Composable Oblivious Extended Permutations

Peeter Laud and Jan Willemson
{peeter.laud|jan.willemson}@cyber.ee

Cybernetica AS

Abstract. An extended permutation is a function $f : \{1, \ldots, m\} \to \{1, \ldots, n\}$, used to map an n-element vector $\boldsymbol{a}$ to an m-element vector $\boldsymbol{b}$ by $b_i = a_{f(i)}$. An *oblivious* extended permutation allows this mapping to be done while preserving the privacy of $\boldsymbol{a}$, $\boldsymbol{b}$ and f in a secure multiparty computation protocol. Oblivious extended permutations have several uses, with private function evaluation (PFE) being the theoretically most prominent one.
In this paper, we propose a new technique for oblivious evaluation of extended permutations. Our construction is at least as efficient as the existing techniques, conceptually simpler, and has wider applicability. Our technique allows the party providing the description of f to be absent during the computation phase of the protocol. Moreover, that party does not even have to exist — we show how to compute the private representation of f from private data that may itself be computed from the inputs of parties. In other words, our oblivious extended permutations can be freely composed with other privacy-preserving operations in a multiparty computation.
Keywords: Secure multiparty computation, Private function evaluation, Extended permutations

1 Introduction

In Secure Multiparty Computation (SMC), k parties compute $(y_1, \ldots, y_k) = f(x_1, \ldots, x_k)$, with the party P_i providing the input x_i and learning no more than the output y_i. Private Function Evaluation (PFE) is a special case of SMC, where the function f is also private, and its description, typically in the form of a circuit, is provided as input by one of the parties. One will thus obtain a solution for PFE, if one designs an SMC system for a universal function f. In SMC systems, f is usually represented as a Boolean or arithmetic circuit. Universal circuits are large (compared to circuits they can execute), hence this approach has not been practical so far.

Recently, Mohassel and Sadeghian [36] have split the task of oblivious circuit evaluation into two parts — obliviously evaluating the gates, and hiding the topology of the circuit in a manner that allows the outputs of the gates to be passed to the inputs of next gates. They introduce *oblivious extended permutations (OEP)* for the second subtask. Their approach increases the performance of PFE over the state of the art by a couple of orders of magnitude, making the private execution of small circuits a realistic proposition.

SMC techniques have seen significant maturation in last years, with the appearance of several frameworks [1,2,5,9,14,21,34] that allow the private computation of certain tasks with practically relevant sizes. There has been a number of successful applications of these frameworks [3, 7, 8, 26, 30]. A common tenet of all existing and in-progress applications is their client-server nature, where the participating entities are partitioned into *input parties* providing the private inputs to a SMC system, *computing parties* that execute the SMC protocols for computing the function f, and *output parties* that receive the results of the computation [4]. Depending on the application, these sets of parties may intersect or even coincide, but such partitioning is nevertheless supported by the used SMC platforms. This flexibility is certainly required in practice, as the active participation of all input parties in all stages of computation is unwanted both for efficiency (if the number of inputs parties is large), as well as organizational (if the input parties do not have the ability to execute complex protocols) reasons.

Mohassel's and Sadeghian's OEP construction does not fit into the model with input, computing and output parties. In their construction, the party providing the description of the private function must participate in the computation, i.e. it must be both an input and a computing party.

In this paper, we propose a multiparty OEP construction that allows the extended permutation to be input to the private computation by a non-computing input party. Even more, our construction allows oblivious extended permutations to be constructed *during* the computation from other private values, thereby removing the need to treat them in any special manner. In fact, all our constructions will be presented in the Arithmetic Black Box (ABB) model, making their use in larger applications straightforward, and also greatly simplifying the security proofs. Our construction is conceptually simpler than [36], and, if the number of computing parties is small, also potentially more efficient (even though a fair comparison is difficult due to different operational profiles). We have implemented our proposed construction and provide benchmarking results.

Our algorithms for constructing an OEP during the execution of a private computation alternate between sorting the matrices along various columns, and computing new columns from the existing ones in parallel manner. These techniques may be of independent interest, as they can be used for various other tasks, including the privacy-preserving grouping and aggregate computation protocols in oblivious databases.

This paper has the following structure. We review the related work in Sec. 2 and give the necessary preliminaries, including the ABB model, in Sec. 3. We will continue in the standard fashion, presenting the desired ideal functionality for OEPs in Sec. 4, followed by the description of the actual protocol set (together with security proofs) in Sec. 5, with the most complicated protocol appearing in Sec. 6. Afterwards, we will focus on optimizations specific for a particular ABB implementation, namely the additive secret sharing as employed by SHAREMIND [6]. We describe the additional functionality offered by this ABB in Sec. 7 and show how it can be used to speed up certain OEP protocols in Sec. 8. In Sec. 9 we present the benchmarking results of our implementation of the OEP protocol; according to our knowledge, this is the first such implementation. In Sec. 10, we discuss some further research directions opened up by our OEP construction.

2 Related Work

A number of existing OEP constructions are based on switching networks employing 2×2 switches that may either pass their inputs unmodified, swap the inputs, or copy one input to both outputs. The network is commonly obtained from Waksman's construction [38]; it is evaluated with SMC techniques. Such constructions appear in [22, 27, 36]. In [35], the construction of [36] is amended to give it security against malicious adversaries. All such constructions require one of the computing parties to know the extended permutation.

An OEP can also be constructed, using homomorphic encryption [22, Sec. 5.3.2]. This construction has better asymptotic complexity than the ones based on switching networks, but it requires many public-key operations. Again, one computing party has to know the extended permutation.

A very simple construction for *shuffling* (permuting) the elements of a vector is given by Laur et al. [31]. Hamada et al. [19, 20] have used this construction to give fast sorting algorithms. Our constructions are also based on this form of shuffling protocols.

OEPs can be used for purposes other than PFE. Guanciale et al. [18] have applied them in the minimization of finite automata obtained through the product construction.

3 Preliminaries

Universal composability (UC) [10] is a standard theoretical framework for stating and proving security of cryptographic constructions. In this framework, a protocol π is defined secure if it is *as secure as* some ideal functionality $\mathcal{F}$ embodying the desired functional and non-functional properties of π in an abstract manner. A functionality $\mathcal{F}_1$ is at least as secure as $\mathcal{F}_2$, if for every user of these functionalities, and every adversary of $\mathcal{F}_1$, there is an adversary of $\mathcal{F}_2$, such that the user cannot see the difference in interacting with $\mathcal{F}_1$ or $\mathcal{F}_2$. UC framework derives its usefulness from the composability of the "*at least as secure as*" relation.

Arithmetic black box. For SMC, the standard ideal functionality is the Arithmetic Black Box (ABB) $\mathcal{F}_{\mathsf{ABB}}$ [13]. It provides an interface for users $P_1, \ldots, P_k$, up to t of which may be corrupted, to perform computations without revealing intermediate values. Here t depends on the protocol set π_{ABB} implementing $\mathcal{F}_{\mathsf{ABB}}$. The functionality $\mathcal{F}_{\mathsf{ABB}}$ is given in Fig. 1. Depending on the implementation π_{ABB}, the adversary and/or certain coalitions of users may also be able to stop the execution of $\mathcal{F}_{\mathsf{ABB}}$. We won't define the behaviour of $\mathcal{F}_{\mathsf{ABB}}$ in exceptional situations (e.g. undefined variables), because their occurrence can be detected from public information.

Internal state: a finite map $\mathbf{S}$ from variable names to values (initially empty)
Exposed commands:

Input data. On input (input, v, x) from some P_i and (input, v) from all other parties, add $\{v \mapsto x\}$ to $\mathbf{S}$.
Classify. On input $(\mathsf{classify}, v, x)$ from all parties, add $\{v \mapsto x\}$ to $\mathbf{S}$.
Compute. On input $(\mathsf{compute}, \otimes, v_1, v_2, v_3)$ from all parties, look up $x_1 = \mathbf{S}(v_1)$ and $x_2 = \mathbf{S}(v_2)$, and add $\{v_3 \mapsto x_1 \otimes x_2\}$ to $\mathbf{S}$.
Declassify. On input $(\mathsf{declassify}, v)$ from all parties, answer with $\mathbf{S}(v)$ to all parties..

When receiving any command except input, the whole command, as well as its answer is also sent to the adversary. For input-commands, (input, v) is sent to the adversary. Some of the parties may be controlled by the adversary; for simplicity, the corruptions are static. For any command $(c, \ldots)$ listed above, $\mathcal{F}_{\mathsf{ABB}}$ accepts the command $(\mathsf{masq}^i, c, \ldots)$ from the adversary, for a corrupted party P_i. Such commands are processed as commands $(c, \ldots)$ from P_i.

Fig. 1: The ideal functionality $\mathcal{F}_{\mathsf{ABB}}$

The interface of $\mathcal{F}_{\mathsf{ABB}}$ does not correspond well to the partitioning of parties into input, computing, and output parties. Still, it can be modeled by precisely defining, which parties are needed to execute different commands, and which parties receive the results.

The values $\mathcal{F}_{\mathsf{ABB}}$ operates on are elements of some algebraic structure depending on π_{ABB}, typically some finite field or ring. The operations $\otimes$ supported by $\mathcal{F}_{\mathsf{ABB}}$ also depend on the actual protocols that are available. Many protocol sets π_{ABB} for SMC have been proposed [12,15–17,24,37,39], several of them also providing security against malicious adversaries. All sets support at least the addition and multiplication of values stored in $\mathbf{S}$. Based on them, one can implement a rich set of arithmetic and relational operations [11], enjoying the same security properties. The protocols we present in this paper need to compare the values in $\mathbf{S}$, and we assume that "equals" and "less than" operations are available in $\mathcal{F}_{\mathsf{ABB}}$.

For a variable name v, it is customary to denote its value, as stored in $\mathbf{S}$, by $[\![v]\!]$. Also, in the description of algorithms executed together with $\mathcal{F}_{\mathsf{ABB}}$, notation $[\![w]\!] = \otimes([\![u]\!], [\![v]\!])$ denotes the calling of $(\mathsf{compute}, \otimes, u, v, w)$ on $\mathcal{F}_{\mathsf{ABB}}$.

Shuffling. An oblivious shuffle, introduced by Laur et al. [31] allows to permute the elements of a private array of length m according to a private permutation $\sigma \in S_m$. The functionality and security of oblivious shuffle can be likewise presented through the notion of ABB. Let the variable names be partitioned into two — names for scalars, and shuffles. In Fig. 1, each variable name refers to a scalar. In Fig. 2 we list the shuffling-related commands of the ABB. Here u, v denote scalar variables, and s denotes shuffle variables.

For ABB implementations based on secret sharing, Laur et al. [31] introduce the following construction. Let $\Gamma \subseteq 2^{\{1,\ldots,k\}}$ be the (monotonic) access structure: A set of parties $A \subseteq \{1, \ldots, k\}$ is allowed to successfully recombine the shares

Input a shuffle. On input $(\mathsf{input}, s, \sigma)$ from some P_i and (input, s, m) from all other parties, if $\sigma \in S_m$ then add $\{s \mapsto \sigma\}$ to $\mathbf{S}$.

Classify a shuffle. On input $(\mathsf{classify}, s, \sigma)$ from all parties, add $\{s \mapsto \sigma\}$ to $\mathbf{S}$.

Make a random shuffle. On input $(\mathsf{rand\_shuffle}, s, m)$ from all parties, pick $\sigma \in_R S_m$ and add $\{s \mapsto \sigma\}$ to $\mathbf{S}$.

Compose private and public shuffle. On input $(\mathsf{compose\_left}, s_1, s_2, \tau)$ from all parties, look up $\sigma = \mathbf{S}(s_2)$ and add $\{s_1 \mapsto \tau \circ \sigma\}$ to $\mathbf{S}$. On input $(\mathsf{compose\_right}, s_1, s_2, \tau)$ do the same, but add $\{s_1 \mapsto \sigma \circ \tau\}$ to $\mathbf{S}$.

Apply a shuffle. On input $(\mathsf{apply}, u_1, \ldots, u_m; v_1, \ldots, v_m; s)$ from all parties, look up $\sigma = \mathbf{S}(s)$ and $x_i = \mathbf{S}(v_i)$ for all $i \in \{1, \ldots, m\}$. Add $\{u_i \mapsto x_{\sigma(i)}\}$ to $\mathbf{S}$ for all i.

Compose shuffles in parallel. On input $(\mathsf{compose\_parallel}, s_1, s_2, s_3)$ from all parties, look up $\sigma_1 = \mathbf{S}(s_1)$ and $\sigma_2 = \mathbf{S}(s_2)$. Let σ_i operate on m_i elements. Let $m_3 = m_1 + m_2$ and define $\sigma_3 \in S_{m_3}$ by $\sigma_3(x) = \sigma_1(x)$, if $x \leq m_1$, and $\sigma_3(x) = \sigma_2(x - m_1) + m_1$, if $x > m_1$. Add $\{s_3 \mapsto \sigma_3\}$ to $\mathbf{S}$.

Invert a shuffle On input $(\mathsf{invert}, s_1, s_2)$ from all parties, look up $\sigma = \mathbf{S}(s_1)$ and add $\{s_2 \mapsto \sigma^{-1}\}$ to $\mathbf{S}$.

Similarly to Fig. 1, all commands are also sent to the adversary, except for input, where only (input, s, m) is sent. Also, $(\mathsf{masq}^i, \ldots)$ commands are accepted from the adversary.

Fig. 2: Shuffle-related operations in $\mathcal{F}_{\mathsf{ABB}}$

if $A \in \mathit{\Gamma}$. A shuffle $\sigma \in S_m$ is represented as $[\![\sigma]\!] = ([\![\sigma]\!]_1, \ldots, [\![\sigma]\!]_l)$, where $[\![\sigma]\!]_i \in S_m$ are random permutations subject to $[\![\sigma]\!]_1 \circ \cdots \circ [\![\sigma]\!]_l = \sigma$. Each $[\![\sigma]\!]_i$ is known by all parties in some set $A_i \in \mathit{\Gamma}$. The number l and the sets A_i must be chosen so, that for each $\bar{A} \subseteq \{1, \ldots, k\}$, $\bar{A} \notin \mathit{\Gamma}$, there exists some i, such that $\bar{A} \cap A_i = \emptyset$. In the shuffling protocol, the values $x_1, \ldots, x_m$ are shuffled using $[\![\sigma]\!]_1, \ldots, [\![\sigma]\!]_l$ (sequentially). Before shuffling with $[\![\sigma]\!]_i$, the current values are shared among the parties in A_i only.

In general, l is exponential in k and t, but the protocol is very practical if k and t are small. For $k = 3$ and $t = 1$ (in which case $\mathit{\Gamma} = \{\{1,2\}, \{1,3\}, \{2,3\}\}$) we have $l = 3$, with each party knowing two permutations.

It is straightforward to securely implement other operations in Fig. 2, based on the protocol of Laur et al. [31]. Note that inverting a shuffle also inverts the sequence of the party sets A_i applying the consecutive permutations in $[\![\sigma]\!]$. Laur et al. also show how to make the protocols secure against malicious adversaries. Alternatively, recent proposals for making passively secure protocols verifiable [23, 28] are readily applicable to described shuffling protocols.

Oblivious shuffles are instrumental for fast sorting algorithms on private values [20]. To sort a vector $[\![u]\!] = ([\![u_1]\!], \ldots, [\![u_m]\!])$, where all values are known to be different, one may generate a random shuffle $[\![\sigma]\!]$ and apply it on $[\![u]\!]$. Afterwards, the elements of $[\![u]\!]$ are randomly ordered and their comparison results may be declassified. In this way expensive, data-oblivious sorting methods [25] do not have to be employed. Sorting, in turn, can be used to transform a vector of values $([\![v_1]\!], \ldots, [\![v_m]\!])$ to a shuffle $[\![\sigma]\!]$, such that $\sigma(i) = v_i$, provided that the private values are a permutation of $(1, \ldots, m)$. See Alg. 1. The algorithm

Algorithm 1: Vector2Shuffle, From a vector of private values to private shuffle

```
Data: Vector of values $(⟦v_1⟧, \ldots, ⟦v_m⟧)$, with $\{v_1, \ldots, v_m\} = \{1, \ldots, m\}$
Result: A shuffle $⟦\sigma⟧$, such that $\sigma(i) = v_i$
$⟦s⟧ \leftarrow \mathsf{rand\_shuffle}(m)$
$(⟦u_1⟧, \ldots, ⟦u_m⟧) \leftarrow \mathsf{apply}(⟦v_1⟧, \ldots, ⟦v_m⟧; ⟦s⟧)$
Sort $(⟦u_1⟧, \ldots, ⟦u_m⟧)$, using $\mathsf{declassify}(⟦\cdot⟧ \leq ⟦\cdot⟧)$ as the comparison function
Let $\tau \in S_m$ be the sorting permutation, i.e. $u_{\tau(i)} = i$.
return $\mathsf{invert}(⟦s⟧ \circ \tau)$
```

Sort. On input $(\mathsf{sort}, u_1^1, \ldots, u_1^l; \ldots; u_m^1, \ldots, u_m^l; s)$, look up the values $x_i^j = \mathbf{S}(u_i^j)$ for all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, l\}$. Find $\sigma \in S_m$, such that

- for all $1 \leq i \leq j \leq n$: $(x_{\sigma(i)}^1, \ldots, u_{\sigma(i)}^l) \leq (u_{\sigma(j)}^1, \ldots, u_{\sigma(j)}^l)$, where the ordering of tuples is defined lexicographically;
- for all $1 \leq i < j \leq n$: if $(x_i^1, \ldots, u_i^l) = (u_j^1, \ldots, u_j^l)$, then $\sigma(i) < \sigma(j)$, i.e. the sorting is stable.

Add $\{s \mapsto \sigma\}$ to $\mathbf{S}$ and forward the sort-command to the adversary.

Fig. 3: Sorting in $\mathcal{F}_{\mathsf{ABB}}$

is secure because the only values output by $\mathcal{F}_{\mathsf{ABB}}$ during its execution are the results of comparisons; these may be made public by the security arguments for sorting algorithms. It is easy to verify that Alg. 1 is also correct.

As sorting turns out to be a useful operation in our protocols, we opt to make it a part of $\mathcal{F}_{\mathsf{ABB}}$. See Fig. 3 for the exact specification. For ease of use, we let our sorting functionality to not actually sort its input, but to output a private shuffle that would sort the input if applied to it. The protocol for computing such a shuffle in π_{ABB} is identical to Alg. 1, except for the omission of the last inversion.

4 Ideal functionality

The notion of *extended permutation (EP)* was introduced in [36] for encoding the topology of arithmetic or Boolean circuits. Mathematically, an EP ϕ from an n-element set to an m-element set is just a function from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$. *Applying* ϕ to a sequence $(x_1, \ldots, x_n)$ produces a sequence $(y_1, \ldots, y_m)$, such that $y_i = x_{\phi(i)}$ for each i. Similarly to shuffles, we want to apply EPs in an oblivious manner, such that the values to which ϕ is applied, as well as ϕ itself remain private.

Let $F_{n,m}$ denote the set of all mappings from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$. Our intended ideal functionality for an ABB with EPs is given in Fig. 4. The functionality maps variables to either private values, private shuffles or private EPs, and allows operations on them. Let the variable names be partitioned into three — names for scalars, shuffles, and EPs. In Fig. 4, u, v denote scalar variables, while f denotes EP variables.

Include the state $\mathbf{S}$ and commands of $\mathcal{F}_{\mathsf{ABB}}$ in Fig. 1 and Fig. 2.
Additional commands are given below.

Input an EP. On input $(\mathsf{input}, f, \phi)$ from some P_i and $(\mathsf{input}, f, n, m)$ from all other parties, if $\phi \in F_{n,m}$ then add $\{f \mapsto \phi\}$ to $\mathbf{S}$.
Classify an EP. On input $(\mathsf{classify}, f, \phi)$ from all parties, add $\{f \mapsto \phi\}$ to $\mathbf{S}$.
Apply an EP. On input $(\mathsf{apply}, u_1, \ldots, u_m; v_1, \ldots, v_n; f)$ from all parties, look up $\phi = \mathbf{S}(f)$ and $x_i = \mathbf{S}(v_i)$. Add $\{u_j \mapsto x_{f(j)}\}$ to $\mathbf{S}$ for all $j \in \{1, \ldots, m\}$. The mapping ϕ must be an element of $F_{n,m}$.
Convert a vector to an EP. On input $(\mathsf{convert}, u_1, \ldots, u_m, n, f)$ from all parties, look up $x_i = \mathbf{S}(u_i)$ for all $i \in \{1, \ldots, m\}$. If $1 \leq x_i \leq n$ holds for all i, let $\phi \in F_{n,m}$ be defined by $\phi(i) = x_i$ and add $\{f \mapsto \phi\}$ to $\mathbf{S}$. Otherwise, the behaviour of $\mathcal{F}_{\mathsf{OEP}}$ is undefined.

The commands are sent to the adversary, and accepted from the adversary similarly to Fig. 1 and Fig. 2.

Fig. 4: The ideal functionality $\mathcal{F}_{\mathsf{OEP}}$

5 Real functionality

Our representation of oblivious extended permutations is based on the following simple result.

Theorem 1. *For any $m, n \in \mathbb{N}$ there exist $\ell_{n,m} = (1 + o(1))m \ln m$ and $g_{n,m} : \{1, \ldots, \ell_{n,m}\} \to \{1, \ldots, n\}$, such that for any function $\phi \in F_{n,m}$, there exist $\sigma \in S_n$ and $\tau \in S_{\ell_{n,m}}$, such that $\phi(x) = (\sigma \circ g_{n,m} \circ \tau)(x)$ for all $x \in \{1, \ldots, m\}$.*

Proof. Define $\ell_{n,m}$ by

$$\begin{aligned} \ell_{0,m} &= 0 \\ \ell_{n,m} &= \ell_{n-1,m} + \lfloor m/n \rfloor \ . \end{aligned}$$

Then $\ell_{n,m} = (1 + o(1))m \ln m$ [32]. Define $g_{n,m}$ by

$$g_{n,m}(x) = k \Leftrightarrow \ell_{k-1,m} < x \leq \ell_{k,m} \ .$$

Let $\phi \in F_{n,m}$ be given. For each $y \in \{1, \ldots, n\}$, let $\phi^{-1}(y) = \{x \mid \phi(x) = y\}$. Let the permutation $\sigma \in S_n$ be such, that $|\phi^{-1}(\sigma(i))| \geq |\phi^{-1}(\sigma(i+1))|$ for all i. Note that $|\phi^{-1}(\sigma(i))| \leq \lfloor m/i \rfloor$.

Let $D_i = \{\ell_{i-1,m} + 1, \ldots, \ell_{i,m}\}$. Note that $|D_i| = \lfloor m/i \rfloor$ and $g_{n,m}(D_i) = \{i\}$. Let the permutation $\tau \in S_{\ell_{n,m}}$ be defined so, that for all $i \in \{1, \ldots, n\}$, we have $\tau(\phi^{-1}(\sigma(i))) \subseteq D_i$. Such permutation τ exists, because $|\phi^{-1}(\sigma(i))| \leq |D_i|$ and different sets D_i are disjoint.

For each $x \in \{1, \ldots, m\}$, we now have $\tau(x) \in D_{\sigma^{-1}(\phi(x))}$, implying $g_{n,m}(\tau(x)) = \sigma^{-1}(\phi(x))$ or $\sigma(g_{n,m}(\tau(x))) = \phi(x)$. $\square$

We see that σ sorts the elements of $\{1, \ldots, n\}$ according to their number of preimages with respect to ϕ. The mapping $g_{n,m}$ creates a sufficient number of copies of each element. These copies are brought to their correct places by τ.

There are machines $M_1, \dots, M_k$ executing the protocols on behalf of the parties $P_1, \dots, P_k$ participating in the protocol. These machines have access to the functionality $\mathcal{F}_{\mathsf{ABB}}$.
There is a public function $f \mapsto (s_f, t_f)$ mapping variable names for EPs to pairs of variable names for shuffles. We assume that these variable names for shuffles are not used outside this protocol set.
A machine M_i responds to the various commands as follows.

Commands for $\mathcal{F}_{\mathsf{ABB}}$. When receiving a command for $\mathcal{F}_{\mathsf{ABB}}$ from the environment, M_i forwards it to $\mathcal{F}_{\mathsf{ABB}}$ and gives back the result.

Input an EP. On input $(\mathsf{input}, f, \phi)$, the machine M_i constructs the shuffles σ and τ corresponding to ϕ according to the proof of Thm. 1. It will then send commands $(\mathsf{input}, s_f, \sigma)$ and $(\mathsf{input}, t_f, \tau)$ to $\mathcal{F}_{\mathsf{ABB}}$.

Input an EP. On input $(\mathsf{input}, f, n, m)$, the machine M_i sends commands (input, s_f, n) and $(\mathsf{input}, t_f, \ell_{n,m})$ to $\mathcal{F}_{\mathsf{ABB}}$.

Classify an EP. On input $(\mathsf{classify}, f, \phi)$, the machine M_i constructs the shuffles σ and τ corresponding to ϕ according to the proof of Thm. 1. Any indeterminacys in the proof are solved in the same, public manner by all parties. Machine M_i will then send the commands $(\mathsf{classify}, s_f, \sigma)$ and $(\mathsf{classify}, t_f, \tau)$ to $\mathcal{F}_{\mathsf{ABB}}$.

Apply an EP. On input $(\mathsf{apply}, u_1, \dots, u_m; v_1, \dots, v_n; f)$, machine M_i will pick $\ell_{n,m}$ new variable names $w_1, \dots, w_{\ell_{n,m}}$ (for scalars). After that, it will
1. send $(\mathsf{apply}, w_1, \dots, w_n; v_1, \dots, v_n; s_f)$ to $\mathcal{F}_{\mathsf{ABB}}$;
2. copy $w_1, \dots, w_n$ to $w_1, \dots, w_{\ell_{n,m}}$ according to $g_{n,m}$, i.e. w_i after copying will be equal to $w_{g_{n,m}(i)}$ before copying (note that this is an operation of $\mathcal{F}_{\mathsf{ABB}}$);
3. send $(\mathsf{apply}, u_1, \dots, u_m, w_{m+1}, \dots, w_{\ell_{n,m}}; w_1, \dots, w_{\ell_{n,m}}; t_f)$ to $\mathcal{F}_{\mathsf{ABB}}$.

Fig. 5: The protocol set π_{OEP} (partially)

Theorem 1 immediately suggests the private encoding for an OEP ϕ. In our implementation π_{OEP} for $\mathcal{F}_{\mathsf{OEP}}$, we will store them as pairs of private shuffles (σ, τ), defined as in the proof of Thm. 1. Fig. 5 depicts the protocol set π_{OEP} (except for the convert-protocol, which is given in Sec. 6), defined in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model.

Theorem 2. *The protocol set π_{OEP}, as depicted in Fig. 5, is at least as secure as $\mathcal{F}_{\mathsf{OEP}}$ without* convert-*commands.*

Proof. We have to show a simulator $\mathcal{S}$ that can translate between the messages at the adversarial interface of $\mathcal{F}_{\mathsf{OEP}}$ and the messages at the adversarial interface of π_{OEP}. The simulator $\mathcal{S}$ has no long-term state and works as follows:

– On an input $(c, \dots)$ from $\mathcal{F}_{\mathsf{OEP}}$ that corresponds to a command for $\mathcal{F}_{\mathsf{ABB}}$, the simulator forwards this input to the adversary.
– On input $(\mathsf{input}, f, n, m)$ from $\mathcal{F}_{\mathsf{OEP}}$, the simulator forwards the commands (input, s_f, n) and $(\mathsf{input}, t_f, \ell_{n,m})$ to the adversary.
– On input $(\mathsf{classify}, f, \phi)$ from $\mathcal{F}_{\mathsf{OEP}}$, the simulator computes the permutations σ and τ according to the proof of Theorem. 1, and forwards $(\mathsf{classify}, s_f, \sigma)$ and $(\mathsf{classify}, t_f, \tau)$ to the adversary.

– On input $(\mathsf{apply}, u_1, \ldots, u_m; v_1, \ldots, v_n; f)$ from $\mathcal{F}_{\mathsf{OEP}}$, the simulator forwards the commands for applying s_f, copying the variables and applying t_f to the adversary. The commands are the same as in Fig. 5.
– On input $(\mathsf{masq}^i, c, \ldots)$ from the adversary to $\mathcal{F}_{\mathsf{ABB}}$, the simulator $\mathcal{S}$ forwards that command to $\mathcal{F}_{\mathsf{OEP}}$, unless the command is part of an adversarial party's activity in the protocols of π_{OEP}. These are recognized through the inclusion of variable names s_f and t_f.
– On input $(\mathsf{masq}^i, \mathsf{input}, s_f, n)$ from the adversary, followed by $(\mathsf{masq}^i, \mathsf{input}, t_f, \ell_{n,m})$: send $(\mathsf{masq}^i, \mathsf{input}, f, n, m)$ to $\mathcal{F}_{\mathsf{OEP}}$.
– On input $(\mathsf{masq}^i, c, s_f, \sigma)$ from the adversary, followed by $(\mathsf{masq}^i, c, t_f, \tau)$, where c is either input or $\mathsf{classify}$: the simulator constructs $\phi = \sigma \circ g_{n,m} \circ \tau$ (where n, m are found from the descriptions of σ and τ), and sends $(\mathsf{masq}^i, c, f, \phi)$ to $\mathcal{F}_{\mathsf{OEP}}$.
– On input $(\mathsf{masq}^i, \mathsf{apply}, w_1, \ldots, w_n; v_1, \ldots, v_n; s_f)$ from the adversary, followed by the requests to copy the variables w_j according to $g_{n,m}$ and the input $(\mathsf{masq}^i, \mathsf{apply}, u_1, \ldots, u_m, w_{m+1}, \ldots, w_{\ell_{n,m}}; w_1, \ldots, w_{\ell_{n,m}}; t_f)$: send $(\mathsf{masq}^i, \mathsf{apply}, u_1, \ldots, u_m; v_1, \ldots, v_n; f)$ to the adversary.

Quite clearly, this simulator provides the necessary translation. Actually, the only non-trivial part of this simulator is the construction of ϕ from σ and τ provided by the adversary. Fortunately, there exists a ϕ for any σ and τ (of correct types). Hence π_{OEP} is secure even against active adversaries (if the protocol set implementing $\mathcal{F}_{\mathsf{ABB}}$ is secure against such adversaries). $\square$

The provided simulator $\mathcal{S}$ is valid for any attacks by the adversary. It can cope with active attacks and with dishonest majority. Hence $\mathcal{F}_{\mathsf{OEP}}$ provides the same security guarantees as $\mathcal{F}_{\mathsf{ABB}}$.

6 Converting a private vector to an OEP

Suppose we are given the numbers m, n, and a vector $([\![v_1]\!], \ldots, [\![v_m]\!])$, such that $1 \leq v_i \leq n$ for all i. We want to construct $[\![\phi]\!]$, such that $\phi \in F_{n,m}$ and $\phi(i) = v_i$ for all i. For this, we have to construct private shuffles $[\![\sigma]\!]$ and $[\![\tau]\!]$ of correct size, such that $\phi = \sigma \circ g_{n,m} \circ \tau$. As we show below, the functionality provided by $\mathcal{F}_{\mathsf{ABB}}$ is sufficient for this construction. However, the construction is more complex than what we have seen before.

Partially specified shuffles. The following subtask occurs in the construction of both σ and τ. Let a vector $([\![v_1]\!], \ldots, [\![v_n]\!])$ be given, such that $v_i \in \{0, 1, \ldots, n\}$ and for each $j \in \{1, \ldots, n\}$ there exists at most one i, such that $v_i = j$. Construct $[\![\sigma]\!]$, where $\sigma \in S_n$ and $\forall i \in \{1, \ldots, n\} : v_i > 0 \Rightarrow \sigma(i) = v_i$.

We cannot directly apply Alg. 1 to obtain the shuffle, because this algorithm assumes that the input vector is a permutation of $\{1, \ldots, n\}$. In particular, the correctness of Alg. 1 hinges on the sorted vector being equal to $(1, \ldots, n)$.

Will we hence first fill the zeroes in the vector $v_1, \ldots, v_n$ with the missing numbers. We use Alg. 2 for that, making the call $\mathsf{FillBlanks}(1, n; [\![v_1]\!], \ldots, [\![v_n]\!])$. After that, we can apply Alg. 1 and obtain a suitable $[\![\sigma]\!]$.

Algorithm 2: FillBlanks, filling the blank squares of a shuffle

```
Data: Bounds $L, H \in \mathbb{N}$
Data: $[\![v_L]\!], [\![v_{L+1}]\!], \ldots, [\![v_H]\!]$, where $v_i \in \{0, L, \ldots, H\}$, and for each
      $j \in \{L, \ldots, H\}$, there is at most one $i$, such that $v_i = j$
Result: $[\![u_L]\!], [\![u_{L+1}]\!], \ldots, [\![u_H]\!]$, where $\{u_L, \ldots, u_H\} = \{L, \ldots, H\}$ and
      $v_i > 0 \Rightarrow u_i = v_i$
1  if $L = H$ then
2      return $[\![L]\!]$
3  $M \leftarrow \lfloor (L+H)/2 \rfloor$
4  $[\![\xi]\!] \leftarrow \mathsf{sort}([\![v_L]\!]; [\![v_{L+1}]\!]; \ldots; [\![v_H]\!])$ ;                    // Fig. 3
5  $([\![v'_L]\!], \ldots, [\![v'_H]\!]) \leftarrow \mathsf{apply}([\![v_L]\!], \ldots, [\![v_H]\!]; [\![\xi]\!])$
6  foreach $i \in \{1, \ldots, H-M\}$ do
7      $[\![b_i]\!] \leftarrow [\![v'_{M+i}]\!] \leq M$
8      $([\![v'_{L+i-1}]\!], [\![v'_{M+i}]\!]) \leftarrow [\![b_i]\!]\ ?\ ([\![v'_{M+i}]\!], [\![v'_{L+i-1}]\!]) : ([\![v'_{L+i-1}]\!], [\![v'_{M+i}]\!])$
9  $([\![v'_L]\!], \ldots, [\![v'_M]\!]) \leftarrow \mathsf{FillBlanks}(L, M; [\![v'_L]\!], \ldots, [\![v'_M]\!])$
10 $([\![v'_{M+1}]\!], \ldots, [\![v'_H]\!]) \leftarrow \mathsf{FillBlanks}(M+1, H; [\![v'_{M+1}]\!], \ldots, [\![v'_H]\!])$
11 foreach $i \in \{1, \ldots, H-M\}$ do
12     $([\![v'_{L+i-1}]\!], [\![v'_{M+i}]\!]) \leftarrow [\![b_i]\!]\ ?\ ([\![v'_{M+i}]\!], [\![v'_{L+i-1}]\!]) : ([\![v'_{L+i-1}]\!], [\![v'_{M+i}]\!])$
13 $([\![u_L]\!], \ldots, [\![u_H]\!]) \leftarrow \mathsf{apply}([\![v'_L]\!], \ldots, [\![v'_H]\!]; \mathsf{invert}([\![\xi]\!]))$
14 return $([\![u_L]\!], \ldots, [\![u_H]\!])$
```

In Alg. 2, we have used a few conventions that have appeared elsewhere in the specifications of privacy-preserving algorithms. In line 7, the variable $[\![b_i]\!]$ will store either 1 or 0, depending on whether the comparison returns true. The line 8 contains an instance of the *binary choice* operator $[\![b]\!]\,?\,[\![x]\!] : [\![y]\!]$. Its result is equal to $[\![x]\!]$ if $b = 1$, and $[\![y]\!]$ if $b = 0$. It is typically computed as $[\![b]\!] \cdot ([\![x]\!] - [\![y]\!]) + [\![y]\!]$. In lines 8 and 12 we are actually using pairs of values in place of $[\![x]\!]$ and $[\![y]\!]$. Hence the effect of these lines is to swap $[\![v'_{L+i-1}]\!]$ and $[\![v'_{M+i}]\!]$ if $b_i = 1$, and otherwise leave them as is.

A number of operations in Alg. 2 can be performed in parallelized fashion. We use the convention that **foreach**-statements indicate vectorized computations. In addition to that, the recursive calls in lines 9 and 10 are executed in parallel.

Clearly, Alg. 2 is secure — it does not declassify any values. Also, it does not input any values from a particular party, hence there are no issues in making sure that these values are valid. Alg. 2 invokes a number of commands of $\mathcal{F}_{\mathsf{ABB}}$ in order to transform a private vector to a different private vector. The adversary's view of Alg. 2 consists of the sequence of the names of these commands. This sequence can be derived from L, H, and the names of the variables input to Alg. 2.

Due to the need to preserve the privacy of $[\![v_i]\!]$, Alg. 2 is quite non-trivial, working in the divide-and-conquer fashion. The main case starts in line 3, and the lines 3–8 are used to rearrange the elements of $v_L, \ldots, v_H$ so, that $v_L, \ldots, v_M$ only contain elements in $\{0, L, \ldots, M\}$, and $v_{M+1}, \ldots, v_H$ only contain elements in $\{0, M+1, \ldots, H\}$. We record ξ and $b_1, \ldots, b_{H-M}$ that are sufficient to undo this rearrangement later. Through recursive calls in lines 9 and 10, we fill in the zeroes among $v'_L, \dots, v'_H$ with missing numbers. Finally, in lines 11–13 we undo the rearrangement we introduced at the beginning.

Assuming that the complexity of sorting is $O(n \log n)$, the overall complexity of FillBlanks is $O(n \log^2 n)$. We could actually simplify the algorithm somewhat — in line 10, the vector that is the argument to the recursive call is already sorted, hence there is no need to sort it again in line 4. This does not reduce the asymptotic complexity, though, as the FillBlanks-call in line 9 still needs to sort its argument vector. In Sec. 7 we show that for certain implementations of the ABB, this vector (which is just a private rotation away from being sorted) can also be sorted more efficiently, bringing the overall complexity of FillBlanks down to $O(n \log n)$, the same as Alg. 1.

Finding vector representations of $[\![\sigma]\!]$ *and* $[\![\tau]\!]$*.* With the help of Alg. 2, we are now ready to present the computation of $[\![\sigma]\!]$ and $[\![\tau]\!]$, such that $\phi = \sigma \circ g_{n,m} \circ \tau$. Algorithm 3 depicts this computation. In the description of this algorithm, we will heavily use the notation $[\![\boldsymbol{v}]\!]$ for vectors with private components (but note that the length of the vector is public). Alg. 3 is secure for the same reasons as Alg. 2.

The algorithm to convert the vector $[\![\boldsymbol{v}^{(1)}]\!] = ([\![v_1^{(1)}]\!], \dots, [\![v_m^{(1)}]\!])$ to the private shuffles $[\![\sigma]\!], [\![\tau]\!]$ has the following components:

– Counting, how many times each value $x \in \{1, \dots, n\}$ occurs among $v_1, \dots, v_m$. We first sort the vector $\boldsymbol{v}^{(1)}$, giving the vector $\boldsymbol{v}^{(1')}$. In this vector, the different values x occur in continuous segments. Vector $\boldsymbol{v}^{(2')}$ marks the start of each segment and $\boldsymbol{v}^{(3')}$ additionally records the positions, where different segments start. Sorting according to $\boldsymbol{v}^{(3')}$ (a stable sort according to $\boldsymbol{v}^{(2')}$ would have had the same effect) brings the start positions together and their differences, recorded in $\boldsymbol{v}^{(4'')}$ are the counts of the values x (the lengths of the segments).
– Computing the vector representing σ, to be used as the argument to Vector2Shuffle. As we sort the vectors in non-decreasing order, the counts end up in the last n elements of $\boldsymbol{v}^{(4'')}$. We want to sort them in non-increasing order, hence we collect their negations in the vector $\boldsymbol{u}^{(2)}$. We apply the sorting permutation to the actual values, whose counts were in $\boldsymbol{u}^{(2)}$. We collect the actual values in $\boldsymbol{u}^{(1)}$, but we must be careful, because not all n values are necessary there. Fortunately, in vector $v^{(2'')}$, there is exactly one “1” for each possible value. Thus we obtain zeroes instead of missing values and can use the FillBlanks-algorithm to fill them out.
– Computing the vector representing τ^{-1}. This vector must have the values $\ell_{i-1,m} + 1, \ell_{i-1,m} + 2, \dots$ in the positions where the original vector $[\![\boldsymbol{v}^{(1)}]\!]$ had the i-th most often occurring values among $\{1, \dots, n\}$. We intend to compute this vector through prefix summation; this takes place in lines 24–25. While doing this prefix summation, we assume that $\boldsymbol{v}^{(1)}$ is sorted, we undo the sorting afterwards. In lines 18–23 we set up the vector $\boldsymbol{v}^{(5')}$ that serves as the argument to prefix summation. We know that in the middle of

Algorithm 3: From a vector of private values to an OEP

```
Data: $m, n \in \mathbb{N}$
Data: $[\![\boldsymbol{v^{(1)}}]\!] = ([\![v_1^{(1)}]\!], \ldots, [\![v_m^{(1)}]\!])$, where $1 \leq v_i^{(1)} \leq n$
Result: $[\![\sigma]\!], [\![\tau]\!]$, such that $(\sigma \circ g_{n,m} \circ \tau)(i) = v_i$ for all $i \in \{1, \ldots, m\}$
1  $[\![\xi_1]\!] \leftarrow \mathsf{sort}([\![\boldsymbol{v^{(1)}}]\!])$
2  $[\![\boldsymbol{v^{(1')}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v^{(1)}}]\!]; [\![\xi_1]\!])$
3  $[\![v_1^{(2')}]\!] \leftarrow 1$
4  foreach $i \in \{2, \ldots, m\}$ do $[\![v_i^{(2')}]\!] \leftarrow 1 - ([\![v_i^{(1')}]\!] \overset{?}{=} [\![v_{i-1}^{(1')}]\!])$
5  foreach $i \in \{1, \ldots, m\}$ do $[\![v_i^{(3')}]\!] \leftarrow i \cdot [\![v_i^{(2')}]\!]$
6  $[\![\xi_2]\!] \leftarrow \mathsf{sort}([\![\boldsymbol{v^{(3')}}]\!])$
7  $[\![\boldsymbol{v^{(1'')}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v^{(1')}}]\!]; [\![\xi_2]\!])$
8  $[\![\boldsymbol{v^{(2'')}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v^{(2')}}]\!]; [\![\xi_2]\!])$
9  $[\![\boldsymbol{v^{(3'')}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v^{(3')}}]\!]; [\![\xi_2]\!])$
10 foreach $i \in \{1, \ldots, m-1\}$ do $[\![v_i^{(4'')}]\!] \leftarrow [\![v_i^{(2'')}]\!] \mathbin{?} ([\![v_{i+1}^{(3'')}]\!] - [\![v_i^{(3'')}]\!]) : 0$
11 $[\![v_m^{(4'')}]\!] \leftarrow m + 1 - [\![v_m^{(3'')}]\!]$
12 foreach $i \in \{1, \ldots, n\}$ do
13     $[\![u_i^{(1)}]\!] \leftarrow m - n + i > 0 \wedge [\![v_{m-n+i}^{(2'')}]\!] \mathbin{?} [\![v_{m-n+i}^{(1'')}]\!] : 0$
14     $[\![u_i^{(2)}]\!] \leftarrow m - n + i > 0 \mathbin{?} -[\![v_{m-n+i}^{(4'')}]\!] : 0$
15 $[\![\xi_3]\!] \leftarrow \mathsf{sort}([\![\boldsymbol{u^{(2)}}]\!])$
16 $[\![\boldsymbol{u^{(1')}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{u^{(1)}}]\!]; [\![\xi_3]\!])$
17 $[\![\sigma]\!] \leftarrow \mathsf{Vector2Shuffle}(\mathsf{FillBlanks}(1, n; [\![\boldsymbol{u^{(1')}}]\!]))$
18 foreach $i \in \{1, \ldots, n\}$ do $[\![u_i^{(3')}]\!] \leftarrow \ell_{i-1,m} + 1$
19 $[\![\boldsymbol{u^{(3)}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{u^{(3')}}]\!]; \mathsf{invert}([\![\xi_3]\!]))$
20 foreach $i \in \{1, \ldots, m\}$ do
21     $j_i \leftarrow i - m + n$
22     $[\![v_i^{(5'')}]\!] \leftarrow \begin{cases} 1, & \text{if } i \leq 0 \vee \neg[\![v_i^{(2'')}]\!] \\ [\![u_{j_i}^{(3)}]\!], & \text{if } [\![v_i^{(2'')}]\!] \wedge \neg[\![v_{i-1}^{(2'')}]\!] \\ [\![u_{j_i}^{(3)}]\!] - [\![u_{j_i-1}^{(3)}]\!] + [\![u_{j_i-1}^{(2)}]\!] + 1, & \text{if } [\![v_{i-1}^{(2'')}]\!] \end{cases}$
23 $[\![\boldsymbol{v^{(5')}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v^{(5'')}}]\!]; \mathsf{invert}([\![\xi_2]\!]))$
24 $[\![v_1^{(6')}]\!] \leftarrow [\![v_1^{(5')}]\!]$
25 for $i = 2$ to $m$ do $[\![v_i^{(6')}]\!] \leftarrow [\![v_{i-1}^{(6')}]\!] + [\![v_i^{(5')}]\!]$
26 $[\![\boldsymbol{v^{(6)}}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v^{(6')}}]\!]; \mathsf{invert}([\![\xi_1]\!]))$
27 foreach $i \in \{m+1, \ldots, \ell_{n,m}\}$ do $[\![v_i^{(6)}]\!] \leftarrow 0$
28 $[\![\tau]\!] \leftarrow \mathsf{invert}(\mathsf{Vector2Shuffle}(\mathsf{FillBlanks}(1, \ell_{n,m}; [\![\boldsymbol{v^{(6)}}]\!])))$
29 return $[\![\sigma]\!], [\![\tau]\!]$
```

continuous segments of $\boldsymbol{v^{(1')}}$, the values in $\boldsymbol{v^{(5')}}$ have to be "1". At the border from i-th most to i'-th most occurring value, however, there should be jumps from the segment $[\ell_{i-1,m}+1,\ldots,\ell_{i,m}]$ to $[\ell_{i'-1,m}+1,\ldots,\ell_{i',m}]$. The length of these jumps depends on i, i', and on the length of the ending segment. These lengths of jumps are computed in line 22 (clearly, the expression there can be converted into a sequence of "? :"-operations). Different cases in this line correspond to the middle of continuous segments, the start of the first segment, and to the starts of following segments, respectively. The vector $\boldsymbol{u^{(2)}}$ contains the *negations* of the lengths of the continuous segments.

The running time of Alg. 3 is dominated by the call to $\mathsf{FillBlanks}$ in line 28. As the size of its argument is $O(m\log m)$, the running time of the algorithm is $O(m\log^3 m)$.

Alg. 3 is used in the protocol set π_{OEP} for converting a private vector to an OEP.

7 Specific operations with shuffles

In Sec. 5 we showed how the operations with extended permutations, defined in Sec. 4, can be implemented on top of the ABB defined in Sec. 3. The resulting implementation will have the same security guarantees as the used ABB, which can be implemented using several different protocol sets. Parts of the OEP implementation, however, can be more efficient if the ABB supports more operations with shuffles. In the following, we will present these operations, as well as their implementations on top of the ABB of Sharemind [6], based on additive sharing over $\mathbb{Z}_N$ by three parties, providing security against one semi-honest party.

Unfortunately, we have to expose an implementation artifact through the interface of $\mathcal{F}_{\mathsf{ABB}}$. Namely, we let the shuffles stored in $\mathbf{S}$ belong to two different *sorts* (this notion is unrelated to "sorting"), which we denote with "0" and "1". Except for inversion, the shuffles of both sorts behave identically to each other. In particular, the result of applying them is the same. The operations for inputing, classifying and making a random shuffle (Fig. 2), as well as sorting (Fig. 3) receive an extra public argument, denoting the sort of the resulting shuffle. The result of the inversion operation will have the sort different from the argument. Let "$s \overset{b}{\mapsto} \sigma$" denote that the variable s is assigned to the shuffle σ of sort b. During the execution, the sorts of all shuffles stored in $\mathcal{F}_{\mathsf{ABB}}$ can be deduced from public information.

The extra operations are depicted in Fig. 6. The permutation $\mathsf{rot}^m_x \in S_m$ is defined by $\mathsf{rot}^m_x(i) \equiv i + x \pmod m$. The operation $\mathsf{mkrotation}$ is another one taking the sort of the result as an argument. The compose operation and its interplay with inversion is the reason for introducing the sorts. The existence of the composition protocol depends on the order of parties performing the shuffles in $[\![\sigma]\!] = ([\![\sigma]\!]_1, [\![\sigma]\!]_2, [\![\sigma]\!]_3)$.

We show how operations in Fig. 6 can be implemented in an ABB based on additive sharing among three parties. In this case, $[\![v]\!] = ([\![v]\!]_1, [\![v]\!]_2, [\![v]\!]_3)$, where

Convert a value to a rotation. On input $(\mathsf{mkrotation}^b, v, s, m)$ from all parties, look up $x = \mathbf{S}(v)$ and add $\{s \overset{b}{\mapsto} \mathsf{rot}_x^m\}$ to $\mathbf{S}$.

Compose private shuffles. On input $(\mathsf{compose}, s_1, s_2, s_3)$ from all parties, where s_1 and s_2 have the same sort b, look up $\sigma_1 = \mathbf{S}(s_1)$ and $\sigma_2 = \mathbf{S}(s_2)$. Add $\{s_3 \overset{b}{\mapsto} \sigma_1 \circ \sigma_2\}$ to $\mathbf{S}$.

Fig. 6: More shuffle-related operations in $\mathcal{F}_{\mathsf{ABB}}$

Algorithm 4: Making a rotation

```
Data: A private value $[\![v]\!]$
Data: A public value $m$ and a bit $b$
Result: Private shuffle $[\![\mathsf{rot}_{v,m}]\!]$ of sort $b$
$[\![v']\!]' \leftarrow \mathsf{change\_modulus}([\![v]\!], m)$
Party $P_i$ sends $[\![v']\!]'_i$ to $P_{i+1}$
Parties $P_{1+2b}$ and $P_2$ define $[\![\sigma]\!]_1 \leftarrow \mathsf{rot}^m_{[\![v']\!]'_{1+b}}$
Parties $P_1$ and $P_3$ define $[\![\sigma]\!]_2 \leftarrow \mathsf{rot}^m_{[\![v']\!]'_3}$
Parties $P_2$ and $P_{3-2b}$ define $[\![\sigma]\!]_3 \leftarrow \mathsf{rot}^m_{[\![v']\!]'_{2-b}}$
return $[\![\sigma]\!]$
```

$v \in \mathbb{Z}_N$, $[\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3 \equiv v \pmod N$ for some fixed modulus N, and party P_i knows $[\![v]\!]_i$. In the implementation of Sharemind, the modulus N is a power of two, making the set of values isomorphic to $(\log N)$-bit unsigned integers. There exist protocols to increase or decrease the bitwidth of these values [6, Sec. 7]. Essentially the same protocols can be used for arbitrary changes of the modulus N. I.e. given N, M, and a sharing $[\![v]\!]$ over $\mathbb{Z}_N$, there is a protocol for the operation $[\![v']\!]' = \mathsf{change\_modulus}([\![v]\!], M)$, where $[\![v']\!]' = ([\![v']\!]'_1, [\![v']\!]'_2, [\![v']\!]'_3)$, and $[\![v']\!]'_1 + [\![v']\!]'_2 + [\![v']\!]'_3 \equiv v \pmod M$. These protocols are secure against one semi-honest adversary. The protocols are made composable by *resharing* [6, Alg. 1] their outputs.

Rotation. In Sharemind, a private shuffle of sort b is stored as $[\![\sigma]\!] = ([\![\sigma]\!]_1, [\![\sigma]\!]_2, [\![\sigma]\!]_3)$ with $[\![\sigma]\!]_1 \circ [\![\sigma]\!]_2 \circ [\![\sigma]\!]_3 = \sigma$. Here the party P_{1+2b} knows $[\![\sigma]\!]_1$ and $[\![\sigma]\!]_2$, party P_2 knows $[\![\sigma]\!]_1$ and $[\![\sigma]\!]_3$, and party P_{3-2b} knows $[\![\sigma]\!]_2$ and $[\![\sigma]\!]_3$. A protocol for converting a value $[\![v]\!]$ to a private rotation $[\![\sigma]\!]$ is given in Alg. 4. All indices of parties are given *modulo* 3.

Algorithm 4 is correct due to the equality $\mathsf{rot}^m_x \circ \mathsf{rot}^m_y = \mathsf{rot}^m_{(x+y) \bmod m}$. It is also secure against one semi-honest party, as the shares of $[\![v']\!]'$ are three random elements of $\mathbb{Z}_m$, subject to their sum being equal to v'. Each party learns two of those shares; these carry no information about v', even if combined with the party's view in the modulus change protocol (due to composability).

Composition. The protocol for composing two shared shuffles of sort 0 is given in Alg. 5. For sort 1, change the roles of P_1 and P_3.

```
Algorithm 5: Composing private shuffles of sort 0
  Data: Private shuffles $[\![\sigma]\!]$ and $[\![\tau]\!]$ of sort 0, where $\sigma, \tau \in S_m$
  Result: Private shuffle $[\![\xi]\!]$ of sort 0, such that $\xi = \sigma \circ \tau$
  Party $P_2$ randomly chooses $\rho_1, \rho_3 \in S_m$, such that $\rho_1 \circ \rho_3 = [\![\sigma]\!]_3 \circ [\![\tau]\!]_1$
  Parties $P_1$ and $P_3$ agree on randomly chosen $\pi_1, \pi_3 \in S_m$
  Party $P_2$ sends $\rho_1$ to $P_1$ and $\rho_3$ to $P_3$
  Party $P_1$ computes $[\![\xi]\!]_1 \leftarrow [\![\sigma]\!]_1 \circ [\![\sigma]\!]_2 \circ \rho_1 \circ \pi_1$ and sends it to $P_2$
  Party $P_3$ computes $[\![\xi]\!]_3 \leftarrow \pi_3 \circ \rho_3 \circ [\![\tau]\!]_2 \circ [\![\tau]\!]_3$ and sends it to $P_2$
  Parties $P_1$ and $P_3$ compute $[\![\xi]\!]_2 \leftarrow \pi_1^{-1} \circ \pi_3^{-1}$
  return $[\![\xi]\!]$
```

```
Algorithm 6: Composing private shuffles generically
  Data: Private shuffles $[\![\sigma_1]\!], \dots, [\![\sigma_k]\!]$, where $\sigma_i \in S_m$
  Result: Private shuffle $[\![\xi]\!]$, such that $\xi = \sigma_1 \circ \cdots \circ \sigma_k$
  foreach $i \in \{1, \dots, m\}$ do $[\![v_i]\!] \leftarrow \mathsf{classify}(i)$
  for $i = 1$ to $k$ do $([\![v_1]\!], \dots, [\![v_m]\!]) \leftarrow \mathsf{apply}([\![v_1]\!], \dots, [\![v_m]\!]; [\![\sigma_i]\!])$
  return Vector2Shuffle($[\![v_1]\!], \dots, [\![v_m]\!]$) // Alg. 1
```

The correctness of Alg. 5 is straightforward to verify. It is also obvious that it is secure against one semi-honest party: each party only receives random elements of S_m. Indeed, party P_1 [resp. P_3] receives the permutation ρ_1 [resp. ρ_3]. Alone, this is a random permutation, carrying no information about $[\![\sigma]\!]_3 \circ [\![\tau]\!]_1$. Party P_2 receives two permutations, $[\![\phi]\!]_1$ and $[\![\phi]\!]_3$. They are both masked with random permutations π_1 and π_3, hence being random themselves.

For completeness, we show that private shuffles can be composed also by using just the operations of $\mathcal{F}_{\mathsf{ABB}}$ in Fig. 1 and Fig. 2. The resulting protocol, shown in Alg. 6, is, however, significantly slower than Alg. 5 due to the need to sort a vector of length m.

8 Speeding up the conversion to OEP

Using the operations in Fig. 6, and assuming the complexities of their implementations to be the same as in Sharemind's set of protocols (constant for mkrotation, linear for composition), we can construct a version of the FillBlanks-algorithm that works in time $O(m \log m)$ for a vector of length m. In fact, as Alg. 3 only uses FillBlanks in composition with Vector2Shuffle, we will present a faster version of their composition, converting a vector representing a permutation with blank spots into a private shuffle.

The algorithm is presented as Alg. 7, with subroutines in Alg. 8 and Alg. 9. We see that structurally, they are similar to Alg. 2, but as certain arrays are almost sorted, we use more efficient methods to actually sort them.

Algorithms 7–9 compute certain permutations of their input vector, but instead of undoing these permutations later (like Alg. 2), they record these permutations and use them as parts of their return values.

Algorithm 7: VectorWithBlanks2Shuffle, From a vector of private values (with blank cells) to private shuffle

```
Data: $b \in \{0,1\}$, denoting the sort of the resulting shuffle
Data: $[\![v_1]\!], \ldots, [\![v_m]\!]$, where $v_i \in \{0,1,\ldots,m\}$, and for each $j \in \{1,\ldots,m\}$,
      there is at most one $i$, such that $v_i = j$
Result: A shuffle $[\![\sigma]\!]$, such that $v_i > 0 \Rightarrow \sigma(i) = v_i$
1 if $m = 1$ then
2     return $[\![\{1 \mapsto 1\}]\!]$
3 $[\![\xi]\!] \leftarrow \mathsf{sort}^{1-b}([\![v_1]\!]; \ldots; [\![v_m]\!])$
4 $([\![u_1]\!], \ldots, [\![u_m]\!]) \leftarrow \mathsf{apply}([\![v_1]\!]; \ldots; [\![v_m]\!]; [\![\xi]\!])$
5 $[\![\tau]\!] \leftarrow \mathsf{VectorWithBlanks2Shuffle}^{b}_{\mathsf{sort}}(1, m; [\![u_1]\!], \ldots, [\![u_m]\!])$
6 return $[\![\tau]\!] \circ \mathsf{invert}([\![\xi]\!])$
```

Algorithm 8: $\mathsf{VectorWithBlanks2Shuffle}_{\mathsf{sort}}$, a subroutine of VectorWithBlanks2Shuffle

```
Data: $b \in \{0,1\}$, denoting the sort of the resulting shuffle
Data: Bounds $L, H \in \mathbb{N}$
Data: Sorted vector $[\![v_L]\!], [\![v_{L+1}]\!], \ldots, [\![v_H]\!]$, where $v_i \in \{0, L, \ldots, H\}$, and for
      each $j \in \{L, \ldots, H\}$, there is at most one $i$, such that $v_i = j$
Result: A shuffle $[\![\sigma]\!]$, where $\sigma \in S_{H-L+1}$ and $v_i > 0 \Rightarrow \sigma(i-L+1) = v_i - L + 1$
1 if $L = H$ then
2     return $[\![\{1 \mapsto 1\}]\!]$
3 $M \leftarrow \lfloor (L+H)/2 \rfloor$
4 foreach $i \in \{1, \ldots, H-M\}$ do
5     $[\![b_i]\!] \leftarrow [\![v_{M+i}]\!] \leq M$
6     $[\![\chi_i]\!] \leftarrow \mathsf{mkrotation}^{1-b}([\![b_i]\!], 2)$
7 $\chi_{H-M+1} \leftarrow H - L$ is even $\;?\; \{1 \mapsto 1\} : \{\Box\}$
8 $[\![\tau]\!] \leftarrow \mu_{H-L+1} \circ ([\![\chi_1]\!] \| \cdots \| [\![\chi_{H-M}]\!] \| \chi_{H-M+1}) \circ \mu^{-1}_{H-L+1}$
9 $([\![v'_L]\!], \ldots, [\![v'_H]\!]) \leftarrow \mathsf{apply}([\![v_L]\!], \ldots, [\![v_H]\!]; [\![\tau]\!])$
10 $[\![\sigma_1]\!] \leftarrow \mathsf{VectorWithBlanks2Shuffle}^{b}_{\mathsf{rotate}}(L, M; [\![v'_L]\!], \ldots, [\![v'_M]\!])$
11 $[\![\sigma_2]\!] \leftarrow \mathsf{VectorWithBlanks2Shuffle}^{b}_{\mathsf{sort}}(M+1, H; [\![v'_{M+1}]\!], \ldots, [\![v'_H]\!])$
12 return $([\![\sigma_1]\!] \| [\![\sigma_2]\!]) \circ \mathsf{invert}([\![\chi]\!])$
```

In Alg. 8, let $\mu_n \in S_n$ be the permutation satisfying

$$\mu_n(2i+1) = i+1$$
$$\mu_n(2i) = \lceil n/2 \rceil + i$$

for all valid values of i. Also, let $\|$ denote the parallel composition of shuffles. This operation exists according to Fig. 2. In all algorithms, we let $\{1 \mapsto 1\}$ denote the only element of S_1 and $\{\Box\}$ denote the only element of S_0.

We see that the complexity of Alg. 7, except for the call to Alg. 8, is $O(m \log m)$, due to the need to sort the input vector. The complexities of both Alg. 8 and Alg. 9, except for the recursive calls, are $O(m)$, as this is the complexity of apply-

Algorithm 9: $\mathsf{VectorWithBlanks2Shuffle}_{\mathsf{rotate}}$, a subroutine of $\mathsf{VectorWithBlanks2Shuffle}$

```
  Data: $b \in \{0,1\}$, denoting the sort of the resulting shuffle
  Data: Bounds $L, H \in \mathbb{N}$
  Data: An almost sorted vector (requires a single rotation to be sorted)
        $[\![v_L]\!], [\![v_{L+1}]\!], \ldots, [\![v_H]\!]$, where $v_i \in \{0, L, \ldots, H\}$, and for each
        $j \in \{L, \ldots, H\}$, there is at most one $i$, such that $v_i = j$
  Result: A shuffle $[\![\sigma]\!]$, where $\sigma \in S_{H-L+1}$ and $v_i > 0 \Rightarrow \sigma(i-L+1) = v_i - L + 1$
1 if $L = H$ then
2     return $[\![\{1 \mapsto 1\}]\!]$
3 foreach $i \in \{1, \ldots, H-L\}$ do $[\![b_i]\!] \leftarrow [\![v_{L+i-1}]\!] > [\![v_{l+1}]\!]$
4 $[\![c]\!] \leftarrow \sum_{i=1}^{H-L} i \cdot [\![b_i]\!]$
5 $[\![\xi]\!] \leftarrow \mathsf{mkrotation}^{1-b}([\![c]\!], H-L+1)$
6 $([\![u_L]\!], \ldots, [\![u_H]\!]) \leftarrow \mathsf{apply}([\![v_L]\!]; \ldots; [\![v_H]\!]; [\![\xi]\!])$
7 $[\![\tau]\!] \leftarrow \mathsf{VectorWithBlanks2Shuffle}^{b}_{\mathsf{sort}}(L, H; [\![u_L]\!], \ldots, [\![u_H]\!])$
8 return $[\![\tau]\!] \circ \mathsf{invert}([\![\xi]\!])$
```

ing the shuffles they've computed, as well as the complexity of composing private shuffles. There are $O(\log m)$ levels of recursive calls, hence the total complexity of Alg. 7 is $O(m \log m)$.

If we use Alg. 7 as a subroutine of Alg. 3, instead of the composition of Alg. 1 and 2, we obtain an algorithm to convert a vector of private values of length m into an OEP with $O(m \log^2 m)$ work.

9 Benchmarks

The asymptotic complexity of our OEP protocol (both communication and computation) is $O(m \log m)$ for an extended permutation $f \in F_{n,m}$ (assuming that m is at least $O(n)$) and for a constant number of parties. The asymptotic complexity of converting a vector of indices to an OEP is $O(m \log^3 m)$ in general and $O(m \log^2 m)$ with optimizations offered by Sharemind.

We have implemented protocols in Fig. 5 on top of the Sharemind secure multiparty computation framework, and tested their performance. In performance testing, we kept in mind the scenario of private function evaluation. For a circuit with I inputs, K binary gates and O outputs, the topology of the circuit is represented by an extended permutation in $F_{I+K,2K+O}$.

Our Sharemind cluster consists of three computers with 48 GB of RAM and a 12-core 3 GHz CPU with Hyper Threading running Linux (kernel v.3.2.0-3-amd64), connected by an Ethernet local area network with link speed of 1 Gbps. On this cluster, we have benchmarked the execution time of the OEP application protocol for extended permutations in $F_{200+K,2K+100}$ (for various values of K), simulating the oblivious evaluation of a circuit with 200 inputs and 100 outputs. The permutations were applied to 32-bit values. The running times are presented

$K/10^6$	running time
0.1	0.5
1	6
5	35
7	49
8	58

Table 1. Execution times for applying an OEP from $(K+200)$ inputs to $(2K+100)$ outputs (times in seconds)

in Table 1. The running time $t(K)$ (in seconds) is very well approximated by $4.54 \cdot 10^{-7} \cdot K \ln K$.

10 Private evaluation of branches

We have proposed a new, efficient construction for oblivious extended permutations, that is fully integrable with secure multiparty computation protocols for other operations. Practically usable private function evaluation is a possible application of our techniques, if combined with private evaluation of the gates in circuits. Recent advances in private function evaluation may make it a practical tool for certain subtasks in secure multiparty computation, e.g. for handling the branching on private values. Currently, such branchings are handled through the execution of both branches, followed by an oblivious choice [33].

It is reasonable to assume that any application of PFE will still attempt to use as much information that can be publicly deduced about the computed function. Flexibility of PFE techniques is necessary, in order to absorb all available information. The oblivious extended permutations proposed in this paper allow a much greater multitude of potential usage scenarios than [36]. It is possible to evaluate a function without anyone knowing, which function is being evaluated. This allows us to obliviously select the representation of a private function and then evaluate it, enabling branching on private values. In this case, we still need to construct private representations of both branches, but this computation can be moved to the offline phase. The actual selection of the privately executed branch can be very efficient [29].

In the following we describe how the representations of private functions could be prepared offline, and how the selection could be made during online phase. Our construction is geared towards the representations and protocols employed by Sharemind, using additive sharing among three parties. Consider the program fragment

```
switch(i) {
  case 1: f_1; break;
  case 2: f_2; break;
     ...
  case k: f_k; break;
}
```

where the value of `i` is secret. Let the topology of the circuit f_i be represented by the EP ϕ_i and the operations in its gates be recorded in the vector $\boldsymbol{v^{(i)}}$. We assume that for all i, the vectors $\boldsymbol{v^{(i)}}$ have the same length and $\phi_i \in F_{n,m}$ for some n and m not depending on i. This assumption is necessary for avoiding leaking information about i through the size of the branch. In practice, we expect padding to be used to make this assumption hold. The EPs ϕ_i and vectors $\boldsymbol{v^{(i)}}$ are public. Let $(\sigma_i, \tau_i) \in S_n \times S_{\ell_{n,m}}$ be the representation of ϕ_i as two permutations. Let $c_1, \ldots, c_k$ be public upper bounds on the number of times that the control flow reaches the *switch*-statement above, while the variable `i` has the value $1, \ldots, k$, respectively. Let $c = c_1 + \cdots + c_k$.

We show how to construct a private vector $[\![\boldsymbol{s}]\!]$ of length c containing c_i copies of σ_i in random order, and how to later select from it according to a private index i. The construction algorithm (for offline running) is given in Alg. 10 and the selection algorithm in Alg. 11. The construction algorithm can be adapted for private function evaluation — instead of σ_i it should receive as input the entire description of the function f_i, consisting of two permutations σ_i and τ_i, as well as the encodings of the gates of f_i. The adapted algorithm would return several vectors instead of $[\![\boldsymbol{s}]\!]$. The selection algorithm stays almost the same, only picking the ℓ-th element of all vectors, not just $\boldsymbol{s}$.

The selection algorithm needs to perform private lookups from vectors $[\![\boldsymbol{w^{(i)}}]\!]$. For this, we could use the algorithms from [29], which perform only a constant number of multiplications (and other expensive operations) during the actual execution, having shifted most of their complexity to the offline pre-processing phase.

We see that the order of private permutations in the vector $[\![\boldsymbol{s}]\!]$ constructed in Alg. 10 is determined by the composition of random permutations $\xi, \pi \in S_c$. No party knows both permutations, hence the it has no information about the order through ξ or π. Also, the resharing operations that the parties perform with the permutations σ_i only make the parties to receive random permutations in S_n, hence they cannot associate the values they receive with the original values σ_i and the order $\xi \circ \pi$ does not leak through it. Finally, the operations Alg. 10 performs to produce the vectors $[\![\boldsymbol{w^{(1)}}]\!], \ldots, [\![\boldsymbol{w^{(k)}}]\!]$ only use the ABB operations and do not leak anything.

In Alg. 11, elements of the vector $[\![\boldsymbol{s}]\!]$ are selected one by one, by exposing the index of the element to be returned. Each element is selected at most once, and the order of selection is completely random — the results of declassification could be simulated by generating random elements of $\{1, \ldots, c\}$ without repeats.

Acknowledgements

This work has been supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 284731 "Usable and Efficient Secure Multiparty Computation (UaESMC)". It was also supported by Estonian Research Council through grant no. IUT27-1, and by European Regional

Algorithm 10: Constructing a vector of private permutations

```
Data: Public $\sigma_1, \dots, \sigma_k \in S_n$, $c_1, \dots, c_k \in \mathbb{N}$
Result: Permuted $\sigma$-s: a vector $[\![\boldsymbol{s}]\!]$ of length $c = c_1 + \cdots + c_k$ containing $c_i$
        shared copies of $\sigma_i$ (of sort 0) in random positions.
Result: For each $i$ a vector $[\![\boldsymbol{w^{(i)}}]\!]$ of length $c_i$, indicating, which elements of $\boldsymbol{s}$
        are equal to $\sigma_i$.
Let $\boldsymbol{z} \in \mathbb{N}^c$ be a non-decreasing vector containing $c_i$ copies of $i$
$P_2$ randomly chooses $\xi \in S_c$
foreach $i \in \{1, \dots, c\}$ do
    $P_2$ sets $\sigma'_i \leftarrow \sigma_{z_{\xi(i)}}$
    $P_2$ chooses random $\sigma'_{i,1}, \sigma'_{i,3} \in S_n$, such that $\sigma'_{i,1} \circ \sigma'_{i,3} = \sigma'_i$
    $P_2$ sends $\sigma'_{i,1}$ to $P_1$
    $P_2$ sends $\sigma'_{i,3}$ to $P_3$
$P_1$ and $P_3$ agree on a random $\pi \in S_c$
foreach $i \in \{1, \dots, c\}$ do
    $P_1$ and $P_3$ agree on random $\pi_{i1}, \pi_{i3} \in S_n$
    $P_1$ sets $[\![s_i]\!]_1 \leftarrow \sigma'_{\pi(i),1} \circ \pi_{i1}^{-1}$ and sends it to $P_2$
    $P_3$ sets $[\![s_i]\!]_3 \leftarrow \pi_{i3}^{-1} \circ \sigma'_{\pi(i),3}$ and sends it to $P_2$
    $P_1$ and $P_3$ set $[\![s_i]\!]_2 \leftarrow \pi_{i1} \circ \pi_{i3}$
$[\![\xi^{-1}]\!] \leftarrow \mathsf{input}(\xi^{-1})$ ;                        // $P_2$ shares $\xi^{-1}$
foreach $i \in \{1, \dots, c\}$ do $P_1$ and/or $P_3$ set $v_{\pi(i)} \leftarrow i$
$[\![\boldsymbol{v}]\!] \leftarrow \mathsf{input}(\boldsymbol{v})$ ;                  // $P_1$ and/or $P_3$ shares $\boldsymbol{v}$
$[\![\boldsymbol{w}]\!] \leftarrow \mathsf{apply}([\![\boldsymbol{v}]\!]; [\![\xi^{-1}]\!])$
foreach $i \in \{1, \dots, k\}$ do
    $\mathit{offset}_i \leftarrow \sum_{j=1}^{i-1} c_j$
    foreach $j \in \{1, \dots, c_i\}$ do $[\![w_j^{(i)}]\!] \leftarrow [\![w_{\mathit{offset}_i + j}]\!]$
return $[\![\boldsymbol{s}]\!], [\![\boldsymbol{w^{(1)}}]\!], \dots, [\![\boldsymbol{w^{(k)}}]\!]$
```

Algorithm 11: Selecting a private permutation

```
Data: Public $\sigma_1, \dots, \sigma_k$, $c_1, \dots, c_k$
Data: Vectors $[\![\boldsymbol{s}]\!], [\![\boldsymbol{w^{(1)}}]\!], \dots, [\![\boldsymbol{w^{(k)}}]\!]$ output by Alg. 10
Data: Private index $[\![idx]\!]$ with $idx \in \{1, \dots, k\}$
Result: A private permutation $[\![t]\!]$, such that $t = \sigma_{idx}$
Internal state: numbers $[\![j_1]\!], \dots, [\![j_k]\!]$ (initially 0)
foreach $i \in \{1, \dots, k\}$ do
    $[\![b_i]\!] \leftarrow ([\![idx]\!] \overset{?}{=} i)$
    $[\![j_i]\!] := [\![j_i]\!] + [\![b_i]\!]$
    $[\![\ell_i]\!] \leftarrow \mathsf{lookup}([\![\boldsymbol{w^{(i)}}]\!], [\![j_i]\!])$
$\ell \leftarrow \mathsf{declassify}(\sum_{i=1}^{k} [\![b_i]\!] \cdot [\![\ell_i]\!])$
return $[\![s_\ell]\!]$
```

Development Fund through the Estonian Center of Excellence in Computer Science (EXCS).

References

1. SecureSCM. Technical report D9.1: Secure Computation Models and Frameworks. `http://www.securescm.org`, July 2008.
2. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, New York, NY, USA, 2008. ACM.
3. Dan Bogdanov and Aivo Kalu. Pushing back the rain—how to create trustworthy services in the cloud. *ISACA Journal*, 3:49–51, 2013.
4. Dan Bogdanov, Liina Kamm, Sven Laur, and Pille Pruulmann-Vengerfeldt. Secure multi-party data analysis: end user validation and practical experiments. Cryptology ePrint Archive, Report 2013/826, 2013. `http://eprint.iacr.org/`.
5. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
6. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.
7. Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *Financial Cryptography*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer, 2012.
8. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.
9. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.
10. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
11. Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.
12. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.
13. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multi-party computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

14. Martin Geisler. *Cryptographic Protocols: Theory and Implementation.* PhD thesis, Aarhus University, February 2010.
15. Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.
16. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
17. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.
18. Roberto Guanciale, Dilian Gurov, and Peeter Laud. Private Intersection of Regular Languages. In *Proceedings of the 12th Annual Conference on Privacy, Security and Trust*. IEEE, 2014.
19. Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive, Report 2014/121, 2014. `http://eprint.iacr.org/`.
20. Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.
21. Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462, New York, NY, USA, 2010. ACM.
22. Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*. The Internet Society, 2012.
23. Dai Ikarashi, Ryo Kikuchi, Koki Hamada, and Koji Chida. Actively private and correct mpc scheme in $t < n/2$ from passively secure schemes with small overhead. Cryptology ePrint Archive, Report 2014/304, 2014. `http://eprint.iacr.org/`.
24. Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In Salil P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 575–594. Springer, 2007.
25. Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122, 2011. `http://eprint.iacr.org/`.
26. Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013.
27. Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In Gene Tsudik, editor, *Financial Cryptography*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008.
28. Peeter Laud and Alisa Pankova. Verifiable computation in multiparty protocols with honest majority. Cryptology ePrint Archive, Report 2014/060, 2014. `http://eprint.iacr.org/`.
29. Peeter Laud and Jan Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. Cryptology ePrint Archive, Report 2013/678, 2013. `http://eprint.iacr.org/`.
30. Sven Laur, Riivo Talviste, and Jan Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security*, volume 7954 of *LNCS*, pages 84–101. Springer, 2013.

31. Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.
32. Johann Peter Gustav Lejeune Dirichlet. Über die Bestimmung der Mittleren Werthe in der Zahlentheorie. *Abhandlungen der Köninglich Preussischen Akademie der Wissenschaften*, pages 69–83, 1849.
33. Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-Model Secure Computation. In *Proceedings of 2014 IEEE Symposium on Security and Privacy*. IEEE, 2014.
34. Lior Malka and Jonathan Katz. Vmcrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584, 2010. `http://eprint.iacr.org/`.
35. Payman Mohassel, Saeed Sadeghian, and Nigel P. Smart. Actively secure private function evaluation. Cryptology ePrint Archive, Report 2014/102, 2014. `http://eprint.iacr.org/`.
36. Payman Mohassel and Seyed Saeed Sadeghian. How to Hide Circuits in MPC: an Efficient Framework for Private Function Evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.
37. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
38. Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.
39. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.