Linear-time oblivious permutations for SPDZ

Peeter Laud

Cybernetica AS
peeter.laud@cyber.ee

Abstract. In this paper, we present a secure multiparty computation (MPC) subroutine for obliviously permuting elements in a private vector. The subroutine makes use of the private data representations used in the SPDZ protocol set; it can be composed with other privacy-preserving operations in this set and it also provides active security with abort. The online computation and communication complexity of the subroutine is linear in the length of the permuted vector.

1 Introduction

Large secure multiparty computation (MPC) protocols are constructed on top of MPC frameworks supporting a small number of primitive operations with private data — input, output, addition, multiplication, conversions between integers and bit-strings. These operations do not hide, which memory locations they access, thus only supporting *data-oblivious* computations.

Certain frameworks support *oblivious permutations* that shuffle a vector of private values in a manner that keeps the reordering private as well. Efficient oblivious permutations are a useful subroutine for certain non-data-oblivious algorithms, in particular the fast algorithms for sorting [11, 6], as well as for the declassification of certain values without leaking their location in memory [5, 1].

Currently, truly efficient oblivious permutations (number of primitive operations being linear in the length of the vector, with small multiplicative overhead) are supported only by MPC protocol sets with passive security and honest majority [16]. In this short paper, we show how to add them to the well-known SPDZ [10] protocol set, which supports dishonest majority, as well as fail-stop security (active attacks are detected). In SPDZ, the computation is split into input-independent offline and efficient online phases. We describe the online phase of oblivious permutations, state which offline precomputations our protocol requires, and discuss the possible implementations of the offline phase. We state the security properties of both the online and offline phase, and argue the security of our protocol.

State of the Art. Applying a sorting network to a random vector is the folk method for random permutations. Laur et al. [16] gave the most widely used permutation protocol for passively secure MPC; it's (communication) complexity is $O(m) \cdot 2^{O(n)}$ (elements of vectors), where m is the length of the vector and n the number of parties in the MPC protocol. The protocol is very efficient for a small n. Asharov et al. [2] gave a protocol with $O(m \log m)$ complexity, based on techniques used for ORAM.

Oblivious (extended) permutations are needed for private function evaluation. Mohassel et al. [17] achieve active security through a combination of MPC and Mix-nets. Laud et al. [14] obtain it through post-execution verification, using precomputed *permutation tuples*. Such tuples form one half of our solution.

Encodings of RAM computations for zero-knowledge proofs requires stating that certain vectors describing memory accesses are permutations of each other. Here, besides the earlier permutation network based solutions [4], Bootle et al. [7] proposed a linear-time construction based on showing the equality of polynomials that have the elements of one of the vectors as its roots [18]. Such polynomial equality checking forms the second half of our solution.

Our approach is also an instance of first performing the computations with passive security (but with privacy also against active attackers), and then verifying the results using an actively secure protocol [12, 15].

Notation. The elements of a vector $\vec{v}$ are denoted $v_1, v_2, \ldots$. Given a vector $\vec{v}$ of length m, and a permutation $\pi \in S_m$ (where S_m is the *symmetric group* of all permutations on m elements), we let $\pi(\vec{v})$ denote the vector $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(m)})$. We write $\vec{w} \leftarrow \vec{u} + \vec{v}$ to denote that $\vec{w}$ is the pointwise sum of $\vec{u}$ and $\vec{v}$; other operations may be used similarly. We use $[m]$ to denote the set $\{1, \ldots, m\}$.

SPDZ [10] is an MPC protocol for n parties, tolerating up to $(n-1)$ static active corruptions, and providing fail-stop security, i.e. misbehaviour by a corrupted party is detected, but the party is not ousted. Also, the misbehaviour does not impact the privacy of honest parties. The protocol executes an arithmetic circuit over a large finite field $\mathbb{F}$, with all values secret-shared among the parties. The protocol works has offline (input-independent) and online phases; in the offline phase each party P_i has selected a private value α_i (denote $\alpha = \alpha_1 + \cdots + \alpha_n$), and parties have executed "heavyweight" protocols to generate correlated values usable for linearizing all operations of the arithmetic circuit. In online phase, the private representation $[\![x]\!]$ of a value x is a tuple of random values $(([\![x]\!]_1, \gamma(x)_1), \ldots, ([\![x]\!]_n, \gamma(x)_n))$, with the i-th party knowning $[\![x]\!]_i$ and $\gamma(x)_i$, and with the values satisfying $[\![x]\!]_1 + \cdots + [\![x]\!]_n = x$ and $\gamma(x)_1 + \cdots + \gamma(x)_n = \alpha \cdot x$ (this is called the "MAC of $[\![x]\!]$"). Linear computations with private values can be done locally by parties. Multiplication triples generated during the offline phase are used for multiplication; the opening of values creates obligations to check that the MAC of the opened value is correct. MAC checks can be done without revealing α. Several MAC checks cost the same as one.

2 Offline phase

In Fig. 1 we present our additions to the ideal functionality $\mathcal{F}_{\text{PREP}}$ for the offline phase of SPDZ. They allow each pair of parties to obtain additive shares of a permutation of a vector, with the first party providing the permutation $\pi \in S_m$ and the second party also learning the vector. The correctness of the outputs is not guaranteed if the parties are corrupt, but privacy still is. We call $(\pi, \vec{z}; \vec{x}, \vec{y})$

On input **Shuffle**(m, P_j, P_k, π) from P_j and **Shuffle**(m, P_j, P_k) from P_k:

0. Ignore the query, if both P_j and P_k are corrupted
1. Pick random vectors $\vec{x}, \vec{y} \in \mathbb{F}^m$. Put $\vec{z} = \pi(\vec{x}) - \vec{y}$
2. Set $X_j = \langle \pi, \vec{z} \rangle$ and $X_k = \langle \vec{x}, \vec{y} \rangle$
3. If P_c is corrupted and P_h is honest ($\{c, h\} = \{j, k\}$), then
   - Output X_c to P_c and the adversary
   - Get the description of a polynomial-time algorithm $\mathcal{A}$ from the adversary
   - Output $\mathcal{A}(X_h)$ to P_h
4. Otherwise (both P_j and P_k are honest), output X_j to P_j and X_k to P_k

Fig. 1. Addition to the ideal functionality $\mathcal{F}_{\text{PREP}}$ of the offline phase

a *permutation tuple.* A possible corresponding real implementation $\mathit{\Pi}_{\text{PREP}}$ is discussed by Chase et al. [8] under the name *Permute-and-Share*, including the trade-offs for executing the protocol several times in parallel with the same π.

3 Online phase

In the online phase, when evaluating an arithmetic circuit, we want to take a number of already computed values, treat them as a vector, apply an oblivious permutation to them, and continue computations with the permuted values. Existing algorithms [11, 13] use oblivious permutations in a couple of different ways. The operations we present below as additions to the SPDZ protocol set $\mathit{\Pi}_{\text{ONLINE}}$ cover all these ways.

3.1 Randomly permuting a vector

There is a private vector $[\![\vec{v}]\!]$ of length m, the elements of which we want to permute, with no party, or a coalition of up to $(n-1)$ parties learning anything about the permutation. The protocol for this operation is given in Fig. 2.

We see that during the i-th iteration of the main loop, the current share of each party P_j $(j \neq i)$ gets permuted with π_i and then additively shared between P_i and P_j. Here the share of P_j is $\vec{y}_1^{(i,j)}$ (for the "value itself") and $\vec{y}_2^{(i,j)}$ (for the MAC), while the share for P_i is $\vec{r}_{ij}$ and $\vec{s}_{ij}$ [14]. Having obtained such shares from every other party, P_i defines his share for the next round be adding them all up (step 1.3). In step (1.1.1), P_j sends his current shares to P_i, but they are masked with random vectors $\vec{x}_k^{(i,j)}$, hence there is no leakage. All parties contribute to the resulting permutation $\pi = \pi_1 \circ \cdots \circ \pi_n$, hence it stays private. After the n-th iteration, parties obtain the shares of the output vector $[\![\vec{v}^{(n)}]\!]$.

In steps (2)–(3), the parties verify that $\vec{v}^{(n)}$ is indeed a permutation of $\vec{v}^{(0)}$. For any vector $\vec{u} \in \mathbb{F}^m$, define the polynomial $f_{\vec{u}}(X) \in \mathbb{F}[X]$ by $f_{\vec{u}}(X) = \prod_{i=1}^{m}(X - u_i)$. Two polynomials $f_{\vec{u}}, f_{\vec{u}'}$ are equal if $\vec{u}$ and $\vec{u}'$ are permutations of each other [18]. In steps (2)–(3), this polynomial equality is tested by evaluating $f_{\vec{v}^{(0)}}$ and $f_{\vec{v}^{(n)}}$ at a random point r and making sure that their difference is 0

```
**Input:** private vector $[\![\vec{v}^{(0)}]\!]$ of length $m$.
**Output:** private vector $[\![\vec{v}^{(n)}]\!]$, such that $\vec{v}^{(n)}$ is a permutation of $\vec{v}^{(0)}$
**Offline:**
(1) Each party $P_i$ selects a random permutation $\pi_i$ of length $m$.
(2) Each pair of parties $P_i, P_j$ runs two instances of $\mathcal{F}_{\text{PREP}}.\mathsf{Shuffle}$, with $P_i$ providing the input $\pi_i$. Let the result of $k$-th instance be $\vec{z}_k^{(i,j)}$ for $P_i$ and $\vec{x}_k^{(i,j)}, \vec{y}_k^{(i,j)}$ for $P_j$
**Online:**
(1) For $i$ going from 1 to $n$, do the following:
  (1.1) For each $j$ different from $i$, do the following:
    (1.1.1) $P_j$ sends $\vec{w}_1^{(i,j)} \leftarrow [\![\vec{v}^{(i-1)}]\!]_j - \vec{x}_1^{(i,j)}$ and $\vec{w}_2^{(i,j)} \leftarrow \gamma(\vec{v}^{(i-1)})_j - \vec{x}_2^{(i,j)}$ to $P_i$
    (1.1.2) $P_i$ computes $\vec{r}_{ij} \leftarrow \pi_i(\vec{w}_1^{(i,j)}) + \vec{z}_1^{(i,j)}$ and $\vec{s}_{ij} \leftarrow \pi_i(\vec{w}_2^{(i,j)}) + \vec{z}_2^{(i,j)}$
    (1.1.3) $P_j$ defines shares of the next round: $[\![\vec{v}^{(i)}]\!]_j = \vec{y}_1^{(i,j)}$ and $\gamma(\vec{v}^{(i)})_j = \vec{y}_2^{(i,j)}$.
  (1.2) $P_i$ computes $\vec{r}_{ii} \leftarrow \pi_i([\![\vec{v}^{(i-1)}]\!]_i)$ and $\vec{s}_{ii} \leftarrow \pi_i(\gamma(\vec{v}^{(i-1)})_i)$
  (1.3) $P_i$ defines shares of the next round: $[\![\vec{v}^{(i)}]\!]_i = \sum_j \vec{r}_{ij}^{(i)}$ and $\gamma(\vec{v}^{(i)})_i = \sum_j \vec{s}_{ij}^{(i)}$.
(2) Parties pick fresh random $[\![r]\!], [\![r']\!] \in \mathbb{F}$, and **Reveal** $r$        // SPDZ supports this
(3) Parties check whether **Reveal**$\big([\![r']\!] \cdot (\prod_{i=1}^{m}(r - [\![v_i^{(n)}]\!]) - \prod_{i=1}^{m}(r - [\![v_i^{(0)}]\!]))\big) = 0$
```

Fig. 2. Obliviously shuffling a private vector

(masking the result with r' before opening it). We know that if the polynomial $f_{\vec{v}^{(n)}} - f_{\vec{v}^{(0)}}$ is non-zero, then it has at most m roots, hence the probability of test (2)–(3) falsely accepting is at most $m/|\mathbb{F}|$.

The equality check for polynomials involves a number of multiplications, whose implementation includes the check of MACs on the multiplicands. Obviously, all these MAC checks have to pass before the value $f_{\vec{v}^{(n)}}(r) - f_{\vec{v}^{(0)}}(r)$ is revealed in the equality check.

Complexity With n computing parties and a vector of length m, the round complexity (of the online phase) of the protocol is $O(n+\log m)$. Here $O(n)$ comes from the the loop (1) and $O(\log m)$ comes from the computation of the products of length m in (3). The $\log m$ term can be removed at the cost increasing the number of (binary) multiplications several times [3].

The communication complexity of the online phase is $O(n^2 m)$, when we consider the elements of $\mathbb{F}$ to have constant size. This holds, because at each iteration of the loop (1), one party exchanges $O(m)$ elements of $\mathbb{F}$ with every other party. The communication complexity of the offline phase is somewhat larger than $O(n^2 m)$, because an implementation of a single invocation of $\mathcal{F}_{\text{PREP}}.$**Shuffle** needs somewhat more than $O(m)$ communication [8].

3.2 Creating a random permutation and applying it

We may also want to "create and store" a random oblivious permutation, such that it can be applied to several vectors during the later steps of the computation. For the creation, we pick a vector $\vec{u}^{(0)} \in \mathbb{F}^m$, where all elements are different, classify it, and apply the protocol in Fig. 2 to it. To represent this permutation π, each party stores his π_i. Also, the protocol stores both $[\![\vec{u}^{(0)}]\!]$ and $[\![\vec{u}^{(n)}]\!]$.

Input: private vector $[\![\vec{v}^{(0)}]\!]$ of length m, and private vectors $[\![\vec{u}^{(0)}]\!]$, $[\![\vec{u}^{(n)}]\!]$ representing a private permutation π. Each party P_i also has π, such that $\pi_1 \circ \cdots \circ \pi_n = \pi$.
Output: private vector $[\![\vec{v}^{(n)}]\!]$, such that $v_i^{(n)} = v_{\pi(i)}^{(0)}$ for all $i \in [m]$.
Offline:
Each pair of parties P_i, P_j runs two instances of $\mathcal{F}_{\text{PREP}}.\mathsf{Shuffle}$, with P_i providing the input π_i. Let the result of k-th instance be $\vec{z}_k^{(i,j)}$ for P_i and $\vec{x}_k^{(i,j)}, \vec{y}_k^{(i,j)}$ for P_j
Online:
Apply the steps (1) of the online phase of Fig. 2, obtaining $[\![\vec{v}^{(n)}]\!]$
(2) Parties pick fresh random $[\![r]\!], [\![r']\!], [\![s]\!] \in \mathbb{F}$ and **Reveal** r, s
(3) Parties evaluate the following expression, open it, and check that it is 0

$$[\![r']\!] \cdot \left(\prod_{i=1}^{m} (r - [\![v_i^{(n)}]\!] - s \cdot [\![u_i^{(n)}]\!]) - \prod_{i=1}^{m} (r - [\![v_i^{(0)}]\!] - s \cdot [\![u_i^{(0)}]\!])\right)$$

Fig. 3. Applying a private permutation to a private vector

The protocol for applying a stored π to a private vector $[\![\vec{v}^{(0)}]\!]$ is given in Fig. 3. It assumes that the identity of the permutation we want to apply is already known during the offline phase. This is a natural assumption, if, during the offline phase, we already know the computation that we want to do once we have the data. Even if the assumption does not hold, the protocol in Fig. 3 can still be used, because a permutation tuple $(\rho, \vec{z}; \vec{x}, \vec{y})$ for parties P_i, P_j with a *random* permutation ρ can easily be converted to another permutation tuple with an arbitrary permutation τ known to P_i by P_i sending $\rho^{-1} \circ \tau$ to P_j, which the parties then apply to $\vec{z}$ and $\vec{y}$. If ρ is a random permutation then it is a sufficient mask for that message, preserving the privacy of τ from P_j.

The functionality and privacy arguments of this protocol are the same as for the protocol in Fig. 2. For the correctness check, consider the bivariate polynomial $g_{\vec{u},\vec{v}} \in \mathbb{F}[X, Y]$ for $\vec{u}, \vec{v} \in \mathbb{F}^m$, defined by $g_{\vec{u},\vec{v}}(X, Y) = \prod_{i=1}^{m} (X - v_i - u_i Y)$. By our arguments in Sec. 3.1, two polynomials $g_{\vec{u},\vec{v}}$ and $g_{\vec{u}',\vec{v}'}$ are equal iff the vector of polynomials $(v_1 + u_1 Y, \ldots, v_m + u_m Y)$ is a permutation of the vector $(v'_1 + u'_1 Y, \ldots, v'_m + u'_m Y)$. But this is only possible if $\vec{u}'$ is a permutation of $\vec{u}$, and $\vec{v}'$ is the same permutation of $\vec{v}$. In Fig. 3, we check the equality of $g_{\vec{u}^{(n)},\vec{v}^{(n)}}$ and $g_{\vec{u}^{(0)},\vec{v}^{(0)}}$ by evaluating their difference at a random point (r, s) and checking that it is zero (again masking with r'). We thus obtain that the permutation that brings $\vec{v}^{(0)}$ into $\vec{v}^{(n)}$ is the same that brought $\vec{u}^{(0)}$ into $\vec{u}^{(n)}$.

3.3 Applying the inverse of a random permutation

After creating and storing the representation $[\![\vec{u}^{(0)}]\!], [\![\vec{u}^{(n)}]\!]$ of a random permutation π, we may want to apply π^{-1} to a private vector $[\![\vec{v}^{(0)}]\!]$ [13]. The protocol for this application is basically the same as the one given in Fig. 3, with the following differences: (a) the offline phase is executed with inputs $\pi_1^{-1}, \ldots, \pi_n^{-1}$, (b) the main loop (iterations indexed by i) runs from n down to 1, and (c) the test in step (3) swaps $[\![v_i^{(n)}]\!]$ and $[\![v_i^{(0)}]\!]$.

3.4 Optimizations

As remarked by Chase et al. [8], running in parallel several instantiations of the implementation of $\mathcal{F}_{\text{PREP}}$.**Shuffle** with the same permutation π may have better communication complexity than running them independently. Our online phase requires several permutation tuples with the same permutation, if the overlaying privacy-preserving computation invokes the protocol in Fig. 3 many times, hence the parallel execution is a natural fit.

In the offline phase, we generate permutation tuples of the form $(\pi, \vec{z}; \vec{x}, \vec{y})$. When applying the inverse of a permutation, we need permutation tuples that contain π^{-1}. We can anticipate that need and create these tuples separately. However, we can also transform the tuple $(\pi, \vec{z}; \vec{x}, \vec{y})$ into the permutation tuple $(\pi^{-1}, \pi^{-1}(-\vec{z}); \vec{y}, \vec{x})$ with no interaction between the parties.

4 Security analysis

In previous section, we presented our additions to the protocol Π_{ONLINE}. The extended Π_{ONLINE} is required to be at least as secure as the extended ideal functionality $\mathcal{F}_{\text{ONLINE}}$. The extensions to latter consist of idealized versions to permute a vector, make a random permutation, and apply it, all working with *handles* to private values, and all abortable by the adversary. For space reasons, we omit precise descriptions.

Security is proved by giving a simulator between $\mathcal{F}_{\text{ONLINE}}$ and Π_{ONLINE}. The simulator runs a copy of the real protocol inside, using real inputs of corrupted parties (which the adversary gave to the machines implementing the protocol on behalf of those parties), and dummy inputs for honest parties. The adversary cannot tell the difference between Π_{ONLINE} and $\mathcal{F}_{\text{ONLINE}} \| \mathcal{S}_{\text{ONLINE}}$, because all messages the honest parties send to corrupted parties are uniformly random. Whenever a value y is revealed, which may happen at the end of the protocol, or at other steps of the overlaying algorithm (e.g. after comparisons in a non-data-oblivious sorting algorithm), $\mathcal{F}_{\text{ONLINE}}$ tells $\mathcal{S}_{\text{ONLINE}}$, what the value of y is. The simulator then obtains the shares of corrupted parties from the adversary, and adjusts the shares of the honest parties, such that they add up to y. The same additive correction is applied to honest parties' shares of $\gamma(y)$ (the simulator knows α). If the subsequent MAC check between the simulator and the adversary fails, then the simulator tells $\mathcal{F}_{\text{ONLINE}}$ to stop.

The evaluation of the check in step (3) of Figs. 2–3 does not involve $\mathcal{F}_{\text{ONLINE}}$. It is performed by $\mathcal{S}_{\text{ONLINE}}$ and the adversary. If it fails (the MACs do not pass the check, or the computed difference is non-zero), then $\mathcal{S}_{\text{ONLINE}}$ tells $\mathcal{F}_{\text{ONLINE}}$ to stop. Again, the view of the adversary during this check consists of uniformly randomly distributed values in $\mathbb{F}$.

5 Conclusions

This paper gives the first presentation of an efficient permutation protocol for secure MPC, with security against active adversaries, and with compatibility towards the well-known SPDZ protocol set. Both the communication complexity and the round complexity of the online phase of the protocol are highly attractive in the context of a small number of computing parties, which is expected to be the most frequent use-case. This opens up a large body of efficient algorithms, built on top of passively secure, honest-majority MPC systems, for conversion onto systems with active security.

Our future work involves the choice of implementation details for both the online and offline phases of our protocols. It is also worthwhile to study, to which extent our protocols are adaptable to other rings, similarly to SPDZ [9].

Acknowledgements. This work has been supported by Estonian Research Council, grant no. PRG920, and by European Regional Development Fund through the ICT Centre of Excellence EXCITE.

References

1. Abidin, A., Aly, A., Cleemput, S., Mustafa, M.A.: An mpc-based privacy-preserving protocol for a local electricity trading market. In: Foresti, S., Persiano, G. (eds.) Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10052, pp. 615–625 (2016)
2. Asharov, G., Chan, T.H., Nayak, K., Pass, R., Ren, L., Shi, E.: Bucket oblivious sort: An extremely simple oblivious sort. In: Farach-Colton, M., Gørtz, I.L. (eds.) 3rd Symposium on Simplicity in Algorithms, SOSA 2020, Salt Lake City, UT, USA, January 6-7, 2020. pp. 8–14. SIAM (2020)
3. Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: Rudnicki, P. (ed.) Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, August 14-16, 1989. pp. 201–209. ACM (1989)
4. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from rams to delegatable succinct constraint satisfaction problems: extended abstract. In: Kleinberg, R.D. (ed.) Innovations in Theoretical Computer Science, ITCS '13, Berkeley, CA, USA, January 9-12, 2013. pp. 401–414. ACM (2013)
5. Bogdanov, D., Kamm, L., Laur, S., Sokk, V.: Rmind: A tool for cryptographically secure statistical analysis. IEEE Trans. Dependable Secur. Comput. **15**(3), 481–495 (2018)
6. Bogdanov, D., Laur, S., Talviste, R.: A practical analysis of oblivious sorting algorithms for secure multi-party computation. In: Bernsmed, K., Fischer-Hübner, S. (eds.) Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8788, pp. 59–74. Springer (2014)
7. Bootle, J., Cerulli, A., Groth, J., Jakobsen, S.K., Maller, M.: Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In: Peyrin, T., Galbraith, S.D. (eds.) Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11272, pp. 595–626. Springer (2018)
8. Chase, M., Ghosh, E., Poburinnaya, O.: Secret-shared shuffle. In: Moriai, S., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security,

Daejeon, South Korea, December 7-11, 2020, Proceedings, Part III. Lecture Notes in Computer Science, vol. 12493, pp. 342–372. Springer (2020)

9. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10992, pp. 769–798. Springer (2018)
10. 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.) Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8134, pp. 1–18. Springer (2013)
11. 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., Kwon, D. (eds.) Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7839, pp. 202–216. Springer (2012)
12. de Hoogh, S., Schoenmakers, B., Veeningen, M.: Certificate validation in secure computation and its use in verifiable linear programming. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9646, pp. 265–284. Springer (2016)
13. Laud, P.: Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. Proc. Priv. Enhancing Technol. **2015**(2), 188–205 (2015)
14. Laud, P., Pankova, A., Jagomägis, R.: Preprocessing based verification of multiparty protocols with honest majority. Proc. Priv. Enhancing Technol. **2017**(4), 23–76 (2017)
15. Laud, P., Pettai, M.: Secure multiparty sorting protocols with covert privacy. In: Brumley, B.B., Röning, J. (eds.) Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10014, pp. 216–231 (2016)
16. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings. Lecture Notes in Computer Science, vol. 7001, pp. 262–277. Springer (2011)
17. Mohassel, P., Sadeghian, S.S., Smart, N.P.: Actively secure private function evaluation. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 486–505. Springer (2014)
18. Neff, C.A.: A verifiable secret shuffle and its application to e-voting. In: Reiter, M.K., Samarati, P. (eds.) CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001. pp. 116–125. ACM (2001)