An efficient way to access an array at a secret index

Timothy Atkinson Marius C. Silaghi

Abstract

We propose cryptographic primitives for reading and assigning the (shared) secret found at a secret index in a vector of secrets. The problem can also be solved in constant round with existing general techniques based on arithmetic circuits and the "equality test" in [4]. However the proposed technique requires to exchange less bits. The proposed primitives require a number of rounds that is independent of the size N of the vector, and only depends (linearly) on the number t of computing servers. A previously known primitive for reading a vector at a secret index works only for 2-party computations. Our primitives work for any number of computing participants/servers.

The proposed techniques are secure against passive attackers, and zero knowledge proofs are provided to show that exactly one index of the array is read/written. The techniques work both with multiparty computations based on secret sharing and with multiparty computations based on threshold homomorphic encryption.

1 Introduction

In many general multi-party computation (MPC) frameworks, secrets s from a ring F are distributed among participants using sharing schemes. In a sharing scheme, each participant A_i gets a share denoted $[s]_i^F$, and at least t participants are required to reconstruct the secret from their shares. Arithmetic circuits can then be evaluated securely over these shares [2, 12, 7, 6]. The proposed primitives also work with MPC schemes where secrets are encrypted with a homomorphic public key cypher E allowing additions of plaintext by operations on ciphertexts [3], and whose secret key is distributed among t servers/participants.

Cryptographic Primitives on Shared Secrets Examples of known primitives working on secret shares in a number of rounds that is independent on the possible values of the secrets are:

- $bits(x)$. Transform the shared secret x (with ℓ bits) into a vector $[x]^B$ of ℓ shared secrets, $[x]^B = b_0, b_1, ..., b_\ell$, with possible values {0,1} and representing the corresponding bits of x [4].

- $EXP(x, [y]^B)$. This primitive computes raises x at exponent y where y is shared on bits [5].
- $+, -, *, =, ==, \&\&, ||, <$. These constant round primitives are equivalent to the corresponding "C" operators but work on shared secrets [2, 4].
- unbounded fan-in multiplications (e.g., [1]).
- SHUFFLE(a). Applies a secret random permutation on vector a [11].
- UNSHUFFLE(b). Applies on vector b the inverse of the secret permutation applied on vector a [11].

We now introduce the following primitives:

- $y = a[\overline{x}]$. Reads in y the item at secret index x in the array a containing N shared secrets.
  Note that it could be implemented with arithmetic circuits using N equality tests, $y = \sum_{i=0}^{N-1}((x == i) * a[i])$.
  [9, 10] gave an efficient version for 2-party computations. We propose next an algorithm for this operation in t rounds for computations with threshold t.
  We propose next an algorithm with less bits exchange and t rounds for this operation.
- $a[\overline{x}] = y$. Writes y at index x in the array of N shared secrets a.
  Note that this primitive could also be implemented in constant number of rounds with arithmetic circuits using N equality tests, $a[i] = (x == i) * (y - a[i]) + a[i]$, $i = [0..(N-1)]$.

We also describe a primitive for changing the moduli of the shares of a secret, by passing throught shares in Z (see Section 4).

2 Accessing arrays at secret indices

As mentioned before, this can be done with other primitives in constant round, but requiring N executions of the equality test operator in [4], and which requires more bits exchanges than our methods.

2.1 Bit-based Access

A fast method we propose is based on bit decomposition and exponentiation with secret index [4, 5]. It works for accessing arrays with size $N < \ell$ where $\ell = \log_2(|F|)$. Given the shared secret index x for a vector of length N, first compute

$$d_0, d_1, ..., d_\ell = bits(EXP(2, [x]^B))$$

by first running the *bits* algorithm [4] followed by the secret exponentiation of [5], and followed again by the *bits* algorithm of [4].

Now one can do array read with

$$\sum_{i=0}^{N}(d_i * a[i]).$$

One can write y the array a at index x with

$$a[i] = a[i] + (y - a[i]) * d_i, \forall i \in [0..N].$$

This version is much faster than the ones above since it needs only two expensive *bits* primitives instead on N of them.

2.2 Mixnet-based secret index access

We show how to achieve the result in t rounds, where t is the number of supposed trusted servers/participants. First, we assume that $x \in [0..(N-1)]$ and is shared among t participants using an additive sharing with shares either from the set of integers Z or from Z_N. Transformations from any sharing to a sharing of this form was described in [8, 3], and a version is described in Section 4.

Read at secret index To perform the operation $y = a[\overline{x}]$ where a is an array of N shared secrets, one can use a mixnet related to the one we proposed in [11]. Each participant A_j encrypts his shares $[a_1]_j, .., [a_N]_j$ of a using a homomorphic encryption scheme E_j for which it holds the secret key and which allows for applying addition in F on its plaintexts via some operation $\oslash$ on ciphertexts [11]. All shares are then passed through a mixnet formed by the t participants holding shares of x. Each A_i generates a vector z of N random sharings of zero, and then for each input:

$$I_{j,i} = |E_j([a_1]_j), ..., E_j([a_N]_j)|$$

computes the output

$$O_{j,i} = |E_j([a_1]_j) \oslash E_j([z_1]_j), ..., E_j([a_N]_j) \oslash E_j([z_N]_j)| <<< [x]_i$$

where $[x]_i$ is A_i's share of x and "$<<< [x]_i$" denotes rotational shift with $[x]_i$ positions.

A_i can prove that he shifted the arrays and did not simply replaced them with new arrays, by generating an interractive zero knowledge proof. The zero knowledge proof is based on generating a set of K additional claims, consisting of vectors obtained with different z and different shifts.

$$C_{j,k} = |E_j([a_1]_j) \oslash E_j([z_1^k]_j), ..., E_j([a_N]_j) \oslash E_j([z_N^k]_j)| <<< s^k, k = [1..K]$$

The verifiers specify a challenge bit c_k for each k. For bits $c_k = 0$, the prover reveals s^k and all shares of z^k, showing that the claims $C_{*,k}$ are a rotation of the input. For bits $c_k = 1$, the prover reveals $s^k - [x]_i$ and all shares of $z - z^k$, showing that the claim is a rotation of the output.

At the end of the mix-net, the last agent in the chain broadcasts all encryptes shares in $O_{*,t}[1]$ and each participant decrypts its shares obtaining $a[\overline{x}]$.

In the case of MPCs based on homomorphic threshold encryption E, the mixnet is run on ciphertexts (operations remain the same but without involving shares):

$$O_i = |E(a_1) \oslash E_j(0), ..., E(a_N) \oslash E_j(0)| <<< [x]_i$$

and the claims in the ZK proof are:

$$C_k = |E([a_1]) \oslash E([z_1^k]), ..., E([a_N]) \oslash E([z_N^k])| <<< s^k, k = [1..K]$$

Write at a secret index To perform the operation $a[\overline{x}] = y$ where a is an array of N shared secrets, one can use a bidirectional mixnet related to the one proposed in [11]. In Phase 1, each participant A_j encrypts his shares $[a_1]_j, .., [a_N]_j$ of a using a homomorphic encryption scheme E_j for which it holds the secret key and which allows for applying addition in F on its plaintext via some operation $\oslash$ on ciphertexts [11]. All shares are then passed through a mixnet formed by the t participants holding shares of x. Each A_i generates a vector z of N random sharings of zero, and then for each input:

$$I_{i,j} = |E_j([a_1]_j), ..., E_j([a_N]_j)|$$

computes the output

$$O_{i,j} = |E_j([a_1]_j) \oslash E_j([z_1]_j), ..., E_j([a_N]_j) \oslash E_j([z_N]_j)| <<< [x]_i$$

where $[x]_i$ is A_i's share of x and $<<< [x]_i$ denotes rotational shift with $[x]_i$ positions towards the left.

At the end of the mix-net, the t^{th} participant in the chain obtains as $O_{*,t}[1]$ the encrypted shares of $a[\overline{x}]$. Now each participant A_j sends to A_t its share of y encrypted with E_j, and A_t replaces $O_{j,t}[1]$ with $E_j([y]_j)$.

In Phase 2, the mix-net is now run in the reverse direction with $O_{*,t}$ as input. Each A_i generates a vector z' of N random sharings of zero, and then for each input:

$$I'_{j,i} = |E_j([a_1]_j), ..., E_j([a_N]_j)|$$

computes the output

$$O'_{j,i} = |E_j([a_1]_j) \oslash E_j([z'_1]_j), ..., E_j([a_N]_j) \oslash E_j([z'_N]_j)| >>> [x]_i$$

where $[x]_i$ is A_i's share of x and $>>> [x]_i$ denotes rotational shift with $[x]_i$ positions towards the right. The result $O'_{*,i}$ is the result vector a.

A_i can prove that he shifted the arrays and did not simply replaced them with new arrays, by generating an interractive zero knowledge proof. This proof also shows that the rotation is with the same number of positions and in the reverse direction as the first phase. The zero knowledge proof is based on generating, besides a set of K claims for Phase 1 as at the previous technique:

$$C_{j,k} = |E_j([a_1]_j) \oslash E_j([z_1^k]_j), ..., E_j([a_N]_j) \oslash E_j([z_N^k]_j)| <<< s^k, k = [1..K]$$

a set of K additional claims, consisting of vectors obtained with different z' and the shifts of the corresponding claims of the first phase.

$$C'_{j,k} = |E_j([a_1]_k) \oslash E_j([z_1'^k]_j), ..., E_j([a_N]_j) \oslash E_j([z_N'^k]_j)| >>> s^k, k = [1..K]$$

The verifiers specify a challenge bit c_k for each k in [1..K]. For bits $c_k = 0$, the prover reveals s^k and all shares of z and z'^k, showing that at both phases the claims $C_{j,*}$ and $C'_{j,*}$ are a rotation of their inputs. For bits $c_k = 1$, the prover reveals $s^k - [x]_i$ and all shares of $z - z^k$ and $z' - z'^k$, showing that at both phases the claims $C_{j,*}$ and $C'_{j,*}$ are rotations of the output.

At the end of the mix-net, the last agent in the chain broadcasts all encrypted shares in $O'_{*,t}$ and each participant decrypts its shares obtaining the result vector a.

As for the read operation, in the case of MPCs based on homomorphic threshold encryption E, the mixnet is run on ciphertexts (operations remain the same but without involving shares). At Phase 1:

$$O_i = |E(a_1) \oslash E_j(0), ..., E(a_N) \oslash E_j(0)| <<< [x]_i$$

and the claims in the ZK proof are:

$$C_k = |E([a_1]) \oslash E([z_1^k]), ..., E([a_N]) \oslash E([z_N^k])| <<< s^k, k = [1..K]$$

at Phase 2:

$$O'_i = |E(a_1) \oslash E_j(0), ..., E(a_N) \oslash E_j(0)| >>> [x]_i$$

and the claims in the ZK proof are:

$$C'_k = |E([a_1]) \oslash E([z_1^k]), ..., E([a_N]) \oslash E([z_N^k])| >>> s^k, k = [1..K]$$

A similar ZK proof as above is performed simultaneously with the claims of both phases.

3 Analysis

The array read requires $t{+}1$ rounds in the passive attacker model (without the ZK proofs which add Kt messages). The number of exchanged bits is $(t{+}1) * t * \ell * N = O(t^2 \ell N)$ for secret sharing and $(t{+}1) * \ell * N = O(t\ell N)$ for MPC with homomorphic threshold encryption when $|F| = \ell$. The array write requires two times this number of operations.

With the $y = \sum_{i=0}^{N-1}((x == i) * a[i])$ implementation of array read, the number of exchanged bits is $O(N * t^2 \ell log(\ell))$ for secret sharing.

4 Changing moduli of a shared secret

Transformations between any sharings was described in [8, 3]. Here we describe a method for transforming from a sharing with shares mod p to shares mod q (via shares that are integers in Z).

Sometimes, it is necessary to take a number a which is currently in Z_p and switch it to Z_q. This may be necessary for something as simple as trying to add a with a number b which is currently Z_q (but also for getting a representation of indices that enables mixnets for accessing arrays as shown above).

As typically a sharing of a with shares mod p is denoted $[a]^p$. To convert a sharing $[a]^p$ to $[a]^q$:

1. First generate a random number, r, on bits: $[r_1]^Z, [r_2]^Z, ..., [r_\ell]^Z$ where $0 \leq [r_i]^Z \leq 1, 0 \leq i \leq \ell, 0 \leq r < p$ and ℓ is the number of bits necessary to represent p.
2. calculate $[r]^Z = [r_1]^Z * 2^0 + [r_2]^Z * 2^1 + ... + [r_\ell]^Z * 2^{\ell-1}$.
3. convert $[r]^Z$ to $[r]^p$ by the following formula $[r]_i^p = [r]_i^Z mod(p)$.
4. calculate $[b]^p = [a]^p - [r]^p$.
5. reveal b.
6. convert b to bits $[b_1]^Z, [b_2]^Z, ..., [b_\ell]^Z$.
7. compute $[a_i]^Z = [b_i]^Z + [r_i]^Z$.
8. if $[a_i]^Z > p$ then subtract p from a (as done in [4])
9. construct $[a]$ by calculating: $[a]^Z = [a_1]^Z * 2^0 + [a_2]^Z * 2^1 + .. + [a_\ell]^Z * 2^{\ell-1}$.
10. convert $[a]^Z$ to $[a]^q$ by $[a]_i^q = [a]_i^Z mod(q)$.

Generating a random number r, $0 \leq r < p$, (p havig ℓ bits) is possible as in step 1. Namely one generates ℓ random bits, that can be compared with a bit representation of p as described in [4]. If the result is too large, then the process is repeated. A sharing of r can then be obtained as in step 2.

References

[1] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds interaction. In *8th ACM Symposium Annual on Principles of Distributed Computing*, pages 201–209, August 1989.

[2] M. Ben-Or, S. Goldwasser, and A. Widgerson. Completeness theorems for non-cryptographic fault-tolerant distributed computating. In *STOC*, pages 1–10, 1988.

[3] R. Cramer, I. Damgøard, and J.B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *BRICS RS-00-14*, 2000.

[4] I. Damgård, M. Fitzi, J. B. Nielsen, and T. Toft. How to split a shared number into bits in constant round and unconditionally secure. Cryptology ePrint Archive, Report 2005/140, 2005. `http://eprint.iacr.org`.

[5] Yevgeniy Dodis, Aleksandr Yampolskiy, and Moti Yung. Threshold and proactive pseudo-random permutations. Cryptology ePrint Archive, Report 2006/017, 2006. `http://eprint.iacr.org/`.

[6] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge, 2004.

[7] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

[8] E. Kiltz. Unconditionally secure constant round multi-party computation for equality, comparison, bits and exponentiation. Cryptology ePrint Archive, Report 2005/066, 2005. `http://eprint.iacr.org`.

[9] M. Naor and K. Nissim. Communication complexity and secure function evaluation. In *ECCC - Electronic Colloquium on Computational Complexity, Report TR01-062*, 2001.

[10] K. Nissim and R. Zivan. Secure discsp protocols - from centralized towards distributed solutions. In *DCR05 Workshop*, 2005.

[11] M.-C. Silaghi. Zero-knowledge proofs for mix-nets of secret shares and a version of ElGamal with modular homomorphism. Cryptology ePrint Archive, Report 2005/079, 2005. `http://eprint.iacr.org`.

[12] A. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, 1982.