Efficient Permutation Protocol for MPC in the Head

Peeter Laud(✉)

Cybernetica AS, Tartu, Estonia
peeter.laud@cyber.ee

Abstract. The *MPC-in-the-head* construction (Ishai et al., STOC'07) gives zero-knowledge proofs from secure multiparty computation (MPC) protocols. This paper presents an efficient MPC protocol for permuting a vector of values, making use of the relaxed communication model that can be handled by the MPC-in-the-head transformation. Our construction allows more efficient ZK proofs for relations expressed in the Random Access Machine (RAM) model. We benchmark our construction and compare it against other reasonable constructions of permutations under the MPC-in-the-head transformation and conclude that it significantly improves on efficiency and the range of applicability.

Keywords: Zero-knowledge proofs · MPC-in-the-head · Random Access Machine

1 Introduction

Zero-knowledge proofs (ZKP) are cryptographic protocols that allow one party—the Prover—to convince another party—the Verifier—in the correctness of a statement, with the Verifier learning nothing besides the fact that the statement holds. The language of statements and their truth values are given in terms of a specified relation $R \subseteq \{0,1\}^* \times \{0,1\}^*$. A statement is some $x \in \{0,1\}^*$, known both to the Prover and the Verifier. The Prover attempts to convince the Verifier that there exists some w (or: the Prover knows some w), such that $(x, w) \in R$.

There exist different techniques for turning the description of the relation R into a ZKP, based on various kinds of interactive proofs, or different secure multiparty computation techniques. They work best when R is represented as an arithmetic circuit, or a boolean circuit. In practice, we express R in some programming language; large parts of it may already be given to us in case we want to present a proof that a certain computer program behaves in a certain manner. Hence, we want to give zero-knowledge proofs for relations expressed in the *Random Access Machine* (RAM) model.

Translating the description of R given in the RAM model into a ZKP is less straightforward. Generic transformations from RAM to circuits incur at least quadratic overheads [37]. For smaller overheads, one separately translates the behaviour of the processing unit, and the behaviour of the memory. These two

R. Roman and J. Zhou (Eds.): STM 2021, LNCS 13075, pp. 62–80, 2021.
https://doi.org/10.1007/978-3-030-91859-0_4

behaviours have to be related to each other, and this requires showing that the load- and store-operations read and write the same values at both sides. Showing the equality of loaded and stored values requires us to sort these actions by the memory addresses; in ZKP, this amounts to proving that two vectors are permutations of each other, and to a sortedness check of one of the vectors.

A universal representation for permutations works by fixing a routing network [4,42], and giving the bits that state how each switching element must route its two incoming values. This representation is equally well usable with any ZKP technique. If there have been m memory operations in the program, then the size of the routing network is *linearithmic*—$O(m \log m)$. If m is close to the total number of operations that the relation R (expressed as a program) performs, then the size of the routing network may be the dominant component in the translation of R into a ZKP.

MPC-in-the-head [25] is a ZKP technique that internally makes use of secure multiparty computation protocols. In practical comparisons with other techniques, it has good running time for the Prover, a decent running time for the Verifier, but longer proofs. Nevertheless, there are a number of ZK proof systems built upon this technique [1,8,18]. The MPC-in-the-head technique is also expected to compose well with other ZKP techniques.

In this paper, we propose a $O(n)$-complexity MPC-in-the-head based method to verify the correctness of the application of a permutation to a vector of values. Our method, which is basically a secure multiparty computation protocol for a communication model that fits into the MPC-in-the-head technique, can be composed with other protocols in the same communication model, hence bringing down the complexity of ZKP protocols for relations represented in the RAM model. We present our construction in Sect. 4, after discussing related work in Sect. 2 and giving the preliminaries in Sect. 3.

To appreciate our result, its location deep down in the technology stack has to be recognized. We give a MPC subroutine, which can be composed with other MPC operations using the same data representations in order to build a MPC protocol in a particular communication model, that evaluates the relation R in a manner that small coalitions of parties do not learn anything about the witness w. Onto this MPC protocol, one can apply the MPC-in-the-head transformation of Ishai et al. [25] that turns it into a ZKP protocol for the relation R. Hence, our subroutine is not a standalone protocol; in particular, it is not a mix-net. Also, any complexity results of our subroutine have to be considered in the context of the MPC-in-the-head transformation.

2 Related Work

Zero-knowledge proofs were first proposed in [22]. In this section, we cannot hope to give an overview of all the advancements thereafter. Rather, we refer to the course notes [40] discussing interactive proofs and their zero-knowledge variants.

The MPC-in-the-head construction was proposed in [25,26]. A number of ZK proof systems have been built on top of this construction. The ZKBoo [18] and

ZKB++ [8] constructions are generic transformations from MPC protocols to ZKP protocols, carefully keeping track of bits that have to be included in the proof vs. can be generated from seeds included in the proof. The construction by Katz et al. [27] gives an improved transformation for MPC protocols that have a separate preprocessing phase. The Ligero transformation [1] applies only to MPC protocols of certain form, but gives proofs of size $O(\sqrt{n})$, where n is the size of the circuit describing R.

Privacy-preserving computations in the RAM model have been studied in the context of *garbled RAM*, which can be seen as RAM analogue for garbled circuits. A heuristic construction was proposed in [34], and constructions based on common hardness assumptions in [17]. Garg et al. [15] proposed a construction that made only black-box use of the underlying cryptographic primitives. Private RAM computation protocols have also been built on top of *oblivious RAM* [21], securely implementing the *client's* operations either on top of garbled circuits [33], or secret-sharing based MPC [28], or a combination of them [12,29].

For ZK proofs, practically most efficient constructions for relations expressed in the RAM model are based on showing that two vectors are permutations of each other. Permutations in ZK proofs and MPC protocols have received their share of attention and so have the means of connecting the processing unit and the memory unit in encoding RAM-based computations in both ZK proofs and MPC protocols. Laur et al. [32] were among the first to propose a composable MPC protocol for secret sharing based protocols; Laud [30] built oblivious reading and writing operations on top of it. For garbled circuits, Zahur and Evans [43] proposed similar constructions. For ZK proofs, Ben-Sasson et al. [2] used routing networks to connect the processing unit and the memory unit in a RAM-based computation. Bootle et al. [6] lifted a technique by Neff [35] for verifying that two encrypted vectors are permutations of each other, into the encodings of relations of ZK proofs; this technique is based on showing the equality of polynomials that have the elements of one of the vectors as its roots.

Making proofs of permutations in private fashion has also been an important component of electronic voting systems. In this context, the proofs—*cryptographic mix-nets* are full-fledged protocols for stating that two sets of ciphertexts encrypt the same bag of plaintexts. These proofs can use any ZKP techniques, and be very short, even down to a constant [23]. An overview of cryptographic mix-nets is given in [24]. There is no straightforward method for composing these protocols with ZKP protocols for an arbitrary relation R.

Current state of the art of oblivious permutations in ZK proofs is definitely not satisfactory. The approaches based on routing networks have linearithmic complexity, if we consider the size of the indices and/or permuted elements to be constant. The approaches based on comparing polynomials have linear complexity, but work only over large fields and introduce extra rounds of interaction into the proof. MPC-in-the-head techniques are more versatile with respect to the algebraic structures they support, and many interesting relations are not best expressed as computations over large fields. Hence we are looking for techniques with the versatility of routing networks, but with linear complexity.

3 Preliminaries

In this paper, $[n]$ denotes the set $\{1, \ldots, n\}$. We use bold font to denote vectors: $\boldsymbol{v} = (v_1, \ldots, v_n)$ is a vector of length n.

3.1 Secure Multiparty Computation

A secure multiparty computation (MPC) protocol allows n parties $P_1, \ldots, P_n$ to jointly evaluate a publicly-known function $f : (\{0,1\}^m)^n \to \{0,1\}^\ell$, where the i-th party supplies the i-th argument of the function. All parties learn the output. Passive security for MPC protocol sets is defined through the simulation paradigm [19]. The *view* of a party in a protocol consists of the inputs of this party, the randomness this party generates, and the messages this party receives from other parties; these values allow one to perform all computations of that party, in particular find the messages it sends to other parties, and the values it outputs at the end of the protocol. The protocol Π_f for n parties is passively secure against the coalition $P_{i_1}, \ldots, P_{i_k}$, if there exists an algorithm $\mathcal{S}$ (the simulator), such that for any $x_1, \ldots, x_n$, the joint view of $P_{i_1}, \ldots, P_{i_k}$ in Π_f, where the input of P_j is x_j, is indistinguishable from the output of $\mathcal{S}(x_{i_1}, \ldots, x_{i_k}, f(x_1, \ldots, x_n))$.

Let $\mathbb{A} \subseteq \{0,1\}^*$ be a finite set. Let $\mathbb{A}_\perp = \mathbb{A} \cup \{\perp\}$, where $\perp$ denotes the absence of a value. A (n, k)-*secret sharing scheme* for $\mathbb{A}$ consists of a randomized algorithm $\mathsf{Share} : \mathbb{A} \to \mathbb{A}^n$ and a deterministic algorithm $\mathsf{Combine} : \mathbb{A}_\perp^n \to \mathbb{A}_\perp$, such that the output of Share, when restricted to at most k positions, is independent from the input, and, for all $x \in \mathbb{A}$, for all $(x_1, \ldots, x_n)$ that can be output by $\mathsf{Share}(x)$, and for all $(x'_1, \ldots, x'_n) \in \mathbb{A}_\perp$, where $x'_i \in \{x_i, \perp\}$ and the number of non-$\perp$ elements x'_i is at least $(k+1)$, we have $\mathsf{Combine}(x'_1, \ldots, x'_n) = x$.

A (n, k)-secret sharing scheme may be a significant component of n-party MPC protocols secure against k parties. In this case, the private values are held by secret-sharing them among the n parties. For operations with private values, one needs cryptographic protocols that take the shares of the inputs of the operation as the input, and return to the parties the shares of the output [16, 20]. Typically, the function f is given by an arithmetic circuit that implements it. The inputs and outputs of f, as well as the intermediate values computed in the circuit are elements of $\mathbb{A}$, which is required to be an algebraic structure, typically a ring (or, more strongly, a field). The inputs of the circuit are shared by the parties holding them. The operations in the circuit are addition and multiplication in the ring $\mathbb{A}$. The parties execute a protocol for each operation in the circuit, eventually obtaining the shares of the output value, which they all learn by running the $\mathsf{Combine}$-algorithm.

Given a value $v \in \mathbb{A}$ that is held in secret-shared form as part of a MPC protocol, we denote the sharing by $[\![v]\!]$, and the individual share of the i-th party by $[\![v]\!]_i$. If $\mathcal{J} \subseteq [n]$, then we let $[\![v]\!]_{\mathcal{J}}$ denote the tuple $([\![v]\!]_i)_{i \in \mathcal{J}}$. The write-up $[\![\boldsymbol{v}]\!]$ denotes a vector, each element of which is secret-shared. The write-up $[\![w]\!] \leftarrow [\![u]\!] + [\![v]\!]$ denotes the execution of the protocol for addition by all the parties, where the inputs are the shares of u and v, and the output shares define the value of w. Similar write-up is used for other operations with secret-shared data. Single-instruction-multiple-data operations are denoted by applying the operations to vectors of values.

A secret sharing scheme over a ring $\mathbb{A}$ is *linear* if the $\mathsf{Combine}$ operation from $\mathbb{A}^n$ to $\mathbb{A}$ is linear [9,39]. In this case, the protocol for $[\![u]\!]+[\![v]\!]$ is just the addition of the corresponding shares of u and v by each party. Similarly, the protocol for $c \cdot [\![u]\!]$, where $c \in \mathbb{A}$ is public, requires each party to multiply its share with c. The protocol for $[\![u]\!] \cdot [\![v]\!]$ is more complex; its details depend on the details of the secret sharing scheme, and it requires communication among participants.

3.2 Honest-Verifier Zero-Knowledge Proofs

Let $R \subseteq \{0,1\}^* \times \{0,1\}^*$, which we also consider as a function $R : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}$. Write $L_R = \{x \in \{0,1\}^* \,|\, \exists w \in \{0,1\}^* : (x,w) \in R\}$. We assume that R is a *NP-relation*, i.e. the function R is polynomial-time computable, and there exists a polynomial p, such for all $x \in L_R$, there exists $w \in \{0,1\}^*$, such that $(x,w) \in R$ and $|w| \leq p(|x|)$.

A protocol $\mathit{\Pi}_R$ is a *Σ-protocol* for a given NP-relation R, if it is a protocol between two parties P and V with the following properties

- **Structure:** both P and V receive $x \in \{0,1\}^*$ as input. P also receives $w \in \{0,1\}^*$ as input. P sends the first message α to V. V generates a random β (does not depend on x or α), and sends it to P as the second message. P sends the third message γ to V. V runs a check on x, α, β, γ and either accepts or rejects.
- **Completeness:** if $(x,w) \in R$, then V definitely accepts.
- **Special soundness:** there exists a number s, such that if the transcripts $(x, \alpha, \beta_i, \gamma_i)$ for $i \in [s]$ with mutually different β_i-s are all accepted by V, then a w satisfying $(x,w) \in R$ can be efficiently found from these transcripts.
- **Special honest-verifier zero-knowledge:** there exists a simulator that on input $x \in L_R$ and a random β, outputs α, γ, such that the distribution of $(x, \alpha, \beta, \gamma)$ is indistinguishable from the transcripts of the real protocol.

A Σ-protocol is an instance of honest-verifier zero-knowledge (HVZK) proofs of knowledge (PoK). It can be turned into a non-interactive ZK PoK using the Fiat-Shamir heuristic [13], which is provably secure in the *Random Oracle Model* (ROM) [38]. The same heuristic is usable if the protocol has more rounds, as long as all challenges from the verifier are freshly generated random values. In this paper, we only consider honest verifiers, as the heuristic is already usable for them.

3.3 Commitments

A *commitment scheme* [7] allows one party to bind (commit) himself to a chosen value, while keeping it hidden from others, and later reveal it, without having the option to change it. The cryptographic primitive of commitment consists of the description of a set $\mathcal{M} \subseteq \{0,1\}^*$, a randomized algorithm $\mathsf{Commit} : \mathcal{M} \rightarrow \{0,1\}^* \times \{0,1\}^*$, and a deterministic algorithm Open :

$\mathcal{M} \times \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$. Here the argument of Commit is the message $m \in \mathcal{M}$ to be commited; its outputs are the commitment c and the opening information d. The algorithm Open takes the message m, commitment c, and the opening information d, and either accepts or rejects. The commitment scheme must be *hiding*—the commitment c hides the message m in a semantically secure manner—, and *binding*—it should be intractable to construct a tuple (m_1, m_2, c, d_1, d_2), such that $m_1 \neq m_2$, but $\mathsf{Open}(m_1, c, d_1)$ and $\mathsf{Open}(m_2, c, d_2)$ both accept.

Commitments can be based on the assumption of hardness of finding discrete logarithms [36]. In the random oracle model, one can commit to $x \in \{0,1\}^*$ by generating a sufficiently long random $r \in \{0,1\}^*$, and setting $c = H(x, r)$ and $d = r$, where H is a random oracle. The Open algorithm verifies that $c = H(x, r)$.

3.4 The IKOS and ZKBoo Constructions

Fix the numbers n and $k \geq 2$, as well as a set $\mathbb{A}$ and a (n, k) secret-sharing scheme for $\mathbb{A}$. For a relation $R \subseteq \{0,1\}^* \times \mathbb{A}$, $n \in \mathbb{N}$ and $x \in \{0,1\}^*$, define the function $f_R^x : \mathbb{A}^n \to \{0,1\}$ by $f_R^x(w_1, \ldots, w_n) = R(x, \mathsf{Combine}(w_1, \ldots, w_n))$. Let $\Pi_{f_R^x}$ be a MPC protocol for f_R^x, passively secure against k parties.

The IKOS construction [25] turns the family of protocols $\{\Pi_{f_R^x}\}_{x \in \{0,1\}^*}$ into a Σ-protocol for the relation R. Let the Prover and the Verifier have an instance $x \in \{0,1\}^*$, and the Prover also have $w \in \mathbb{A}$, such that $(x, w) \in R$. In the IKOS construction, the prover first constructs a secret sharing of w by $(w_1, \ldots, w_n) \leftarrow \mathsf{Share}(w)$. He then executes the protocol $\Pi_{f_R^x}$ with inputs $w_1, \ldots, w_n$ "in his head", i.e. he performs the computations of all n *virtual* parties by himself. Through this computation, the Prover obtains the views of all n virtual parties. The Prover commits to the views of each virtual party, and sends the commitments to the Verifier. The latter randomly picks a set of indices $\{i_1, \ldots, i_k\}$. The Prover opens the views of the i_1-th, i_2-th, …, i_k-th virtual party to the Verifier, who checks that the obtained output is 1 for all virtual parties whose views were opened, and that these views are consistent with each other.

A MPC protocol consists of two kinds of steps. In the first kind, a party performs local computations. The second kind of steps is the sending of a message from one party to another, the latter receiving the same message. We can express the message send and receive as a two-party functionality of the form $(x, \perp) \mapsto (\perp, x)$, stating how the inputs of the parties are transformed into outputs.

In the IKOS construction, the Verifier checks the correctness of both kinds of steps for all virtual parties whose views have been opened. The steps of first kind are checked by the Verifier repeating the computations of the virtual parties. The steps of second kind can be checked only if both the sender and the receiver of the message are among the virtual parties whose views have been opened. In this case, the Verifier recomputes sender's message and checks that it appears in receiver's view.

For verifying the steps of the second kind, the actual two-party functionality being executed makes no difference; it may be more complex than sending and receiving a message. Indeed, for any two-party functionality $(x, y) \mapsto (g_1(x, y), g_2(x, y))$, where g_1 and g_2 are deterministic functions, the Verifier can recompute x in the first virtual party's view, y in the second virtual party's view, and then check that $g_1(x, y)$ and $g_2(x, y)$ appear in their views. In the following, we call MPC protocols, which additionally make use of such more general two-party functionalities, *MPC-in-the-head protocols.*

A n-party MPC-in-the-head protocol for evaluating arithmetic circuits over a finite ring $\mathbb{A}$ with passive security against $(n-1)$ parties was introduced and used in the ZKBoo [18] ZK proof system. The two-party functionality used by their protocol is *oblivious linear evaluation* (OLE), where the first party ("sender") inputs a pair of values $(x, r) \in \mathbb{A}^2$, the second party ("receiver") inputs a value $y \in \mathbb{A}$, the sender obtains nothing, and the receiver obtains $xy - r$.

In the ZKBoo MPC-in-the-head protocol, private values are additively shared, i.e. $v \in \mathbb{A}$ is represented as $[\![v]\!]$, where each $[\![v]\!]_i$ is a random element of $\mathbb{A}$, subject to the condition $\sum_{i=1}^{n} [\![v]\!]_i = v$. Hence the algorithm $\mathsf{Share}(v)$ generates random $[\![v]\!]_1, \ldots, [\![v]\!]_{n-1} \leftarrow \mathbb{A}$, and computes $[\![v]\!]_n = v - \sum_{i=1}^{n-1} [\![v]\!]_i$. The algorithm $\mathsf{Combine}([\![v]\!]_1, \ldots, [\![v]\!]_n)$ adds up all its arguments, none of which may be $\perp$. In order to add two private values in the underlying MPC protocol, or to multiply a private value with a constant, each party performs that same operation with his shares. For multiplying private values $[\![u]\!]$ and $[\![v]\!]$, the parties execute the protocol in Algorithm 1. We see that each pair of parties (P_i, P_j) runs an instance of OLE in order to share between themselves the product $[\![u]\!]_i \cdot [\![v]\!]_j$.

```
**Data**: private values $[\![u]\!], [\![v]\!]$
**Data**: private value $[\![w]\!]$, such that $w = uv$
**foreach** $i, j \in [n],\ i \neq j$ **do**
    $P_i$ picks a random $r_{ij}^{(i)} \xleftarrow{\$} \mathbb{A}$
    $P_i$ and $P_j$ run the following two-party functionality:
        $P_i$ inputs $([\![u]\!]_i, r_{ij}^{(i)})$
        $P_j$ inputs $[\![v]\!]_j$
        $P_i$ obtains nothing
        $P_j$ obtains $r_{ij}^{(j)} \leftarrow [\![u]\!]_i \cdot [\![v]\!]_j - r_{ij}^{(i)}$
**foreach** $i \in [n]$ **do**
    $P_i$ computes $[\![w]\!]_i \leftarrow [\![u]\!]_i \cdot [\![v]\!]_i + \sum_{1 \leq j \leq n, j \neq i} (r_{ij}^{(i)} + r_{ji}^{(i)})$
**Return** $[\![w]\!]$
```

Algorithm 1: Multiplying two private values in ZKBoo

In Algorithm 1, we have introduced the notation for two-party functionalities, generalizing the notation "$P_i \to P_j : M$" of one party sending a message to another party. In our notation, we specify the inputs each party gives to the functionality, and the outputs they get, together with the computations of the outputs from the inputs. Note that the two-party functionality will add only the output of each party to the view of that party, and nothing else.

The protocol in Algorithm 1, together with the protocols for adding private values and multiplying them with public constants, as well as protocols for secret-sharing an input value (the party doing the sharing generates a random element of $\mathbb{A}$ as the share of each party, subject to their sum being equal to the value to be shared), and recovering an output of the computation (all parties send their shares to all other parties; each party adds up the shares), is a n-party protocol passively secure against $(n-1)$ parties. Indeed, all messages a party receives, either during the sharing an input value, or as the receiver in an OLE functionality, or during the recovery of outputs, are uniformly random elements of $\mathbb{A}$ (in case of output recovery, subject to their sum being equal to the actual output, which is given to the simulator), hence can be simulated as such. These values remain uniformly random if we combine the views of up to $(n-1)$ parties.

The ZKBoo protocol considers the case $n = 3$ in particular, because this leads to the shortest proofs, due to the MPC-in-the-head protocol being *(2,3)-decomposable*. The latter condition basically means that the view of a virtual party P_i must be constructible from the random seed of this party, from his shares of private inputs, and from the view of $P_{(i+1) \bmod 3}$. In particular, there can be no information flow from $P_{(i-1) \bmod 3}$ to P_i. In Algorithm 1, this necessitates the reversal of the flow in the OLE, whenever $i + 1 \equiv j \pmod 3$. Namely, instead of P_i generating a random $r_{ij}^{(i)}$ before the start of OLE, we let P_j randomly generate $r_{ij}^{(j)}$ instead. The parties will then execute OLE with reversed roles, with P_i inputting only $[\![u]\!]_i$, P_j inputting both $[\![v]\!]_j$ and $r_{ij}^{(j)}$, P_i learning $r_{ij}^{(i)} = [\![u]\!]_i \cdot [\![v]\!]_j - r_{ij}^{(j)}$, and P_j learning nothing.

3.5 Motivation: Simulating Computations

Existing MPC protocols, and ZK proof protocols built on top of them, are suitable if the computed function f or the relation R is represented as an arithmetic circuit. In practice, such f and R are usually represented differently. They are usually given in a format executable by a computer, i.e. as programs in an imperative language, i.e. as programs for a *Random Access Machine (RAM)*. These programs can invoke storing and loading operations against memory, the cells of which are addressable with the elements of $\mathbb{A}$. These operations, and the memory structure are not easily converted into an arithmetic circuit. Examples of such R include the evaluation of a particular program, showing that certain inputs lead to certain (faulty) outputs. Another example is showing the upper or lower bounds of the length of a shortest path between two vertices in a graph, where the structure of the graph and/or the lengths of edges must remain private.

For verifying that $R(x, w) = 1$, where R is given as a RAM program, one commonly splits the execution of R on the RAM into two parts, proves the correctness of execution separately, and then shows that the two parts are connected in the right manner [2]. The first part of execution is the *processing unit*; the proof shows that at each execution step, the instruction was decoded correctly, and the result of the instruction was correctly computed from its inputs. The second part of the execution is the *memory*; the proof shows that for each memory cell, the value read from it is the same that was written to it previously. The two parts have to be connected—the sequence of load- and store-operations has to be the same at both sides. The ZK proof must check that the same sequence appears at both parts.

At processor side, it is natural to order the sequence of load- and store-operations by timestamps. When verifying the correctness of the steps made by the processor, at each execution step we need to know what value was loaded from the memory, or what value was stored there (if any). At memory side, it is natural to order this sequence first by memory address, and then by timestamps. In this manner, it is easy to verify that for each memory cell, the value loaded from there was the same that was either stored there, or loaded from there the previous time the same cell was accessed. Hence we need to show that two sequences are permutations of each other. For added flexibility, we want to have the permutation as a separate object, because we may need to show that several sequences are related to each other through the same permutation.

4 Our Construction

4.1 The Protocol

We will now present our permutation protocol, which can be used for the permutation functionality in a MPC-in-the-head protocol set that represents private values through additive sharing. Let S_m denote the group of permutations of m elements. Given a private representation of a permutation $\sigma \in S_m$, and a vector of shared values $[\![\boldsymbol{v}]\!] = ([\![v_1]\!], \ldots, [\![v_m]\!])$, where $v_i \in \mathbb{A}$, we want to have a protocol for obtaining $[\![\sigma(\boldsymbol{v})]\!] = ([\![v_{\sigma(1)}]\!], \ldots, [\![v_{\sigma(m)}]\!])$. If the protocol is executed by $n \geq 3$ parties, then we want it to be passively secure against a coalition of $(n-1)$ parties.

The permutation σ originates as a part of the witness, as we do not have any operations implemented by the MPC protocol set that result in a private permutation. In order to apply the IKOS construction to our permutation protocol, σ has to be secret-shared among the n parties using a $(n, n-1)$ secret-sharing scheme. We use the following scheme: the private representation of σ is $[\![\sigma]\!] = ([\![\sigma]\!]_1, \ldots, [\![\sigma]\!]_n)$, where $[\![\sigma]\!]_i \in S_m$ is a *random* permutation of m elements, subject to the constraint $\sigma = [\![\sigma]\!]_n \circ \cdots \circ [\![\sigma]\!]_1$. Hence the sharing algorithm $\mathsf{Share}(\sigma)$ uniformly randomly picks the permutations $[\![\sigma]\!]_1, \ldots, [\![\sigma]\!]_{n-1} \in S_m$ and computes $[\![\sigma]\!]_n = \sigma \circ [\![\sigma]\!]_1^{-1} \circ \cdots \circ [\![\sigma]\!]_{n-1}^{-1}$. The algorithm $\mathsf{Combine}([\![\sigma]\!]_1, \ldots, [\![\sigma]\!]_n)$ computes $[\![\sigma]\!]_n \circ \cdots \circ [\![\sigma]\!]_1$, requiring that none of the arguments is $\perp$. The i-th computing party will hold $[\![\sigma]\!]_i$. We are not going to specify how $[\![\sigma]\!]_i$ is represented as a bit-string. If the representation allows to express also values that are not elements of S_m, then the i-th computing party must check that $[\![\sigma]\!]_i \in S_m$. Note that through the IKOS construction, this checking requirement carries over to the Verifier in the ZKP protocol, if he selects the view of the i-th computing party for opening.

The protocol for obtaining $[\![\sigma(\boldsymbol{v})]\!]$ from $[\![\sigma]\!]$ and $[\![\boldsymbol{v}]\!]$ is given in Algorithm 2. Its structure is rather similar to the multiplication protocol in Algorithm 1. It uses a two-party functionality that is similar to oblivious linear evaluation; this similarity shows when thinking of the permutations as the action of the group S_m on the Abelian group $\mathbb{A}^m$. The algebraic identity—$\sigma(\boldsymbol{u}+\boldsymbol{v}) = \sigma(\boldsymbol{u})+\sigma(\boldsymbol{v})$—is used in the design of the protocol. Compared to the multiplication in rings, the group action lacks the other distributive law; hence there is less parallelism in Algorithm 2 than in Algorithm 1. For $n = 3$, the protocol can be made (2,3)-decomposable similarly to Algorithm 1, and it can be composed with the rest of the ZKBoo protocol set in order to express computations that consist of arithmetic operations and permutations.

```
Data: private vector $[\![\boldsymbol{v}]\!]$, private permutation $[\![\sigma]\!]$
Result: private vector $[\![\boldsymbol{w}]\!]$, where $w_i = v_{\sigma(i)}$
$[\![\boldsymbol{w}^{(0)}]\!] \leftarrow [\![\boldsymbol{v}]\!]$
for $i = 1$ to $n$ do
    foreach $j \in [n]\backslash\{i\}$ do
        $P_i$ generates random $\boldsymbol{r}_{ij}^{(i)} \in \mathbb{A}^m$
        Parties $P_i$ and $P_j$ run the following two-party functionality:
            $P_i$ inputs $[\![\sigma]\!]_i$ and $\boldsymbol{r}_{ij}^{(i)}$
            $P_j$ inputs $[\![\boldsymbol{w}^{(i-1)}]\!]_j$
            $P_i$ obtains nothing
            $P_j$ obtains $\boldsymbol{r}_{ij}^{(j)} \leftarrow [\![\sigma]\!]_i([\![\boldsymbol{w}^{(i-1)}]\!]_j) - \boldsymbol{r}_{ij}^{(i)}$
                                        /* Elementwise subtraction of vectors */
        $P_j$ defines $[\![\boldsymbol{w}^{(i)}]\!]_j \leftarrow \boldsymbol{r}_{ij}^{(j)}$
    $P_i$ defines $[\![\boldsymbol{w}^{(i)}]\!]_i \leftarrow [\![\sigma]\!]_i([\![\boldsymbol{w}^{(i-1)}]\!]_i) + \sum_{\substack{1\le j\le n\\ j\ne i}} \boldsymbol{r}_{ij}^{(i)}$
Return $[\![\boldsymbol{w}^{(n)}]\!]$
```

Algorithm 2: Private permutation PrivPerm

Theorem 1. *Algorithm 2 computes* $[\![\sigma(\boldsymbol{v})]\!]$.

Proof. This is established by the following loop invariant:

$$\boldsymbol{w}^{(i)} = [\![\sigma]\!]_i([\![\sigma]\!]_{i-1}(\cdots [\![\sigma]\!]_1(\boldsymbol{v})\cdots)) \ . \tag{1}$$

Indeed, the vector $\boldsymbol{w}^{(0)}$ is initialized as $\boldsymbol{v}$. During the main loop, $[\![\boldsymbol{w}^{(i)}]\!]$ is constructed by permuting the additive shares of the private vector $[\![\boldsymbol{w}^{(i-1)}]\!]$ with the permutation $[\![\sigma]\!]_i$. The permutation of the i-th share will be learned by the i-th party, while the permutation of the j-th share ($j \neq i$) will be additively shared between the i-th and j-th parties. Due to the definition of $[\![\sigma]\!]$, we have $\boldsymbol{w}^{(n)} = \sigma(\boldsymbol{v})$. This vector is then returned in secret-shared manner. □

4.2 Security

Theorem 2. *Algorithm 2 is secure against a passive adversary corrupting at most* $(n-1)$ *parties.*

Proof. Let $\mathcal{J} = \{j_1, \ldots, j_k\} \subset [n]$, where $k \leq (n-1)$. Consider the joint view of a set of parties $P_{j_1}, \ldots, P_{j_k}$. Their view at the start of the protocol consists of their shares of $[\![\boldsymbol{v}]\!]$ and $[\![\sigma]\!]$. Consider the i-th iteration of the protocol. If $i \in \mathcal{J}$, then no new values are added to their joint view while P_i runs the two-party functionality with all other parties; the only values added into the view are the random vectors generated by P_i. If $i \notin \mathcal{J}$, then the vectors $\boldsymbol{r}^{(j_1)}_{ij_1}, \ldots, \boldsymbol{r}^{(j_k)}_{ij_k}$ are added to the joint view. These are vectors of random values, perfectly masked with the random vectors $\boldsymbol{r}^{(i)}_{ij_1}, \ldots, \boldsymbol{r}^{(i)}_{ij_k}$. We see that for each vector $[\![\boldsymbol{w}^{(i)}]\!]_j$, where $i \in [n]$ and $j \in \mathcal{J}$, there is at least one newly generated random vector contributing to its value. We also see that newly generated random vectors mask the expressions containing either $[\![\sigma]\!]_j$ or $[\![\boldsymbol{w}^{(i)}]\!]_j$ for $j \notin \mathcal{J}$.

Hence we can simulate the joint view of $P_{j_1}, \ldots, P_{j_k}$ as shown in Algorithm 3. The simulator first generates the shares of the vectors $\boldsymbol{w}^{(i)}$, which are random, as explained above. It will then fill out the values of the vectors $\boldsymbol{r}^{(i)}_{ij}$ and $\boldsymbol{r}^{(j)}_{ij}$, iff these vectors are visible to some of the parties $P_{j_1}, \ldots, P_{j_k}$. It is straightforward to verify that all these vectors are random, subject only to the equalities between them that are prescribed in Algorithm 2.

4.3 Complexity

When discussing the complexity of MPC protocols, we typically care about three quantities—the number of bits exchanged by the communication parties, the number of necessary round-trips of communication, and the computational complexity of the local computations. If the IKOS transformation is applied to the protocol, then its round complexity becomes moot—the protocol will be executed in the head of the Prover without any latency. In this transformation, the Prover has to perform the computations of all parties, hence the complexity of these is relevant. However, for information-theoretically secure MPC protocols, which Algorithm 1 and Algorithm 2 are examples of, the computational complexity tends to be small, consisting of simple arithmetic operations, and randomness generation (which is usually implemented by calls to a pseudorandom function). Hence the computation complexity is considered to be subsumed by the communication complexity, and has not received significant attention in the literature.

The same three complexity categories matter for ZKP protocols. There is also the fourth category—the soundness error. This shows the probability of the verifier accepting an invalid proof in a single session of the protocol; repetition is used to lower it.

The round complexity of the IKOS transformation is small—the generic construction is a Σ-protocol. The soundness error is $1/n$, where n is the number of the parties and the underlying MPC protocol must be secure against $(n-1)$ parties. The soundness error cannot be influenced by the design of the MPC protocol. The computation complexity of the resulting ZKP protocol is similar to the underlying MPC protocol. The communication complexity depends on that

```
Data: Shares $[\![\boldsymbol{v}]\!]_{\mathcal{J}}$, $[\![\sigma]\!]_{\mathcal{J}}$, $[\![\boldsymbol{w}]\!]_{\mathcal{J}}$
Result: Views of the parties $\{P_i\}_{i\in\mathcal{J}}$ in Alg. 2
Let $i_*$ be an element of $[n]\backslash\mathcal{J}$
foreach $j \in \mathcal{J}$ do
    $[\![\boldsymbol{w}^{(0)}]\!]_j \leftarrow [\![\boldsymbol{v}]\!]_j$
    $[\![\boldsymbol{w}^{(n)}]\!]_j \leftarrow [\![\boldsymbol{w}]\!]_j$
foreach $j \in \mathcal{J}$, $i \in [n-1]$ do
    Randomly generate $[\![\boldsymbol{w}^{(i)}]\!]_j \in \mathbb{A}^m$
foreach $i \in [n]$ do
    if $i \in \mathcal{J}$ then
        foreach $j \in \mathcal{J}\backslash\{i\}$ do
            $\boldsymbol{r}_{ij}^{(j)} \leftarrow [\![\boldsymbol{w}^{(i)}]\!]_j$
            $\boldsymbol{r}_{ij}^{(i)} \leftarrow [\![\sigma]\!]_i([\![\boldsymbol{w}^{(i-1)}]\!]_j) - \boldsymbol{r}_{ij}^{(j)}$
        foreach $j \in [n]\backslash(\mathcal{J} \cup \{i_*\})$ do
            Randomly generate $\boldsymbol{r}_{ij}^{(i)} \in \mathbb{A}^m$
        $\boldsymbol{r}_{ii_*}^{(i)} \leftarrow [\![\boldsymbol{w}^{(i)}]\!]_i - [\![\sigma]\!]_i([\![\boldsymbol{w}^{(i-1)}]\!]_i) - \sum_{\substack{1\le j\le n \\ j\notin\{i,i_*\}}} \boldsymbol{r}_{ij}^{(i)}$
    else
        foreach $j \in \mathcal{J}$ do
            $\boldsymbol{r}_{ij}^{(j)} \leftarrow [\![\boldsymbol{w}^{(i)}]\!]_j$
Return all values $[\![\boldsymbol{w}^{(i)}]\!]_j$, $\boldsymbol{r}_{ij}^{(i)}$, $\boldsymbol{r}_{ij}^{(j)}$
```

Algorithm 3: Simulator for the view of the set of parties $\{P_i\}_{i\in\mathcal{J}}$, where $|\mathcal{J}| \le (n-1)$

of the MPC protocol, but the dependence is not very straightforward—while the size of the views of virtual parties is basically the same as the communication complexity of the MPC protocol, the verifier is able to regenerate some of it based on the views of other parties that were opened to him. A rule of thumb is, that when the general IKOS transformation is used, then the communication complexity of the resulting ZKP protocol is similar to the amount of communication from parties with unopened view to parties with opened view [8,18]. If the views of all but one party are opened, then this amounts to the size of communication originated from the last party. In case of more general two-party functionalities, the "communication" are the outputs to parties.

We see that in Algorithm 2, the communication originating from each party is $(n-1)m$ elements of $\mathbb{A}^m$. This can be seen as the contribution of our permutation protocol to the total communication complexity of the resulting ZKP protocol. This amount of communication is equal to m invocations of the multiplication protocol in Algorithm 1. The concrete communication complexity for circuits containing addition and multiplication gates has been reported to be $274\log_2|\mathbb{A}|$ bits per multiplication gate in ZKBoo [18] for the soundness error 2^{-80}; this complexity is halved for ZKB++ [8]. The permutation operation will contribute to the length of the proof in the same manner.

Benchmarking. We have implemented the prover and verifier for a ZK proof system similar to ZKB++, making use of the MPC-in-the-head protocols for addition and multiplication (Algorithm 1) of both shared values and constants, as well as the permutation protocol in Algorithm 2. The system uses $n = 3$ virtual parties with (2,3)-decomposable protocols; it is made non-interactive using the Fiat-Shamir transform. It is implemented in Haskell, using the HsOpenSSL bindings for cryptographic operations—AES-128 in CBC mode is used for expanding the random seed, and SHA-256 is used as the hash function. The system has not been optimized for execution speed and memory usage, and the running times we report are expected to be much improved; however, the length of the produced proof is the same as would be produced by an optimized system.

We have benchmarked our implementation on a program that first inputs m 32-bit values and a permutation for m elements as part of the witness, and then applies the permutation to the values; we have varied m between 2^5 and 2^{15}. The fragment of inputting a permutation and applying it to a vector of private values will appear in the encodings of relations R represented in the RAM model—the private values are the memory addresses and values in the load- and store-operations the program representing R has performed. The permutation sorts this vector by the memory addresses. Hence it makes sense to benchmark this fragment, as it precisely characterizes the cost of a crucial step of encoding R as a ZKP.

The soundness error of a single run of ZKB++ is 2/3 [8]. In our benchmarks, we have executed the protocol 218 times in parallel, bringing the soundness error below 2^{-128}. After generating the views of the virtual parties in all 218 runs, we use the hash function according to the Fiat-Shamir transform to obtain the challenges for all runs; the challenge determines, the views of which parties in which runs have to be made available to the Verifier as the main part of the proof. In Fig. 1 we show the running times of our prover and verifier, as well as the length of the transmitted proof. We see the running times growing slightly more than linearly, and the proof size is linear in the length of the vector.

5 Comparison Against Alternatives

Let us compare the efficiency of our protocol against possible alternative implementations of a permuting a private vector in an MPC protocol suitable for the IKOS transformation, for n parties, with passive security against $(n - 1)$ parties, based on additive secret sharing, and with the values being elements of a ring (secret sharing is over the same ring). Obviously, our greatest interest is towards the case, where the values are N-bit integers for some value of N, i.e. the underlying ring is $\mathbb{Z}_{2^N}$. The following two implementation approaches are natural.

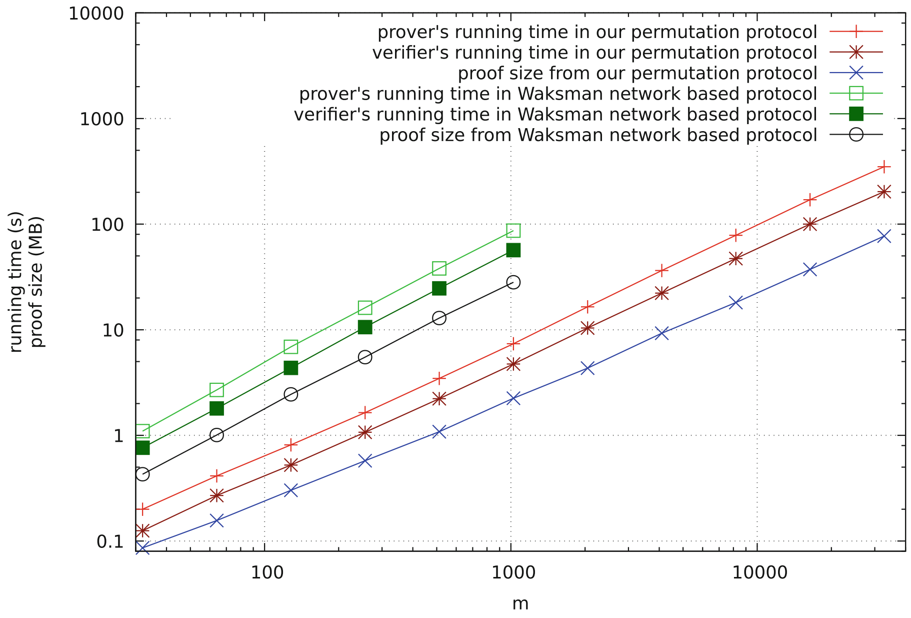

Fig. 1. Execution time and proof size for our permutation protocol, and protocol based on Waksman networks, for various lengths m of the permuted vector

5.1 Using a Routing Network

We can use a routing network to permute a vector of length m; this network has $O(m \log m)$ switches, each of which requires a single multiplication to process. If m is a power of two, then the number of switches in Waksman's network [42] is exactly $m \log_2 m - m + 1$. For other values of m, the size is larger. Hence the communication complexity of the permutation is at least $(m \log_2 m - m + 1)(n-1)$ elements of $\mathbb{A}$, which is greater than the complexity of Algorithm 2, as soon as $m \geq 4$.

However, this is not yet the entire complexity of the protocol based on routing networks. The control bits of the network, representing the permutation, are part of the witness w, secret-shared among the virtual parties by the prover. The protocol must make sure that these are indeed bits. The verification that the value b, represented as $[\![b]\!]$, is a bit can happen by b being shared not over $\mathbb{Z}_{2^N}$, but over $\mathbb{Z}_2$. In the following, let $\mathbf{R}[\![v]\!]$ denote that the value v is a private value, represented by additively sharing it over the ring $\mathbf{R}$.

When we have the representations of the bits $\mathbb{Z}_2[\![b]\!]$ used to control the routing network, we have to convert them to $\mathbb{Z}_{2^N}[\![b]\!]$ in order to use them in the multiplication protocol over $\mathbb{Z}_{2^N}$. Converting between different rings in MPC protocols over rings is a problem that has not received general attention. Bogdanov et al. [5] propose a conversion method for $\mathbb{Z}_2$ to larger rings for three-party computations with passive security, but this does not fit our use-case because it is secure against a single party only.

Suppose now that the control bits have been shared over $\mathbb{Z}_{2^N}$. In this case, the MPC protocol must verify that they are indeed bits. If b is a bit then $b(b-1) = 0$. Over a field, only bits satisfy this equality, which is in wide use in ZKP protocols over fields. Over rings, the satisfaction of $b(b-1) = 0$ is not always sufficient for b to be bit. Fortunately, it is sufficient for the rings $\mathbb{Z}_{2^N}$.

Hence the prover can input the control bits shared over $\mathbb{Z}_{2^N}$, and the protocol can verify that they are indeed bits. The verification will require another $(m \log_2 m - m + 1)$ multiplications, doubling the size of the proof. The verification will also require the declassification of the results of the checks; we assume that the amortized cost of these declassifications in addition to the making available the output of R is close to zero.

Benchmarking. For practical comparison of our protocol against an existing approach, we have also benchmarked our ZK proof system on a program that first inputs m 32-bit integers as part of the witness, with m varying between 2^5 and 2^{10}. As next, it will input a number of bits (represented as 32-bit integers) equal to the number of switches in a Waksman network for m inputs and outputs. The program verifies that these bits are indeed bits, and then applies Waksman network to the m integers, using the bits as the control bits of the switches. The execution times and lengths of the proofs are shown in Fig. 1. We see that these times and sizes are larger, and grow somewhat faster than for our protocol.

5.2 Verifying a Polynomial Equality

The second alternative is to use Neff's technique [35]: if two vectors $\boldsymbol{u}, \boldsymbol{v}$ of length m are permutations of each other, then the polynomial $Q_{\boldsymbol{u},\boldsymbol{v}}(X) = \prod_{i=1}^{m}(X - u_i) - \prod_{i=1}^{m}(X - v_i)$ is the zero polynomial. If the polynomial is over a field, then $Q_{\boldsymbol{u},\boldsymbol{v}} \equiv 0$ is also sufficient for $\boldsymbol{u}$ to be a permutation of $\boldsymbol{v}$. If the field is large, then this condition can be verified (with a small soundness error, proportional to the length m, and inversely proportional to the size of the field) by evaluating $Q_{\boldsymbol{u},\boldsymbol{v}}$ at a random point, selected by the Verifier. The verification requires $2m - 2$ multiplications, which is greater than the complexity of Algorithm 2, as soon as $m \geq 3$. We see that even when working over a field, our protocol outperforms the alternatives in an IKOS-based ZKP protocol.

Moreover, Neff's check cannot be used in the ring $\mathbb{Z}_{2^N}$, because its structure is very different from a field. We could use the protocol in Algorithm 4 (taken from [31]) to convert an additive sharing over $\mathbb{Z}_{2^N}$ to an additive sharing over $\mathbb{Z}_2^N$, i.e. into a bitwise sharing. The latter can be thought of as a sharing over the finite field $GF(2^N)$, as the additive operation in both structures is the same. At this point, Neff's check can be used.

We can estimate the communication cost of Algorithm 4. The sharing done by P_i consists of P_i sending random values to every other party. These values could be generated from pairwise shared random seeds, and no actual communication would be necessary (P_i would select its own share so, that $\bigoplus_{j=1}^{n} \mathbb{Z}_2^N[\![u_i]\!]_j = u_i$). The addition of two N-bit values requires $(N-1)$ AND-operations. There are

```
**Data**: number of parties $n$, bit-length $N$, private value $\mathbb{Z}_{2^N}[\![v]\!]$
**Result**: private value $\mathbb{Z}_2^N[\![v]\!]$
**foreach** $i \in \{1, \ldots, n\}$ **do**
    Party $P_i$ defines $u_i \leftarrow \mathbb{Z}_{2^N}[\![v]\!]_i$, shares $\mathbb{Z}_2^N[\![u_i]\!]$
Parties privately execute the summation circuit for $n$ $N$-bit values, computing
$\mathbb{Z}_2^N[\![u]\!] \leftarrow \sum_{i=1}^{n} \mathbb{Z}_2^N[\![u_i]\!]$
**Return** $\mathbb{Z}_2^N[\![u]\!]$
```

Algorithm 4: Bit extraction

n values to be added, hence the addition has to be repeated $(n-1)$ times. This has to be further multiplied by the length of the vector m. One addition requires $(n-1)$ bits to be communicated, which is N times less than one multiplication according to Algorithm 1. The total communication cost of converting m private values from $\mathbb{Z}_{2^N}$ to $GF(2^N)$ is equivalent to the cost of $(n-1)m(N-1)/N$ multiplications.

6 Discussion

We have proposed a passively secure MPC-in-the-head protocol for permutation. More efficient constructions of ZK proofs from MPC-in-the-head protocols are known, if the underlying protocols with several parties are *actively* secure for at least a constant fraction of the parties. Existing efficient linear secret sharing based MPC protocols [3,11] make use of homomorphic MACs, which are updated by each operation in the arithmetic circuit encoding the computation. It is unclear, what would be a suitable MAC for permutation, as it would have to have suitable homomorphic properties with respect to the application of that permutation to a vector of values.

There exist methods to turn passively secure protocols into actively secure protocols with the help of replication [10]. Most probably, these methods will not help in increasing the efficiency of the resulting ZK proof, compared to the use of the underlying passively secure protocol, because the IKOS technique would dismantle the passive-to-active construction.

Still, our construction will be useful in encoding the relations represented in the RAM model as ZK proofs built using the IKOS technique. Its efficiency can perhaps be further improved by considering a separate preprocessing step as in [27] and compressing the representations of random permutations and vectors as much as possible (e.g. as in [14]).

Acknowledgements. This research has been funded by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-20-C-0083. The views, opinions, and/or findings expressed are those of the author and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This research has also been supported by European Regional Development Fund through the Estonian Centre of Excellence in ICT Research (EXCITE), and Estonian Research Council through grant PRG920. We thank Said Daoudagh and the PC members of STM 2021 for their valuable comments.

References

1. Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: lightweight sublinear arguments without a trusted setup. In: Thuraisingham, et al. [41], pp. 2087–2104. https://doi.org/10.1145/3133956.3134104
2. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In: Kleinberg, R.D. (ed.) Innovations in Theoretical Computer Science, ITCS 2013, Berkeley, CA, USA, 9–12 January 2013, pp. 401–414. ACM (2013). https://doi.org/10.1145/2422436.2422481
3. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 169–188. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20465-4_11
4. Beneš, V.E.: Mathematical Theory of Connecting Networks and Telephone Traffic. Academic Press, Cambridge (1965)
5. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. Int. J. Inf. Secur. **11**(6), 403–418 (2012). https://doi.org/10.1007/s10207-012-0177-2
6. Bootle, J., Cerulli, A., Groth, J., Jakobsen, S., Maller, M.: Arya: nearly linear-time zero-knowledge proofs for correct program execution. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11272, pp. 595–626. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03326-2_20
7. Brassard, G., Chaum, D., Crépeau, C.: Minimum disclosure proofs of knowledge. J. Comput. Syst. Sci. **37**(2), 156–189 (1988). https://doi.org/10.1016/0022-0000(88)90005-0
8. Chase, M., et al.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. In: Thuraisingham, et al. [41], pp. 1825–1842. https://doi.org/10.1145/3133956.3133997
9. Cramer, R., Damgård, I., Maurer, U.: General secure multi-party computation from any linear secret-sharing scheme. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 316–334. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_22
10. Damgård, I., Orlandi, C., Simkin, M.: Yet another compiler for active security or: efficient MPC over arbitrary rings. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 799–829. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_27
11. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38
12. Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: Thuraisingham, et al. [41], pp. 523–535. https://doi.org/10.1145/3133956.3133967
13. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
14. Fleischhacker, N., Simkin, M.: On publicly-accountable zero-knowledge and small shuffle arguments. In: Garay, J.A. (ed.) PKC 2021. LNCS, vol. 12711, pp. 618–648. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-75248-4_22

15. Garg, S., Lu, S., Ostrovsky, R.: Black-box garbled RAM. In: Guruswami, V. (ed.) IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17–20 October 2015, pp. 210–229. IEEE Computer Society (2015). https://doi.org/10.1109/FOCS.2015.22
16. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: Coan, B.A., Afek, Y. (eds.) Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1998, Puerto Vallarta, Mexico, 28 June–2 July 1998, pp. 101–111. ACM (1998). https://doi.org/10.1145/277697.277716
17. 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). https://doi.org/10.1007/978-3-642-55220-5_23
18. Giacomelli, I., Madsen, J., Orlandi, C.: ZKBoo: faster zero-knowledge for Boolean circuits. In: Holz, T., Savage, S. (eds.) 25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, 10–12 August 2016, pp. 1069–1083. USENIX Association (2016). https://www.usenix.org/conference/usenixsecurity16
19. Goldreich, O.: The Foundations of Cryptography - Volume 2: Basic Applications. Cambridge University Press, Cambridge (2004). https://doi.org/10.1017/CBO9780511721656
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.V. (ed.) 1987 Proceedings of the 19th Annual ACM Symposium on Theory of Computing, New York, New York, USA, pp. 218–229. ACM (1987). https://doi.org/10.1145/28395.28420
21. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996). https://doi.org/10.1145/233551.233553
22. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: Sedgewick, R. (ed.) Proceedings of the 17th Annual ACM Symposium on Theory of Computing, Providence, Rhode Island, USA, 6–8 May 1985, pp. 291–304. ACM (1985). https://doi.org/10.1145/22145.22178
23. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49896-5_11
24. Haines, T., Müller, J.: SoK: techniques for verifiable mix nets. In: 33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, 22–26 June 2020, pp. 49–64. IEEE (2020). https://doi.org/10.1109/CSF49147.2020.00012
25. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge from secure multiparty computation. In: Johnson, D.S., Feige, U. (eds.) Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, 11–13 June 2007, pp. 21–30. ACM (2007). https://doi.org/10.1145/1250790.1250794
26. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Zero-knowledge proofs from secure multiparty computation. SIAM J. Comput. **39**(3), 1121–1152 (2009). https://doi.org/10.1137/080725398
27. Katz, J., Kolesnikov, V., Wang, X.: Improved non-interactive zero knowledge with applications to post-quantum signatures. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 15–19 October 2018, pp. 525–537. ACM (2018). https://doi.org/10.1145/3243734.3243805

28. Keller, M.: The oblivious machine. In: Lange, T., Dunkelman, O. (eds.) LATINCRYPT 2017. LNCS, vol. 11368, pp. 271–288. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25283-0_15
29. Keller, M., Yanai, A.: Efficient maliciously secure multiparty computation for RAM. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 91–124. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_4
30. Laud, P.: Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. Proc. Priv. Enhancing Technol. **2015**(2), 188–205 (2015). https://doi.org/10.1515/popets-2015-0011
31. Laud, P., Randmets, J.: A domain-specific language for low-level secure multiparty computation protocols. In: Ray, I., Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015, pp. 1492–1503. ACM (2015). https://doi.org/10.1145/2810103.2813664
32. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 262–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_18
33. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: a programming framework for secure computation. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, 17–21 May 2015, pp. 359–376. IEEE Computer Society (2015). https://doi.org/10.1109/SP.2015.29
34. Lu, S., Ostrovsky, R.: How to garble RAM programs? In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 719–734. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_42
35. Neff, C.A.: A verifiable secret shuffle and its application to e-voting. In: Reiter, M.K., Samarati, P. (eds.) CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, 6–8 November 2001, pp. 116–125. ACM (2001). https://doi.org/10.1145/501983.502000
36. 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). https://doi.org/10.1007/3-540-46766-1_9
37. Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM **26**(2), 361–381 (1979). https://doi.org/10.1145/322123.322138
38. Pointcheval, D., Stern, J.: Security proofs for signature schemes. In: Maurer, U. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 387–398. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68339-9_33
39. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979). https://doi.org/10.1145/359168.359176
40. Thaler, J.: Proofs, Arguments, and Zero-Knowledge (2021). Course notes. http://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html
41. Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.): Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, 30 October–03 November 2017. ACM (2017). https://doi.org/10.1145/3133956
42. Waksman, A.: A permutation network. J. ACM **15**(1), 159–163 (1968). https://doi.org/10.1145/321439.321449
43. Zahur, S., Evans, D.: Circuit structures for improving efficiency of security and privacy tools. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, 19–22 May 2013, pp. 493–507. IEEE Computer Society (2013). https://doi.org/10.1109/SP.2013.40