Secure Multiparty Sorting Protocols with Covert Privacy

Peeter Laud and Martin Pettai

Cybernetica AS, Estonia
{peeter.laud|martin.pettai}@cyber.ee

Abstract. We introduce the notion of *covert privacy* for secret-sharing-based secure multiparty computation (SMC) protocols. We show how covertly or actively private SMC protocols, together with recently introduced verifiable protocols allow the construction of SMC protocols secure against active adversaries. For certain computational problems, the relative overhead of our protocols, when compared to protocols secure against passive adversaries only, approaches zero as the problem size increases.
We analyse the existing adaptations of sorting algorithms to SMC protocols and find that unless they are already using actively secure primitive protocols, none of them are covertly private or verifiable. We propose a covertly private sorting protocol based on radix sort, the relative overhead of which again approaches zero, when compared to the passively secure protocol. Our results reduce the computational effort needed to counteract active adversaries for a significant range of SMC applications, where sorting is used as a subroutine.

1 Introduction

Secure multiparty computation (SMC) [16] allows a set of mutually distrustful parties to jointly perform computations on the data they have, without anyone of them learning anything beyond the result of the computation. While starting out as a mere theoretical curiosity [17, 31, 10], both the protocols and the computing infrastructure have improved over the last two decades, such that significant real-world problems may be solved with them [8, 2, 3]. In these applications, the inputs may come from, and outputs may be learned by many different parties, but the protocols for actual computations are executed by just a small set of *computing parties* (typically two or three), upon which the other parties have to place their trust. The trust has a *threshold* nature: the parties believe that at most a certain fraction of computing parties have been corrupted. E.g. the Sharemind SMC platform [6], which is the basis of the work reported here, has three computing parties, one of which may be passively corrupted.

A passive adversary learns the internal states, the inputs and outputs of corrupted parties, but these parties still follow the prescribed protocol. More desirable is security against an active adversary, which may additionally instruct the parties to change the messages they send out. Security against active adversaries requires significantly more computational effort and/or communication to be used, either during the protocol execution or at some other time (pre- or post-computation) [13].

For certain computations, the verification of the correctness of the result is simpler than actually performing the computation. For privately performing such computations, we could use a passively secure SMC protocol for computing the result *and the proof of its correctness* and an actively secure protocol for the verification. An existing implementation of this idea for linear programming [14] ensures that the result of the computation is correct and the verification phase does not leak anything to an active adversary. In order to not leak anything at all to an active adversary, the computation protocol has to be *actively private* [4]. This property is stronger than passive security, but weaker than active security which also implies correctness of results. Sharemind's protocols for simple operations are actively private [29] and this property composes [4].

In this paper we study protocols for oblivious sorting, i.e. protocols that transform a vector of private values into another vector of private values that is sorted and is equal to the input vector as multisets. It is easier to verify the sortedness of a vector than it is to sort it. The proof of correctness of sorting is a *private permutation* [27] that transforms the input vector to the output vector; the verification of such proof means applying the permutation and checking the equality of two vectors. Sorting is an important primitive in SMC protocols, used extensively in data analysis and as a "substitute" for operations that cannot be naturally converted into SMC protocols [3, 23].

Existing passively secure oblivious sorting protocols either do not naturally produce proofs of correctness, or are not actively private, nor do they seem to be amenable to simple changes which would give active privacy without unduly hampering the performance. To obtain more efficient protocols, but still retain protection against adversaries that deviate from the protocol, intermediate adversarial models have been considered. A *covert adversary* is willing to deviate, but only if it is not caught in the process [1]. One can argue that for many, if not most practical applications of SMC, the protection against a covert adversary is as good as the protection against an active adversary, if detected deviations from the protocol can be brought in front of a court and appropriate punishments levied. Techniques of verifiable computation [15] may be used to turn passively secure SMC protocols into covertly secure protocols [25, 26]. These techniques introduce a *post-execution* verification phase to the protocols, where the parties check each other on whether they followed the protocol correctly.

In this paper we define *covert privacy.* A protocol is covertly private if an active adversary may affect its correctness, but if it learns something about the inputs of others, the honest parties will likely be notified of a potential information leak. A covertly private sorting protocol (also producing the proof permutation) together with a verification mechanism is sufficient for dealing with active adversaries only during verification. But it turns out that sorting protocols based on shuffling and data-dependent sorting algorithms are not even covertly private. As the main result of this paper, we propose a sorting protocol based on radix sort, which is covertly private and has complexity similar to the passively secure protocol.

2 Related work

Let us give an overview of SMC protocols for sorting. Sorting networks were the first tool to be adapted to SMC [30, 20]. Shuffling protocols for secret sharing based SMC protocol sets were first proposed in [27] and applied for sorting in [32]. There exist well-engineered SMC sorting protocols based on quicksort [19] and on radix sort [18]. The performance of different sorting protocols in a uniform setting has been evaluated in [5].

This paper also proposes an improved method for certain privacy-preserving computations in a manner that provides security against active adversaries. The current best-performing SMC protocol sets with such security are based on secret sharing over finite fields, making use of message authentication codes to detect misbehaviour [12, 22]. Such protocols make use of precomputed shared tuples of values that assist in performing non-linear operations (in particular, multiplication of shared values). This precomputation used to be several orders of magnitude more expensive than the actual execution of the protocol, but recent advances have brought down the cost [21]. An alternative, which we are following in this paper, is to verify the computation parties after the execution [25, 26] to make sure that they did not deviate from the protocol.

Our verifiable sorting protocol has similarities to the verifiable SMC protocols for linear programming [14]. Similarly to them, we use passively secure protocols to perform the actual computation and to find the proof of correctness, and higher-security protocols to verify the proof, thereby detecting incorrect results. Differently from them, we also require covert privacy from the actual computation, hence detecting any privacy leaks there may be.

3 Preliminaries

3.1 Universal Composability and Secure Multiparty Computation

We use SMC protocols to build large privacy-preserving applications, for which the security claims can reasonably only be made in compositional manner. Hence we present all our constructions in the *universal composability* (UC) framework [9], which allows us to state that a certain protocol is *at least as secure as* (or *simulates*) a certain ideal functionality.

Let $\mathcal{F}$ be an ideal functionality for n parties, i.e. it is a probabilistic interactive Turing machine with interfaces to communicate with n users, and with an adversary. Let π be an n-party protocol, i.e. it consists of n probabilistic interactive Turing machines $M_1, \ldots, M_n$, each communicating with a user (also called the *environment*), with an adversary and possibly also with some ideal functionalities (which are also part of π).

Definition 1. *An n-party protocol* π black-box simulates *the ideal functionality $\mathcal{F}$, if there exists a machine* Sim*, such that for all users $H = (P_1, \ldots, P_n)$ and adversaries $\mathcal{A}$, $\mathit{view}_{H\|\pi\|\mathcal{A}}(H) \approx \mathit{view}_{H\|\mathcal{F}\|(\mathsf{Sim}\|\mathcal{A})}(H)$.*

Here the view of H encompasses all messages it exchanges with either π or $\mathcal{F}$, and with $\mathcal{A}$. The probability distribution over such sequences of messages is denoted by $\mathit{view}_{\mathbf{C}}(H)$, where $\mathbf{C}$ is the system that contains H. We use "$\approx$" to denote the similarity of views; this similarity may mean either equality, or statistical or computational indistinguishability.

An adversary may corrupt some machine M_i by sending it a $\mathsf{corrupt}$-message; the machine forwards it to the i-th user (the party P_i). Afterwards, M_i will send to the adversary everything it has seen or will see. Also, the adversary determines which subsequent messages M_i sends to other machines and the user. An adversary is *passive* if it instructs M_i to always send the same messages it would have sent without being corrupted.

Denote $[n] = \{1, \ldots, n\}$. Let f be a *one-shot n-party functionality*, i.e. it is used to compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$, possibly in randomized manner. The *ideal functionality for f* is an interactive randomized Turing machine $\mathcal{F}^f_{\mathsf{sec}}$ that communicates with n parties and the adversary. It performs by first receiving the set of *corrupted parties* $\mathcal{C} \subseteq [n]$ from the adversary and sending $\mathsf{corrupt}$ to each party P_i, where $i \in \mathcal{C}$. The machine then receives x_i from each P_i. If $i \in \mathcal{C}$, then $\mathcal{F}^f_{\mathsf{sec}}$ forwards x_i to the adversary. For each $i \in \mathcal{C}$, the adversary sends x'_i to $\mathcal{F}^f_{\mathsf{sec}}$. For each $i \in [n]\backslash\mathcal{C}$, define $x'_i = x_i$. The machine $\mathcal{F}^f_{\mathsf{sec}}$ computes $(y'_1, \ldots, y'_n) = f(x'_1, \ldots, x'_n)$ and sends y'_i to the adversary for $i \in \mathcal{C}$. The adversary responds either with (stop, j) for some $j \in \mathcal{C}$, or with the values y_i for all $i \in \mathcal{C}$. In the first case, $\mathcal{F}^f_{\mathsf{sec}}$ sends (stop, j) to all parties and stops. In the latter case, define $y_i = y'_i$ for all $i \in [n]\backslash\mathcal{C}$. The machine $\mathcal{F}^f_{\mathsf{sec}}$ sends y_i to P_i for all $i \in [n]$ and stops. An adversary for the functionality $\mathcal{F}^f_{\mathsf{sec}}$ is *passive* if it always defines $x'_i = x_i$ and $y_i = y'_i$ for $i \in \mathcal{C}$.

There are a couple of different ways to specify the ideal functionality for SMC. One example of such idealization is the *arithmetic black box* [24], which is convenient to use when constructing higher-level SMC protocols from more basic ones (like addition or multiplication), obtaining the security proof (in sense of Def. 1) of the higher-level protocol almost for free. In this paper, our focus is different, we are discussing different security properties of certain kinds of SMC protocols.

We are interested in SMC protocols based on secret sharing. These protocols are built up from basic protocols for certain operations that turn the sharings of the inputs of that operation into sharing of outputs. If x is a value then we let $[\![x]\!] = ([\![x]\!]_1, \ldots, [\![x]\!]_n)$ denote an arbitrary sharing of x, where $[\![x]\!]_i$ is the i-th share held by the i-th party. A one-shot functionality $f_\otimes$ for an operation $\otimes$ receives as inputs the sharings of the arguments of $\otimes$. It reconstructs the arguments, applies $\otimes$ to obtain the result and returns an arbitrary sharing of it.

In our modeling, there is a particular functionality that we want to securely compute, we consider it as a composition $f_1; f_2; \cdots ; f_k$ of one-shot functionalities (each f_i may also have an arbitrary number of arguments that it simply passes through). The ideal functionality $\mathcal{F}_{\mathrm{sec}}^{f_1;\cdots;f_k}$ is the sequential composition of $\mathcal{F}_{\mathrm{sec}}^{f_1};\cdots;\mathcal{F}_{\mathrm{sec}}^{f_k}$. Here only $\mathcal{F}_{\mathrm{sec}}^{f_1}$ receives its inputs from H and only $\mathcal{F}_{\mathrm{sec}}^{f_k}$ sends its outputs back to it. Otherwise, the inputs are received from previous functionality and passed on to the next one. If $\mathcal{F}_{\mathrm{sec}}^{f_i}$ is securely implemented by the protocol π_i, then the sequential composition of these protocols securely implements $\mathcal{F}_{\mathrm{sec}}^{f_1};\cdots;\mathcal{F}_{\mathrm{sec}}^{f_k}$, as implied by the universal composability theorem [9] (which is actually more general than that).

3.2 Privacy vs. Security

Privacy is a security property that for many protocols, including SMC protocols based on secret sharing may be easier to achieve than "full security". For a given one-shot functionality f, privacy is defined as simulating a different ideal functionality $\mathcal{F}_{\mathrm{priv}}^{f}$. The machine $\mathcal{F}_{\mathrm{priv}}^{f}$ works identically to $\mathcal{F}_{\mathrm{sec}}^{f}$ until computing resulting values $(y_1,\ldots,y_n) = f(x'_1,\ldots,x'_n)$. Afterwards the machine $\mathcal{F}_{\mathrm{priv}}^{f}$ stops, i.e. no other machine actually receives the computed values. A protocol π is a private realization of f if it black-box simulates $\mathcal{F}_{\mathrm{priv}}^{f}$ (Def. 1).

The values computed by $\mathcal{F}_{\mathrm{priv}}^{f}$ are still used when sequentially composing such functionalities. In the composition $\mathcal{F}_{\mathrm{priv}}^{f_1};\cdots;\mathcal{F}_{\mathrm{priv}}^{f_k}$, each $\mathcal{F}_{\mathrm{priv}}^{f_i}$ passes the result values to $\mathcal{F}_{\mathrm{priv}}^{f_{i+1}}$ to be used as inputs. The outputs from $\mathcal{F}_{\mathrm{priv}}^{f_k}$ still go nowhere. In addition to that, the inputs received by $\mathcal{F}_{\mathrm{priv}}^{f_2},\ldots,\mathcal{F}_{\mathrm{priv}}^{f_k}$ are not sent to the adversary, nor can the adversary adjust them. Hence privacy means only that the adversary's view during the protocol, but not necessary after it, can be simulated from just the inputs to the adversarially controlled parties.

Sequential composition of private protocols is again private [4]. In [4], this result is technically shown only for passive adversaries, but nowhere does the proof make use of the passiveness.

3.3 Protocols for Oblivious Sorting

Suppose we are given a vector of secret-shared values $[\![\boldsymbol{x}]\!] = ([\![x_1]\!],\ldots,[\![x_k]\!])$. SMC frameworks, e.g. Sharemind, contain protocols for comparing, adding and multiplying shared values (the results are again shared). These can be combined to an oblivious *compare-exchange node* that computes $[\![\min\{y,z\}]\!]$ and $[\![\max\{y,z\}]\!]$ from $[\![y]\!]$ and $[\![z]\!]$. Such nodes can be used to construct sorting networks for secret-shared data.

Common sorting algorithms are harder to adapt for SMC, because there is no simple means to access an element of a vector by a secret-shared index. Declassifying the accessed indices would leak information about the initial ordering of the vector elements. This can be overcome by first performing a random shuffle of the elements of the vector. The shuffling protocol [27], adapted for Sharemind, works as follows. Let the elements of the vector $[\![\boldsymbol{x}]\!]$ of length k be additively shared (over $\mathbb{Z}_N$ for some integer N) among three parties, two of which are honest and third may be passively corrupted. A permutation $\sigma \in S_k$ is shared as

```
  Data: shared vector $[\![\boldsymbol{x}]\!]$, private permutation $[\![\![\sigma]\!]\!] = (\sigma_1, \sigma_2, \sigma_3)$
  Result: vector $[\![\boldsymbol{x}]\!]$ permuted according to $\sigma$
1 for $i = 1$ to $3$ do
2     $P_i$ randomly generates $\boldsymbol{r} \in \mathbb{Z}_N^k$
3     $P_i$ sends $\boldsymbol{r}$ to $P_{i-1}$ and $\boldsymbol{y} := [\![\boldsymbol{x}]\!]_i - \boldsymbol{r}$ to $P_{i+1}$
4     $P_{i-1}, P_i, P_{i+1}$ update
      $([\![\boldsymbol{x}]\!]_{i-1}, [\![\boldsymbol{x}]\!]_i, [\![\boldsymbol{x}]\!]_{i+1}) := ([\![\boldsymbol{x}]\!]_{i-1} + \boldsymbol{r}, (0, 0, \ldots, 0), [\![\boldsymbol{x}]\!]_{i+1} + \boldsymbol{y})$
5     $P_{i-1}$ and $P_{i+1}$ reorder $[\![\boldsymbol{x}]\!]_{i-1}$ and $[\![\boldsymbol{x}]\!]_{i+1}$ by $\sigma_i$
6 $\forall i : P_i$ randomly generates $\boldsymbol{r}_i \in \mathbb{Z}_N^k$ and sends it to $P_{i+1}$
7 $\forall i : P_i$ updates $[\![\boldsymbol{x}]\!]_i := [\![\boldsymbol{x}]\!]_i + \boldsymbol{r}_i - \boldsymbol{r}_{i-1}$
```
Algorithm 1: Shuffling protocol of [27]

three random permutations $[\![\![\sigma]\!]\!] = (\sigma_1, \sigma_2, \sigma_3) \in S_k^3$, satisfying $\sigma_3 \circ \sigma_2 \circ \sigma_1 = \sigma$, such that party P_i knows the permutations σ_{i-1} and σ_{i+1} (the indices are *modulo* 3). The shuffling protocol is given in Alg. 1. Its main component is for each pair of parties, to turn the additive sharing among three parties into additive sharing among this pair and apply one of the permutations σ_i. The protocol ends with a *resharing* step, after which all shares are random again. A variation of this protocol works for values shared with Shamir's 2-out-of-3 sharing.

After shuffling the vector, the results of comparing its elements may be made public, as long as all elements are different. They can be made different by adding an extra field to the comparison keys, which has the least significance in comparisons and is different for all elements. All pairs of elements may be compared in parallel [32], which has excellent round complexity, but requires $O(k^2)$ of work. Alternatively, comparisons may be made according to some sorting algorithm, e.g. the quicksort [19], moving around the elements of $[\![\boldsymbol{x}]\!]$ as indicated by the comparison results. The private shuffle $[\![\![\sigma]\!]\!]$ together with the public permutation (that latter can be composed with the last component of $[\![\![\sigma]\!]\!]$, giving a single private shuffle $[\![\![\sigma']\!]\!]$) applied during the sorting of shuffled $[\![\boldsymbol{x}]\!]$ provides the proof that the vectors before and after sorting had the same elements. In the rest of the paper, we will assume that all elements of $\boldsymbol{x}$ are different.

Counting sort can also be adapted to SMC, as shown in Alg. 2. Here private data in $\boldsymbol{y}$ is reordered according to the keys in $\boldsymbol{x}$. The sorting algorithm computes where each element of $\boldsymbol{x}$ and $\boldsymbol{y}$ would go if the key were equal to 0 (stored in $\overline{\boldsymbol{c}}$) or 1 (stored in $\boldsymbol{c}$; offset by the number of 0-s in $\boldsymbol{x}$, which is equal to $\overline{c}_k + 1$). The actual position is computed in line 4, performing an oblivious choice over $[\![x_i]\!]$, selecting either $[\![\overline{c}_i]\!]$ or $([\![c_i]\!] + [\![\overline{c}_k]\!] + 1)$. Note that while $x_i \in \{0, 1\}$, the elements of $[\![\boldsymbol{x}]\!]$ have to be shared over a larger ring (at least $\mathbb{Z}_k$) to make sure that the computations do not overflow; this may require the use of the *share conversion* protocol [7, Alg. 5]. The positions are randomly shuffled together with $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$. The declassification returns a random permutation of $\{1, \ldots, k\}$, leaking nothing about $[\![\boldsymbol{x}]\!]$. The composition of $[\![\![\sigma]\!]\!]$ and $\boldsymbol{o}$ is the proof of sameness of vectors. Counting sort can be extended to radix sort for multi-bit keys as in [5] and the sameness proofs can be composed as in Alg. 3.

```
    Data: keys $[\![\boldsymbol{x}]\!]$, data $[\![\boldsymbol{y}]\!]$, where $\boldsymbol{x} \in \mathbb{Z}_2^k$, $\boldsymbol{y} \in \mathbb{Z}_N^k$
    Result: $[\![\boldsymbol{x}]\!]$, $[\![\boldsymbol{y}]\!]$ stably sorted according to $\boldsymbol{x}$
1   foreach $i \in \{1, \dots, k\}$ do $[\![\overline{x}_i]\!] := 1 - [\![x_i]\!]$
2   $[\![\boldsymbol{c}]\!] := \mathsf{prefixsum}([\![\boldsymbol{x}]\!])$
3   $[\![\overline{\boldsymbol{c}}]\!] := \mathsf{prefixsum}([\![\overline{\boldsymbol{x}}]\!])$
4   foreach $i \in \{1, \dots, k\}$ do $[\![o_i]\!] := 1 + [\![\overline{c}_i]\!] + [\![x_i]\!] \cdot ([\![c_i]\!] - [\![\overline{c}_i]\!] + [\![\overline{c}_k]\!] + 1)$
5   Generate a random private permutation $[\![\![\sigma]\!]\!] \in S_k$
6   Shuffle $[\![\boldsymbol{x}]\!]$, $[\![\boldsymbol{y}]\!]$ and $[\![\boldsymbol{o}]\!]$ according to $[\![\![\sigma]\!]\!]$
7   $\boldsymbol{o} := \mathsf{declassify}([\![\boldsymbol{o}]\!])$
8   foreach $i \in \{1, \dots, k\}$ do $[\![x_i]\!] := [\![x_{o_i}]\!]$; $[\![y_i]\!] := [\![y_{o_i}]\!]$
```

Algorithm 2: Counting sort as a SMC protocol

```
    Data: Private permutations $[\![\![\sigma^{(1)}]\!]\!], \dots, [\![\![\sigma^{(m)}]\!]\!] \in S_k$
    Result: Private permutation $[\![\![\sigma]\!]\!]$ satisfying $\sigma = \sigma^{(m)} \circ \dots \circ \sigma^{(1)}$
1   $[\![\boldsymbol{o}]\!] := (1, 2, \dots, k)$
2   for $i = 0$ to $m - 1$ do
3       Shuffle $[\![\boldsymbol{o}]\!]$ according to $[\![\![\sigma^{(m-i)}]\!]\!]^{-1}$ (reverse the loop in Alg. 1)
4   Generate a random private permutation $[\![\![\sigma']\!]\!] \in S_k$
5   Shuffle $[\![\boldsymbol{o}]\!]$ according to $[\![\![\sigma']\!]\!]$
6   $\boldsymbol{o} := \mathsf{declassify}([\![\boldsymbol{o}]\!])$
7   return the composition of $[\![\![\sigma']\!]\!]$ and $\boldsymbol{o}$
```

Algorithm 3: Composing oblivious shuffles

3.4 Covert Security

Covertly secure protocols are secure against adversaries that may deviate from the protocol, but do not want to get caught [1].

Consider an ideal functionality $\mathcal{F}$ for n parties and a real protocol implementing it. A machine M_i may give a special output accuse_J to the party P_i, indicating that it suspects the parties in $J \subseteq [n]$ of deviating. We require the protocol π to be *detection accurate*, meaning that if for any honest P_i, the machine M_i outputs accuse_J, then $J \subseteq \mathcal{C}$, i.e. only corrupt parties can be caught cheating. We say that a run of π *catches a cheater* (denoted $\pi\Downarrow$) if all honest M_i output accuse_{J_i} to P_i, and the intersection of all the sets J_i is not empty.

Definition 2. *Let* $\varepsilon \in [0, 1]$. *A detection accurate protocol* π black-box simulates *the ideal functionality* $\mathcal{F}$ in the presence of covert adversaries with ε-deterrent *[1] if there exists a machine* Sim, *such that for all users* H *and adversaries* $\mathcal{A}$,

$$\varepsilon \cdot \Delta(\mathit{view}_{H\|\pi\|\mathcal{A}}(H), \mathit{view}_{H\|\mathcal{F}\|(\mathsf{Sim}\|\mathcal{A})}(H)) \leq \Pr[\pi\Downarrow] \enspace .$$

Here $[0, 1]$ is the set of real numbers between 0 and 1, and Δ is the *statistical distance* between probability distributions: $\Delta(\mu, \mu') = \frac{1}{2}\sum_{x \in X} |\mu(x) - \mu'(x)|$, where $\mu, \mu' : X \to [0, 1]$ are two probability distributions over the set X. This form of simulatability with covert adversaries is preserved by sequential composition (taking the minimum of ε-s), as shown by a simple hybrid argument [1].

4 Covertly Private SMC

Covert privacy. The definitions of privacy and covert security can easily be combined. In covert privacy, we let the machines M_i output accuse to the party P_i. Note that we do not specify the set of accused parties here. We define $\pi\Downarrow$ if all honest parties output accuse. We also relax the notion of *detection accuracy*, only requiring that an honest party does not output accuse if $\mathcal{C} = \emptyset$.

Definition 3. *Let $\varepsilon \in [0, 1]$. A detection-accurate protocol π is a* covertly private SMC protocol with ε-deterrent *for a functionality f, if there exists a machine* Sim*, such that for all users H, adversaries $\mathcal{A}$,*

$$\varepsilon \cdot \Delta(view_{H\|\pi\|\mathcal{A}}(H), view_{H\|\mathcal{F}^f_{\mathrm{priv}}\|(\mathsf{Sim}\|\mathcal{A})}(H)) \leq \Pr[\pi\Downarrow] \ .$$

We see that the covert privacy of a SMC protocol means the following. An active adversary may change the outcome of the protocol without the honest parties noticing it. An active adversary may also learn something about the inputs of honest parties, but if it does so, the honest parties will be notified with significant probability.

Covert privacy composes in the same manner as active or passive privacy. The proof in [4] carries over without modifications.

From covert privacy to covert security. Consider a covertly private SMC protocol π for a functionality f. Also let π compute for each party P_i a *verification value* v_i; we think of this verification value as not an output to the party P_i, but as input to subsequent protocols. There exist transformations [25, 26] that turn passively secure SMC protocols into covertly secure protocols (with 1-deterrent). The transformations perform the following steps:

- binding the parties to the messages they've sent using signatures;
- adding a verification protocol that uses the signed incoming and outgoing messages as verification values.

Given a protocol π, we let $\mathsf{s}[\pi]$ denote the protocol where all outgoing messages are signed. This protocol also outputs the signed messages as verification values. We let $\mathsf{v}[\pi]$ denote the verification protocol for $\mathsf{s}[\pi]$ constructed as in [25] or [26]. The protocol $\mathsf{v}[\pi]$ outputs a set $J_i \subseteq [n]$ of parties to be accused to each honest party P_i. If $J_i = \emptyset$ then no deviations were detected and the result output by π may be used. Note that the execution of $\mathsf{v}[\pi]$ is **much more expensive** (by two or more orders of magnitude) than the execution of π.

Covertly secure sorting. Let π_0 be the protocol for checking the correctness of sorting, it is given in Alg. 4. It performs some simple checks, verifying that the original and the sorted vector are the same, and that the sorted vector actually is sorted. Let π_1 be a covertly private sorting protocol that also outputs the proof of sameness of vectors. The following is one of the main results of this paper.

Theorem 1. *The protocol in Alg. 5 is a covertly secure sorting protocol.*

```
   Data: shared vectors $[\![\boldsymbol{x}]\!]$, $[\![\boldsymbol{y}]\!]$ of length $k$; private permutation $[\![\![\sigma]\!]\!] \in S_k$
   Result: yes/no, stating whether $[\![\![\sigma]\!]\!]$ proves that $[\![\boldsymbol{y}]\!]$ is the sorted version of $[\![\boldsymbol{x}]\!]$
1  Shuffle $[\![\boldsymbol{x}]\!]$ according to $[\![\![\sigma]\!]\!]$
2  foreach $i \in \{1, \ldots, k\}$ do $[\![b_i]\!] := ([\![x_i]\!] = [\![y_i]\!])$?
3  foreach $i \in \{1, \ldots, k-1\}$ do $[\![b'_i]\!] := ([\![y_i]\!] \leq [\![y_{i+1}]\!])$?
4  return declassify($\bigwedge_{i=1}^{k} [\![b_i]\!] \wedge \bigwedge_{i=1}^{k-1} [\![b'_i]\!]$)
```

Algorithm 4: Protocol π_0: checking the correctness of sorting

```
   Data: shared vector $[\![\boldsymbol{x}]\!]$
   Data: Covertly private SMC sorting protocol $\pi_1$ (with sameness proof)
   Result: sorted $[\![\boldsymbol{x}]\!]$; or accusations against misbehaving parties
1  $([\![\boldsymbol{x}']\!], [\![\![\sigma]\!]\!], accuse, \boldsymbol{v}) \leftarrow \mathsf{s}[\pi_1]([\![\boldsymbol{x}]\!])$
2  if $accuse = \mathsf{true}$ then go to 8
3  $(b, \boldsymbol{v}') \leftarrow \mathsf{s}[\pi_0]([\![\boldsymbol{x}]\!], [\![\boldsymbol{x}']\!], [\![\![\sigma]\!]\!])$
4  $(J_1, \ldots, J_n) \leftarrow \mathsf{v}[\pi_0](\boldsymbol{x}, \boldsymbol{x}', [\![\![\sigma]\!]\!], b, \boldsymbol{v}')$
5  each party $P_i$ does the following
6      if $J_i \neq \emptyset$ then return $\mathsf{accuse}_{J_i}$
7  if $b = \mathsf{true}$ then return $[\![\boldsymbol{x}']\!]$
8  $(J_1, \ldots, J_n) \leftarrow \mathsf{v}[\pi_1](\boldsymbol{x}, \boldsymbol{x}', [\![\![\sigma]\!]\!], accuse, \boldsymbol{v})$      // $\forall i \in [n] \backslash \mathcal{C} : J_i \neq \emptyset$
9  each party $P_i$ returns $\mathsf{accuse}_{J_i}$
```

Algorithm 5: Covertly secure SMC protocol for sorting

Proof. Let Sim_0, Sim_1 be simulators for $(\mathsf{s}[\pi_0], \mathsf{v}[\pi_0])$ and $(\mathsf{s}[\pi_1], \mathsf{v}[\pi_1])$, respectively. The simulator Sim for the protocol in Alg. 5 will receive the shares of $[\![\boldsymbol{x}]\!]$ for corrupted parties and invoke Sim_1 with them. At some point, Sim_1 computes the corrupted parties' shares of $[\![\boldsymbol{x}']\!]$ and $[\![\![\sigma]\!]\!]$, as well as the accusation bit. At this point Sim invokes Sim_0 with the shares it has computed. It does not return to continue with Sim_1.

The simulator Sim shows that Alg. 5 black-box simulates the ideal sorting functionality (which receives the shares of the elements of $\boldsymbol{x}$ and returns the shares of the elements of the sorted vector) in the presence of covert adversaries. Indeed, Sim can compute the corrupted parties' shares of $[\![\boldsymbol{x}']\!]$ and $[\![\![\sigma]\!]\!]$ indistinguishably from the real protocol due to Sim_1 being a simulator for $(\mathsf{s}[\pi_1], \mathsf{v}[\pi_1])$ and these shares belonging to the view of the adversary. If the accusation bit is true in the real protocol then some corrupt party will be accused by all honest parties in line 8 by the security properties of $\mathsf{v}[\pi_1]$. Such accusations do not have to be simulated by Sim according to Def. 2. If the accusation bit is false then Sim will produce a good simulation of the real protocol due to Sim_0 being a simulator for $(\mathsf{s}[\pi_0], \mathsf{v}[\pi_0])$. If the bit b is false in line 7 and the real protocol continues with the invocation of $\mathsf{v}[\pi_1]$, then again some corrupted party will definitely be accused in line 8 and this part of the protocol does not need simulating. $\square$

A conceptually simpler covertly secure sorting protocol would unconditionally jump from line 2 to line 8. But the full protocol in Alg. 5 is more efficient in executions where no party tries to deviate from the protocol; it is natural to expect most executions to be like that. While the conceptually simpler protocol would always execute $\mathsf{v}[\pi_1]$, the protocol in Alg. 5 executes $\mathsf{s}[\pi_0]$ and $\mathsf{v}[\pi_0]$ instead. We expect the protocol π_0 which only checks for sortedness to be $O(\log k)$ times cheaper than the sorting protocol π_1. The verification is similarly cheaper.

5 Analysis of Oblivious Sorting Methods

We have discussed SMC protocols based on securely implementing sorting networks and argued that they are actively private. Unfortunately, the sequence of swaps that they produce is not easily converted into a single private shuffle. One can convert the comparison results of each layer of compare-exchange nodes into an oblivious shuffle; these $O(\log^2 k)$ oblivious shuffles (for input vectors of length k) can be composed into a single oblivious shuffle with $O(k \log^2 k)$ work using Alg. 3. As we show below, at least parts of this algorithm are not covertly private for the same reason as Alg. 2.

5.1 Methods Based on Shuffling and Comparison

We now show that sorting protocols that first shuffle the vector $[\![\boldsymbol{x}]\!]$ and then declassify the results of comparisons cannot be covertly private, at least for additive secret sharing. For this purpose we present a pair $(H, \mathcal{A})$, such that no simulator Sim can make the sorting protocol π indistinguishable from $\mathcal{F} \| \mathsf{Sim}$, where $\mathcal{F}$ is the ideal sorting functionality.

H and $\mathcal{A}$ first agree on a bit b, followed by $\mathcal{A}$ corrupting one of the parties, and H submitting a vector $[\![\boldsymbol{x}]\!]$ of length 2 to be sorted. The elements of $\boldsymbol{x}$ are $x_1 = 0$ and $x_2 = 1 + 2b$; H additively shares them before submission. In the real protocol, vector $[\![\boldsymbol{x}]\!]$ is shuffled (i.e. perhaps the elements are swapped) and at some moment, $[\![x_1]\!]$ and $[\![x_2]\!]$ are compared to each other. Before the comparison, $\mathcal{A}$ tells the corrupted party to add 2 to its share of $[\![x_2]\!]$. Due to additive sharing, this means that x_2 is increased by 2. The comparison result is declassified. In real execution, the comparison result depends on the bit b. If $b = 0$, then $x_1 < x_2$, because $|x_2 - x_1| = 1$ before the adversary's interference and the increasing of x_2 was sufficient to make it larger than x_1. If $b = 1$ then $|x_2 - x_1| = 3$ before the adversary's interference. In this case the increase of x_2 did not affect the comparison result and either result is possible with 50% probability. The simulator does not know b and hence does not know, from which probability distribution the simulated result of the comparison should be sampled. Nor can the honest parties in the real execution notice that something is wrong.

5.2 Counting sort

Most steps of the counting sort protocol (Alg. 2) are covertly, and even actively private [29], except for the declassification step in line 7. The protocol can be seen as consisting of two parts, the first of them computing the positions for reordering the elements of $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{y}]\!]$, and the second one (lines 5–8) actually performing the reordering. We show that the second part is not covertly secure, if $[\![\boldsymbol{o}]\!]$ is

```
Data: Private permutation $[\![\![\sigma]\!]\!] \in S_{k+1}$, index $x_1 \in \{1,\ldots,k+1\}$
Result: Private permutation $[\![\![\sigma']\!]\!] \in S_k$, such that $\sigma' = \sigma{\downarrow}_{x_0}$
1 for $i = 1$ to $3$ do
2     $P_{i-1}$ and $P_{i+1}$ send $x_{i+1} = \sigma_i(x_i)$ to $P_i$
3     if $P_i$ receives different $x_{i+1}$-s then $P_i$ outputs accuse
4 foreach $i \in \{1,2,3\}$ do $P_{i-1}$ and $P_{i+1}$ define $\sigma'_i := \sigma_i{\downarrow}_{x_i}$
```
Algorithm 6: Puncturing a private permutation (in Sharemind)

shared over $\mathbb{Z}_k$ (in this case, $0 \equiv k$). We analyse the lines 5–7, because the last line only performs public operations. Note that exactly the same operations are performed in Alg. 3, lines 4–6. The honest parties may try to detect adversarial interference by noticing that the declassified $\boldsymbol{o}$ is not a permutation of $1,\ldots,k$.

We again present $(H,\mathcal{A})$ for which no simulator Sim exists. H and $\mathcal{A}$ first agree on a bit, after which $\mathcal{A}$ corrupts a party and H shares an arbitrary $\boldsymbol{x}$ and $\boldsymbol{y}$ of length k among the computing parties. It also shares the vector $\boldsymbol{o} = (b,b,\ldots,b)$ (of length k). Even though $\boldsymbol{o}$ should be a permutation in a "normal" execution of the lines 5–7, this does not have to be the case if the previous steps of a larger protocol have also been affected by the adversary. In the execution, $\mathcal{A}$ selects a random permutation $\boldsymbol{o}' \in \mathbb{Z}_k^k$. It lets the shuffling protocol (for $[\![\boldsymbol{o}]\!]$) execute normally, except that during the resharing step (lines 6–7 in Alg. 1), it tells the corrupted party to add $\boldsymbol{o}'$ to its share. Hence the declassification in line 7 of Alg. 2 produces a permutation and adversarial interference will not be detected. The declassified permutation is equal to $\boldsymbol{o}' + (b,b,\ldots,b)$. The simulator Sim does not know b, thus cannot simulate this.

6 Covertly Private Reordering

A covertly private reordering protocol (replacing steps 5–7 in Alg. 2 and steps 4–6 in Alg. 3) is all that is needed for a covertly private sorting algorithm that can be used in Alg. 5. We start its presentation by an auxiliary algorithm for *puncturing* a private permutation.

For $i \in \mathbb{N}$, define the mappings $\mathsf{ins}_i, \mathsf{del}_i : \mathbb{N} \to \mathbb{N}$ by $\mathsf{ins}_i(j) = \mathsf{del}_i(j) = j$ if $j < i$ and $\mathsf{ins}_i(j) = j+1$, $\mathsf{del}_i(j) = j-1$ for $j \geq i$. Let $\tau \in S_{k+1}$ and $i \in \{1,\ldots,k+1\}$. *Puncturing* τ at i gives us the permutation $\tau{\downarrow}_i = \mathsf{del}_{\tau(i)} \circ \tau \circ \mathsf{ins}_i \in S_k$. A private permutation $[\![\![\sigma]\!]\!] \in S_{k+1}$ can also be easily punctured while leaking the value $\sigma(i)$ in the process, as shown in Alg. 6. Clearly, nothing else is leaked, because no other points of σ_i-s are made public. The protocol is covertly private because everything a party receives, it receives from two other parties, one of which has to be honest. Without the last line in Alg. 6, this protocol covertly securely computes and makes public $x_4 = \sigma(x_1)$.

The covertly private reordering protocol is given in Alg. 7. It introduces the *replicated secret sharing* [11] $[\![\![y]\!]\!]$ of a value $y \in \mathbb{Z}_N$. In our case of additive secret sharing with Sharemind security model, $[\![\![y]\!]\!]$ consists of three random elements of $\mathbb{Z}_N$ summing up to y, with each party knowing two of them. Random

```
Data: Shared vectors $[\![\boldsymbol{x}]\!], [\![\boldsymbol{o}]\!]$ of length $k$, where $\boldsymbol{o}$ is a permutation of
      $\{1,\ldots,k\}$
Result: $[\![\boldsymbol{x}]\!]$ reordered according to $[\![\boldsymbol{o}]\!]$
1  Generate random $[\![\![y_1]\!]\!], \ldots, [\![\![y_m]\!]\!]$
2  foreach $i \in \{1,\ldots,m\}$ do $[\![o_{k+i}]\!] := [\![y_i]\!]$
3  Generate a random private permutation $[\![\![\sigma]\!]\!] \in S_{k+m}$
4  Shuffle $[\![\boldsymbol{o}]\!]$ according to $[\![\![\sigma]\!]\!]$
5  $\boldsymbol{o} :=$ declassify($[\![\boldsymbol{o}]\!]$)
6  foreach $i \in \{1,\ldots,m\}$ do
7      $z_i := \sigma(k+i)$                          // Alg. 6 without last line
8      if $o_{z_i} \neq$ declassify($[\![\![y_i]\!]\!]$) then output accuse
9  Delete positions $z_1,\ldots,z_m$ from $\boldsymbol{o}$
10 if $\boldsymbol{o}$ is not a permutation of $\{1,\ldots,k\}$ then output accuse
11 Shuffle $[\![\boldsymbol{x}]\!]$ according to $[\![\![\sigma{\downarrow_{k+m}}{\downarrow_{k+m-1}}\cdots{\downarrow_{k+1}}]\!]\!]$
12 foreach $i \in \{1,\ldots,k\}$ do $[\![x_i]\!] := [\![x_{o_i}]\!]$
```

Algorithm 7: Protocol for covertly private reordering

replicated shared values are generated in the same manner as random private permutations. Conversion from $[\![\![y]\!]\!]$ to $[\![y]\!]$ means just dropping one of the shares. In declassifying $[\![\![y]\!]\!]$, each party sends to both other parties the share they do not yet know. In this manner, each party will learn the missing share from both other parties and the adversary cannot send a wrong share without being detected.

We see that in Alg. 7 we add extra elements to the index vector $[\![\boldsymbol{o}]\!]$ in order to catch the adversary manipulating many elements of it, the number of added elements m functions as security parameter. After shuffling, we determine where the added elements had to end up, and check that they were not changed by the adversary. Finally, we drop the points $k+1,\ldots,k+m$ from the private permutation σ (this can be done locally after running Alg. 6 to find $z_1,\ldots,z_m$) and use the result to shuffle $[\![\boldsymbol{x}]\!]$ as before.

Theorem 2. *Let $c = m/k$. Alg. 7 is covertly private with ε-deterrent, where $\varepsilon = 1-(c+1)^{-c}$.*

Proof. We need to construct Sim simulating the view of the corrupted party from this party's input shares. The simulator does not know the honest parties' shares of $[\![\boldsymbol{x}]\!]$ and $[\![\boldsymbol{o}]\!]$, hence it does not know $\boldsymbol{x}$ and $\boldsymbol{o}$. During the run, it receives all messages the corrupted party sends to honest parties, and must generate honest parties' messages to the corrupt party, thereby learning all these messages. Hence it can still follow Alg. 7 as follows.

In line 1, Sim will either generate or receive all three shares of $[\![\![y_1]\!]\!], \ldots, [\![\![y_m]\!]\!]$, hence it knows $y_1,\ldots,y_m$. Similarly, in line 3 it learns all three shares of $[\![\![\sigma]\!]\!]$ and therefore σ itself. In line 4, it simulates the invocation of a shuffle (Alg. 1), which is actively private. In line 5, Sim must simulate the result of declassifying shuffled $\boldsymbol{o}$. In declassified $\boldsymbol{o}$, the simulator puts $y_1,\ldots,y_m$ to positions $\sigma(k+1),\ldots,\sigma(k+m)$ and a uniformly random permutation of $\{1,\ldots,k\}$ to the remaining k positions.

The simulator has all information (shares of $[\![\![\sigma]\!]\!]$ and $[\![\![\boldsymbol{y}]\!]\!]$) to simulate the lines 6–8. The shuffle in line 11 is actively private and the operations in lines 9,10,12 do not involve communication between parties. We must now justify that the simulation of the declassification in line 5 is sufficiently similar to the real protocol to achieve the claimed deterrent. We make the following claims.

Claim 1. Consider a protocol where first a vector $\boldsymbol{z} \in \mathbb{Z}_N^k$ is generated and shared by an active adversary $\mathcal{A}$ (having corrupted a party), then a private permutation $[\![\![\sigma]\!]\!]$ is generated and applied to $[\![\boldsymbol{z}]\!]$ using the protocol in Alg. 1, and finally $\boldsymbol{z}$ is declassified. Any such $\mathcal{A}$ can be emulated by an adversary that selects $\boldsymbol{z}, \boldsymbol{z}' \in \mathbb{Z}_N^k$ and learns $\sigma'(\boldsymbol{z}) + \boldsymbol{z}'$ for an unknown, uniform $\sigma' \in S_k$.

The claim follows from the construction of Alg. 1. At each resharing and updating of shares in lines 3–4 and 6–7 of Alg. 1, the adversary can add a known vector to $\boldsymbol{z}$. The adversary also knows all but one permutations used in line 5. The addition of vectors before the application of the unknown permutation corresponds to selecting a different $\boldsymbol{z}$ in the beginning. The addition after this application corresponds to $\boldsymbol{z}'$. During declassification, the adversary can add yet another vector to the current $\boldsymbol{z}$.

Claim 1 explains how much an adversary can affect $\boldsymbol{o}$ in line 5 of Alg. 7. It can choose some $\boldsymbol{o}$ (not necessarily a permutation of $\{1, \ldots, k\}$) in the beginning and, after it has been permuted, add a known $\boldsymbol{d}$ to it. Assuming that the addition did not affect any elements of $\boldsymbol{y}$ (which would result in an immediate accusation), the adversary hopes to obtain a non-uniformly chosen permutation of $\{1, \ldots, k\}$.

Claim 2. Let $|\boldsymbol{v}| = k$, with $\boldsymbol{v}$ having s different elements. If $\sigma \in S_k$ is uniformly chosen then $\Pr[\sigma(\boldsymbol{v}) = \boldsymbol{v}] \leq (k-s+1)!/k!$.

Indeed, if $w_1, \ldots, w_s$ are the elements occuring in $\boldsymbol{v}$ with c_i being the count of w_i, then the number of permutations in S_k leaving $\boldsymbol{v}$ in place is $c_1! \cdots c_s! = \prod_{i=1}^{s} \prod_{j=1}^{c_i-1} (1+j)$. This product has $k-s$ factors, bounded by $2, 3, \ldots, (k-s+1)$. Hence it is at most $(k-s+1)!$. The set S_k has $k!$ elements.

Claim 3. Let $\boldsymbol{v}, \boldsymbol{d}, \boldsymbol{u} \in \mathbb{Z}_N^k$ with $\boldsymbol{u}$ being a permutation of $\{1, \ldots, k\}$. Let $t = \|\boldsymbol{d}\|_0$, denoting the number of non-zero elements (the *Hamming weight*) of $\boldsymbol{d}$. If $\sigma \in S_k$ is uniformly chosen then $\Pr[\sigma(\boldsymbol{v}) + \boldsymbol{d} = \boldsymbol{u}] \leq (t+1)!/k!$.

Indeed, if there is no σ_0, such that $\sigma_0(\boldsymbol{v}) + \boldsymbol{d} = \boldsymbol{u}$, then the claim holds. Otherwise, the probability is equal to $\Pr[(\sigma \circ \sigma_0^{-1})(\sigma_0(\boldsymbol{v})) = \sigma_0(\boldsymbol{v})]$. By claim 2, this probability as at most $(k-s+1)!/k!$, where s is the number of different elements in $\sigma_0(\boldsymbol{v}) = \boldsymbol{u} - \boldsymbol{d}$. Vector $\boldsymbol{u}$ has k different elements, hence $\boldsymbol{u} - \boldsymbol{d}$ has at least $k - t$ different elements.

In the following, let $\boldsymbol{d} \in \mathbb{Z}_N^{k+m}$ be the difference in $\boldsymbol{o}$ that the adversary has caused in line 5 of Alg. 7. Let E be the event that accuse does not occur in the loop in lines 6–8 of the real execution of Alg. 7. Let $t = \|\boldsymbol{d}\|_0$.

Claim 4. (trivial) $\Pr[E] \leq \binom{k}{t} / \binom{k+m}{t}$.

Claim 5. If $t \leq c$ (i.e. $m \geq tk$) then $\Pr[E] \leq 1/(t+1)!$.

Indeed, we have $\binom{k}{t} / \binom{k+m}{t} = \prod_{i=0}^{t-1} \frac{k-i}{k+m-i} \leq (\frac{k}{k+m})^t$. Applying the inequality $m \geq tk$ gives

$$\Pr[E] \leq \left(\frac{k}{k+m}\right)^t \leq \left(\frac{k}{k+tk}\right)^t = \left(\frac{1}{t+1}\right)^t = \frac{1}{(t+1)^t} \leq \frac{1}{(t+1)!} \ .$$

Claim 6. If $t \geq c$ then $\Pr[E] \leq (c+1)^{-c}$.

Similarly to previous claim, we get

$$\Pr[E] \leq \left(\frac{k}{k+m}\right)^t = \left(\frac{k}{k+ck}\right)^t = \left(\frac{1}{c+1}\right)^t \leq \left(\frac{1}{c+1}\right)^c = \frac{1}{(c+1)^c} \ .$$

Claim 7. Let $\boldsymbol{u} \in \mathbb{Z}_N^k$ be a permutation of $\{1, \ldots, k\}$. If $m \geq tk$ then the probability of $\boldsymbol{o}$ being equal to $\boldsymbol{u}$ in line 10 of the real execution of Alg. 7 is at most $1/k!$.

Indeed, to get to line 10, the event E must occur. If E occurred then all t changes the adversary made to $\boldsymbol{o}$ must have happened to the positions not taken by the elements of $\boldsymbol{y}$. Hence, if E occurred then by Claim 3, the probability of $\boldsymbol{o} = \boldsymbol{u}$ in line 10 is at most $(t+1)!/k!$. This probability must be multiplied with $\Pr[E]$ and the resulting product is at most $1/k!$.

Claim 7 shows that if m is sufficiently large then no declassification result in the real execution may occur with larger probability than in the simulated execution. This justifies Sim outputing a uniformly random permutation of $\{1, \ldots, k\}$.
Claim 8. If $t \geq c$ then accuse is output in the real execution with probability at least $\varepsilon = 1 - (c+1)^{-c}$.

Indeed, the probability of accuse being output is at least $1 - \Pr[E] \geq \varepsilon$.

The value of t is a random variable determined by the adversary. We have now analysed both the cases $t \leq c$ and $t \geq c$ and shown that in both cases, the adversary cannot get a large difference of views in real and ideal execution. Indeed, if $t \leq c$, then no particular value of $\boldsymbol{o}$ in line 10 can be obtained with greater probability in the real execution than in the simulated execution. Any difference in probabilities is due to accuse being output in the real execution. Thus, if $t \leq c$, we have 1-deterrent against adversaries trying to breach the privacy.

If $t \geq c$ then accuse is output with probability at least ε. As the distance between the views in the real and ideal execution cannot be larger than 1, we have at least ε-deterrent here.

□

7 Conclusions

We have presented a covertly private SMC reordering protocol that may be used to build a covertly private SMC sorting protocol based on the radix sort algorithm. The overhead of the sorting protocol, compared to a passively secure protocol, is only about three times (i.e. $m = 2k$), already giving the probability of ca. 90% for catching a misbehaving adversary. The resulting covertly secure sorting protocol has only $o(k)$ overhead (over the passively secure protocol) on an input of size k.

It remains to be seen whether the presented reordering protocol is sufficient to construct a covertly private sorting protocol based on the quicksort algorithm.

References

1. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010.
2. Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the Estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. *Financial Cryptography (FC) 2015*, LNCS 8975:227–234.
3. Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Social Study Using Secure Computation. *Proceedings of Privacy Enhancing Technologies (PoPETS)*, 2016.
4. Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. From Input Private to Universally Composable Secure Multi-party Computation Primitives. CSF 2014, pp. 184–198.
5. Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. *NordSec 2014*, LNCS 8788:59–74.
6. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. *ESORICS 2008*, LNCS 5283:192–206.
7. 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.
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. *Financial Cryptography (FC) 2009*, LNCS 5628:325–343.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. *FOCS 2001*, pp. 136–145.
10. Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. *STOC 2002*, pp. 494–503.
11. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. *TCC 2005*, LNCS 3378:342–362.
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. *ESORICS 2013*, LNCS 8134:1–18.
13. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO 2012*, LNCS 7417:643–662.
14. Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. *AFRICACRYPT 2016*, LNCS 9646:265–284.
15. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. *CRYPTO 2010*, LNCS 6223:465–482.
16. Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
17. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. *STOC 1987*, pp. 218–229.

18. 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.
19. Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. *ICISC 2012*, LNCS 7839:202–216.
20. Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. Cryptology ePrint Archive, Report 2011/122, 2011.
21. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016.
22. Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. *CCS 2013*, pp. 549–560.
23. Peeter Laud. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *Proceedings of Privacy Enhancing Technologies*, 2015(2):188–205, 2015.
24. Peeter Laud. Stateful Abstractions of Secure Multiparty Computation. *Applications of Secure Multiparty Computation*, volume 13 of *Cryptology and Information Security*, pp 26–42. IOS Press, 2015.
25. Peeter Laud and Alisa Pankova. Verifiable Computation in Multiparty Protocols with Honest Majority. *ProvSec 2014*, LNCS 8782:146–161.
26. Peeter Laud and Alisa Pankova. Preprocessing-based verification of multiparty protocols with honest majority. Cryptology ePrint Archive, Report 2015/674, 2015.
27. Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. *ISC 2011*, LNCS 7001:262–277.
28. Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. *PKC 2007*, LNCS 4450:343–360.
29. Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. *CSF 2015*, pp. 75–89.
30. Guan Wang, Tongbo Luo, Michael T. Goodrich, Wenliang Du, and Zutao Zhu. Bureaucratic protocols for secure two-party sorting, selection, and permuting. *ASIACCS 2010*, pp. 226–237.
31. Andrew C. Yao. Protocols for secure computations. *FOCS 1982*, pp. 160–164.
32. Bingsheng Zhang. Generic constant-round oblivious sorting algorithm for MPC. *ProvSec 2011*, LNCS 6980:240–256.