinct proofs based on PCP/SNARKs, on the other hand, computation cost for the prover is always proportional to $|M|$.
– The protocol is proven UC-secure based only on a *global*, non-programmable random oracle. In particular, there are no trusted setup assumptions.

On Non-standard Assumptions. Our protocol uses a non-programmable random oracle. We point out that if one wishes to achieve UC security in a succinct protocol, then some non-standard-model assumption is required. In particular, the simulator must be able to extract the dataset M of a corrupt prover during the commitment phase. In the standard model, this would require the prover to send at least $|M|$ bits of data in the protocol.[1]

A global (in the sense of [12]), non-programmable random oracle is arguably the mildest non-standard-model assumption. We point out that SNARKs also use non-standard-model assumptions, such as the knowledge of exponent assumptions (KEA), which are incompatible with the UC framework [28].

2 Our Techniques

Our goal is to construct ZK proofs where the overhead of the verifier does not depend on $|M|$, not even in the initialization phase. Moreover we insist the computational overhead for P when computing a proof is proportional only to the running time of the RAM program representing $\mathcal{R}(M, w)$, and not on $|M|$. The latter requirement immediately rules out any circuit-based approach, such as PCP proof, or SNARKs where the relation $\mathcal{R}(M, w)$ is unrolled into a boolean circuit of size at least $|M|$.

Towards achieving complexity that is proportional only to the running time of $\mathcal{R}$, the starting point is to represent $\mathcal{R}$ as a (oblivious) RAM program. An oblivious RAM [31] is a RAM program whose *access pattern* (i.e., the set $\mathcal{I}$ of memory addresses accessed, along with whether the accesses are reads or writes) leaks nothing about the private intermediate values of the computation. The transformation from an arbitrary RAM computation to an oblivious one incurs a small polylogarithmic overhead in running time and in the size of the memory. However, once the memory is in an ORAM-suitable format, it can be persistently reused for many different ORAM computations.

Hu et al. [24] provide a ZK proof of knowledge protocol for RAM programs that is sublinear in the amortized sense: the protocol has an initial setup phase

[1] Work that pre-dates the UC security model avoids this problem by using a simulator that rewinds the prover — a technique that is not possible in the UC model.

in which both parties expend effort proportional to $|M|$. After this initialization phase, each proof of the form "$\exists w : \mathcal{R}(M, w) = 1$" has cost (for both parties) proportional only to the running time of $\mathcal{R}(M, w)$. There are other works [2,15,16,18,29] that can be used to construct malicious-secure two-party computation of general functionalities based on RAM programs. Compared to [24], these other techniques are overkill for the special case of ZK functionalities. All of these techniques result in sublinear performance only in the amortized sense described above.

Our goal is to achieve similar functionality as [24] without expensive effort by the verifier in the initialization phase. Looking more closely at the initialization phase of [24], the two parties engage in a secure two-party protocol where they jointly compute a shared representation of each block of M (specifically, a garbled sharing, where the verifiers has values l_0, l_1 for each bit, while the prover learns l_b if the corresponding bit of M is b).

Towards removing the verifier's initial overhead, a natural approach is to remove the participation of V in the setup phase, and have P commit succinctly to the memory using a **Merkle Tree.** Then later in the proof phase, P can prove that the RAM program accepts when executed on the values stored within the Merkle Tree.

Technical Challenge (Extraction): Unfortunately, this natural approach leads to challenges in the UC model. Consider a malicious prover who convinces a verifier of some statement. For UC security, there must exist a simulator that can extract the (large) witness M. But since the main feature of this proof is that the total communication is much shorter than $|M|$, it is information-theoretically impossible for the simulator to extract M in the standard model.

Instead, we must settle for a non-standard-model assumption. We use the global random oracle (gRO) of [12], which equips the UC model with a global, non-programmable random oracle. Global here means that the same random oracle is used by all the protocol executions that are run in the world, and this framework was introduced precisely to model the real world practice of instantiating the random oracle with a single, publicly known, hash function.

A non-programmable random oracle allows the simulator to observe the queries made by an adversary. Suppose such an oracle is used as the hash function for the Merkle tree. Then the simulator can use its ability to observe an adversary's oracle queries to reconstruct the entire contents of the Merkle tree from just the root alone.

Now that the Merkle tree is constructed with a random oracle as its hash function, authenticating a value to the Merkle tree is a computation that involves the random oracle. Hence, we cannot use a standard ZK proof to prove a statement that mentions the logic of authenticating values in the Merkle tree. Any Merkle-tree authentication has to take place "in the open." Consequently, the leaves of the Merkle tree need to be revealed "in the open" for each authentication. Therefore, the leaves of the Merkle tree must not contain actual blocks of the witness, but commitments to those blocks (more specifically, UC commitments so the simulator can further extract the RAM program's memory).

Another challenge for extraction comes from the fact that the Merkle tree contains only an ORAM-ready encoding of the logical data M. A simulator can extract the contents of the Merkle tree, but must provide the corresponding *logical* data M to the ideal functionality. We therefore require an ORAM scheme with the following nonstandard *extractability* property: Namely, there should be a way to extract, from any (possibly malicious) ORAM-encoded initial memory, corresponding logical data M that "explains" the ORAM-encoded memory. We formally define this property and show that a simple modification of the Path ORAM construction [36] achieves it.

Consistency Across Multiple ORAM Executions. An oblivious RAM program necessarily performs both physical reads and writes, even if the underlying logical RAM operations are read-only. This means that each proof about the contents of M *modifies* M. Now that the verifier has no influence on the Merkle-tree commitment of M, we need a mechanism to ensure that the Merkle-tree commitment to M remains consistent across many executions of ORAM programs.

Additionally, an ORAM also requires a persistent client state, shared between different program executions. However, in our setting it suffices to simply consider a distinguished block of memory — say, $M[0]$ — as the storage for the ORAM client state.

To manage the modifications made by RAM program executions, we have the prover present commitments to both the initial value and final value of each memory block accessed by the program. The prover (A) proves that the values inside these commitments are consistent with the execution of the program; (B) authenticates the commitments of initial values to the current Merkle tree; (C) updates the Merkle tree to contain the commitments to the updated values. In this way, the verifier can be convinced that RAM program accepts, and that the Merkle tree always encodes the most up-to-date version of the memory M.

In more detail, the protocol proceeds as follows. In the **initialization phase**, the prover processes M to make it an ORAM memory. She commits individually to each block of M and places these commitments in a Merkle tree. She sends the root of the Merkle tree to the verifier.

Then (repeatedly) to prove $\mathcal{R}(M) = 1$ for an oblivious RAM program $\mathcal{R}$, the parties do the following:

1. The prover runs $\mathcal{R}$ in her head. Let I be the set of blocks that were accessed in this execution. Let $M[I]$ denote the initial values in M at those positions, and let $M'[I]$ denote the values in those positions after $\mathcal{R}$ has terminated.
2. The prover sends I to the verifier, which leaks no information if the RAM program is oblivious.
3. The prover sends the commitments to the $M[I]$ blocks which are stored in the Merkle tree. She authenticates each of them to the root of the Merkle tree.
4. The prover generates commitments to the blocks of $M'[I]$ and sends them to the verifier. She gives authenticated updates to the Merkle tree to replace the previous $M[I]$ commitments with these new ones.

5. The prover then proves in zero-knowledge that the access pattern I, the values inside the commitments to $M[I]$, and the values inside the commitments to $M'[I]$ are *consistent with an accepting execution* of $\mathcal{R}$ (i.e., $\mathcal{R}$ indeed generates access pattern I and accepts when $M[I]$ contains the values within the commitments that the prover has shown/authenticated). Importantly, the witness to this proof consists of only the openings of the commitments to $M[I]$ and $M'[I]$ and not the entire contents of M. We can instantiate this proof using any traditional (linear-time) ZK proof protocol.

Note that the cost to the prover is linear in $|M|$ in the initialization phase, although the communication cost is constant. The cost to both parties for each proof depends only on the running time of $\mathcal{R}$. Also, all Merkle-tree authentications are "in the open," so the approach is compatible with a random-oracle-based Merkle tree.

Note that ORAM computations inherently make read/write access to their memory, even if their logical computation is a read-only computation. Hence our protocol has no choice but to deal with reads and writes by the program $\mathcal{R}$. As a side effect, our protocol can be used without modification to provably perform read/write computations on a dataset.

Technical Challenge (Black-Box Use of Commitments): Since our construction will already use the global random oracle model, we would like to avoid any further setup assumptions. This means that the UC commitments in our scheme will use the random oracle.

At the same time, the last step of our outline requires a zero-knowledge proof about the contents of a commitment scheme. We therefore need a method to prove statements about the contents of commitments in a way that *treats the commitment scheme in a black-box way.*

Towards this, we borrow well-known techniques from previous work on black-box (succinct) zero-knowledge protocol [23,25,32]. Abstractly, suppose we want to commit to a value m and prove that the committed value satisfies $f(m) = 1$. A black-box commitment to m will consist of UC-secure commitments to the components of $(e_1, \ldots, e_n)$, where $(e_1, \ldots, e_n) \leftarrow \mathsf{Code}(m)$ is an encoding of m in an error correcting code. The prover uses $(e_1, \ldots, e_n)$ as a witness in a standard ZK proof that $f(\mathsf{Decode}(e_1, \ldots, e_n)) = 1$. The statement being proven does not mention commitments at all. However, we show how to modify the ZK proof so that it reveals to the verifier a random subset of the e_i components as a side-effect. The verifier can then ask the prover to open the corresponding e_i-commitments to prove that they match.

Suppose the error-correcting encoding has high minimum distance. Then in order to cheat successfully, the prover must provide a witness to the ZK proof with many e_i values that don't match the corresponding commitments. But, conditioned on the fact that enough e_i's are revealed, this would lead to a high chance of getting caught. Hence the prover is bound to use a witness in the ZK proof that coincides with the contents of the commitment.

We note that each black-box commitment is used in at most two proofs — one when that block of memory is written and another when that block of memory is read. This fact allows us to choose a coding scheme for which no information about m is revealed by seeing two random subsets of e_i's.

In summary it suffices to construct a modified ZK proof protocol that reveals a random subset of the e_i witness components. We show two instantiations:

- In the "MPC in the head" approach of [25], the prover commits to views of an imagined MPC interaction, and opens some subset of them. For example, the computation of $f(\mathsf{Decode}(e_1, \ldots, e_n))$ may be expressed as a virtual 3-party computation where each simulated party has an additive share of the e_i's. The prover commits to views of these parties and the verifier asks for some of them to be opened, and checks for consistency.
  We modify the protocol so that the prover commits not only to each virtual party's view, but also commits individually to each virtual party's *share of each* e_i. A random subset of these can also be opened (for *all* virtual parties), and the verifier can check them for consistency. Intuitively, the e_i's that are fully revealed are *bound* to the ZK proof. That is, the prover cannot deny that these e_i values were the ones actually used in the computation $f(\mathsf{Decode}(e_1, \ldots, e_n))$.
- The ZK protocol of Jawurek et al. [26] is based on garbled circuits. In fact, their protocol is presented as a 2PC protocol for the special class of functions that take input from just one party (and gives a single bit of output). This special class captures zero-knowledge, since we can express a ZK proof as an evaluation of the function $f_x(w) = R(x, w)$ for an NP-relation R, public x, and private input w from the prover. In other words, ZK is a 2PC in which the verifier has no input.
  We show that their protocol extends in a very natural way to the case of 2PC for functions of the form $f(x, y) = (y, g(x, y))$ — i.e., functions where both parties have input but one party's input is made public. Then in addition to proving that $f(\mathsf{Decode}(e_1, \ldots, e_n)) = 1$, we can let the verifier have input that chooses a public, random subset of e_i's to reveal. As above, the prover cannot deny that these are the e_i values that were actually used in the computation of $f(\mathsf{Decode}(e_1, \ldots, e_n)) = 1$.

Technical Challenge (Non-interactive UC-Commitments in the gRO*)*: In the above outline, we assume that the commitment scheme used in the construction is instantiated with a UC-secure commitment scheme in the gRO model. For our application we crucially need a UC-commitment with **non-interactive commitment phase**, meaning that a committer can compute a commitment without having to interact with the verifier. To see why this is crucial, recall that in the Setup phase the prover needs to commit to each block of the memory M using a UC-commitment. If the commitment procedure was interactive, then the verifier (who is the receiver of the commitment) will need to participate. This would lead to a linear (in $|M|$) effort required for the verifier.

Unfortunately, known UC-commitments in the gRO model [12] are interactive[2]. Therefore, as an additional contribution, in this work we design a new commitment scheme that is UC-secure in the gRO and has non-interactive commitment and decommitment. Our new commitment scheme is described in Fig. 5.

Optimal Complexity by Combining ORAM and PCP. It is possible to achieve optimal complexity (i.e., polylog$|M|$ for V and $O(T)$ for P, where T is the program's running time) by combining ORAM and PCP-based ZK proofs as follows. Upon each proof, P runs the ORAM in his head and succinctly commits to the ORAM states (using Merkle Tree, for example). Then P proves that the committed ORAM states are correct and consistent with the *committed* memory M, using PCP-based ZK. The use of PCP guarantees that V only reads a few positions of the proof, while the use of ORAM bounds the work of P to $O(T)$. Unfortunately, this approach requires a non-black-box use of the hash function, and as such it is not compatible with the use of random oracles, and does not yield itself to efficient implementation.

Note that plugging in the black-box succinct ZK proof developed in [23] would not give the desired complexity. Very roughly, this is because proving consistency of T committed positions using [23]'s techniques, requires to open at least T paths.

3 Preliminaries

3.1 The gRO model

This *global* random oracle model was introduced by Canetti et al. in [12] to model the fact that in real world random oracles are typically replaced with a single, publicly known, hash function (e.g., SHA-2) which is globally used by all protocols running in the world. The main advantage of adopting gRO, besides being consistent with the real world practice of using a global hash function, is that we are not assuming any trusted setup assumption. In order to be global, the gRO must be *non programmable*. This means that the power of the simulator lies exclusively in his ability to observe the queries made by an adversary to gRO. Therefore, when modeling a functionality in the gRO model, [12] provides a mechanism that allows the simulator for a session sid to obtain all queries to gRO that start with sid.

The global random oracle functionality $\mathcal{G}_{\mathsf{gRO}}$ of [12] is depicted Fig. 1. $\mathcal{G}_{\mathsf{gRO}}$ has the property that "leaks" to an adversary (the simulator) all the illegitimate queries[3]. The reader is referred to [12] for further details on the gRO model.

[2] Several techniques exist that construct equivocal commitments from an extractable commitment, and could be potentially adopted in the gRO model. Unfortunately, all such techniques require interaction.

[3] In each session sid, an illegitimate query to $\mathcal{G}_{\mathsf{gRO}}$ is a query that was made with an index $sid' \neq sid$.

3.2 Ideal Functionalities

We require a commitment functionality $\mathcal{F}_{t\mathsf{com}}$ for the gRO model; we defer details to the full version.

The main difference with the usual commitment functionality is that in $\mathcal{F}_{t\mathsf{com}}$, the simulator of session sid, requests the set $\mathcal{Q}_{\mathsf{sid}}$ of queries starting with prefix sid submitted to $\mathcal{G}_{\mathsf{gRO}}$.

Our final protocol realizes the zero-knowledge functionality described in Fig. 2. It captures proving recurring statements about a large memory M where M can be updated throughout the process. This functionality consists of two phases: in the **Setup** phase, the prover sends a dataset M, for a session sid. This is a one-time phase, and all subsequent proofs will be computed by $\mathcal{F}_{zk}$ over the committed dataset M. In the **Proof** phase, P simply sends the relation $\mathcal{R}_l$ that he wishes to run over the data M, and possibly a witness w. A relation can be seen as a RAM program that takes in input (M, w). The evaluation of the RAM program can cause M to be updated.

Our main protocol can be seen as a way to reduce $\mathcal{F}_{zk}$ (succinct ZK of RAM execution) to a series of smaller zero-knowledge proofs about circuits. The functionality $\mathcal{F}_{check}^{C_1,C_2}$ (Fig. 3) captures a variant of ZK proofs for boolean circuits that we require. In particular, while in standard ZK only the prover has input (the witness), in this generalization the verifier also has input, but its input will be revealed to the prover by the end of the proof. Later we show how to instantiate this functionality using either the garbled-circuit-based protocol of [26] or the MPC-in-the-head approach of [19,25].

3.3 Encoding Scheme

A pair of polynomial time algorithms $(\mathsf{Code}, \mathsf{Decode})$ is an encoding scheme with parameters (d, t, κ) if it satisfies the following properties.

- The output of Code is a vector of length κ.
- Completeness. For all messages m, $m = \mathsf{Decode}(\mathsf{Code}(m))$.
- Minimum distance: For any $m \neq m'$, the two codewords $\mathsf{Code}(m)$ and $\mathsf{Code}(m')$ are different in at least d indices.
- Error correction: For any m, and any codeword C that is different from $\mathsf{Code}(m)$ in at most $d/2$ positions, $m \leftarrow \mathsf{Decode}(C)$.
- t-Hiding. For any m, any subset of $2t$ indices of $\mathsf{Code}(m)$ information-theoretically hide m.

Let $s \in \mathbb{N}$ denote the statistical security parameter. We observe that we can use Reed-Solomon codes to obtain an encoding satisfying the above properties with $\kappa = 4s, d = 2s$, and $t = s$. To encode a message m from a finite field $\mathbb{F}$, we generate a random polynomial P of degree $2s$ over $\mathbb{F}$ such that $P(0) = m$. The codeword is the evaluation of P at $\kappa = 4s$ different points i.e. $C = (P(1), \ldots, P(4s))$. To decode a message, we use the well-known decoding algorithm of Berlekamp and Welch for Reed-Solomon codes.

Functionality $\mathcal{G}_{\mathsf{gRO}}$

Parameters: output length $\ell(\lambda)$ and a list $\bar{\mathcal{F}}$ of ideal functionality programs.

1. Upon receiving a query x, from some party $P = (\mathsf{pid}, \mathsf{sid})$ or from the adversary S do:
   - If there is a pair (x, v) for some $v \in \{0,1\}^{\ell(\lambda)}$ in the (initially empty) list $\mathcal{Q}$ of past queries, return v to P. Else, choose uniformly $v \in \{0,1\}^{\ell(\lambda)}$ and store the pair (x, v) in $\mathcal{Q}$. Return v to P.
   - Parse x as (s, x'). If $\mathsf{sid} \neq s$ then add (s, x', v) to the (initially empty) list of `illegitimate` queries for SID s, that we denote by $\mathcal{Q}_{|s}$.
2. Upon receiving a request from an instance of an ideal functionality in the list $\bar{\mathcal{F}}$, with SID s, return to this instance the list $\mathcal{Q}_{|s}$ of illegitimate queries for SID s.

Fig. 1. $\mathcal{G}_{\mathsf{gRO}}$

Functionality $\mathcal{F}_{zk}$
Parties: P and V and adversary Sim.

- **Setup.** On input $(\mathsf{sid}, \mathsf{INIT}, M)$ from P, if no previous init command has been given, then record M. Else, do nothing.
- l-th **Proof.** On input $(\mathsf{PROVE}, \mathsf{sid}|l, \mathcal{R}_l, w)$ from P: evaluate $(M', b) = \mathcal{R}_l(M, w)$. If $b = 1$ send $(\mathsf{ACCEPT}, \mathsf{sid}|l)$ to V. Set $M = M'$.
- When asked by the adversary, obtain from $\mathcal{G}_{\mathsf{gRO}}$ the list $\mathcal{Q}_{\mathsf{sid}}$ of illegitimate queries that pertain to SID sid, and send $\mathcal{Q}_{\mathsf{sid}}$ to S.

Fig. 2. $\mathcal{F}_{zk}$

Functionality $\mathcal{F}_{check}^{C_1,C_2}$
Parameterized by: Two check circuits C_1, C_2 that each take one input.
Parties: P and V and adversary Sim.

- Challenge. On input $(\mathsf{CHALLENGE}, sid, r)$ from V, if there is no previous sid and $C_2(r) = 1$, record r. Else, do nothing.
- Proof. On input $(\mathsf{PROVE}, sid, \mathbf{W})$ from P, if $C_1(\mathbf{W}) = 1$, the $\mathcal{F}_{check}$ outputs $(\mathsf{ACCEPT}, sid, (\{\mathbf{W}[i]\}_{i, r_i = 1})$ to party V and (r, sid) to P.

Fig. 3. $\mathcal{F}_{check}^{C_1,C_2}$

Hiding follows from the security of Shamir's Secret Sharing: any $t = 2s$ points on a polynomial of degree $2s$ do not leak any information about the secret $P(0)$. Minimum distance $d = 2s$ follows from the observation that if two encodings agree in more than $2s$ points, then they must in fact be the same polynomial and hence encode the same value. Error correction follows from the Berlekamp-Welch decoding algorithm, which can efficiently correct errors up to half the minimum distance.

3.4 Oblivious RAM Programs

Oblivious RAM (ORAM) programs were first introduced by Goldreich and Ostrovsky [21]. ORAM provides a wrapper that encodes a logical dataset as a physical dataset, and translates each logical memory access into a series of physical memory accesses so that the physical memory access pattern leaks nothing about the underlying logical access pattern.

Syntactically, let Π be a RAM program that operates on memory M and also takes an additional auxiliary input w. We write $(M', z) \leftarrow \Pi(M, w)$ to denote that when Π runs on memory M and input w, it modifies the memory to result in M' and outputs z.

We use M to represent the logical memory of a RAM program and $\widehat{M}$ to indicate the physical memory array in Oblivious RAM program. We consider all memory to be split into **blocks**, where $M[i]$ denotes the ith block of M.

An Oblivious RAM (wrapper) consists of algorithms $(\mathsf{RamInit}, \mathsf{RamEval})$ with the following meaning:

- $\mathsf{RamInit}$ takes a security parameter and logical memory M as input, and outputs a physical memory $\widehat{M}$ and state st.
- $\mathsf{RamEval}$ takes a (plain) RAM program Π, auxiliary input w, and state st as input, and outputs an updated memory $\widehat{M}'$, updated state st, and RAM output z.

In general these algorithms are randomized. When we wish to explicitly refer to specific randomness used in these algorithms, we write it as an additional explicit argument ω. When we omit this extra argument, it means the randomness is chosen uniformly.

Definition 1. *Let* $(\mathsf{RamInit}, \mathsf{RamEval})$ *be an ORAM scheme. For all* M *and sequences of RAM programs* $\Pi_1, \ldots, \Pi_n$ *and auxiliary inputs* $w_1, \ldots, w_n$*, and all random tapes* $\omega_0, \ldots, \omega_n$*, define the following values:*

- $\mathsf{RealOutput}(M, \Pi_1, \ldots, \Pi_n, w_1, \ldots, w_n)$*:*
  Set $M_0 = M$*. Then for* $i \in [n]$*, do* $(M_i, z_i) = \Pi_i(M_{i-1}, w_i)$*. Return* $(z_1, \ldots, z_n)$*.*
- $\mathsf{OblivOutput}(M, \Pi_1, \ldots, \Pi_n, w_1, \ldots, w_n, \omega_0, \ldots, \omega_n)$*:*
  Set $(\widehat{M}_0, st_0) = \mathsf{RamInit}(1^k, M; \omega_0)$*. Then for* $i \in [n]$*, do* $(\widehat{M}_i, st_i, z'_i) = \mathsf{RamEval}(\Pi_i, \widehat{M}_{i-1}, st_{i-1}, w_i; \omega_i)$*. Return* $(z'_1, \ldots, z'_n)$*.*

The ORAM scheme is ***correct*** *if* $\mathsf{RealOutput}(M, \Pi_1, \ldots, \Pi_n, w_1, \ldots, w_n)$ *and* $\mathsf{OblivOutput}(M, \Pi_1, \ldots, \Pi_n, w_1, \ldots, w_n, \omega_0, \ldots, \omega_n)$ *agree with overwhelming probability over choice of random* ω_i*.*

The ORAM scheme is ***sound*** *if* for all $\omega_0, \ldots, \omega_n$*, the vectors* $\mathsf{RealOutput}(M, \Pi_1, \ldots, \Pi_n, w_1, \ldots, w_n)$ *and* $\mathsf{OblivOutput}(M, \Pi_1, \ldots, \Pi_n, w_1, \ldots, w_n, \omega_0, \ldots, \omega_n)$ *disagree only in positions where the latter vector contains* $\perp$*.*

In our protocol, we allow the adversary to choose the randomness to the ORAM construction. The soundness property guarantees that the adversary cannot use this ability to falsify the output of the RAM program. At worst, the adversary can influence the probability that the RAM program aborts.

In our protocol, the simulator for a corrupt prover can extract only the ORAM-initialized memory $\widehat{M}$. However, the simulator must give the logical memory M to the ideal functionality. For this reason, we require an ORAM construction that is **extractable** in the following sense:

Definition 2. *An ORAM scheme* $(\mathsf{RamInit}, \mathsf{RamEval})$ *is* ***extractable*** *if there is a function* $\mathsf{RamExtract}$ *with the following property. For all (possibly maliciously generated)* $(\widehat{M}, st)$*, all* M *and sequences of RAM programs* $\Pi_1, \ldots, \Pi_n$ *and auxiliary inputs* $w_1, \ldots, w_n$ *define the following:*

- *Set* $M_0 \leftarrow \mathsf{RamExtract}(\widehat{M}, st)$*. Then for* $i \in [n]$*, do* $(M_i, z_i) = \Pi_i(M_{i-1}, w_i)$*. Return* $(z_1, \ldots, z_n)$*.*
- *Set* $(\widehat{M}_0, st_0) = (\widehat{M}, st)$*. Then for* $i \in [n]$*, do* $(\widehat{M}_i, st_i, z'_i) = \mathsf{RamEval}(\Pi_i, \widehat{M}_{i-1}, st_{i-1}, w_i)$*. Return* $(z'_1, \ldots, z'_n)$*.*

Then with overwhelming probability $z'_i \in \{z_i, \perp\}$ *for each* i*.*

In other words, $\mathsf{RamExtract}$ produces a plain RAM memory that "explains" the effect of $(\widehat{M}, st)$. The only exception is that a malicious $\widehat{M}, st$ could cause the ORAM construction to abort more frequently than a plain RAM program.

Let $\mathsf{AccessPattern}(\Pi, \widehat{M}, w, st; \omega)$ denote the **access pattern** describing the accesses to physical memory made by $\mathsf{RamEval}(\Pi, \widehat{M}, w, st; \omega)$. The access pattern is a sequence of tuples of the form (READ, id) or (WRITE, id), where id is a block index in $\widehat{M}$.

Definition 3. *We say that a scheme* $(\mathsf{RamInit}, \mathsf{RamEval})$ *is* ***secure*** *if there exists an efficent* $\mathcal{S}$ *such that, for all* M, Π*, and* w*, the following two distributions are indistinguishable:*

- *Run* $\mathcal{S}(1^k, |M|, \Pi, |w|)$*.*
- *Run* $(\widehat{M}, st) \leftarrow \mathsf{RamInit}(1^k, M)$*, then return* $\mathsf{AccessPattern}(\Pi, \widehat{M}, w, st)$*.*

In other words, the access pattern leaks no information about M *or* w*.*

Note that the output of $\mathsf{AccessPattern}$ contains only the memory *locations* and not the *contents* of memory. Hence, we do not require the ORAM construction to encrypt/decrypt memory contents — they will be protected via other mechanisms in our protocol.

Our definitions of *soundness* and *extractability* are non-standard. We discuss how to modify existing ORAM constructions to achieve these definitions in the full version.

3.5 Trapdoor Commitment

We construct UC commitments from trapdoor commitments with the following properties: (a) the trapdoor is used only to compute the decommitment, (b) knowledge of the trapdoor allows to equivocate any previously computed commitment (as long as the state z is known). Such a commitment scheme can be based on Pedersen's perfectly hiding commitment scheme [35]. Details and formal definitions for this instantiation are given in the full version.

4 Succinct Zero-Knowledge Proof for RAM Programs

4.1 Protocol Description

Overview. The protocol consists of two phases: a (one-time) setup phase, and a proof phase.

In the setup phase the prover commits to the ORAM memory $\widehat{M}$ in a black-box friendly manner. That is, for each memory location $\widehat{M}[i]$, P first computes an encoding of $\widehat{M}[i]$ resulting in shares $(x_{i,1}, \ldots, x_{i,\kappa})$, then it commits to each share $x_{i,j}$ independently, obtaining commitments $N_i = (cx_{i,1}, \ldots, cx_{i,\kappa})$. Committing to each share independently will allows the prover to later selectively open a subset of t shares. N_i is then placed in the i-th leaf of the Merkle Tree. Similarly, P will also commit to the ORAM state st used to computed $\widehat{M}$, by committing to its shares $(s_1, \ldots, s_\kappa)$. At the end of the setup phase, the verifier receives the root of the Merkle Tree, and the commitments to the encoding of the initial ORAM state.

In the l-th proof phase, the prover first runs the ORAM program corresponding to relation $\mathcal{R}_l$ in her head. From this, she will obtain the access pattern $\mathcal{I}$, the updated contents of memory, and the final ORAM state st'.

P will then commit to this information, using again the black-box friendly commitment outlined above. The verifier at this point receives the set of positions $\mathcal{I}$ as well as commitments to all the encodings. Then, to prove consistency of such computation in a black-box manner, P will invoke the $\mathcal{F}_{check}$ functionality (Fig. 3) that does the following:

1. Decode the shares received in input and reconstruct initial ORAM state st, initial memory blocks $\{\widehat{M}[i]\}$ read by the ORAM computatation, the final ORAM state st' and the updated value $\{\widehat{M}[i]\}$ of any memory blocks accessed during the ORAM computation.
2. Run the ORAM evaluation on input st and the given initial memory block. Check that the program indeed generates access pattern $\mathcal{I}$, updates the memory to the values provided, and outputs the updated state provided.
3. If the check above is successful, then output a subset of t shares from each encoding received in input.

This invocation of $\mathcal{F}_{check}$ is described in greater detail below. It checks only that the encodings provided by P lead to an accepting computation. As it is, this does

not prove anything about whether this computation is consistent with the initial memory committed in the setup phase, and with the previous proofs. To glue such encodings to the values that P has committed outside the functionality, we have P open also to a subset of t commitments. In this way, the verifier can be convinced that the values that made the $\mathcal{F}_{check}$ functionality accept are consistent with the ones committed by P.

Notation. We use upper case letters to denote vectors, while we use lower case letters to denote a string. For example, notation $Z = (z_1, \ldots, z_n)$ means that vector Z has components $z_1, \ldots, z_n$. Notation $Z[i]$ denotes the ith component of vector Z and is equivalent to value z_i. We use bold upper case to denote a collection of vectors. For example, $\mathbf{S} = \{S_1, S_2, \ldots\}$.

Moreover, in the protocol, we shall use notation X_i to denote the value of memory block i *before* the proof is computed, while we use notation Y_i to denote the value of memory block i *after* the proof. Similarly we used notation S, S' to denote the encoding of a pre-proof and post-proof ORAM state, respectively.

Let $\mathsf{UCCom} = (\mathsf{Gen}, \mathsf{Com}, \mathsf{Dec}, \mathsf{Ver})$ be a UC-secure commitment scheme that has *non-interactive* commitment and the decommitment phase. In Sect. 5 we give an instantiation of such a scheme in the gRO model. Let $(\mathsf{Code}, \mathsf{Decode})$ be an encoding scheme with parameters (d, t, κ). Let $(\mathsf{RamInit}, \mathsf{RamEval})$ be a secure ORAM scheme. Our (stateful) ZK protocol $\Pi = (\Pi.\mathsf{Setup}, \Pi.\mathsf{Proof})$ is described in Figs. 4 and 5.

Setup $\Pi.\mathsf{Setup}(M, 1^\lambda)$
Let $m = |M|$.

- Run the initialization of the UC commitment scheme to obtain parameters pk.
- Initialize ORAM. Run $(\widetilde{M}, st) \leftarrow \mathsf{RamInit}(M, 1^\lambda)$.
- Encode memory $\widetilde{M}$. For $i \in [m]$:
  (encoding) $X_i = (x_{i,1}, \ldots, x_{i,\kappa}) \xleftarrow{\$} \mathsf{Code}(\widetilde{M}[i])$.
  (commitment) $CX_i = (cx_{i,1}, \ldots, cx_{i,\kappa}) \xleftarrow{\$} \mathsf{Com}(pk, x_{i,1}), \ldots, \mathsf{Com}(pk, x_{i,\kappa})$.
  (decommitment) $DX_i = (dx_{i,1}, \ldots, dx_{i,\kappa})$
- Encode state st.
  (encoding) $S = (s_1, \ldots, s_\kappa) \xleftarrow{\$} \mathsf{Code}(st)$.
  (commitment) $CS = (cs_1, \ldots, cs_\kappa) \xleftarrow{\$} \mathsf{Com}(pk, s_1), \ldots, \mathsf{Com}(pk, s_\kappa)$.
  (decommitment) $DS = (ds_1, \ldots, ds_\kappa)$
- Build the tree. Let $d = \log m$.
  Leaves. For $i = 1, \ldots, m$. Set leaf $N_i = \mathsf{gRO}(sid, i, \mathrm{P}, CX_i)$.
  Internal nodes. For $v \in \{0,1\}^{\leq d-1}$, set $N_v = \mathsf{gRO}(sid, N_{v0} | N_{v1})$.
  Root. Let $h = N_\epsilon$.
- Publish values.
  - ◇ Commitment to the ORAM state $CS = (cs_1, \ldots, cs_\kappa)$.
  - ◇ Root of the Merkle Tree h.

Fig. 4. Setup phase

$l+1$**-th Zero-knowledge Proof.** $\Pi.\mathsf{Proof}(state)$

Public Inputs: Relation $\mathcal{R}_l = (\Pi_l, x)$. Root of the Merkle tree h, commitment to ORAM sate: $CS = (cs_1, \ldots, cs_\kappa)$.
Private Inputs for P**:** Witness w. Memory $\widetilde{M} = \widetilde{M}^l$, ORAM state $st = st^l$. Encodings, commitments and decommitment information for memory blocks and ORAM state. That is: (memory) X_i, CX_i, DX_i, for $i \in [m]$; (state) S, CS,DS.

1. Program evaluation and commitments to the updated memory/state.
   - **Program Evaluation.** P runs $(\mathcal{I}, \widetilde{M}', st') \xleftarrow{\$} \mathsf{RamEval}(\Pi_l, \widetilde{M}, x, w, st)$. Let $\mathsf{access}(\mathcal{I}) = \mathsf{read}(\mathcal{I}) \cup \mathsf{write}(\mathcal{I})$. Let $\widetilde{M}'$ denotes the final version of the memory $\widetilde{M}$.
   - **Commitment to updated memory blocks**. For $i \in \mathsf{access}(\mathcal{I})$.
     (Encoding) $Y_i = (y_{i,1}, \ldots, y_{i,\kappa}) \xleftarrow{\$} \mathsf{Code}(\widetilde{M}'[i])$.
     (Commitment) $CY_i = (cy_{i,1}, \ldots, cy_{i,\kappa}) \xleftarrow{\$} \mathsf{Com}(pk, y_{i,1}), \ldots, \mathsf{Com}(pk, y_{i,\kappa})$.
     (Decommitments) $DY_i = (dy_{i,1}, \ldots, dy_{i,\kappa}) \xleftarrow{\$} \mathsf{Dec}(cy_{i,1}), \ldots, \mathsf{Dec}(cy_{i,\kappa})$
   - **Commitment to updated state.**
     (Encoding) $S' = (s'_i, \ldots, s'_\kappa) \leftarrow \mathsf{Code}(st')$.
     (Commitment) $CS' = (cs'_1, \ldots, cs'_\kappa) \xleftarrow{\$} \mathsf{Com}(pk, s'_1), \ldots, \mathsf{Com}(pk, s'_\kappa)$.
     (Decommitments) $DS' = (ds'_1, \ldots, ds'_\kappa) \xleftarrow{\$} \mathsf{Dec}(cs'_1), \ldots, \mathsf{Dec}(cs'_\kappa)$.
   - **Update root**. Recompute the Merkle Tree root h'. Compute circuits $C_{1,\mathcal{I}}, C_{2,\mathcal{I}}$
   - Set $\mathbf{W} = (w\ S, S', \{X_i\}_{i \in \mathsf{read}(\mathcal{I})}, \{Y_i\}_{i \in \mathsf{access}(\mathcal{I})})$.
   - **Send to** V**.** Access pattern $\mathcal{I}$; commitments CY_i $(\forall i \in \mathsf{access}(\mathcal{I}))$; CS', new root h' to V.
2. $\mathcal{F}^C_{check}$
   - V sends $(\mathsf{CHALLENGE}, sid_\ell, r)$ to $\mathcal{F}^{C_1,C_2}_{check}$, for $r \xleftarrow{\$} \{0,1\}^\kappa$. P sends $(\mathsf{PROVE}, \mathsf{sid}, \mathbf{W})$ to $\mathcal{F}^{C_1,C_2}_{check}$
   - V obtains partial encodings $Y_j[\gamma]$, $X_i[\gamma]$, $S[\gamma], S'[\gamma]$ for γ s.t. $r_\gamma = 1$. P receives r.
3. **Verification.**
   - P sends: (1) decommitments $DX_i[\gamma]$ and authentication path π_i from h to N_i, $\forall i \in \mathsf{read}(\mathcal{I})$. (2)decommitments $DY_j[\gamma]$ for $j \in \mathsf{access}(\mathcal{I})$, (3) decommitments $DS[\gamma]$, $DS'[\gamma]$.
   - V checks validity[a] of path π_i, $\forall i \in [\mathsf{read}(\mathcal{I})]$, and the validity of decommitments by running procedure $\mathsf{Ver}(pk, \cdot)$.

[a] An authentication path π_i for a node N_i is a chain of hash values. In order to check validity of a path, V checks that node $N_i = \mathsf{gRO}(sid, i, \mathrm{P}, CX_i)$ and that for each internal node v, $N_v = \mathsf{gRO}(sid, N_{v0} \| N_{v1})$.

Fig. 5. Proof phase

The $\mathcal{F}^{C_1,C_2}_{check}$ Circuits

ORAM Components: Let $\mathcal{I}$ be an ORAM memory access sequence. We define $\mathsf{read}(\mathcal{I}) = \{i \mid (\textsc{read}, i) \in \mathcal{I}\}$, $\mathsf{write}(\mathcal{I}) = \{i \mid (\textsc{write}, i) \in \mathcal{I}\}$, and $\mathsf{access}(\mathcal{I}) = \mathsf{read}(\mathcal{I}) \cup \mathsf{write}(\mathcal{I})$; *i.e.*, the indices of blocks that are read/write/accessed in $\mathcal{I}$. If $S = \{s_1, \ldots, s_n\}$ is a set of memory-block indices, then we define $M[S] = (M[s_1], \ldots, M[s_n])$.

Next, we describe the exact check circuits C_1 and C_2 we need for our main protocol. The check circuit $C_{2,\mathcal{I}}(r)$ is straightforward. Given bit string r, it returns 1 if $r_\gamma = 1$ in at most t locations.

Given an ORAM access pattern $\mathcal{I}$, we let the witness W consist of the auxiliary input w and a collection of *encodings* of: the initial ORAM state S, the final ORAM state S', the input memory blocks $\mathbf{X} = (X_1, \ldots, X_{|\mathsf{read}(\mathcal{I})|})$, and the output/resulting memory blocks $\mathbf{Y} = (Y_1, \ldots, Y_{|\mathsf{access}(\mathcal{I})|})$. The check circuit $C_{1,\mathcal{I}}(\mathrm{W})$ is defined as follows:

$C_{1,\mathcal{I}}(w, S, S', \mathbf{X}, \mathbf{Y})$:

$st := \mathsf{Decode}(S)$

simulate $\mathsf{RamEval}(\Pi, \widehat{M}, st, w)$ in the following way:

whenever a block i of $\widehat{M}$ is accessed:

if $i \notin \mathsf{access}(\mathcal{I})$ then return 0

else if the access is a READ: take $\mathsf{Decode}(X_i)$ as the result of the access

else if the access is (WRITE, v), set $\widehat{M}'[i] = v$

if the above simulation of $\mathsf{RamEval}$ does not return 1, then return 0

if the above simulation does not result in access pattern $\mathcal{I}$, then return 0

if the above simulation results in ORAM state $st' \neq \mathsf{Decode}(S')$ then return 0

for $i \in \mathsf{access}(\mathcal{I})$: if $\widehat{M}'[i] \neq \mathsf{Decode}(Y_i)$ then return 0

return 1

4.2 Instantiation $\mathcal{F}_{check}$

Instantiating $\mathcal{F}_{check}^{C}$ using JKO Protocol. JKO refers to a zero-knowledge protocol of Jawurek et al. [26]. The protocol is based on garbled circuits and is quite efficient, requiring only a single garbled circuit to be sent.

We first give an overview of the JKO protocol. Abstractly, suppose the prover would like to prove knowledge of a witness w such that $R(w) = 1$, where R is a public function/circuit.

1. The verifier generates a garbled circuit implementing R. The parties then perform instances of oblivious transfer, where the verifier acts as receiver. The verifier sends the garbled inputs for the garbled circuit, and the prover picks up a garbled input encoding the witness w.
2. The verifier sends the garbled circuit and the prover evaluates it, resulting in a garbled output. Since R has a single output bit, this is a single wire label (the wire label encoding output "true", if the prover is honest). The prover commits to this garbled output.
3. The verifier opens the garbled circuit so the prover can check that it was garbled correctly. In the JKO protocol, this is done using *committed OT* in step (1). The verifier "opens" its inputs to these OTs, revealing the entire set of garbled inputs. This is enough for the prover to verify the correctness of the garbled circuit.

4. If the prover is satisfied that the circuit was garbled correctly, then she opens her commitment to the garbled output.
5. The verifier accepts the proof if the prover's commitment is opened to the "true" output wire label of the garbled circuit.

The protocol is zero-knowledge because a simulator can extract the entire set of garbled inputs from the OTs in step (1). Then the simulator can compute the "true" output wire label and commit to it instep (2).

The protocol is sound due to the *authenticity* property of the garbled circuit. Namely, given a garbled input encoding w and the garbled circuit, it should be hard to guess an output wire label other than the one encoding truth value $R(w)$. (See [3] for the formal definition) This authenticity property holds in step (2) when the prover must commit to the output wire label. After step (3), the prover can compute any garbled output for the garbled circuit, but the prover has already committed to the garbled output at that point.

Importantly, the prover is the only party with private input to the garbled circuit. But the prover plays the role of garbled circuit *evaluator*. Hence, the protocol does not use the traditional *privacy* security property of garbled circuits. This is also the reason that the same garbled circuit can be both evaluated and checked. Doing this in a more general 2PC is problematic since opening/checking a circuit would reveal the secrets of the garbled circuit's generator. In this case, that party is the verifier and has no secrets to reveal.

Modifications. With some minor modifications, the JKO protocol can be used to efficiently instantiate the $\mathcal{F}_{check}^{C_1,C_2}$ functionality. The main differences are:

- The computation gives more than a single bit output.
- The computation takes input from the verifier (r) as well as the prover. We are able to handle this in the JKO protocol paradigm because r is eventually made public to the prover.

The modified JKO protocol proceeds as follows.

1. The verifier generates a garbled circuit computing the function $\widetilde{C}(\mathbf{W}) = [\text{if } C_2(r) \text{ then } C_1(\mathbf{W}, r) \text{ else } \perp]$. The parties perform a committed OT for each input bit, in which the prover obtains garbled input encoding $\mathbf{W}$.
2. The verifier sends the garbled circuit and the prover evaluates it, resulting in a garbled encoding of the (many-bit) output $z = \widetilde{C}(\mathbf{W})$. The prover commits to the garbled output.
3. The verifier opens the committed OTs, revealing all garbled inputs. The verifier also sends r at this point. The prover can check whether the garbled circuit was generated correctly.
4. The prover, if satisfied, opens the commitment to the garbled output and sends the plain output $z = \widetilde{C}(\mathbf{W})$. The prover outputs (r, sid).
5. The verifier outputs (z, sid) if the commitment is opened to the valid garbled encoding of z.

Lemma 4. *The modified JKO protocol above is a UC-secure realization of* $\mathcal{F}_{check}^{C_1,C_2}$*, in the committed-OT + commitment hybrid model, if the underlying garbling scheme satisfies the authenticity property.*

Instantiating $\mathcal{F}_{check}^{C_1,C_2}$ using IKOS. IKOS refers to the general approach introduced in [25] for obtaining ZK proofs in the commitment-hybrid model for arbitrary NP statements, given any generic MPC protocol. Recently, Giacomelli et. al [19] explored and implemented a concrete instantiation of the IKOS approach based on the GMW protocol [20] among three parties. Their optimized construction is only slightly less efficient than the JKO protocol [26] but instead has the advantage of being a public-coin Σ protocol that can be efficiently made a non-interactive Zero-knowledge proof using the Fiat-Shamir transform.

We first recall the IKOS approach and show how we can modify it to realize the $\mathcal{F}_{check}^{C_1,C_2}$ functionality for any circuits C_1, C_2. As mentioned above, the main ingredient is a Σ protocol with special soundness and honest-verifier Zero-knowledge property:

The prover has an input W and wants to prove that $C_1(\mathrm{W}) = 1$ where C_1 can be any public circuit. Let Π be a t-private n-party MPC protocol with perfect correctness. The protocol proceeds as follows.

- Prover generates n random values $\mathbf{W}_i$ such that $\mathbf{W} = \bigoplus_{i=1}^{n} \mathbf{W}_i$.
- Prover runs (on its own) the n-party MPC Π for computing $C_1(\bigoplus_i \mathbf{W}_i)$ where party P_i's input is $\mathbf{W}_i$, and obtains the view $v_i = \mathsf{View}_{P_i}(\mathbf{W})$ for all $i \in [n]$.
- Prover commits to $v_1, \ldots, v_n$.
- Verifier chooses a random subset $E \subset [n]$ where $|E| = t$, and sends E to prover.
- Prover opens the commitment to v_e for all $e \in E$.
- Verifier checks that:
  - For all $e \in E$, v_e yields the output 1 for P_e.
  - For all $e, e' \in E$, the view of P_e and $P_{e'}$ (v_e and v'_e) are consistent.
  - If any of the checks fail it rejects. Else it accepts.

The above protocol has a soundness probability that is a function of n and t. But this probability can be easily amplified by repeating the protocol multiple times in parallel for different runs of Π and using different random challenges E each time. This parallel version remains a Σ protocol as desired.

We need to enhance the above protocol to also take a random string r satisfying $C_2(r)$ for a circuit C_2 as Verifier's input and reveal those locations in the witness $\mathrm{W}[i]$ where $r_i = 1$. The above Σ protocol can be easily extended to handle this case. We simply have the verifier send r along with E to the Prover. Prover checks that $C_2(r) = 1$ and if the case, it opens commitments $\mathbf{W}[i]$ for all i where $r_i = 1$. This is in addition to the views it opens to achieve soundness.

1. Prover generates n random values $\mathbf{W}_i$ such that $\mathbf{W} = \bigoplus_{i=1}^{n} \mathbf{W}_i$.
2. Prover runs (on its own) the n-party MPC Π for computing $C_1(\bigoplus_i \mathbf{W}_i)$ where party P_i's input is $\mathbf{W}_i$, and obtains the view $v_i = \mathsf{View}_{P_i}(\mathbf{W})$ for all $i \in [n]$.

3. Prover commits to $\mathbf{W}_1[j], \ldots, \mathbf{W}_n[j]$ for all $j \in [|\mathbf{W}|]$ and $v_1, \ldots, v_n$.
4. Verifier chooses a random subset $E \subset [n]$ where $|E| = t$, and sends E and its input r to the prover.
5. Prover aborts if $C_2(r) \neq 1$. Else it opens commitment to $\mathbf{W}_i[j]$ for all $i \in [n]$ and all j where $r_j = 1$.
6. Prover also opens the commitment to v_e for all $e \in E$ and to $\mathbf{W}_e[j]$ for all $j \in [|\mathbf{W}|]$.
7. Verifier checks that:
   (a) For all $e \in E$, the opened $\mathbf{W}_e$ and v_e are consistent, i.e. W_e is correctly embedded in v_e.
   (b) For all $e \in E$, v_e yields the output 1 for P_e.
   (c) For all $e, e' \in E$, the view of P_e and $P_{e'}$ (v_e and v'_e) are consistent.
   (d) If any of the checks fail it rejects. Else it accepts.

The above protocol is a public-coin, honest-verifier protocol. We can transform it into a zero-knowledge protocol by letting the verifier commit to his random challenge before the prover sends the first message.

Lemma 5. *The modified IKOS protocol above is a secure realization of the $\mathcal{F}_{check}^{C_1,C_2}$ functionality, when the commitments are instantiated with UC commitments.*

5 A New UC-Commitment in the gRO Model

In [12], Canetti et al. show a UC commitment scheme that is secure in the gRO model. Such a commitment scheme is based on trapdoor commitments (e.g., Pedersen's Commitment). The main idea is to have the receiver choose parameters $(\mathsf{pk}, \mathsf{sk})$ of a trapdoor commitment, have the sender commit using pk, and later, in the decommitment phase, before revealing the opening, have the receiver reveal the trapdoor sk (this is done in such a way that despite revealing sk, binding is still preserved). This trick allows to achieve equivocability without programming the RO. On the other hand, this trick has the fundamental drawback of requiring that each commitment is computed under a fresh public key pk. (To see why, note that if more than one commitment is computed under the same public key, then binding holds only if all such commitments are opened at the same time.). This is highly problematic in our setting, where the prover commits to each element of the memory, as the verifier would need to provide as many public keys as the size of the memory.

Therefore, we design a new commitment scheme in the gRO model that satisfies the crucial property that the receiver can send one public key pk at the beginning, and the sender can re-use it for all subsequent commitments.

The idea behind our new scheme is fairly simple. The receiver R will pick two public keys $(\mathsf{pk}^0, \mathsf{pk}^1)$ for a trapdoor commitment scheme. Additionally, R computes a non-interactive witness indistinguishable proof of knowledge (NIWI) π, proving knowledge of one of the secret keys sk^b. R then sets the parameters of the commitment as $pk = (\mathsf{pk}^0, \mathsf{pk}^1, \pi)$. NIWI proofs of knowledge can be constructed

from any Σ-protocol in the gRO model using the transformation of [14,34]. A self-contained description of this technique is deferred to the full version. For concrete efficiency, one can instantiate the trapdoor commitment with Pedersen's commitment. In this case the public keys are of the form $\mathsf{pk}^0 = g_0, h^{\mathsf{trap}_0}$ and $\mathsf{pk}^1 = g_1, h^{\mathsf{trap}_1}$, and proving knowledge of the secret key sk^b corresponds to simply prove knowledge of the exponent trap_b. The parameters pk so generated are used for all subsequent commitments.

To commit to a message m, S first splits m as m^0, m^1 s.t. $m = m^0 \oplus m^1$. Then S computes commitments C^0 and C^1 to m^0 and m^1 as follows.

First, commit to m^b, i.e., $c^b_{\mathsf{msg}} = \mathsf{TCom}(\mathsf{pk}^b, m^b)$ using the trapdoor commitment scheme. Then, S queries gRO with the *opening* of c^b_{msg}, and receives an answer a^b_C. At this point, S commits to the answer a^b_C using again TCom, resulting in commitment c^b_{ro}. The commitment C^b will then consist of the pair $C^b = (c^b_{\mathsf{msg}}, c^b_{\mathsf{ro}})$. Intuitively, the commitment is extractable in the gRO model since S is forced to commit to the *answer* of gRO, and hence the extractor can simply extract the decommitments by observing the queries to gRO, and checking that there exists at least a query q that corresponds to a valid opening of c^b_{msg}.

In the decommitment phase S simply opens the two commitments, and R checks that c^b_{ro} is indeed the commitment of the answer of gRO, on input the decommitment of c^b_{msg}. Note that the receiver R does not reveal any trapdoor (as she already proved knowledge of one of them), and therefore the same pk can be used again for a new commitment. To equivocate, the simulator simply extracts the trapdoor sk^b from NIWI proof π (recall that π is straight-line extractable in the gRO model), and uses it to equivocate commitments $c^b_{\mathsf{msg}}, c^b_{\mathsf{ro}}$.

We describe the protocol in more details below. Further details proving knowledge of a Pedersen commitment trapdoor are given in the full version.

Protocol UCCom. **A New UC Commitment in the** gRO **Model**. Let sid denote the session identifier.

Setup Phase $\langle \mathsf{Gen}(C(1^\lambda), R(1^\lambda)) \rangle$.

- R computes $(\mathsf{pk}^0, \mathsf{sk}^0) \leftarrow \mathsf{TCGen}(1^\lambda)$, and $(\mathsf{pk}^1, \mathsf{sk}^1) \leftarrow \mathsf{TCGen}(1^\lambda)$. R computes a NIWI proof of knowledge π for proving knowledge of sk^d for a random bit d. R sends $pk = (\mathsf{pk}^0, \mathsf{pk}^1, \pi)$ to C.
- If π is accepting, C records parameters $\mathsf{pk}^0, \mathsf{pk}^1$.

i-th Commitment Phase $\mathsf{Com}(\mathsf{sid}, i, m)$**:** C randomly picks m^0, m^1 such that $m = m^0 \oplus m^1$. Then for each m^b:

- Commit to m^b: $(c^b_{\mathsf{msg}}, d^b_{\mathsf{msg}}) \leftarrow \mathsf{TCom}(\mathsf{pk}, m^b)$.
- Query gRO on input $(\mathsf{sid}, i, \mathrm{S} \| m^b \| d^b_{\mathsf{msg}} \| s^b)$, where $s^b \xleftarrow{\$} \{0,1\}^\lambda$. Let a_C be the answer of gRO.
- Commit to a^b_C: $(c^b_{\mathsf{ro}}, d^b_{\mathsf{ro}}) \leftarrow \mathsf{TCom}(\mathsf{pk}, a_C)$. Set $C^b = (c^b_{\mathsf{msg}}, c^b_{\mathsf{ro}})$.

Send $C = [C^0, C^1]$ to R.

i-th Decommitment Phase: $\mathsf{Dec}(state)$

- S sends $D = [m^b, d^b_{\mathsf{msg}}, d^b_{\mathsf{ro}}, a^b_C, s^b]$ to R for each $b \in \{0,1\}$.
- $\mathsf{Ver}(\mathsf{pk}, D)$. The receiver R accepts m as the decommitted value iff *all* of the following verifications succeed: (a) (b) $\mathsf{TRec}(c^b_{\mathsf{ro}}, a^b_C, d^b_{\mathsf{ro}}) = 1$, (c) $a^b_C = \mathsf{gRO}(\mathsf{sid}, \mathrm{C} \| m^b \| d^b_{\mathsf{msg}} \| s^b)$, (d) $\mathsf{TRec}(c^b_{\mathsf{msg}}, m^b, d^b_{\mathsf{msg}}) = 1$.

Theorem 6. *Assume that* $(\mathsf{TCGen}, \mathsf{TVer}, \mathsf{TCom}, \mathsf{TRec}, \mathsf{TEquiv})$ *is a Trapdoor commitment scheme, that on-line extractable NIWI proof of knowledge exist in the* gRO *model, then* UCCom *is UC-secure commitment scheme in the* gRO *model.*

Proof (Sketch).

Case R^* is Corrupted. We show that there exists a simulator, that for convenience we call SimCom, that is able to equivocate any commitment. The strategy of SimCom is to first extract the trapdoor of sk^b for some bit b from the NIWI π, then use the tradpoor sk^b to appropriately equivocate the commitment C^b. The key point is that, because $m = m^0 \oplus m^1$, equivocating one share m^b will be sufficient to open to any message m. The completed description of the simulator SimCom is provided below.

Simulator SimCom

To generate a simulated commitment under parameters pk and sid:

- Parse pk as $\mathsf{pk}^0, \mathsf{pk}^1, \pi$. Extract sk_b from π (for some $b \in \{0,1\}$) running the extractor associated to the NIWI protocol, and by observing queries to gRO for session sid. If the extractor fails, output `Abort` and halt.
- Compute $c^{\bar{b}}_{\mathsf{msg}}, d^{\bar{b}}_{\mathsf{msg}} = \mathsf{TCom}(\mathsf{pk}^{\bar{b}}, m^{\bar{b}})$, where $m^{\bar{b}}$ is a random string.
- Query gRO and obtain: $a^{\bar{b}}_C = \mathsf{gRO}(\mathsf{sid}, \mathrm{C} \| m^{\bar{b}} \| d^{\bar{b}}_{\mathsf{msg}} \| s^{\bar{b}})$.
- Compute $c^{\bar{b}}_{\mathsf{ro}}, d^{\bar{b}}_{\mathsf{ro}} = \mathsf{TCom}(\mathsf{pk}^{\bar{b}}, a^{\bar{b}}_C)$.
- Compute $c^b_{\mathsf{msg}}, c^b_{\mathsf{ro}}$ as commitments to 0.

To equivocate the simulated commitment to a value m:

- Compute $m^b = m \oplus m^{\bar{b}}$. Compute $d^b_{\mathsf{msg}} = \mathsf{TEquiv}(\mathsf{sk}^b, c_{\mathsf{msg}}, m^b)$.
- Query gRO and obtain: $a^b_C = \mathsf{gRO}(\mathsf{sid}, \mathrm{C} \| mb \| d^b_{\mathsf{msg}} \| s^b)$. Compute $d^b_{\mathsf{ro}} = \mathsf{TEquiv}(\mathsf{sk}^b, c^b_{\mathsf{ro}}, a^b_C)$.
- Output $(d^e_{\mathsf{msg}}, d^e_{\mathsf{ro}}, s^e)$ for $e = 0, 1$.

Indistinguishability. The difference between the transcript generated by SimCom and an honest S is in the fact that SimCom equivocates the commitments using the trapdoor extracted form pk, and that SimCom will abort if such trapdoor is not extracted. Indistinguishability then follows from the extractability property of π (which holds unconditionally in the gRO model) and due to the trapdoor property of the underlying trapdoor commitment scheme.

Case S^* is Corrupted. We show that there exists a simulator, that we denote by SimExt, that is able to extract the messages m^0, m^1 already in the commitment phase, by just observing the queries made to $\mathcal{G}_{\mathsf{gRO}}$ (with SID sid).

The extraction procedure follows identically the extraction procedure of the simulator shown in [12]. We describe SimExt in details below.
$\mathsf{SimExt}(\mathsf{sid}, pk, C = [C^0, C^b])$.

- Parse $pk = \mathsf{pk}^0, \mathsf{pk}^1, \pi$. If π is not accepting halt. Else, parse $C^b = c^b_{\mathsf{msg}}, c^b_{\mathsf{ro}}$ for $b = 0, 1$. Let $\mathcal{Q}_{\mathsf{sid}}$ be the list of queries made to gRO by any party.
- For $b = 0, 1$. If there exists a query q of the form $q = \mathsf{sid}\|\text{'C'}\|m^b\|d^b_{\mathsf{msg}}\|s^b$ such that $\mathsf{TRec}(c^b_{\mathsf{msg}}, m^b, d^b_{\mathsf{msg}}) = 1$, the record m^b, otherwise set $m^b = \perp$. Set $m = m^0 \oplus m^1$.
- Send $(\mathsf{commit}, \mathsf{sid}, \text{'C'}, \text{'R'}, m')$ to $\mathcal{F}_{t\mathsf{com}}$.
- *Decommitment phase:* If the openings is not accepting, halt. Else, let m^* be the valid messages obtained from the decommitment. If $m^* = m$, it sends the message $(\mathsf{decommit}, \mathsf{sid}, \text{'C'}, \text{'R'})$ to the trusted party. Otherwise, if $m^* \neq m$, then output `Abort` and halt.

Indistinguishability. The indistinguishability of the output of SimExt follows from the witness indistinguishability property of the proof system, and the biding property of the trapdoor commitment.

Due to the WI of π, any S^* cannot extract secret key sk_b used by R. Thus, if SimExt fails in extracting the correct opening, it must be that S^* is breaking the binding of commitment scheme. In such a case we can build an adversary $\mathcal{A}$ that can use S^* and the queries made by S^* to gRO to extract two openings for commitment $c^{\bar{b}}_{\mathsf{msg}}, c^{\bar{b}}_{\mathsf{ro}}$.

6 Security Proof

Theorem 7. *If* $\mathsf{UCCom} = (\mathsf{Gen}, \mathsf{Com}, \mathsf{Dec}, \mathsf{Ver})$ *is a UC-secure commitment scheme, with non-interactive commitment and decommitment phase,* $(\mathsf{Code}, \mathsf{Decode})$ *is an encoding scheme with parameters* (d, t, κ)*,* $(\mathsf{RamInit}, \mathsf{RamEval}, S_{\mathsf{oram}})$ *is a secure ORAM scheme, then protocol* $\Pi = (\Pi.\mathsf{Setup}, \Pi.\mathsf{Proof})$ *(Figs. 4 and 5), securely realizes* $\mathcal{F}_{zk}$ *functionality (Fig. 2).*

Proof. The proof follows from Lemma 9 and Lemma 8.

6.1 Case P is Corrupted

Lemma 8. *If* UCCom *is UC-secure in the* gRO *model,* $(\mathsf{Code}, \mathsf{Decode})$ *is a encoding scheme with parameters* (d, t, κ)*,* $(\mathsf{RamInit}, \mathsf{RamEval}, S_{\mathsf{oram}})$ *is a secure ORAM scheme. Then, protocol* $\Pi = (\Pi.\mathsf{Setup}, \Pi.\mathsf{Proof})$ *in Fig. 5 and Fig. 4 securely realizes* $\mathcal{F}_{zk}$ *in the* $\mathcal{F}^{C_1,C_2}_{check}$ *(resp.,* $\mathcal{F}^{C}_{check}$*) hybrid model, in presence of malicious PPT prover* P^**.*

Proof. The proof consists in two step. We first describe a simulator Sim for the malicious P^*. Then, we prove that the output of the simulator is indistinguishable from the output of the real execution.

Simulator *Intuition.* At high level, the simulator Sim proceeds in two steps. In the setup phase, Sim extracts the value committed in the nodes of the Merkle Tree. Recall that a leaf N_i of the tree is just a concatenation of commitments of shares of the memory block $\widehat{M}[i]$ (indeed, $N_i = CX_i = (cx_{i,1}, \ldots, cx_{i,\kappa})$). Sim is able to extract all commitments in CX_i by observing the queries made to gRO that are consistent with the published root h. Moreover, given such commitments, Sim is able to further extract the shares by exploiting the extractability property of UCCom (which, in turns, uses the observability of gRO.) Therefore, by the end of the setup phase, Sim has extracted shares for each block $i \in [m]$, and reconstructed "its view" of the memory, that we denote by $\widehat{M}^\star$, as well as the initial ORAM state st. Given $\widehat{M}^\star$, Sim will then be able to determine the memory $M^\star$, by running extractor $\mathsf{RamExtract}(\widehat{M}, st)$, and sends it to the ideal functionality $\mathcal{F}_{zk}$.

In the proof phase, the goal of the simulator Sim is to continuously monitor that each computation (each proof) is *consistent* with the memory $M^\star$ initially sent to $\mathcal{F}_{zk}$. Intuitively, the computation is consistent if the memory values input by P^* in each successful execution of $\mathcal{F}_{check}$ (which are represented in encoded form $X_i = [x_{i,1}, \ldots, x_{i,\kappa}]$), are "consistent" with the memory $M^\star$ that Sim has computed by extracting from the commitments; or more precisely, with the *encoding* of the block memory extracted so far.

Upon the first proof, the simulator will check that the shares of $M[i]$ submitted to $\mathcal{F}_{check}$ agree with the shares for block $M^\star[i]$ extracted in the setup phase. Here agree means that they decode to the same values. (Note that we do not require that all shares agree with the ones that were extracted by Sim, but we required that enough shares agree so that they decode to the same value).

After the first proof, P^* will also send commitments to the *updated* version of the blocks j touched during the computation. (Precisely, the shares of each block). As in Setup phase, Sim will extract these *new blocks* and update his view of $M^\star$ accordingly. In the next proof then, Sim will check consistency just as in the first proof, but consistency is checked against the newly extracted blocks.

In each proof, when checking consistency, two things can go wrong. Case 1. (Binding/extraction failure) When decommitting to the partial encodings (Step 3 of Fig. 5), P^* correctly opens values that are different from the ones previously extracted by Sim. If this happens, that P^* either has broken the extractability property of UCCom or has found a collision in the output of gRO. Thus, due to the security of UCCom, this events happens with negligible probability.

Case 2. (Encoding failure) Assume that the t shares extracted by Sim correspond to the t shares decommitment by P^*, but that among the $\kappa - t$ shares that were not open, there are at least d shares that are different. This means that the values decoded by $\mathcal{F}_{check}$ are inconsistent with the values that are decoded from the extracted shares, which means that the computation in the protocol is taking a path that is inconsistent with the path dictated by the $M^\star$ initially submitted by Sim.

We argue that this events also happen with negligible probability. Indeed, due to the security of $\mathcal{F}_{check}$ we know that the position γ that P^* will need to

decommit are unpredictable to P^*. Thus, the probability that P^* is able to open t consistent shares, while committing to d bad shares is bounded by: $(1 - \frac{d}{\kappa})^t$ which is negligible.

The Algorithm Sim. We now provide a more precise description of the simulator Sim. *Notation.* We use notation $X^\star$ to denote the fact that this is the "guess" that Sim has on the value X after extracting from the commitment of CX. During the proof phase, Sim will keep checking if this guess is consistent with the actual values that P^* is giving in input to $\mathcal{F}_{check}$.

Let SimExt be the extractor associated to UCCom and outlined in Sect. 5

Setup Phase. Run SimExt for the generation algorithm Gen. Upon receiving commitments: $CS = (cs_1, \ldots, cs_\kappa)$ and root h from P.

1. (**Extract Commitments at the Leaves of Merkle Tree**) For each query made by P^* to gRO $(sid, i \| l, P, C)$, set $CX_i^\star[l] = C$ iff $sid' = sid$ and the outputs of gRO along the paths to i are consistent with the root h. This is done by obtaining the list of queries $\mathcal{Q}_{|\mathsf{sid}}$ from $\mathcal{G}_{\mathsf{gRO}}$. At the end of this phase, Sim has collected commitments $CX_i^\star[i]$ that need to be extracted.
2. (**Extract Shares**.) Invoke extractor SimExt on input $(\mathsf{sid}, pk, CX_i^\star[l])$ for all $i \in [m]$ and $l \in [\kappa]$. Let $X_i^\star = (x_{i,1}^\star, \ldots, x_{i,\kappa}^\star)$ denote the openings extracted by SimExt. Similarly, invoke SimExt on input $(\mathsf{sid}, pk, CS[l])$ with for $l \in [\kappa]$ and obtain shares $s_1^\star, \ldots, s_\kappa^\star$ for the initial state. Note that the extracted values could be $\perp$. Record all such values.
3. (**Decode memory blocks** $\widehat{M}^\star[i]$) For each $i \in m$, run $b_i = \mathsf{Decode}(x_{i,1}^\star, \ldots, x_{i,\kappa}^\star)$. If Decode aborts, then mark $b_i = \perp$. Set block memory: $\widehat{M}^\star[i] = b_i$. Similarly, set $st = \mathsf{Decode}(s_1, \ldots, s_\kappa$.
4. Determine the real memory $M^\star$ as follows: $M^\star = \mathsf{RamExtract}(\widehat{M}, st)$. Send $(\mathsf{sid}, \mathsf{INIT}, M^\star)$ to $\mathcal{F}_{zk}$.

l-proof. Input to this phase: (Public Input) Statement $\mathcal{R}_l$, x. **Private input for** Sim. For each memory block i, Sim has recorded the most updated shares extracted: $X_i^\star = [x_{i,1}^\star, \ldots, x_{i,\kappa}^\star]$. The first time $X_i^\star$ are simply the ones extracted in the setup phase. In the l sub-sequent proof, $X_i^\star$ is set to the values extracted from the transcript of the $l-1$ proof. Similarly, Sim has recorded the extracted encodings of the ORAM state $S^\star = [s_1^\star, \ldots, s_\kappa^\star]$.

1. Upon receiving commitments CY_i $(\forall i \in \mathsf{access}(\mathcal{I}))$; CS', and new root h'. Run SimExt on inputs (sid, pk, CY_i) and obtain encoding $Y_i^\star$, and on input (sid, CS') to obtain the encoding of the ORAM state $S'^\star$.
2. Invoke $\mathsf{Sim}_{\mathcal{F}_{check}}$. If $\mathsf{Sim}_{\mathcal{F}_{check}}$ aborts, then abort and output $\mathcal{F}_{check}$ `failure!!`. If $\mathsf{Sim}_{\mathcal{F}_{check}}$ halts, then halt.
   Else, obtain P^*'s inputs to $\mathcal{F}_{check}$: W $= (w \; S, S', \mathbf{X}, \mathbf{Y})$. Recall that $\mathbf{X} = \{X_1, \ldots, X_{|\mathsf{read}(\mathcal{I})|}\}$ and $\mathbf{Y} = \{Y_1, \ldots, Y_{|\mathsf{access}(\mathcal{I})|}\}$, where X_i, Y_j are encodings of blocks in position i and j. Sim records the above values as comparison values for later.

3. Upon receiving decommitments $DX_i[\gamma]$, and authentication paths π_i for $i \in \mathsf{read}(\mathcal{I})$; $DY_j[\gamma]$ for $j \in \mathsf{access}(\mathcal{I})$ and $DS[\gamma]$, $DS'[\gamma]$. Let $X_i[\gamma], X_j[\gamma], S[\gamma], S'[\gamma]$ the value obtained from the decommitment.
   Perform the verification step as an honest verifier V (Step. 3 of Fig. 5). If any check fails, alt and output the transcript obtained so far. Else, perform the following consistency checks.
   (a) **Check consistency of the commitments stored in the Merkle tree.** If there is exists an i s.t., the commitment $CX_i^\star$ extracted in $\Pi.\mathsf{Setup}$ phase, is different from the commitment CX_i opened in the proof phase (with accepting authentication path π_i), then abort and output `Collision Failure!!!`.
   (b) **Check binding/extraction**. Check that, for all $i \in \mathsf{read}(\mathcal{I})$, all γ s.t. $r_\gamma = 1$ $X_i[\gamma] = X_i^\star[\gamma]$, and for all $j \in \mathsf{access}(\mathcal{I})$ $Y_i[\gamma] = Y_i^\star[\gamma]$, and $S[\gamma] = S^\star[\gamma]$, $S'[\gamma] = S'[\gamma]$. If not, abort and output `Binding Failure!!`.
   (c) **Check correct decoding.** Check that, for all $i \in \mathsf{read}(\mathcal{I})$, $\mathsf{Decode}(X_i^\star) = \mathsf{Decode}(X_i)$; that for all $j \in \mathsf{access}(\mathcal{I})$, $\mathsf{Decode}(Y_i^\star) = \mathsf{Decode}(Y_i)$, and that $\mathsf{Decode}(S^\star) = \mathsf{Decode}(S)$, $\mathsf{Decode}(S') \neq \mathsf{Decode}(S'^\star)$. If any of this check fails, abort and output `Decoding Failure!!`.
4. Send $(\mathsf{PROVE}, sid, \mathcal{R}_l, w)$ to $\mathcal{F}_{zk}$.
5. **Update extracted memory and extracted state**. For each $i \in \mathsf{access}(\mathcal{I})$: Set $X_i^\star = Y_i^\star$, and $S^\star = S'^\star$.[4]

Indistinguishability Proof. The proof is by hybrids arguments. As outlined at the beginning of the section, the crux of the proof is to show that the memory $M^\star$ extracted by Sim in $\Pi.\mathsf{Setup}$, is consistent with all the proofs subsequently provided by P^*. In other words, upon each proof, the updates performed to the memory in the real transcript are consistent with the updates that $\mathcal{F}_{zk}$ performs on the memory $M^\star$ sent by Sim in the ideal world.

Recall that, for each proof, Sim continuosly check that the memory blocks used in $\mathcal{F}_{check}$ are consistent with the memory blocks committed (and extracted by Sim). If this consistency is not verified, then Sim will declare failure and abort.

Intuitively, proving that the simulation is succesfull corresponds to prove that the probability that Sim declares failure is negligible. Assuming secure implementation of $\mathcal{F}_{check}$, the above follows directly from the (on-line) extractability of UCCom, the collision resistance of gRO and the *d-distance* property of the encoding scheme. We now proceed with the description of the hybrid arguments.

H_0 **(Real world).** This is the real world experiment. Here Sim runs just like an honest verifier. It outputs the transcript obtained from the executions.

H_1 **(Extracting witness from $\mathcal{F}_{check}$).** In this hybrid experiment Sim deviates from the algorithm of the verifier V in the following way. In the proof phase, Sim

[4] Recall that we use notation X_i to denote the in "initial version" of block i (before the proof was computed) while we use notation Y_i to denote the version of block i after the proof. Similarly, S denotes the initial ORAM state (pre-proof), while S' denotes the ORAM state obtained after the proof has been computed.

obtain the witness W used by P^* in $\mathcal{F}_{check}$, and it aborts if it fails in obtaining such inputs. Due to the security of $\mathcal{F}_{check}$ H_1 and H_0 are computationally indistinguishable.

H_2 **(Extracting the leaves of the Merkle Tree).** In this hybrid Sim uses to observability of gRO to obtain commitments $CX_i^\star[l]$ for $i \in [m], l \in [\kappa]$, and it aborts if an (accepting) path π_i, revealed by P^* in the proof phase, lead to a commitment $CX_i[l] \neq CX_i^\star[l]$. This corresponds to the event `Collision Failure!!!`. Due to the collision resistance property of gRO, probability of event `Collision Failure!!!` is negligible, and hence, the transcript generated in H_1 and H_2 are statistically close.

H_3 **(Extracting openings from commitments).** In this hybrid Sim invokes SimExt to extract the opening from all commitments. The difference between H_3 and H_2 is that in H_3 Sim aborts every time event `Binding Failure!!` occurs, which is negligible under the assumption that UCCom is an extractable commitment.

H_4 **(Decoding from extracted shares).** In this hybrid Sim determines each memory block $\widehat{M^\star}[i]$ by running Decode algorithm on the *extracted* shares $X_i^\star$. That is, $\widehat{M^\star}[i]$=Decode$(X_i^\star)$.

Moreover, it checks that all the extracted encodings (i.e., Y_i, S, S') decodes to the same values used in $\mathcal{F}_{check}$ (Step (c) in the algorithm Sim). In this hybrid Sim aborts everytime events `Decoding Failure!!` happen.

Hence, to prove that experiment H_3 and H_4 are statistically indistinguishable, it is sufficient to prove that: $Pr[Event \quad \texttt{DecodingFailure!!}] = negl(\kappa)$. As we argued in the high-level overview, event `Decoding Failure!!`happens with probability $(1 - \frac{d}{\kappa})^t$, which is negligible in κ for $t = 1/2\kappa$.

H_5 **(Submit to $\mathcal{F}_{zk}$ the extracted memory $\widehat{M^\star}$) Ideal World.** In this hybrid Sim plays in the ideal world, using the memory $\widehat{M^\star}$ extracted in the Setup phase.

We have proved that the value extracted by Sim are consistent with the values sent in input to $\mathcal{F}_{check}$. (Indeed, we have proved that all the failure events happen with negligible probability). Due to the security of $\mathcal{F}_{check}$ it follows that each proof l, is the a correct computation given the input blocks and the input ORAM state[5]. Due to the above arguments we know that the value sent to $\mathcal{F}_{check}$ are consistent with the memory blocks and ORAM state extracted so far. Putting the two things together, we have that any accepting proof is computed on values that are consistent with the committed values (extracted by Sim), which in turn are generated from the first version of the memory $\widehat{M^\star}$ extracted by Sim. This experiment corresponds to the description of the simulator Sim, proving the lemma.

[5] Here we are also using crucially the fact that choosing a bad ORAM state does not effect the correctness of the ORAM computation, but it can only effect the security guarantees for the prover.

6.2 Case V is corrupted

Lemma 9. *If* UCCom *is an equivocal commitment scheme in the* gRO *model,* (Code, Decode) *is an encoding scheme with parameters* $(d, 2k, \kappa)$, (RamInit, RamEval, S_{oram}) *is a secure ORAM scheme. Then, protocol* $\Pi = (\Pi.\mathsf{Setup}, \Pi.\mathsf{Proof})$ *in Fig. 5 and Fig. 4 securely realizes* $\mathcal{F}_{zk}$ *in the* $\mathcal{F}_{check}^{C_1,C_2}$ *(resp.,* $\mathcal{F}_{check}^{C}$*) hybrid model, in presence of malicious PPT verifier* V^*.

Proof Intuition. At high-level, assuming the $\mathcal{F}_{check}$ is securely implemented, the transcript of the verifier simply consists of a set of commitments, and partial encodings for each block memory touched in the computation and the ORAM state. Due to the hiding (in fact equivocability) properties of the commitments as well as the $2k$ hiding property of the encodings, it follows that by looking at $< 2t$ shares, V^* cannot distinguish the correct values of the memory/state from commitments to 0. Moreover, due to the security of ORAM, the access pattern $\mathcal{I}$ disclosed upon each proof, does not reveal any additional information about the memory/ORAM state.

Following this intuition, the simulator for V^* follows a simple procedure. It computes all commitments so that they are equivocal (i.e., it runs procedure SimCom guaranteed by the security property of commitment scheme UCCom). Upon each proof, Sim will run S_{oram} to obtain the access pattern $\mathcal{I}$, and the simulator $\mathsf{Sim}_{\mathcal{F}_{check}}$ to compute the transcript of $\mathcal{F}_{check}$, and to obtain the partial encodings that V^* is expected to see. Finally, Sim will simply equivocate the commitments that must be opened, so that they actually open to the correct partial encodings. The precise description of the simulator Sim is provided below.

The Algorithm Sim.

Setup Phase. Compute all commitments using algorithm $\mathsf{SimCom}(\mathsf{sid}, pk, com, \cdot)$. Compute Merkle tree correctly.

l-**proof.** Upon receiving $(\mathsf{PROVE}, sid, \mathcal{R}_l, 1)$ from $\mathcal{F}_{zk}$.

1. Run ORAM simulator $\mathcal{S}(1^\lambda, |\widehat{M}|)$ and obtain $\mathcal{I}$.
2. Run SimCom to obtain commitments CY_i for all $i \in \mathsf{access}(\mathcal{I})$ and commitments CS'. Update the root of the Merkle Tree accordingly.
3. Run $\mathsf{Sim}_{\mathcal{F}_{check}}$ to obtain the transcript for $\mathcal{F}_{check}$ and obtain the partial encodings: $X_i[\gamma], Y_j[\gamma]$ for $i \in \mathsf{read}(\mathcal{I})$ and $j \in \mathsf{access}(\mathcal{I})$; $S[\gamma], S'[\gamma]$, where γ is such that $r_\gamma = 1$, where r is the verifier's input to $\mathcal{F}_{check}$.
4. Equivocate commitments.
   - For each $i \in \mathsf{read}(\mathcal{I})$, compute $DX_i[\gamma] \leftarrow \mathsf{SimCom}(\mathsf{sid}, pk, equiv, CX_i[\gamma], X_i[\gamma])$ Moreover, retrieve path π in the tree.
   - For each $j \in \mathsf{access}(\mathcal{I})$, compute $DY_i[\gamma] \leftarrow \mathsf{SimCom}(\mathsf{sid}, pk, equiv, CY_i[\gamma], Y_i[\gamma])$.
   - Compute $DS[\gamma] \leftarrow \mathsf{SimCom}(\mathsf{sid}, pk, equiv, DS[\gamma], DS[\gamma])$, $DS'[\gamma] \leftarrow \mathsf{SimCom}(\mathsf{sid}, pk, equiv, DS'[\gamma], DS'[\gamma])$.
5. Send decommitments to V^*.

Indistinguishability Proof. The proof is by hybrid arguments. We will move from an experiment where Sim computes the transcript for V^* using real input M and following the algorithm run by P (hybrid H_0), to an hybrid where Sim has not input at all (hybrid H_3).

H_0. This is the real world experiment. Sim gets in input M and simply follows the algorithm of P (Figs. 4 and 5).

H_1 **(Compute Equivocal Commitments using** SimCom**).** In this hybrid Sim computes commitments using procedure $\mathsf{SimCom}(\mathsf{sid}, pk, com, \cdot)$, which requires no inputs, and decommit using $\mathsf{SimCom}(pk, equiv, \cdots)$, using the correct encodings computed from $\widehat{M}$. The difference between H_1 and H_2 is only in the way commitments are computed. Due to the equivocability property of the commitment scheme (in the gRO) model, it follows that H_1 and H_2 are statistically indistinguishable. Note that at this point Sim still uses real values for $\widehat{M}$ to compute the shares that will be later committed, and some of which will be opened.

H_2 (Run $\mathsf{Sim}_{\mathcal{F}_{check}}$). In this hybrid argument Sim computes the transcript of $\mathcal{F}_{check}$ by running simulator $\mathsf{Sim}_{\mathcal{F}_{check}}$, and decommit to the share given in output by $\mathcal{F}_{check}$. Note that $\mathcal{F}_{check}$ will output t encodings for each block memory and state. Note also that, if a block memory was accessed in a previous execution, then t shares of the encodings have been already revealed. For example the encoding of the final state $S'[1], \ldots, S'[\kappa]$, which is the output state in an execution ℓ, will the the encoding used as initial state in proof $\ell + 1$. This means that for each encoding, the adversary R^* collects $2k$ partials encodings. Due to the security of $\Pi_{\mathcal{F}_{check}}$, and to the $2k$ hiding property of the encoding scheme, hybrids H_2 and H_1 are computationally indistinguishable.

H_3 (**Use ORAM simulator** S_{oram})**.** In this hybrid Sim will replace executions of $\mathsf{RamInit}$ and $\mathsf{RamEval}$ with S_{oram}. This is possible because the actual values computed by $\mathsf{RamInit}$ and $\mathsf{RamEval}$ are not used anywhere at this point. Due to the statistical security of $(\mathsf{RamInit}, \mathsf{RamEval}, S_{\mathsf{oram}})$ hybrids H_3 and H_4 are statistically instinguishable. Note that in this experiment the actual memory M is not used anywhere. This experiment corresponds to the description of the simulator Sim, proving the lemma.

References

1. libSNARK: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark
2. Afshar, A., Hu, Z., Mohassel, P., Rosulek, M.: How to efficiently evaluate RAM programs with malicious security. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 702–729. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46800-5_27

3. Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Yu, T., Danezis, G., Gligor, V.D., (eds.) ACM CCS 2012, pp. 784–796. ACM Press, October 2012
4. Ben-Sasson, E., Chiesa, A., Gabizon, A., Virza, M.: Quasi-linear size zero knowledge from linear-algebraic PCPs. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 33–64. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49099-0_2
5. Ben-Sasson, E., Chiesa, A., Garman, C., Green, M., Miers, I., Tromer, E., Virza, M., Zerocash: Decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, pp. 459–474. IEEE Computer Society Press, May 2014
6. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: On the concrete efficiency of probabilistically-checkable proofs. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC, pp. 585–594. ACM Press, June 2013
7. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40084-1_6
8. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable zero knowledge via cycles of elliptic curves. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 276–294. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_16
9. Ben-Sasson, E., Kaplan, Y., Kopparty, S., Meir, O., Stichtenoth, H.: Constant rate PCPs for circuit-SAT with sublinear query complexity. In 54th FOCS, pp. 320–329. IEEE Computer Society Press, October 2013
10. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: Recursive composition and bootstrapping for SNARKS and proof-carrying data. In: Boneh, D., Roughgarden, T., Feigenbaum, J. (eds.) 45th ACM STOC, pp. 111–120. ACM Press, June 2013
11. 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
12. Canetti, R., Jain, A., Scafuro, A.: Practical UC security with a global random oracle. In: Ahn, G.-J., Yung, M., Li, N. (eds.) ACM CCS 2014, pp. 597–608. ACM Press, November (2014)
13. Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: versatile verifiable computation. In: 2015 IEEE Symposium on Security and Privacy, pp. 253–270. IEEE Computer Society Press, May 2015
14. Fischlin, M.: Communication-efficient non-interactive proofs of knowledge with online extractors. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 152–168. Springer, Heidelberg (2005). doi:10.1007/11535218_10
15. Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled RAM. In: Guruswami, V. (ed.) 56th FOCS, pp. 210–229. IEEE Computer Society Press, October 2015
16. Garg, S., Lu, S., Ostrovsky, R., Scafuro, A.: Garbled RAM from one-way functions. In: Servedio, R.A., Rubinfeld, R., (eds.) 47th ACM STOC, pp. 449–458. ACM Press, June 2015
17. 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

18. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 405–422. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55220-5_23
19. Giacomelli, I., Madsen, J., Orlandi, C.: ZKBoo: faster zero-knowledge for Boolean circuits. In: 25th USENIX Security Symposium (USENIX Security 2016) (2016)
20. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Aho, A. (ed.) 19th ACM STOC, pp. 218–229. ACM Press, May 1987
21. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM (JACM) **43**, 431–473 (1996)
22. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM CCS 2012, pp. 513–524. ACM Press, October 2012
23. Goyal, V., Ostrovsky, R., Scafuro, A., Visconti, I.: Black-box non-black-box zero knowledge. In: Shmoys, D.B. (ed.) 46th ACM STOC, pp. 515–524. ACM Press, May/June 2014
24. Hu, Z., Mohassel, P., Rosulek, M.: Efficient zero-knowledge proofs of non-algebraic statements with sublinear amortized cost. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 150–169. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48000-7_8
25. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Johnson, D.S., Feige, U. (eds.) 39th ACM STOC, pp. 21–30. ACM Press, June 2007
26. Jawurek, M., Kerschbaum, F., Orlandi, C.: Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013, pp. 955–966. ACM Press, November 2013
27. Kilian, J.: A note on efficient zero-knowledge proofs and arguments (extended abstract). In: 24th ACM STOC, pp. 723–732. ACM Press, May 1992
28. Kosba, A.E., Zhao, Z., Miller, A., Qian, Y., Chan, T.H., Papamanthou, C., Pass, R., Shelat, A., Shi, E.: How to use SNARKS in universally composable protocols. IACR Cryptology ePrint Archive 2015:1093 (2015)
29. Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 377–396. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36594-2_22
30. Micali, S.: CS proofs (extended abstracts). In: 35th FOCS, pp. 436–453. IEEE Computer Society Press, November 1994
31. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: 22nd ACM STOC, pp. 514–523. ACM Press, May 1990
32. Ostrovsky, R., Scafuro, A., Venkitasubramanian, M.: Resettably Sound zero-knowledge arguments from OWFs - the (semi) black-box way. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9014, pp. 345–374. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46494-6_15
33. Parno, B., Howell, J., Gentry, C., Raykova, M., Pinocchio: nearly practical verifiable computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society Press, May 2013
34. Pass, R.: On deniability in the common reference string and random oracle model. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 316–337. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45146-4_19

35. Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 129–140. Springer, Heidelberg (1992). doi:10.1007/3-540-46766-1_9
36. Stefanov, E., Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi, A.-R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013, pp. 299–310. ACM Press, November 2013