Composable Oblivious Extended Permutations*

Peeter Laud and Jan Willemson

Cybernetica AS, Estonia
{peeter.laud|jan.willemson}@cyber.ee

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 [39] have split the task of oblivious circuit evaluation into two parts — obliviously evaluating the gates, and hiding the

* Supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 284731 "Usable and Efficient Secure Multiparty Computation (UaESMC)", by Estonian Research Council through grant no. IUT27-1, and by European Regional Development Fund through the Estonian Center of Excellence in Computer Science (EXCS).

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 [1,2,5,8,14,23,37] that allow the private computation of certain tasks with practically relevant sizes. There have been a number of successful applications of these frameworks [3, 6, 7, 28, 33]. 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] (these sets of parties may overlap). 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 input 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 or 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 [13], making their use in larger applications straightforward, and also greatly simplifying the security proofs. Our construction is conceptually simpler than [39], 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). It increases the variety of deployment scenarios for PFE applications, among which credit evaluation and background checks have been proposed [29]. With our construction, the private computation can be outsourced, and the evaluated function itself may be obtained through secure computation. We have implemented our proposed construction and provide benchmarking results.

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. In Sec. 4 we present the desired ideal functionality for OEPs, as well as the actual protocol set (together with security proofs), with the most complicated protocol appearing in Sec. 5. In Sec. 6 we present the benchmarking results of our implementation of the OEP protocol; according to our knowledge, this is the first such implementation. In Sec. 7, 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 [42]; it is evaluated with SMC techniques. Such constructions appear in [24, 29, 39]. In [38], the construction of [39] 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 with homomorphic encryption [24, Sec. 5.3.2]. This construction has better asymptotic complexity than the ones based on switching networks, but it requires many expensive public-key operations. Again, one computing party has to know the extended permutation.

Oblivious RAM (ORAM) [17, 19, 41] is a functionality that allows a server's memory to be read and written according to a client's private address. It is a more flexible construction than OEP, which fixes the indices ahead of time and performs many reads in parallel. The implementation of ORAM algorithms on top of SMC (which would be necessary to emulate OEP) is non-trivial and brings high overheads [12, 36].

A very simple construction for *shuffling* (permuting) the elements of a vector is given by Laur et al. [34]. Hamada et al. [21, 22] have used this construction to give fast sorting algorithms. Our constructions are also based on this form of shuffling protocols.

3 Preliminaries

Universal composability (UC) [9] 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: If protocol π_1 is at least as secure as the (ideal) functionality $\mathcal{F}_1$, and protocol π_2 incorporating $\mathcal{F}_1$ (i.e. π_2 has been realized in the $\mathcal{F}_1$-*hybrid model*) is at least as secure as $\mathcal{F}_2$, then π_2, where the calls to $\mathcal{F}_1$ have been replaced with the invocations of π_1, is also at least as secure as $\mathcal{F}_2$.

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 (for simplicity, we only consider static corruptions), 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

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. I.e. one does not attempt to hide from the adversary, which protocols are executed (who could determine it through traffic analysis). Only the processed values are hidden.
Some of the parties may be *corrupted*. 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.
The execution of $\mathcal{F}_{\mathsf{ABB}}$ takes place in *rounds*, with each party submitting its command for the current round, and the adversary submitting commands for corrupt parties. In implementations π_{ABB}, many rounds may take place in parallel.

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

coalitions of users may also be able to stop the execution of $\mathcal{F}_{\mathsf{ABB}}$. We will not define the behaviour of $\mathcal{F}_{\mathsf{ABB}}$ in exceptional situations (e.g. undefined variables), because their occurrence can be detected from public information.

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 [11, 15, 16, 18, 26, 40, 43], 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 [10], 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. [34] 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

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.

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 $\mathcal{F}_{\mathsf{ABB}}$

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. [34] introduce a construction, where 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 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. The sets A_i have to be carefully selected for the scheme to be secure. For $k = 3$ and $t = 1$, we may take $l = 3$ and $A_i = \{1, 2, 3\} \backslash \{i\}$.

It is straightforward to securely implement other operations in Fig. 2, based on the protocol of Laur et al. [34]. 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 [25, 30] are readily applicable to described shuffling protocols.

Oblivious shuffles are instrumental for fast sorting algorithms on private values [22]. 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 [27] 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 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.

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

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

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

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 of lexicographically, stably sorting the rows of a $m \times l$ table. 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 Our OEP functionality

The notion of *extended permutation (EP)* was introduced in [39] for encoding the topology of arithmetic or Boolean circuits. Mathematically, an EP ϕ from a length-n to a length-m sequence 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.

The representation of oblivious extended permutations used by our implementation π_{OEP} of $\mathcal{F}_{\mathsf{OEP}}$ is based on the following simple result.

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 $\mathcal{F}_{\mathsf{OEP}}$

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 $\ell_{n,m} = (1 + o(1))m \ln m$ [35]. 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 $g_{n,m}(z) = i$ for all $z \in D_i$. 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 $\sigma(g_{n,m}(\tau(x))) = \phi(x)$. □

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.

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 $\mathsf{convert}$-protocol, which is given in Sec. 5), defined in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model.

There are machines $M_1, \ldots, M_k$ executing the protocols on behalf of the parties $P_1, \ldots, P_k$ 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 indeterminacies 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 $(\mathsf{apply}, u_1, \ldots, u_m; v_1, \ldots, v_n; f)$, machine M_i will pick $\ell_{n,m}$ new variable names $w_1, \ldots, w_{\ell_{n,m}}$ (for scalars). After that, it will

1. send $(\mathsf{apply}, w_1, \ldots, w_n; v_1, \ldots, v_n; s_f)$ to $\mathcal{F}_{\mathsf{ABB}}$;
2. copy $w_1, \ldots, w_n$ to $w_1, \ldots, w_{\ell_{n,m}}$ 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 $(\mathsf{apply}, u_1, \ldots, u_m, w_{m+1}, \ldots, w_{\ell_{n,m}}; w_1, \ldots, w_{\ell_{n,m}}; t_f)$ to $\mathcal{F}_{\mathsf{ABB}}$.

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

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 $(c, \ldots)$ 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). □

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

5 Converting a private vector to an OEP

Suppose we are given the numbers m, n, and a vector $(\llbracket v_1 \rrbracket, \ldots, \llbracket v_m \rrbracket)$, such that $1 \leq v_i \leq n$ for all i. We want to construct $\llbracket \phi \rrbracket$, such that $\phi \in F_{n,m}$ and $\phi(i) = v_i$ for all i. For this, we have to construct private shuffles $\llbracket \sigma \rrbracket$ and $\llbracket \tau \rrbracket$ 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 $(\llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket)$ 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 $\llbracket \sigma \rrbracket$, 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 $(1, \ldots, n)$.

We will 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 $\mathsf{FillBlanks}(1, n; \llbracket v_1 \rrbracket, \ldots, \llbracket v_n \rrbracket)$. After that, we can apply Alg. 1 and obtain a suitable $\llbracket \sigma \rrbracket$.

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 $\llbracket b_i \rrbracket$ will store either 1 or 0, depending on whether the comparison returns true. The line 8 contains an instance of the *binary choice* operator $\llbracket b \rrbracket \,?\, \llbracket x \rrbracket : \llbracket y \rrbracket$. Its result is equal to $\llbracket x \rrbracket$ if $b = 1$, and $\llbracket y \rrbracket$ if $b = 0$. It is typically computed as $\llbracket b \rrbracket \cdot (\llbracket x \rrbracket - \llbracket y \rrbracket) + \llbracket y \rrbracket$. In lines 8 and 12 we are actually using pairs of values in place of $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$.

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** $([\![u_L]\!], \ldots, [\![u_H]\!])$

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 $v'_L, \ldots, v'_H$ 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 the full version of this paper [32] 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 $[\![\boldsymbol{v^{(1)}}]\!] = ([\![v_1^{(1)}]\!], \ldots, [\![v_m^{(1)}]\!])$ to the private shuffles $[\![\sigma]\!], [\![\tau]\!]$ performs the following steps. First, it counts how many times each value $x \in \{1, \ldots, n\}$ occurs among $v_1, \ldots, v_m$. 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).

Second, the algorithm computes 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.

Third, the algorithm computes the vector representing τ^{-1}. This vector must have the values $\ell_{i-1,m}+1, \ell_{i-1,m}+2, \ldots$ in the positions where the original vector $[\![\boldsymbol{v^{(1)}}]\!]$ had the i-th most often occurring values among $\{1, \ldots, n\}$. 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 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

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** $[\![v_i^{(2')}]\!] \leftarrow 1 - ([\![v_i^{(1')}]\!] \stackrel{?}{=} [\![v_{i-1}^{(1')}]\!])$
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** $[\![v_i^{(4'')}]\!] \leftarrow [\![v_i^{(2'')}]\!] \,?\, ([\![v_{i+1}^{(3'')}]\!] - [\![v_i^{(3'')}]\!]) : 0$
11 $[\![v_m^{(4'')}]\!] \leftarrow m + 1 - [\![v_m^{(3'')}]\!]$
12 **foreach** $i \in \{1, \ldots, n\}$ **do**
13 $\quad [\![u_i^{(1)}]\!] \leftarrow m - n + i > 0 \wedge [\![v_{m-n+i}^{(2'')}]\!] \,?\, [\![v_{m-n+i}^{(1'')}]\!] : 0$
14 $\quad [\![u_i^{(2)}]\!] \leftarrow m - n + i > 0 \,?\, -[\![v_{m-n+i}^{(4'')}]\!] : 0$
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 $\quad j_i \leftarrow i - m + n$
22 $$\quad [\![v_i^{(5'')}]\!] \leftarrow \begin{cases} 1, & \text{if } i \leq 0 \vee \neg[\![v_i^{(2'')}]\!] \\ [\![u_{j_i}^{(3)}]\!], & \text{if } [\![v_i^{(2'')}]\!] \wedge \neg[\![v_{i-1}^{(2'')}]\!] \\ [\![u_{j_i}^{(3)}]\!] - [\![u_{j_i-1}^{(3)}]\!] + [\![u_{j_i-1}^{(2)}]\!] + 1, & \text{if } [\![v_{i-1}^{(2'')}]\!] \end{cases}$$
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** $[\![\sigma]\!], [\![\tau]\!]$

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 FillBlanks in line 28. As the size of its argument is $O(m \log m)$, the running time of the algorithm

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

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

is $O(m \log^3 m)$. For ABB implementations based on additive sharing, it can be reduced to $O(m \log^2 m)$ [32].

Alg. 3 is used in the protocol set π_{OEP} for converting a private vector to an OEP. The security of this protocol trivially follows from universal composability (π_{OEP} provides the same security guarantees as π_{ABB} with regards to the number of parties the adversary can corrupt, and the kinds of attacks they can perform). Indeed, as declassification is not used in Alg 2 and 3 (we assume that Vector2Shuffle is implemented with the help of the sort-command), the entire communication on the interface between the protocol and the adversary consists of the names of the commands $\mathcal{F}_{\mathsf{ABB}}$ is executing. The sequence of these commands depends only on the problem size and can be trivially generated by the simulator.

6 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 $O(m \log^3 m)$.

We have implemented protocols in Fig. 5 on the Sharemind secure multiparty computation platform (providing security against passive attacks by one party out of three in total) 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 performance tests are performed on a cluster 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 in Table 1. The running time $t(K)$ (in seconds) is very well approximated by $4.54 \cdot 10^{-7} \cdot K \ln K$. As this has been the first implementation of OEPs (as far as we know), these numbers constitute the baseline for comparing further realizations.

The benchmarking of Alg. 3 is outside the scope of this paper.

7 Discussion

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.

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 [39]. 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 [31].

OEPs can be used for purposes other than PFE. Guanciale et al. [20] have implicitly applied them in the minimization of finite automata obtained through the product construction.

References

1. SecureSCM. Technical report D9.1: Secure Computation Models and Frameworks. `http://www.securescm.org` (July 2008)
2. Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security. pp. 257–266. ACM, New York, NY, USA (2008)
3. Bogdanov, D., Kalu, A.: Pushing back the rain—how to create trustworthy services in the cloud. ISACA Journal 3, 49–51 (2013)
4. Bogdanov, D., Kamm, L., Laur, S., Pruulmann-Vengerfeldt, P.: Secure multi-party data analysis: end user validation and practical experiments. Cryptology ePrint Archive, Report 2013/826 (2013), `http://eprint.iacr.org/`
5. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) ESORICS. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008)
6. Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: Keromytis, A.D. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 7397, pp. 57–64. Springer (2012)
7. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Financial Cryptography. pp. 325–343 (2009)

8. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: USENIX Security Symposium. pp. 223–239. Washington, DC, USA (2010)
9. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS. pp. 136–145. IEEE Computer Society (2001)
10. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC. Lecture Notes in Computer Science, vol. 3876, pp. 285–304. Springer (2006)
11. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS. Lecture Notes in Computer Science, vol. 8134, pp. 1–18. Springer (2013)
12. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious ram without random oracles. In: Ishai, Y. (ed.) TCC. Lecture Notes in Computer Science, vol. 6597, pp. 144–163. Springer (2011)
13. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 247–264. Springer (2003)
14. Geisler, M.: Cryptographic Protocols: Theory and Implementation. Ph.D. thesis, Aarhus University (February 2010)
15. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In: PODC. pp. 101–111 (1998)
16. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) STOC. pp. 169–178. ACM (2009)
17. Gentry, C., Goldman, K.A., Halevi, S., Jutla, C.S., Raykova, M., Wichs, D.: Optimizing oram and using it efficiently for secure computation. In: Cristofaro, E.D., Wright, M. (eds.) Privacy Enhancing Technologies. Lecture Notes in Computer Science, vol. 7981, pp. 1–18. Springer (2013)
18. Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: STOC. pp. 218–229. ACM (1987)
19. Goldreich, O., Ostrovsky, R.: Software Protection and Simulation on Oblivious RAMs. J. ACM 43(3), 431–473 (1996)
20. Guanciale, R., Gurov, D., Laud, P.: Private Intersection of Regular Languages. In: Proceedings of the 12th Annual Conference on Privacy, Security and Trust, pp. 112–120. IEEE (2014)
21. Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive, Report 2014/121 (2014), `http://eprint.iacr.org/`
22. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: Kwon, T., Lee, M.K., Kwon, D. (eds.) ICISC. Lecture Notes in Computer Science, vol. 7839, pp. 202–216. Springer (2012)
23. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: CCS '10: Proceedings of the 17th ACM conference on Computer and communications security. pp. 451–462. ACM, New York, NY, USA (2010)
24. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS. The Internet Society (2012)

25. Ikarashi, D., Kikuchi, R., Hamada, K., Chida, K.: 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/
26. Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: Vadhan, S.P. (ed.) TCC. Lecture Notes in Computer Science, vol. 4392, pp. 575–594. Springer (2007)
27. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122 (2011), http://eprint.iacr.org/
28. Kamm, L., Bogdanov, D., Laur, S., Vilo, J.: A new way to protect privacy in large-scale genome-wide association studies. Bioinformatics 29(7), 886–893 (2013)
29. Kolesnikov, V., Schneider, T.: A practical universal circuit construction and secure evaluation of private functions. In: Tsudik, G. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 5143, pp. 83–97. Springer (2008)
30. Laud, P., Pankova, A.: Verifiable Computation in Multiparty Protocols with Honest Majority. In: Chow, S.S.M., Liu, J.K. (eds.) Provable Security - 8th International Conference, ProvSec 2014, Hong Kong, October 9-10, 2014. Proceedings. Lecture Notes in Computer Science, Springer (2014), to appear
31. Laud, P., Willemson, J.: Universally composable privacy preserving finite automata execution with low online and offline complexity. Cryptology ePrint Archive, Report 2013/678 (2013), http://eprint.iacr.org/
32. Laud, P., Willemson, J.: Composable Oblivious Extended Permutations. Cryptology ePrint Archive, Report 2014/400 (2014), http://eprint.iacr.org/
33. Laur, S., Talviste, R., Willemson, J.: From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In: Applied Cryptography and Network Security, LNCS, vol. 7954, pp. 84–101. Springer (2013)
34. Laur, S., Willemson, J., Zhang, B.: Round-Efficient Oblivious Database Manipulation. In: Proceedings of the 14th International Conference on Information Security. ISC'11. pp. 262–277 (2011)
35. Lejeune Dirichlet, J.P.G.: Über die Bestimmung der Mittleren Werthe in der Zahlentheorie. Abhandlungen der Köninglich Preussischen Akademie der Wissenschaften pp. 69–83 (1849)
36. Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. In: TCC. pp. 377–396 (2013)
37. Malka, L., Katz, J.: Vmcrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584 (2010), http://eprint.iacr.org/
38. Mohassel, P., Sadeghian, S., Smart, N.P.: Actively secure private function evaluation. Cryptology ePrint Archive, Report 2014/102 (2014), http://eprint.iacr.org/
39. Mohassel, P., Sadeghian, S.S.: How to Hide Circuits in MPC: an Efficient Framework for Private Function Evaluation. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT. Lecture Notes in Computer Science, vol. 7881, pp. 557–574. Springer (2013)
40. Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)
41. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM Conference on Computer and Communications Security. pp. 299–310. ACM (2013)
42. Waksman, A.: A permutation network. J. ACM 15(1), 159–163 (1968)
43. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: FOCS. pp. 160–164. IEEE (1982)