Efficient coding for secure computing with additively-homomorphic encrypted data

Thijs Veugen

Unit ICT, TNO,
The Hague, The Netherlands
and
Research Group Cryptology, CWI,
Amsterdam, The Netherlands
Email: thijs.veugen@tno.nl

Abstract: A framework is introduced for efficiently computing with encrypted data. We assume a semi-honest security model with two computing parties. Two different coding techniques are used with additively homomorphic encryption, such that many values can be put into one large encryption, and additions and multiplications can be performed on all values simultaneously. For more complicated operations such as comparisons and equality tests, bit-wise secret sharing is proposed as an additional technique that has a low computational and communication complexity, and which allows for precomputing. The framework is shown to significantly improve the computational complexity of state-of-the-art solutions on generic operations such as secure comparisons and secure set intersection.

Keywords: packing; batching; homomorphic encryption; secure comparison; secure equality; secure set intersection; vector addition chain.

Reference to this paper should be made as follows: Veugen, T. (xxxx) 'Efficient coding for secure computing with additively-homomorphic encrypted data', *Int. J. Applied Cryptography*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Thijs Veugen received his two MSc degrees, in Mathematics and Computer Science, both cum laude, and a PhD in Information Theory, all from Eindhoven University of Technology. After that, he worked as a Scientic Software Engineer at Statistics Netherlands, Heerlen, the Netherlands. Since 1999, he has been a Senior Scientist Applied Cryptography at TNO, currently within the unit ICT in The Hague, the Netherlands. From 2008 till 2016, he was also affiliated as a Senior Researcher with the Cyber Security group of Delft University of Technology. Since 2016, he has been part-time member of the Cryptology group at CWI in Amsterdam, the Netherlands. He has written many scientific papers on computing with encrypted data, serves frequently as a member of the program committee board of information security related conferences, and holds numerous related patents in various countries.

1 Introduction

The area of computing with encrypted integers has been developing steadily over the past couple of years (Lagendijk and Barni, 2013). Many applications are known, where different parties need to compute with encrypted data, without having to decrypt intermediate values, to assure that sensitive data is not leaked. Although a high level of security can be achieved in this way, current solutions still suffer from efficiency drawbacks, which delay their exploitation. In the current paper we introduce a framework that contributes to mitigating these disadvantages.

We consider a semi-honest security model, where two parties jointly compute with encrypted data using efficient coding schemes that are designed and fine-tuned for this purpose. In particular, we deploy the common setting of party $\mathcal{A}$ having all encrypted values, and party $\mathcal{B}$ holding the decryption key (Lagendijk and Barni, 2013). This setting, which originates from the domain of encrypted signal processing (Lagendijk and Barni, 2013), is different from several efficient secure multi-party computation platforms that have been developed over the last few years (Keller et al., 2016; Furukawa et al., 2017). Although the crypto system can be any additively homomorphic scheme, the well-known Paillier scheme (Paillier, 1999) is used.

In our model, party $\mathcal{A}$ is able to add different encrypted values without needing support from party $\mathcal{B}$, due to the additively homomorphic property of the scheme. For more complicated operations such as multiplications or comparisons, a common approach is to have party $\mathcal{A}$ additively blind encrypted values by adding a large random number, which can safely be decrypted by party $\mathcal{B}$, and using this decrypted value in a protocol between $\mathcal{A}$ and $\mathcal{B}$, which implements the required operation (Damgård et al., 2008) at the cost of additional communication and computation.

An important cause of the lack of efficiency when computing with encrypted data is the blow-up of integers during encryption. For security reasons, each encryption has a size of one or more kilobit (Kb), irrespective of the size of the plaintext integer. This increases both communication and computational complexity. Therefore, the main goal of our framework is to reduce the difference in size between plaintext and ciphertext. For integer operations like additions and multiplications, this is achieved by using coding schemes that put many small integers into one large integer. More complex, bit-wise operations like comparisons and equality tests are implemented by binary secret sharings, where the size of the 'ciphertext' is very small.

Since encoding (putting many small values into one large integer) and decoding in the encrypted domain takes some effort, the best approach is to do this in the plain domain, or to perform most operations in the encrypted domain on encoded data, and to only decode in the end. This also limits the number of decryptions, which are quite costly.

An interesting alternative, avoiding additional communication, would be to use fully homomorphic encryption, for which similar coding techniques are known (Smart and Vercauteren, 2014). However, despite some recent advances, to date fully homomorphic encryption schemes are mainly of theoretical interest and far too inefficient to be used in practice (Lagendijk and Barni, 2013). Additively homomorphic schemes sincerely reduce the computational effort, but require some additional communication to enable secure multiplications.

In the next subsection, we explain our contribution more explicitly, followed by the notation used throughout the paper. In Sections 2 and 3, we introduce the two coding schemes, namely packing and batching. Section 4 shows how bit-wise secret sharing can be used to implement the more complicated bit-wise operations in the encrypted domain. This is combined in Section 5 with secure comparison of packed, or batched, encryptions. A couple of new applications, which require batching, are illustrated in Section 6. In Section 7, the security properties of the framework are explained. The performance of our framework is compared to state-of-the-art solutions in Section 8, and we end by giving directions for further research and our conclusions.

1.1 Our contribution

We show two efficient coding schemes for basic computations, such as additions and multiplications, on encrypted integers, in the semi-honest two-party model. For more complicated, bit-wise operations, such as comparison and equality, we propose a bit-wise secret sharing scheme. Decoding is preferably performed in the plain domain, since secure comparison protocols are needed to decode in the encrypted domain. In terms of computation and communication, the most intensive part of this secret sharing scheme can be precomputed, and is independent of the actual data. We propose efficient solutions that allow the two parties to jointly precompute. Alternatively, a trusted dealer can be used.

We show how to securely compare batched, or packed, integers in the encrypted domain, and how to determine whether they are equal. We show how our framework can be used to improve the efficiency of state-of-the-art solutions to generic operations like secure multiplications, secure comparison, and secure equality, but also secure set intersection. This results, amongst others, in seven new cryptographic protocols, shown throughout the paper.

1.2 Notation and parameters

Throughout this paper we make use of various symbols, which we have defined in Table 1. The remainder of x after reduction modulo M is denoted by $y = x \bmod M$, such that $0 \le y < M$. On the other hand, $y = x \pmod M$ refers to the equivalence relation, so $(y - x) \bmod M = 0$, but not necessarily $0 \le y < M$.

Table 1 Notation

ℓ	Bit length of inputs
n	Number of compartments
ν	Size of one packed compartment
N	Size of plaintext
$[.]$	(Paillier) encryption
$\mathcal{A}, \mathcal{B}$	Parties
X	Packed, or batched, integer
$(x < y)$	The bit representing the Boolean $x < y$
γ_i, δ_i	Binary outcomes of comparison or equality
θ	Blinded batch, or packing
$(.)_B$	Batch
$\div$	Integer division
K	Decryption key
κ	Statistical security parameter
σ	Bit length of one batched compartment
m_i	Size of batched compartment i
M	Modulus of one batch
$\langle . \rangle$	Secret sharing
$x^{\mathcal{A}}, x^{\mathcal{B}}$	Privately known integers
x_i	Content of compartment i
$(x = y)$	The bit representing the Boolean $x = y$
$x_{(i)}$	The ith bit of integer x, $x_{(0)}$ being the right-most bit
R	Random number used for additive blinding
$(.)_P$	Packing
$\oplus$	Exclusive-or

We assume that the plaintext size N has 2,048 bits, a reasonable security parameter nowadays, and that the statistical security parameter $\kappa = 80$. Because we choose Paillier as the additively homomorphic crypto system, all computations on cipher texts are done modulo N^2. However, we will omit this modulo reduction to avoid distracting the reader with details. Furthermore, we sometimes write $\log_2$, the logarithm with base two, but neglect that this is usually not an integer value, and the outcome should be rounded up.

2 Batching

We use a technique known from fully homomorphic encryption called batching (Smart and Vercauteren, 2014). We optimise this coding scheme for additively homomorphic encryption, show efficient methods for encoding and decoding in the encrypted domain, and an efficient approach for multiplying encrypted batches. These methods use a new addition chain algorithm for vectors of width two.

Let m_i, $1 \le i \le n$, be a collection of integers, which are pairwise coprime. Let $M = \prod_{i=1}^{n} m_i$ be their product. The idea is that n small integers, each of size m_i, can be batched into one large integer of size M, and additions and multiplications on the large integer will translate to identical operations on the n small integers, according to the well-known Chinese remainder theorem.

To be able to compute with batchings in the encrypted domain, we need $\log_2 M + \kappa < \log_2 N$, where $\kappa = 80$ is the security parameter, and N is the modulus of the crypto system.

2.1 Encoding and decoding in the plain domain

To batch n integers into one large integer, we compute

$$(x_1, x_2, \ldots, x_n)_B = \sum_{i=1}^{n} \mu_i \frac{M}{m_i} x_i,$$

where μ_i is the multiplicative inverse of $\frac{M}{m_i}$ modulo m_i (Knuth, 1997).

Decoding a batched integer X into n integers is simply computing $x_i = \mathrm{X} \bmod m_i$.

2.2 Encoding and decoding in the encrypted domain

To encode n encrypted integers into one encrypted batch $[X]$, we can use the additively homomorphic property of the crypto system:

$$[X] = \prod_{i=1}^{n} [x_i]^{\mu_i \frac{M}{m_i}}.$$

This product can be computed efficiently by means of a vector addition chain. The algorithm provided in Appendix A takes roughly $(1 + \log_2 n) \cdot 0.8 \cdot \sigma n$ multiplications of two encryptions, where $\sigma = \log_2 m_i$.

To decode an encrypted batch $[X]$ to n encrypted integers, we compute $[X \bmod m_i]$ for all i, $1 \le i \le n$, in parallel, as shown in Protocol 1.

Protocol 1 Debatching in the encrypted domain

Party	$\mathcal{A}$	$\mathcal{B}$
Input	$[X]$	K
Output	$[x_i]$, $1 \le i \le n$	
Constraints	$X = (x_1, x_2, \ldots, x_n)_B$ and $0 \le x_i < m_i$, $1 \le i \le n$, $0 \le \mathrm{X} < N \cdot 2^{-\kappa}$	

1. Party $\mathcal{A}$ generates a large random variable R, and additively blinds X: $[\theta] = [X] \cdot [R] = [X + R]$.
2. Party $\mathcal{A}$ sends $[\theta]$ to $\mathcal{B}$.
3. Party $\mathcal{B}$ decrypts it, 'debatches' θ, and sends the n integers $\theta_i = \theta \bmod m_i$, $1 \le i \le n$, separately encrypted, to party $\mathcal{A}$.
4. Parties $\mathcal{A}$ and $\mathcal{B}$ perform n parallel secure comparisons, such that $\mathcal{A}$ obtains n encrypted comparison bits $[\delta_i]$, where $\delta_i = (\theta_i < (R \bmod m_i))$.
5. Party $\mathcal{A}$ computes $[x_i] = [\theta_i] \cdot [-R \bmod m_i] \cdot [\delta_i]^{m_i}$, for each i, $1 \le i \le n$.

The correctness of debatching in the encrypted domain is shown in Veugen (2014). In Subsection 4.3, a secure comparison protocol is described to compute the δ_i's. The exponentiation $[\delta_i]^{m_i}$ can be avoided by translating $\langle \delta_i \rangle$ directly to $[m_i \cdot \delta_i]$.

2.3 Multiplication of encrypted batches

The main advantage of batched coding is that many multiplications can be combined into one (secure) multiplication:

$$(x_1, x_2, \ldots, x_n)_B \cdot (y_1, y_2, \ldots, y_n)_B = (x_1 \cdot y_1, x_2 \cdot y_2, \ldots, x_n \cdot y_n)_B \pmod{M}.$$

To take full advantage, the number of compartments n, and therefore also the number M, should be as large as possible. However, the secure multiplication is computed modulo N, the plaintext size of the crypto system, and a carry-over modulo N would destroy the correctness of the result. The easiest way to avoid a carry-over modulo N, is to ensure $M^2 < N$. This halves the number of compartments we can put into one encryption. Furthermore, since we also want to be able to additively blind (and debatch) intermediate results, we require $M^2 < N \cdot 2^{-\kappa}$.

After one secure multiplication, the batched result has grown from size M to M^2. This does not pose a serious problem if the remaining batched operations are additions. However, to be able to compute another batched multiplication, we need to reduce the size of the batch, without affecting its result modulo M. Computing the exact result modulo M would require a large secure comparison (Bianchi et al., 2009), so we approximate the result, similar to the approach of Veugen (2014), which computes the modular remainder, possibly added with M. This can be combined with a standard secure multiplication (Erkin et al., 2012) to Protocol 2.

Protocol 2 Multiplying two encrypted batches

Party	$\mathcal{A}$	$\mathcal{B}$
Input	$[X]$ and $[Y]$	K
Output	$[A]$	
Constraints	$0 \leq X, Y < N \cdot 2^{-\kappa}$ and $A = (X \cdot Y) \pmod M$, $0 \leq A < 4M^2$	

1 Party $\mathcal{A}$ has the encrypted batches $[X]$ and $[Y]$. He generates two large random numbers R_X and R_Y, to additively blind them: $[\theta_X] = [X] \cdot [R_X]$ and $[\theta_Y] = [Y] \cdot [R_Y]$.

2 Party $\mathcal{A}$ sends $[\theta_X]$ and $[\theta_Y]$ to party $\mathcal{B}$, who decrypts them.

3 Party $\mathcal{B}$ computes $(\theta_X \bmod M)$, $(\theta_Y \bmod M)$, and their product $(\theta_X \bmod M) \cdot (\theta_Y \bmod M)$, encrypts them, and returns the three encryptions to $\mathcal{A}$.

4 Party $\mathcal{A}$ computes the approximate modular remainder of the product $X \cdot Y$:

$$[A] = [(\theta_X \bmod M) \cdot (\theta_Y \bmod M)] \cdot [\theta_X \bmod M]^{(-R_Y) \bmod M} \cdot [\theta_Y \bmod M]^{(-R_X) \bmod M} \cdot [((-R_X) \bmod M) \cdot ((-R_Y) \bmod M)].$$

In Veugen (2014), it is shown that $(\theta_X \bmod M) + ((-R_X) \bmod M)$ equals either $(X \bmod M)$ or $(X \bmod M) + M$, and similarly for $(\theta_Y \bmod M) + ((-R_Y) \bmod M)$. Therefore, A, which is the product of these two approximations, equals XY modulo M, and is upper bounded by $(2M) \cdot (2M) = 4M^2$.

This secure multiplication protocol for batches takes one communication round, and mainly costs two decryptions by $\mathcal{B}$, and two exponentiations by $\mathcal{A}$. It approximates the product XY, and reduces the size of the approximate product A to order M^2, without disturbing its remainder modulo M. The two exponentiations by $\mathcal{A}$ can be efficiently computed by a vector addition chain, as explained in Subsection A.1.

2.4 Partial debatching in the encrypted domain

Obviously, instead of debatching all compartments, it is possible to retrieve the contents of only a small number of compartments. This makes it possible to, for example, change the contents of a particular compartment, or switch the content of two different compartments. The switching can be accomplished by computing $[X] \cdot [(0, 0, \ldots, x_j - x_i, 0, \ldots, x_i - x_j, 0, \ldots, 0)_B]$, which switches the content of compartments i and j, $1 \leq i < j \leq n$. To compute the latter encrypted batch, containing only $[x_j - x_i]$ and $[x_i - x_j]$, we compute $\left([x_j - x_i]^{\mu_i m_j} \cdot [x_i - x_j]^{\mu_j m_i}\right)^{\frac{M}{m_i m_j}} = [x_j - x_i]^{(\mu_i m_j - \mu_j m_i)\frac{M}{m_i m_j}}$, which is easier than computing a complete batch in the encrypted domain.

2.5 Choosing the m_i

To be able to batch, we need to find n coprime numbers m_i, which should not be too large. For example, when all integers are at most ℓ bits, we need to find n coprime numbers between 2^ℓ and $2^{\ell+1}$. The average number of primes smaller than x is $x/\ln x$, so the average number of primes in this interval is $2^\ell \dfrac{\ell - 1}{\ell(\ell+1)\ln 2}$ (see also Table 2). Common values for $\log_2 N$ and κ are 2,048 and 80, so assuming our goal is a full batch, we need at least $n \approx (2{,}048 - 80)/(\ell + 1)$ coprime numbers of $\ell + 1$ bits. If we only allowed the m_i to be prime, we would need $2^\ell \dfrac{\ell - 1}{\ell \ln 2} \geq 1{,}968$, thus $\ell \geq 11$. Since they only need to be pairwise coprime, we have more options, but for smaller ℓ it seems that some of the m_i should be chosen somewhat larger than $2^{\ell+1}$ in order to have sufficient coprime numbers. We should also take into account that the compartment size σ usually exceeds the input size ℓ, since an output value, which must fit into one compartment, is the result of a series of operations on inputs, and the size of the intermediate results grows during these operations.

Table 2 Number of primes per bit size

# bits	3	4	5	6	7
# primes	2	2	5	7	13
# bits	8	9	10	11	12
# primes	23	43	75	137	255

3 Packing

A different technique for putting multiple integers into one encryption, and performing parallel operations on all integers, is called packing (Erkin et al., 2012). We introduce a way of unpacking in the encrypted domain, which is more efficient than existing methods.

In packing, the (encrypted) integers are simply concatenated, which enables parallel additions with integers. Packing also allows parallel multiplications with a constant value: $(x_1, x_2, \ldots, x_n)_P \cdot y = (x_1 \cdot y, x_2 \cdot y, \ldots, x_n \cdot y)_P$, but not with different values as in batching.

Although it is possible to have a different size for each compartment, the compartment size is usually constant. We denote the compartment size with ν, which is often a power of two.

3.1 Encoding and decoding in the plain domain

In packing, n integers of size ν are packed together into one large integer, by concatenating them:

$$(x_1, x_2, \ldots, x_n)_P = \sum_{i=1}^{n} \nu^{i-1} x_i.$$

Decoding a packed integer, also called unpacking, is the reconstruction of the n integers of size ν:

```
1  $X = (x_1, x_2, \ldots, x_n)_P$
2  for $i = 1$ to $n - 1$ do
     $\{X = (x_i, x_{i+1}, \ldots, x_n)_P\}$
     a  $x_i = X \bmod \nu$
```

```
    b   $X = (X - x_i)/\nu$
3   $x_n = X$.
```

If ν is a power of two, unpacking is simply grouping the bits of X into n integers.

3.2 *Encoding and decoding in the encrypted domain*

If the n integers are encrypted with an additively homomorphic crypto system, packing can be done efficiently using Horner's rule:

```
1   $[X] = [x_n]$
2   for $i = n - 1$ down to 1 do $\{[X] = [(x_{i+1}, \ldots, x_n)_P]\}$
    a   $[X] = [X]^{\nu}$
    b   $[X] = [X] \cdot [x_i]$
```

This mainly requires $n - 1$ exponentiations to the power ν, which costs roughly $(n - 1) \cdot 1.5 \log_2 \nu$ multiplications of large integers.

Unpacking in the encrypted domain is a bit more involved. The algorithm from Bianchi et al. (2009) translates the unpacking algorithm from the plain domain straightforward to the encrypted domain, which requires a large exponentation in each iteration in order to divide by ν. Moreover, since the encrypted content of each compartment is computed one-by-one, the number of communication rounds is linear in n, and n decryptions are needed.

We suggest an alternative approach, similar to debatching in the encrypted domain. We additively blind X with a large random number R, and ask $\mathcal{B}$ to unpack $\theta = X + R$. The addition of R might cause an overflow in some compartments. The difference with batching is that these overflows create a carry-over, which influences the content of the next compartment: $\theta_i = (x_i + R_i + c_{i-1}) \bmod \nu$, where c_{i-1} is the carry-over bit from the previous compartment. This can be solved by unpacking $2X$ instead of X, and dividing the content of each compartment by two, discarding the remainder. Since adding one (carry-over) bit to the even number $2x_i$ will not change the outcome $x_i = (2x_i + c_{i-1}) \div 2$, we have a solution that is redundant to possible carry-covers. We only need to increase the compartment size with one bit to ensure $2x_i < \nu$. We also assume ν is even. Protocol 3 shows how to unpack in the encrypted domain.

Before step 6, we have $\theta_i = (2x_i + R_i + c_{i-1}) \bmod \nu$, $1 \leq i \leq n$, so either $2x_i + c_{i-1} = \theta_i - R_i$, or $2x_i + c_{i-1} = (\theta_i + \nu) - R_i$. To decide whether there was an overflow in the ith compartment, we need to compute the n encrypted comparison bits $[\delta_i]$, where $\delta_i = (\theta_i < R_i)$. These can be computed efficiently using n parallel secure comparisons, as described in Subsection 4.3.

The correctness of the computation of x_i in step 8 follows from observing $2x_i + c_{i-1} = \theta_i - R_i$. In this case, $x_i = (2x_i + c_{i-1}) \div 2 = (\theta_i \div 2) - (R_i \div 2) - (\theta_i \bmod 2 < R_i \bmod 2)$ (Veugen, 2014). Only if θ_i is even, and R_i is odd, does the result of the integer division need to be corrected by one. If we have an overflow in the ith compartment, then $\delta_i = 1$, and we have the relation $2x_i + c_{i-1} = (\theta_i + \nu) - R_i$, for which similarly holds $x_i = ((\theta_i + \nu) \div 2) - (R_i \div 2) - ((\theta_i + \nu) \bmod 2 < R_i \bmod 2)$. Since ν is even, $(\theta_i + \nu) \div 2 = (\theta_i \div 2) + \nu/2$, and $(\theta_i + \nu) \bmod 2 = \theta_i \bmod 2$.

Protocol 3 Unpacking in the encrypted domain

Party	$\mathcal{A}$	$\mathcal{B}$
Input	$[X]$	K
Output	$[x_i]$, $1 \leq i \leq n$	
Constraints	$X = (x_1, x_2, \ldots, x_n)_P$ and $0 \leq 2x_i < \nu$, $1 \leq i \leq n$ and ν is even and $0 \leq X < N \cdot 2^{-(\kappa+1)}$	

1. Party $\mathcal{A}$ generates a large random variable R, and additively blinds $2X$: $[\theta] = [X]^2 \cdot [R] = [2X + R]$.
2. Party $\mathcal{A}$ sends $[\theta]$ to $\mathcal{B}$.
3. Party $\mathcal{B}$ decrypts it, and unpacks $\theta \bmod \nu^n$ to θ_i, $1 \leq i \leq n$.
4. For each i, party $\mathcal{B}$ computes $\theta_i \div 2$ and $\theta_i \bmod 2$, and sends the $2n$ integers, separately encrypted, to party $\mathcal{A}$.
5. Party $\mathcal{A}$ unpacks $R \bmod \nu^n$ to R_i, $1 \leq i \leq n$.
6. Parties $\mathcal{A}$ and $\mathcal{B}$ perform n parallel secure comparisons, such that $\mathcal{A}$ obtains n encrypted comparison bits $[\delta_i]$, where $\delta_i = (\theta_i < R_i)$.
7. For each i, party $\mathcal{A}$ computes $[\gamma_i] = [-(\theta_i \bmod 2 < R_i \bmod 2)]$. If $R_i \bmod 2 = 0$, $[\gamma_i] = [0]$, else $[\gamma_i] = [(\theta_i \bmod 2) - 1] = [\theta_i \bmod 2] \cdot [-1]$.
8. Party $\mathcal{A}$ computes $[x_i] = [\theta_i \div 2] \cdot [-R_i \div 2] \cdot [\gamma_i] \cdot [\delta_i]^{\nu/2}$, for each i, $1 \leq i \leq n$.

3.3 *Packing versus batching*

Packing and batching are both techniques for putting many integers into one encryption, and allowing parallel operations on them. From a functional point of view, the main difference is that batching allows both adding and multiplying, while packing only allows adding. Encoding a batch is more complex and computationally more intensive than encoding a packing, both in the plain and in the encrypted domain. The effort required for decoding them is comparable, although debatching in the encrypted domain is slightly less intensive than unpacking in the encrypted domain.

In batching, adding a single constant to a batched integer results in adding the constant to all compartments. In packing, the constant is only added to the first compartment. To add the same constant value to all compartments, the constant needs to be packed first. Multiplying a packing with a constant results in multiplying each compartment with this constant, similar to batching. By batching different, publicly known constants, each compartment can be multiplied with a different value. This is not possible with packing.

An important difference is that packing allows for more compartments than batching, since a multiplication of two batched values doubles its bit size (which has to be reduced afterwards). For both techniques, partial decoding, i.e., obtaining the (encrypted) result of a few compartments, is possible.

4 Bit-wise operations on private integers

Batching can be used for additions and multiplications of encrypted integers. For more complex, bit-wise operations, like comparisons and equalities, we suggest a different cryptographic tool. This setting, which, as we will see later, combines nicely with packing and batching is where both parties privately hold an integer, on which they would like to perform a bit-wise operation.

4.1 Bit-wise secret sharing

A common solution for comparing two privately held integers is by Damgård et al. (2008, 2009). Although it uses a fine-tuned homomorphic crypto system, in the semi-honest model the protocol can be significantly sped up by using secret sharing modulo two (De Cock et al., 2017). In this crypto system, secrets are binary values that are secret-shared between party $\mathcal{A}$ and $\mathcal{B}$. More precisely, let $c \in \{0, 1\}$ be a bit, then the sharing $\langle c \rangle$ consists of a bit $c^{\mathcal{A}}$, held by party $\mathcal{A}$, and a bit $c^{\mathcal{B}}$, held by party $\mathcal{B}$, such that $c^{\mathcal{A}} \oplus c^{\mathcal{B}} = c$, where $\oplus$ denotes exclusive-or, the addition modulo two.

Two shared secrets can be added, by locally adding the shares. To multiply two shared secrets, we need two random sharings $\langle r_1 \rangle$ and $\langle r_2 \rangle$, and the sharing of their product $\langle r_1 \cdot r_2 \rangle$. The multiplication protocol by Beaver (1998) is as follows:

1 parties $\mathcal{A}$ and $\mathcal{B}$ secretly share bits b_1 and b_2

2 they locally compute $\langle \beta_1 \rangle = \langle b_1 \rangle \oplus \langle r_1 \rangle = \langle b_1 \oplus r_1 \rangle$ and $\langle \beta_2 \rangle = \langle b_2 \rangle \oplus \langle r_2 \rangle = \langle b_2 \oplus r_2 \rangle$

3 the parties open β_1 and β_2, which means they show each other shares

4 they locally compute $\langle b_1 \cdot b_2 \rangle = \langle r_1 \cdot r_2 \rangle \oplus (\beta_2 \cdot \langle r_1 \rangle) \oplus (\beta_1 \cdot \langle r_2 \rangle) \oplus (\beta_1 \cdot \beta_2) = \langle (r_1 r_2) \oplus (\beta_2 r_1) \oplus (\beta_1 r_2) \oplus (\beta_1 \beta_2) \rangle$.

Correctness follows from $(\beta_1 \oplus r_1)(\beta_2 \oplus r_2) = (\beta_1 \beta_2) \oplus (\beta_1 r_2) \oplus (r_1 \beta_2) \oplus (r_1 r_2)$.

4.2 Precomputing triplets

The only open question remaining is how to generate the random triplets $(\langle r_1 \rangle, \langle r_2 \rangle, \langle r_1 \cdot r_2 \rangle)$, especially since each multiplication requires a different triplet. De Cock et al. (2017) suggest to use a trusted dealer, who could easily generate the triplets and distribute the shares to $\mathcal{A}$ and $\mathcal{B}$. In our setting such a dealer is not available. However, it is possible for $\mathcal{A}$ and $\mathcal{B}$ to jointly precompute the triplets. Protocol 4 computes $\log_2 N$ sharings of the product of two (random) bits, with each bit held by a different party. It uses additively homomorphic encryption and packing.

Protocol 4 Secure multiplication of private bits

Party	$\mathcal{A}$	$\mathcal{B}$
Input	$r_i^{\mathcal{A}},\ 0 \le i < \log_2 N$	$r_i^{\mathcal{B}},\ 0 \le i < \log_2 N$ and K
Output	$\rho_i^{\mathcal{A}},\ 0 \le i < \log_2 N$	$\rho_i^{\mathcal{B}},\ 0 \le i < \log_2 N$
Constraints	$\rho_i^{\mathcal{A}} \oplus \rho_i^{\mathcal{B}} = r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}},\ 0 \le i < \log_2 N$	

1 Party $\mathcal{B}$ encrypts its bits $r_i^{\mathcal{B}}$ separately, with Paillier, and sends them to $\mathcal{A}$.

2 Party $\mathcal{A}$ multiplies, for each i, $r_i^{\mathcal{B}}$ with $r_i^{\mathcal{A}}$:
if $r_i^{\mathcal{A}} = 0$, $[r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}}] = [0]$, else $[r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}}] = [r_i^{\mathcal{B}}]$.

3 For each i, party $\mathcal{A}$ generates a random bit $\rho_i^{\mathcal{A}}$, and computes $[c_i] = [(r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}}) \oplus \rho_i^{\mathcal{A}}]$:
if $\rho_i^{\mathcal{A}} = 0$, $[c_i] = [r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}}]$, else $[c_i] = [1] \cdot [r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}}]^{-1}$.

4 Party $\mathcal{A}$ packs all bits $[c_i]$ into one encryption $[C] = [(c_0, c_1, \ldots, c_{(\log_2 N)-1})_P]$, with compartment size $\nu = 2$, and sends $[C]$ to party $\mathcal{B}$.

5 Party $\mathcal{B}$ decrypts C, and unpacks it to $\rho_i^{\mathcal{B}} = c_i$, $0 \le i < \log_2 N$.

At the end, the parties have $\log_2 N$ sharings $\langle \rho_i \rangle$, such that $\rho_i = \rho_i^{\mathcal{A}} \oplus \rho_i^{\mathcal{B}} = r_i^{\mathcal{A}} \cdot r_i^{\mathcal{B}}$. This building block can be used to compute $\log_2 N$ multiplication triplets, by using the following protocol.

1 Party $\mathcal{A}$ generates $\log_2 N$ pairs of random bits $r_{1i}^{\mathcal{A}}$ and $r_{2i}^{\mathcal{A}}$, and similarly for party $\mathcal{B}$.

2 The parties use Protocol 4 twice. First to compute $\log_2 N$ sharings $\langle r_{1i}^{\mathcal{A}} \cdot r_{2i}^{\mathcal{B}} \rangle$, and second to compute $\log_2 N$ sharings $\langle r_{2i}^{\mathcal{A}} \cdot r_{1i}^{\mathcal{B}} \rangle$.

3 The parties compute, for each i, $\langle r_{1i} \cdot r_{2i} \rangle = (r_{1i}^{\mathcal{A}} \cdot r_{2i}^{\mathcal{A}}) \oplus \langle r_{1i}^{\mathcal{A}} \cdot r_{2i}^{\mathcal{B}} \rangle \oplus \langle r_{2i}^{\mathcal{A}} \cdot r_{1i}^{\mathcal{B}} \rangle \oplus (r_{1i}^{\mathcal{B}} \cdot r_{2i}^{\mathcal{B}})$.

The products $r_{1i}^{\mathcal{A}} \cdot r_{2i}^{\mathcal{A}}$ can be computed locally by party $\mathcal{A}$, and similarly $r_{1i}^{\mathcal{B}} \cdot r_{2i}^{\mathcal{B}}$ by party $\mathcal{B}$. This protocol generates $\log_2 N$ multiplication triplets, which can be used to securely multiply $\log_2 N$ sharings. It requires an average total of 8.75 $\log_2 N$ multiplications of Paillier encryptions.

Since each shared secret is binary, these triplets can also be generated by oblivious transfers. Suppose party $\mathcal{A}$ constructs $x_0 = (r^{\mathcal{A}} \cdot 0) \oplus \rho^{\mathcal{A}}$ and $x_1 = (r^{\mathcal{A}} \cdot 1) \oplus \rho^{\mathcal{A}}$, then obliviously transferring $\rho^{\mathcal{B}} = x_{r^{\mathcal{B}}}$ to party $\mathcal{B}$ generates the same sharing $\langle \rho \rangle$ as above. A recent result for extending oblivious transfers by Kolesnikov and Kumaresan (2013) can be used to efficiently generate multiplication triplets. Although the method described above fits nicely into our framework, extended oblivious transfers will sincerely reduce the precomputation effort, because less public key cryptography is involved. They require 130 bits of communication for the oblivious transfer of one bit (Kolesnikov and Kumaresan, 2013). We need to send one

Paillier encryption of 2,048 bits (and one additional bit) to achieve the same transfer, which suggests an improvement by roughly a factor 16.

4.3 Secure comparison of private integers

De Cock et al. (2017) translated the DGK comparison protocol to this secret sharing setting. We slightly improve their secure comparison protocol.

To obtain sharings of the bits of privately held integers x and y, we simply define $x_i^{\mathcal{A}} = x_{(i)}$, $x_i^{\mathcal{B}} = 0$, and similarly $y_i^{\mathcal{A}} = 0$, $y_i^{\mathcal{B}} = y_{(i)}$, $0 \le i < \ell$.

1 Let $\langle x_{(i)} \rangle$ and $\langle y_{(i)} \rangle$, $0 \le i < \ell$, be secret sharings of the bits of integers x and y, $0 \le x, y < 2^\ell$.

2 For each i, $0 \le i < \ell$, parties $\mathcal{A}$ and $\mathcal{B}$ derive sharings of the bits $d_i = (x_{(i)} < y_{(i)})$, by computing in parallel $\langle d_i \rangle = \langle y_{(i)} \rangle \cdot (1 \oplus \langle x_{(i)} \rangle)$, requiring ℓ multiplications of sharings in total.

3 They locally compute sharings of $e_i = (x_{(i)} = y_{(i)})$ by $\langle e_i \rangle = \langle x_{(i)} \rangle \oplus \langle y_{(i)} \rangle \oplus 1$, $1 \le i < \ell$.

4 For each i, $0 \le i < \ell$, parties $\mathcal{A}$ and $\mathcal{B}$ compute $\langle c_i \rangle = \langle d_i \rangle \prod_{j=i+1}^{\ell-1} \langle e_j \rangle$.

5 They locally compute $\langle \delta \rangle = 1 \oplus \sum_{i=0}^{\ell-1} \langle c_i \rangle$.

It can be shown that the shared bit δ is the outcome of the comparison $(x \le y)$ (De Cock et al., 2017). To minimise the number of multiplications of shares, we suggest to combine the computation of c_i and δ to $\delta = 1 \oplus e_\ell \cdot (d_{\ell-1} \oplus e_{\ell-1} \cdot (d_{\ell-2} \oplus \ldots \oplus e_2 \cdot (d_1 \oplus e_1 \cdot d_0) \ldots))$. In this way, the computational and communication complexity for comparing two privately held ℓ-bit integers comes down to a total of 2ℓ multiplications of shares. Since only binary sharings are used, the total amount of communication and computation is very small, even negligible when comparing it to the complexity of computing with large encryptions. The number of communication rounds is $\ell + 1$, but could be reduced to $1 + \log_2 \ell$ (Veugen, 2018).

When precomputing the 2ℓ multiplications triplets by means of packed encryptions, the precomputation costs for comparing two privately held integers is 8.75ℓ multiplications of Paillier encryptions, for inputs consisting of ℓ bits. This is much lower than the average of $7.5 + 151.5\ell$ multiplications needed for standard DGK comparison (Veugen, 2012).

5 Secure comparison

The main advantage of securely comparing batched, or packed, encrypted integers is that we can additively blind all integers at once, and party $\mathcal{B}$ only needs one decryption. The output of the batched (or packed) secure comparison protocol will be n, separately encrypted bits $[\gamma_i]$, $1 \le i \le n$. In order to securely compare two batches or packings of encrypted integers, we need a protocol for securely comparing two privately held integers, as described in the previous section.

Although the size of each compartment may vary, for simplicity we assume each encrypted integer has at most ℓ bits. As shown in Subsection 5.3, bounding the number of bits also allows us to reduce the precomputation costs. Since we need a little extra room to compute the comparison, we have $2^{\ell+1} \le m_i$, in case of batching, and $2^{\ell+2} \le \nu$, in case of packing.

5.1 Secure comparison of batched values

Instead of adding, or multiplying, batched values, it is also possible to securely compare batched values. Suppose we have an encrypted batch $[X]$, containing (encrypted) integers x_i, $1 \le i \le n$, a similar batch $[Y]$, and we would like to securely compute the encrypted bit $\gamma_i = (x_i < y_i)$, for all i. Assume $0 \le x_i, y_i < 2^\ell$, and $2^{\ell+1} \le m_i$, then $\gamma_i = (z_i < 2^\ell)$, where $z_i = x_i - y_i + 2^\ell$ has at most $\ell + 1$ bits. Protocol 5 computes the comparison bits γ_i.

Protocol 5 Secure comparison of batched values

party	$\mathcal{A}$	$\mathcal{B}$
Input	$[X]$ and $[Y]$	K
Output	$[\gamma_i]$, $1 \le i \le n$	
Constraints	$X = (x_1, x_2, \ldots, x_n)_B$ and $Y = (y_1, y_2, \ldots, y_n)_B$, $0 \le X, Y < N \cdot 2^{-(\kappa+1)}$, $0 \le x_i, y_i < 2^\ell \le m_i/2$, $1 \le i \le n$, $\gamma_i = (x_i < y_i)$, $1 \le i \le n$.	

1 Party $\mathcal{A}$ computes the encrypted batch $[Z]$, such that $z_i = x_i - y_i + 2^\ell$, for each i: $[Z] = [X] \cdot [Y]^{-1} \cdot [2^\ell]$.

2 Party $\mathcal{A}$ computes a large random number R, and additively blinds Z with it: $[\theta] = [Z] \cdot [R]$.

3 Party $\mathcal{A}$ sends $[\theta] = [Z + R]$ to $\mathcal{B}$.

4 Party $\mathcal{B}$ decrypts it, and 'debatches' θ to $\theta_i = \bmod m_i$, $1 \le i \le n$.

5 Parties $\mathcal{A}$ and $\mathcal{B}$ jointly compute the comparison bits $[\delta_i]$, $[\delta_i']$, and $[\delta_i'']$, for each i, $1 \le i \le n$, where $\delta_i = (\theta_i < (R \bmod m_i))$, δ_i' $(\theta_i + m_i < (R \bmod m_i) + 2^\ell)$, and $\delta_i'' = (\theta_i < (R \bmod m_i) + 2^\ell)$, by executing in parallel $3n$ secure comparison protocols on private inputs.

6 Using $[\delta_i]$, $[\delta_i']$, and $[\delta_i'']$, party $\mathcal{A}$ computes $[\gamma_i] = [\delta_i \cdot \delta_i' + (1 - \delta_i) \cdot \delta_i''] = [\delta_i' + \delta_i'' - \delta_i] = [\delta_i'] \cdot [\delta_i''] \cdot [\delta_i]^{-1}$. This is done for each i.

In step 1, we have the relation $z_i = (X - Y + 2^\ell) \bmod m_i$, even if $X - Y + 2^\ell < 0$, and since R always exceeds $| X - Y + 2^\ell|$, there is no reduction modulo N in the computation of θ in step 3.

In step 5, we have $(\theta - R) \bmod m_i = z_i$, so either $z_i = \theta_i - (R \bmod m_i)$, or $z_i = \theta_i - (R \bmod m_i) + m_i$. Just as in secure debatching, $[\delta_i]$ is computed to find out whether there was an overflow in the i^{th} compartment. If $\delta_i = 0$, then $\theta_i \ge R \bmod m_i$, so $z_i = \theta_i - (R \bmod m_i)$, and $\gamma_i = \delta_i''$. Otherwise, there was an overflow in the i^{th} compartment, and $\gamma_i = \delta_i'$. By construction of the δ_i's, we have $0 \le \delta_i' \le \delta_i \le \delta_i'' \le 1$, which explains the correctness of computing $[\gamma_i]$ in the final step.

For the three secure comparison protocols from step 5, we use the comparison protocol based on sharings modulo two. This means that the secret-shared outputs of these comparison protocols have to be translated to encrypted bits. This can be done in a fairly straightforward way. Party $\mathcal{B}$ encrypts his share $\delta^{\mathcal{B}}$, and sends the encryption to party $\mathcal{A}$, who computes $[\delta^{\mathcal{A}} \oplus \delta^{\mathcal{B}}]$, by flipping the encrypted bit in case $\delta^{\mathcal{A}} = 1$.

5.2 Secure comparison of packed values

Suppose we have $[X] = [(x_1, x_2, \ldots, x_n)_P]$, $[Y] = [(y_1, y_2, \ldots, y_n)_P]$, and we would like to compute $[\gamma_i] = [(x_i < y_i)]$, $1 \le i \le n$. Just as in the previous batched case, assume $0 \le x_i, y_i < 2^\ell$.

Protocol 6	Secure comparison of packed values	
Party	$\mathcal{A}$	$\mathcal{B}$
Input	$[X]$ and $[Y]$	K
Output	$[\gamma_i]$, $1 \le i \le n$	
Constraints	$X = (x_1, x_2, \ldots, x_n)_P$ and $Y = (y_1; y_2; \ldots, y_n)_P$, $0 \le X, Y < N \cdot 2^{-(\kappa+1)}$, $0 \le x_i, y_i < 2^\ell \le \nu/4$, $1 \le i \le n$, $\gamma_i = (x_i < y_i)$, $1 \le i \le n$.	

1. Party $\mathcal{A}$ computes the encrypted packing $[Z]$, such that $z_i = x_i - y_i + 2^\ell$, for each i: $[Z] = [X] \cdot [Y]^{-1} \cdot [(2^\ell, 2^\ell, \ldots, 2^\ell)_P]$.
2. Party $\mathcal{A}$ computes a large random number R, and additively blinds $2Z$ with it: $[\theta] = [Z]^2 \cdot [R]$.
3. Party $\mathcal{A}$ sends $[\theta] = [2Z + R]$ to $\mathcal{B}$.
4. Party $\mathcal{B}$ decrypts it, and unpacks $\theta \bmod \nu^n$ to θ_i, $1 \le i \le n$.
5. Party $\mathcal{A}$ unpacks $R \bmod \nu^n$ to R_i, $1 \le i \le n$.
6. Parties $\mathcal{A}$ and $\mathcal{B}$ jointly compute the comparison bits $[\delta_i]$, $[\delta_i']$, and $[\delta_i'']$, for each i, $1 \le i \le n$, where $\delta_i = (\theta_i < R_i)$, $\delta_i' = (\theta_i + \nu < R_i + 2^{\ell+1})$, and $\delta_i'' = (\theta_i < R_i + 2^{\ell+1})$, by executing in parallel $3n$ secure comparison protocols on private inputs.
7. Using $[\delta_i]$, $[\delta_i']$, and $[\delta_i'']$, party $\mathcal{A}$ computes $[\gamma_i] = [\delta_i \cdot \delta_i' + (1 - \delta_i) \cdot \delta_i''] = [\delta_i' + \delta_i'' - \delta_i]$. This is done for each i.

Our solution will be similar to the previous batched case. We additively blind Z with a large random number R, and ask $\mathcal{B}$ to unpack $\theta = Z + R$. The addition of R might cause an overflow in some compartments. The difference with batching is that these overflows create a carry-over, which influences the content of the next compartment: $\theta_i = (z_i + R_i + c_{i-1}) \bmod \nu$, where c_{i-1} is the carry-over bit from the previous compartment. This can be solved by not computing $\gamma_i = (z_i < 2^\ell)$, but $\gamma_i = (2z_i + c_{i-1} < 2^{\ell+1})$. Since adding one (carry-over) bit to the even number $2z_i$ will not change the outcome γ_i of the comparison, we have a solution, which is redundant to possible carry-covers. We only need to increase the compartment size with one bit: $2^{\ell+2} \le \nu$.

The entire protocol for securely comparing n pairs from two packed encrypted integers is Protocol 6.

In step 6, we have $\theta_i = (2z_i + c_{i-1} + R_i) \bmod \nu$, so either $2z_i + c_{i-1} = \theta_i - R_i$, or $2z_i + c_{i-1} = \theta_i - R_i + \nu$. Just as in secure unpacking, $[\delta_i]$ is computed to find out whether there was an overflow in the i^{th} compartment. If $\delta_i = 0$, then $\theta_i \ge R_i$, so $2z_i + c_{i-1} = \theta_i - R_i$, and $\gamma_i = \delta_i'$. Otherwise, there was an overflow in the i^{th} compartment, and $\gamma_i = \delta_i'$. By construction of the δ_i's, we have $0 \le \delta_i' \le \delta_i \le \delta_i'' \le 1$, which explains the correctness of computing $[\gamma_i]$ in the final step.

5.3 Combining the three comparisons of private integers

In the batched secure comparison protocol, we use, for each i, three subprotocols for securely comparing two privately held integers. More precisely, we compute $\delta_i = (\theta_i < (R \bmod m_i))$, $\delta_i' = (\theta_i + m_i < (R \bmod m_i) + 2^\ell)$, and $\delta_i'' = (\theta_i < (R \bmod m_i) + 2^\ell)$. The computations of δ_i and δ_i'' can be combined, by first computing $\epsilon_i = (\theta_i \bmod 2^\ell < (\mathrm{R} \bmod m_i) \bmod 2^\ell)$, and extending the result to both δ_i and δ_i''. This is shown in Appendix B.

In the packed secure comparison protocol, we need to compute $\delta_i = (\delta_i < R_i)$, $\delta_i' = (\theta_i + \nu < R_i + 2^{\ell+1})$, and $\delta_i'' = (\theta_i < R_i + 2^{\ell+1})$. We can similarly combine the computation of δ_i and δ_i''. But since we can choose $\nu = 2^{\ell+2}$, this also holds for δ_i and δ_i'. Consequently, instead of $3 \cdot 2(\ell + 2)$, we only need $2\ell + 3 \cdot 4$ multiplications of sharings, which reduces the precomputation costs.

5.4 Secure equality

Just as secure comparison, one can perform secure equality with packed, and batched, encryptions. The protocols for equality or comparison are nearly identical, expect for the subprotocols on private inputs. In case of batching, we batch $z_i = (x_i - y_i) \bmod m_i$, where $0 \le x_i, y_i < m_i$, and compute $\delta_i = (\theta_i = (R \bmod m_i))$. Since $(\theta - R) \bmod m_i = z_i$, we have $\delta_i = (z_i = 0)$.

In case of packing, we pack $2z_i$, where $z_i = x_i - y_i + 2^\ell$, and choose $\nu = 2^{\ell+2}$. When $\mathcal{B}$ unpacks $\theta \bmod n$, it has $\theta_i = (2z_i + c_{i-1} + R_i) \bmod \nu$. Since $0 < z_i < 2^{\ell+1}$ and $\nu = 2^{\ell+2}$, we have $z_i = 2^\ell$, if and only if, $(\theta_i - R_i) \bmod 2^{\ell+1} = c_{i-1}$. Therefore, it is sufficient to compute $\delta_i = (\theta_i \bmod 2^{\ell+1} = R_i \bmod 2^{\ell+1})$ and $\delta_i' = (\theta_i \bmod 2^{\ell+1} = (R_i + 1) \bmod 2^{\ell+1})$, to obtain $\gamma_i = \delta_i + \delta_i' = (x_i = y_i)$.

The subprotocol, securely computing a sharing of $\delta = (x = y)$, given privately held integers x and y, is similar to the protocol for securely comparing them. In fact, it is sufficient to locally compute sharings $\langle e_i \rangle$, for each bit i of the inputs, where $e_i = (x_{(i)} = y_{(i)})$, just as in the comparison protocol, and securely multiply them to $\langle \delta \rangle \prod_i \langle e_i \rangle$, requiring $\ell - 1$ multiplications of sharings.

6 Security

In our framework, we combine two different techniques, each with their own security properties, and assume a semi-honest security model. The first technique is additively homomorphic encryption, which we use in the standard two-party setting, party $\mathcal{B}$ having the decryption key. By additively blinding all values sent to $\mathcal{B}$, using a random integer that has κ more bits, we achieve statistical security towards $\mathcal{B}$ (Veugen, 2014). The security towards party $\mathcal{A}$ is guaranteed by the semantic security of the homomorphic crypto system, leading to computational security (Veugen, 2014).

The second technique is (bit-wise) secret sharing, which by itself is unconditionally secure (De Cock et al., 2017). However, this is only true in case of a trusted dealer, who computes and distributes the multiplication triplets to both parties. When generating the multiplication triplets by additively homomorphic encryption, as described in Subsection 4.2, we achieve the same security properties as described above.

Therefore, our framework achieves statistical security towards $\mathcal{B}$, and computational security towards $\mathcal{A}$.

We give a formal security proof for Protocol 1, the other security proofs are analogous. We closely follow Goldreich's (2001) notation so $\mathcal{A}$'s input is $\overline{x}$, $\mathcal{B}$'s input is $\overline{y}$, and the output $f(\overline{x},\overline{y})$ equals the pair $(f_1(\overline{x},\overline{y}),$ $f_2(\overline{x},\overline{y}))$, where f_1 denotes $\mathcal{A}$'s output function and f_2 $\mathcal{B}$'s output function. The following definition precisely states what we have to prove, which loosely speaking comes down to "whatever can be computed by $\mathcal{A}$ or $\mathcal{B}$ from their view of a protocol execution, can be computed from their input and output".

Definition of privacy with respect to semi-honest behaviour [Goldreich, (2001), Definition 7.2.1]: Let Π be a two-party protocol for computing f. The view of party $\mathcal{A}$ (resp., $\mathcal{B}$) during an execution of Π on $(\overline{x},\overline{y})$, denoted $V_1^{\pi}(\overline{x},\overline{y})$ (resp., $V_2^{\pi}(\overline{x},\overline{y})$), is $(\overline{x}, r, m_1, \dots, m_t)$ (resp., $(\overline{y}, r, m_1, \dots, m_t)$), where r represents the outcome of party $\mathcal{A}$'s (resp., party $\mathcal{B}$'s) internal coin tosses, and m_i represents the ith message it has received.

For a deterministic functionality f, we say that Π privately computes f if there exist probabilistic polynomial-time algorithms, denoted S_1 and S_2, such that

$$\left\{S_1\left(\overline{x}, f_1\left(\overline{x},\overline{y}\right)\right)\right\}_{\overline{x},\overline{y}\in\{0,1\}^*} \overset{c}{\equiv} \left\{V_1^{\pi}\left(\overline{x},\overline{y}\right)\right\}_{\overline{x},\overline{y}\in\{0,1\}^*}, \quad \text{and}$$

$$\left\{S_2\left(\overline{x}, f_2\left(\overline{x},\overline{y}\right)\right)\right\}_{\overline{x},\overline{y}\in\{0,1\}^*} \overset{c}{\equiv} \left\{V_2^{\pi}\left(\overline{x},\overline{y}\right)\right\}_{\overline{x},\overline{y}\in\{0,1\}^*},$$

where $\overset{c}{\equiv}$ denotes computational indistinguishability.

Theorem: Assume the additively homomorphic cryptosystem denoted by [.] is semantically secure, and assume the secure comparison protocol used in step 4 privately computes the encrypted comparison result of both private inputs.

Then on inputs $\overline{x} = ([X])$ and $\overline{y} = (K)$, Protocol 1 privately computes the output $f(\overline{x},\overline{y})$ = $(([x_1], \dots, [x_n]), \perp)$.

Proof: Since we use the comparison protocol as a building block of f, we can present it as an oracle in our proofs and use Goldreich's Composition Theorem (7.3.3) (2001). The only assumption we made about the private comparison protocol is that the comparison result is privately computed, which fulfils Goldreich's premise for applying the composition theorem.

In Protocol 1, the view of $\mathcal{A}$ consists of its private number $[X]$, its random number R (of $\log_2 N - 1$ bits), its output $([x_1], \dots, [x_n])$, and all intermediate messages received from $\mathcal{B}$: the encrypted batches $[\theta_i]$, $1 \le i \le n$, and the encrypted comparison bits $[\delta_i]$, $1 \le i \le n$. Summarising, the view of $\mathcal{A}$ equals

$$V_1 = \left([X], R, [x_1], \dots, [x_n], [\theta_1], \dots, [\theta_n], [\delta_1], \dots, [\delta_n]\right).$$

According to Definition 7.2.1 (Goldreich, 2001), it suffices to show that there exists a probabilistic polynomial-time algorithm S_1 such that $S_1(\overline{x}, f_1, (\overline{x},\overline{y}))$ is computationally indistinguishable from V_1. Since the encryption algorithm is semantically secure, every pair of encryptions is computationally indistinguishable (Goldreich, 2001), so by letting S_1 randomly generate $3n$ encrypted integers $[x_i]$, $[\theta_i]$, $[\delta_i]$, $1 \le i \le n$, and a random number r_R of $\log_2 N - 1$ bits, this condition is easily verified.

The view of $\mathcal{B}$ consists of the decryption key K, and all intermediate messages received from $\mathcal{A}$: the encrypted number $[\theta]$, where $\theta = X + R$. Since $\mathcal{B}$ owns the decryption key, $[\theta]$ can be decrypted to θ. Summarising, the view of $\mathcal{B}$ is equivalent to

$$V_2 = (K, \theta).$$

Again, we have to show that there exists a probabilistic polynomial-time algorithm S_2 such that $S_2(\overline{y}, f_2(\overline{x},\overline{y}))$ is computationally indistinguishable from V_2. This is easily satisfied by letting S_2 randomly generate an integer r_θ of $\log_2 N - 1$ bits. For any value of r_θ, we have

$$\begin{aligned}\Pr(\theta = r_\theta) &= \sum\nolimits_{x=0}^{r_\theta} \Pr(X = x)\Pr(R = r_\theta - x) \\ &= 2^{-(\log_2 N - 1)} \Pr(X \le r_\theta).\end{aligned}$$

Let $X^{\max}$ be the maximal value of X, then $X^{\max} \cdot 2^{\kappa} < N$. It follows that

$$\begin{aligned}\sum\nolimits_{r_\theta} \left|\Pr(r_\theta) - \Pr(\theta)\right| &= \sum\nolimits_{r_\theta} 2^{-(\log_2 N - 1)}\left(1 - \Pr\left(X \le r_\theta\right)\right) \\ &= 2^{-(\log_2 N - 1)} \sum\nolimits_{r_\theta} \Pr\left(X > r_\theta\right) \\ &= 2^{-(\log_2 N - 1)} \sum\nolimits_{r_\theta} \Pr\left(X = r_\theta\right) \cdot r_\theta \\ &= 2^{-(\log_2 N - 1)} E(X) \\ &\le 2^{-(\log_2 N - 1)} X^{\max} < 2^{1-\kappa},\end{aligned}$$

which decreases faster than the reciprocal of any polynomial for sufficiently large security parameter κ, so θ and r_θ are statistically indistinguishable, and thus also computationally indistinguishable (Goldreich, 2001). □

We conclude that Protocol 1 privately computes $\mathcal{A}$'s output ($[x_1]$, …, $[x_n]$) in the semi-honest model. In fact, we showed that the integer X is even statistically secure towards $\mathcal{B}$. Whether this holds for the entire protocol will depend on the chosen comparison protocol.

7 Applications of batching

The applications of packing (Bianchi et al., 2009; Erkin et al., 2012) have been illustrated by others. We have already shown how to perform multiple secure comparisons in parallel using both packing and batching. Since batching with additively homomorphic encryption is less well known, we illustrate a few other possibilities.

As shown in Protocol 2, batching can be used to perform n secure multiplications by means of only one secure multiplication of two batched integers. The principle of parallel secure multiplications could be extended to secure inner products, and even matrix-vector multiplications. For small matrices, it is possible to first batch each column, and second batch all columns into one matrix batch. Then one secure multiplication with a batched vector performs a secure matrix-vector multiplication. A different application of batching is secure set intersection.

7.1 Set intersection

Securely computing the intersection of two sets is needed for several applications, for example when checking which users are on a revocation list or not. In secure set intersection, party $\mathcal{A}$ holds k integers $x_1, x_2, \ldots, x_k$, party $\mathcal{B}$ privately holds n integers $y_1, y_2, \ldots, y_n$, and both parties want to jointly compute the elements of their intersection, without leaking information on the remaining elements.

Many solutions are known to solve secure set intersection, for example by using fully homomorphic encryption (Chen et al., 2017), or somewhat homomorphic encryption (Saha and Koshiba, 2016). Since our setting is based on additively homomorphic encryption, we give an efficient solution based on earlier work by Freedman et al. (2004). We follow their solution, and show how batching (and packing) can be used to improve its efficiency. We assume p is a public prime number, (slightly) larger than the inputs x_j of party $\mathcal{A}$. Protocol 7 shows our solution.

The ith element of the batched integer Y equals $\sum_{j=0}^{k}(\alpha_j \bmod p)\cdot(y_i^j \bmod p)$, which equals $p(y_i)$ modulo p. Since $p(y_i) \bmod p = 0$, if and only if, $p(y_i) = 0$, the parties compute $\gamma_i = ((Y \bmod m_i) \bmod p = 0)$, which is one, if and only if, y_i is in the intersection. If $\delta_i = 0$, then there was no overflow in the ith compartment, and $Y \bmod m_i = \theta_i - (R \bmod m_i)$, and otherwise, $Y \bmod m_i = \theta_i - (R \bmod m_i) + m_i$. In the first case $\gamma_i = \delta_i'$, otherwise $\gamma_i = \delta_i''$. This shows the correctness of the computation of the bit γ_i.

Protocol 7 Secure set intersection

Party	$\mathcal{A}$	$\mathcal{B}$
Input	$x_1, x_2, \ldots, x_k$ and p	$y_1, y_2, \ldots, y_n, p$ and K
Output		$\{x_1, x_2, \ldots, x_k\} \cap \{y_1, y_2, \ldots, y_n\}$
Constraints	$0 \le x_i < p,\ 1 \le i \le k$, p is prime, and $(k \cdot p^2)^n < N \cdot 2^{-\kappa}$	

1. Party $\mathcal{B}$ computes the k batched integers $Y^j = (y_1^j \bmod p, y_2^j \bmod p, \ldots, y_n^j \bmod p)_B$, $1 \le j \le k$, encrypts them, and sends them to $\mathcal{A}$.
2. Party $\mathcal{A}$ computes $k + 1$ integers α_j, possibly negative, such that $p(x) = \prod_{j=1}^{k}(x - x_j) = \sum_{j=0}^{k}\alpha_j x_j$. Clearly, $p(y) = 0$, if and only if, y is in the set $\{x_1, x_2, \ldots, x_k\}$.
3. Party $\mathcal{A}$ computes $[Y] \approx [(p(y_1), p(y_2), \ldots, p(y_n))_B]$: $[Y] = [\alpha_0 \bmod p] \cdot \prod_{j=1}^{k}[Y^j]^{\alpha_j \bmod p}$.
4. Party $\mathcal{A}$ additively blinds Y, by computing a large random number R: $[\theta] = [Y] \cdot [R]$, and sends $[\theta]$ to $\mathcal{B}$.
5. Party $\mathcal{B}$ decrypts θ, and 'debatches' it to $\theta_i = \theta \bmod m_i$, $1 \le i \le n$.
6. Parties $\mathcal{A}$ and $\mathcal{B}$ run n secure comparison protocols, and $2n$ secure equality protocols in parallel, to compute sharings $\langle\delta_i\rangle$, $\langle\delta_i'\rangle$ and $\langle\delta_i''\rangle$ of $\delta_i = (\theta_i < R \bmod m_i)$, $\delta_i' = ((\theta \bmod m_i) \bmod p = (R \bmod m_i) \bmod p)$, and $\delta_i'' = ((\theta \bmod m_i + m_i) \bmod p = (R \bmod m_i) \bmod p)$.
7. The parties compute $\langle\gamma_i\rangle = \langle\delta_i'\rangle + \langle\delta_i\rangle \cdot (\langle\delta_i''\rangle - \langle\delta_i'\rangle)$, and open the sharings of γ_i, $1 \le i \le n$, to party $\mathcal{B}$, who learns the elements of the set intersection.

In Freedman et al. (2004), the encryptions $[p(y_i)]$ are computed one-by-one, and each $[p(y_i) \cdot r_i + y_i]$ is sent to party $\mathcal{B}$. The multiplicative blinding with random number r_i is expensive, and so are the n Paillier decryptions needed by $\mathcal{B}$, which we reduce to one decryption of θ, by means of batching. Since we computed the shared bits γ_i, it is also easy to securely compute the cardinality of the set intersection: $\sum_i \gamma_i$.

The set intersection will be computed correctly and securely, as long as $m_i > p - 1 + k(p - 1)^2$, and $M < N \cdot 2^{-\kappa}$, so $n\sigma < \log_2 N - \kappa$, where compartment size $\sigma = \log_2 k + 2\log_2 p$. This means that for integers x_j of bit size $\ell = 10$ (and $\log_2 p = 11$), for example, we can compute the set intersection up to $k = n = 64$ elements, using only one encrypted batch. For larger sets we need to split Y into several batched encryptions, each computing a part of the n bits γ_i.

8 Performance

In order to determine the computational complexity, we compute the average number of multiplications of encryptions. To estimate the improvement of our solution, we compare the complexity of our solution with the state-of-the-art. First for computing secure multiplications, then for doing secure comparisons and secure equalities, and finally for securely computing set intersections.

Although decoding is preferably done in the plain domain, or only once at the end of all computations, we also present the performance of including decoding in the encrypted domain. In our framework, an important part of the bit-wise operations is generating the multiplication triplets. Since the triplets can be precomputed by additively homomorphic encryption, we mention this effort separately. Especially when extended oblivious transfers are used for precomputing the triplets, as described in Subsection 4.2, the precomputation effort seems to be negligible compared to the remaining computations.

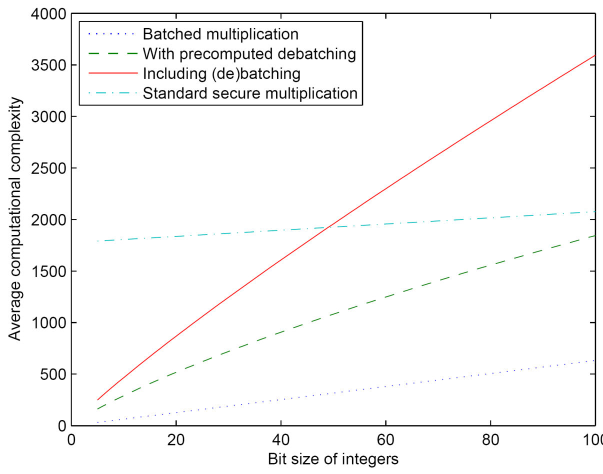
8.1 *Secure multiplication*

We compare one secure multiplication with the batched approach, and compute the average number of multiplications of encryptions to measure the computational complexity per secure multiplication. The number n of compartments is determined by $2n \cdot \sigma = \log_2 N - \kappa$, where compartment size $\sigma = 2\ell$.

For one standard secure multiplication protocol, we roughly need two decryptions and two exponentiations with a power of $\sigma + \kappa$ bits. A Paillier decryption takes on average $\frac{3}{8}\log_2 N$ multiplications, so we need $2(\sigma + \kappa)1.5 + \frac{3}{4}\log_2 N$ multiplications in total.

In the batched approach, we first batch the $2n$ encryptions into two encryptions, then securely multiply them, and finally unbatch the multiplication result. This would take $2(1 + \log_2 n) \cdot 0.8 \cdot \sigma n + (\frac{3}{4}\log_2 N + 1.6n \cdot \sigma) + (\frac{3}{8}\log_2 N + n \cdot 8.75 \cdot \sigma)$ multiplications.

Figure 1 Average computational complexity per secure multiplication (see online version for colours)



In Figure 1, the light blue line '-.' indicates the performance of the standard protocol, the dark blue line '..' the average costs per compartment when multiplying two encrypted batches, and the red line '-' the batched multiplication, including the costs for batching and debatching. We let ℓ increase from 5 to 100, and set n accordingly to get a full batch.

For $\ell = 5$, a batched multiplication reduces the average costs of a secure multiplication from 1,791 to 33 multiplications of encrypted integers. Even when including the costs for batching and debatching, the average costs are still smaller when $\ell < 50$.

Table 3 Number of communicated encryptions for n secure multiplications

	Standard	*Batch*	*Batching*	*Debatching*	*Computing triplets for debatching*
Encryptions sent	$3n$	3	0	$n + 1$	$4\sigma n$

We also compare the number of communicated encryptions, see Table 3. During a standard secure multiplication, three encryptions need to be sent. Two by party $\mathcal{A}$, and one by party $\mathcal{B}$. This is the same for a secure multiplication of two encrypted batches, which combines n secure multiplications (excluding the two additional encryptions needed for reducing the size of the product). For batching n encrypted integers into one encryption, no communication is needed. Furthermore, for debatching we need $n + 1$ encryptions: $\mathcal{A}$ sends the batched encryption to $\mathcal{B}$, and gets n encryptions back. And finally, we need to precompute the multiplication triplets for debatching, which we assume is done with packing and additively homomorphic encryption. There are $n \cdot 2\sigma$ triplets needed, and for each triplet two encryptions are sent from $\mathcal{B}$ to $\mathcal{A}$. Therefore, the batched multiplication reduces the communication costs with a factor n. In a setting where batching is used for only one batched multiplication, the reduction factor is almost 3, excluding the effort for precomputing multiplication triplets.

8.2 *Secure comparison*

A well-known solution for securely comparing two encrypted integers is the DGK protocol. In this secure comparison algorithm it is common to intermediately switch to the DGK crypto system, which is fine-tuned to small plain texts. Since the size of a cipher text in DGK is N instead of N^2, we count four multiplications of DGK-encrypted numbers as one multiplication of Paillier encryptions. Then the average computational complexity of DGK is comparable to $7.5 + \frac{3}{8}\log_2 N + 151.5\ell$ multiplications of integers of size N^2 (Veugen, 2012). We will compare n secure DGK comparisons with our packed approach, by counting the number of multiplications of integers of size N^2.

To securely compare n pairs of encrypted integers through packing, we follow the solution described in Subsection 5.2, which consists of:

1 party $\mathcal{A}$ computing the encrypted packings $[X]$, $[Y]$, and $[Z]$

2 blinding Z to get $[\theta]$

3 party $\mathcal{B}$ decrypting $[\theta]$ and unpacking it

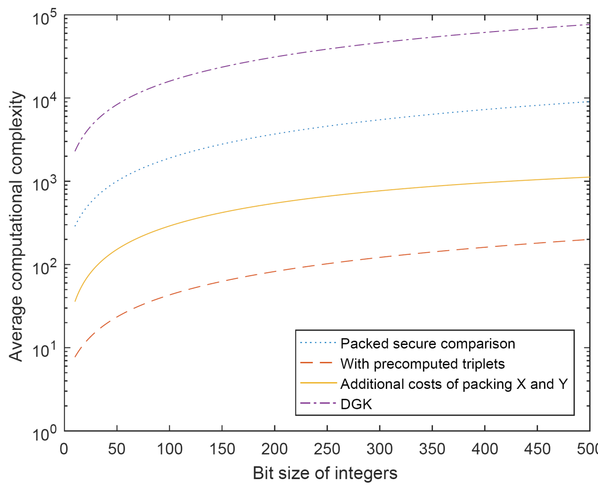
4 performing n secure comparisons of privately held integers, as described in Subsection 5.3

5 obtaining the $[\gamma_i]$ from the three $[\delta_i]$'s.

We choose the compartment size $\nu = 2^{\ell+2}$, and maximise n, such that $n \cdot (\ell + 2) < \log_2 N - \kappa$. The packed secure comparisons will take (1) $3 + 2(n - 1) \cdot 1.5 \log_2 (\ell + 2)$ + (2) 2 + (3) $\frac{3}{8}\log_2 N$ + (4) $n(2\ell + 12)8.75$ + (5) $3n$ multiplications.

The performance of securely comparing n pairs of encrypted integers by means of packing is depicted in Figure 2. Because of the large differences, the complexity axis has a logarithmic scale. Since the DGK protocol communicates one encryption per input bit, our packed version reduces the communication complexity with roughly a factor ℓ.

Figure 2 Average computational complexity per secure comparison (see online version for colours)



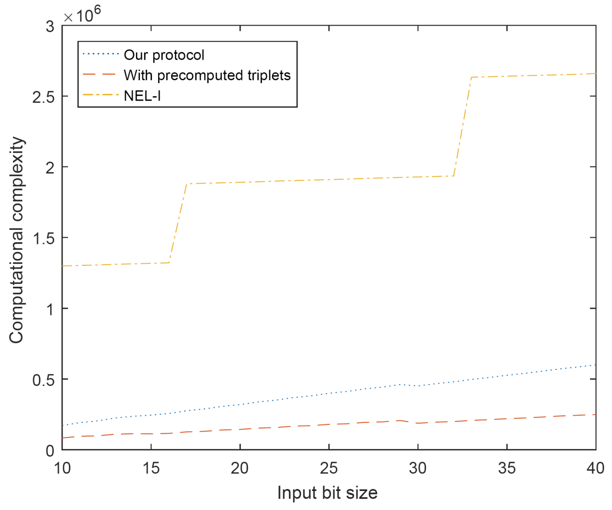
8.3 Secure equality

Suppose party $\mathcal{A}$ holds 1,000 pairs of encrypted integers, $[x_i]$, $[y_i]$, $1 \leq i \leq 1{,}000$, for which $\mathcal{A}$ and $\mathcal{B}$ jointly want to determine whether they are equal or not, such that $\mathcal{A}$ obtains $[\delta_i] = [(x_i = y_i)]$. The state-of-the-art solution for performing this, is described by Nateghizad et al. (2016), who also consider 1,000 equality tests. They improve other existing cryptographic protocols by using, among other techniques, packing.

We suggest to use batching, and compare their results with our secure equality approach. We vary the bit length of the inputs from $\ell = 10$ to $\ell = 40$, and compute the number of multiplications with Paillier encryptions, as a measure for the computational complexity. We batch the encrypted integers $[z_i] = [x_i + 2^\ell - y_i]$ into a small number of encryptions, and determine sharings of $\delta_i = (z_i = 2^\ell)$. The maximal number n of compartments we can have in one encryption is determined by $n \cdot \sigma < \log_2 N - \kappa$, where $\sigma = \ell + 1$ and varies from $n = 48$ (for $\ell = 40$) to $n = 178$ (for $\ell = 10$). This exceeds the number of compartments from Nateghizad et al. (2016), since they need $n \cdot (\ell + \kappa + 1) < \log_2 N$, which gives $n = 22$ (for $\ell = 10$) only.

Figure 3 shows the computational improvement of our equality protocol compared to the protocol NEL-I (Nateghizad et al., 2016), the gain of which increases for growing input size. The communication complexity of both solutions is comparable.

Figure 3 Average computational complexity for 1,000 secure equalities (see online version for colours)



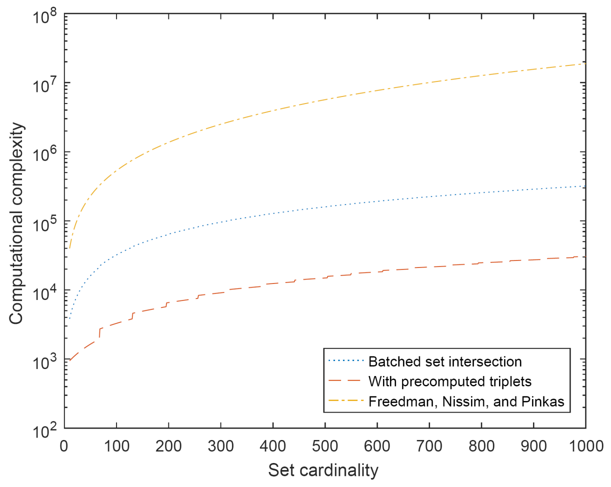
In step 8 of NEL-I, the authors use coefficients α_i of a Lagrange polynomial. They oversaw that these coefficients are rational numbers, which destroys the efficiency of their solution. However, this can be solved by computing $[t^i \cdot \alpha^{-1}]$, instead of $[t^i]$, $1 \leq i \leq \log_2 \ell$, where α^{-1} is the multiplicative inverse of $\alpha = \prod_{i=2}^{\log_2 \ell} (i-1)$ modulo N.

8.4 Set intersection

We compare our batched solution for securely computing the set intersection, with the protocol of Freedman et al. (2004). We fix the number of bits ℓ of the set elements to 10, and let $n = k$ increase from 10 till 1,000. The results are depicted in Figure 4.

Our approach shows a decreased computational complexity, the gain of which grows for larger sets. Because of the large differences, the complexity axis has a logarithmic scale.

Figure 4 Computational complexity of set intersection (see online version for colours)



8.5 Summary

The highest gain of our platform is achieved in case many numbers need to be securely compared. Packing will reduce the computational complexity by roughly a factor thousand, and the communication complexity by a factor ℓ (ignoring precomputations). A similar gain in computational complexity is experienced when using batching for secure set intersection.

For the multiplication and equality operations, the advantage of batching w.r.t. computational complexity is less, but still remarkable. On the other hand, the average communication complexity of a secure multiplication is reduced by a factor n.

9 Conclusions and further research

We introduced a framework that can be used for efficiently computing with encrypted data between two parties in the semi-honest security model. It uses additively homomorphic encryption for adding and multiplying a large number of encrypted integers, and bit-wise secret sharing for more complicated operations on the encrypted integers. The framework is shown to be computationally and communication-wise more efficient than state-of-the-art solutions for a number of generic operations. To increase its performance, an important, data-independent part can be precomputed. This includes the decoding operations in case they need to be performed in the encrypted domain.

Our framework is expected to improve the performance of several applications, when computing in the encrypted domain is required. Ideally, encoding is performed once during the input phase, and decoding is done at the end. The exact implications have yet to be determined. We currently precompute the multiplication triplets by additively homomorphic encryption, but the efficiency is likely to be improved by using extended oblivious transfers as shown in Subsection 4.2.

References

Beaver, D. (1998) 'One-time tables for two-party computation', in Goos, G., Hartmanis, J., van Leeuwen, J., Hsu, W-L. and Kao, M-Y. (Eds.): *Computing and Combinatorics*, Vol. 1449, pp.361–370, Springer Berlin Heidelberg, Berlin, Heidelberg [online] http://link.springer.com/10.1007/3-540-68535-9_40.

Bianchi, T., Veugen, T., Piva, A. and Barni, M. (2009) 'Processing in the encrypted domain using a composite signal representation: pros and cons', *IEEE International Workshop on Information Forensics and Security*.

Chen, H., Laine, K. and Rindal, P. (2017) 'Fast private set intersection from homomorphic encryption', *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security – CCS '17*, Dallas, Texas, USA, pp.1243–1255, ACM Press [online] http://dl.acm.org/citation.cfm?doid=3133956.3134061.

Damgård, I., Geisler, M. and Krøigaard, M. (2008) 'Homomorphic encryption and secure comparison', *Journal of Applied Cryptology*, Vol. 1, No. 1, pp.22–31.

Damgård, I., Geisler, M. and Krøigaard, M. (2009) 'A correction to efficient and secure comparison for online auctions', *Journal of Applied Cryptology*, Vol. 1, No. 4, pp.323–324.

De Cock, M., Dowsley, R., Horst, C., Katti, R., Nascimento, A., Poon, W-S. and Truex, S. (2017) 'Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation', *IEEE Transactions on Dependable and Secure Computing*, p.1 [online] http://ieeexplore.ieee.org/document/7873244/.

Erkin, Z., Veugen, T., Toft, T. and Lagendijk, R.L. (2012) 'Generating private recommendations efficiently using homomorphic encryption and data packing', *IEEE Transactions on Information Forensics and Security*, Vol. 7, No. 3, pp.1053–1066.

Freedman, M.J., Nissim, K. and Pinkas, B. (2004) 'Efficient private matching and set intersection', in Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Terzopoulos, D., Tygar, D., Vardi, M.Y., Cachin, C. and Camenisch, J.L. (Eds.): *Advances in Cryptology – EUROCRYPT 2004*, Vol. 3027, pp.1–19, Springer Berlin Heidelberg [online] http://link.springer.com/10.1007/978-3-540-24676-3_1.

Furukawa, J., Lindell, Y., Nof, A. and Weinstein, O. (2017) 'High-throughput secure three-party computation for malicious adversaries and an honest majority', Coron, J-S. and Nielsen, J.B. (Eds.): *Advances in Cryptology – EUROCRYPT 2017*, Vol. 10211, pp.225–255, Springer International Publishing, Cham [online] http://link.springer.com/10.1007/978-3-319-56614-6_8.

Goldreich, O. (2001) *Foundations of Cryptography: Basic Applications*, Vol. 2, Cambridge University Press, New York.

Keller, M., Orsini, E. and Scholl, P. (2016) 'MASCOT: faster malicious arithmetic secure computation with oblivious transfer', *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security – CCS'16*, Vienna, Austria, pp.830–842, ACM Press [online] http://dl.acm.org/citation.cfm?doid=2976749.2978357.

Knuth, D.E. (1997) *The Art of Computer Programming*, 3rd ed., Addison-Wesley, Reading, Mass.

Kolesnikov, V. and Kumaresan, R. (2013) 'Improved OT extension for transferring short secrets', in Canetti, R. and Garay, J.A. (Eds.): *Advances in Cryptology – CRYPTO 2013*, Vol. 8043, pp.54–70, Springer Berlin Heidelberg, Berlin, Heidelberg [online] http: //link.springer.com/10.1007/978-3-642-40084-1_4.

Lagendijk, R.L. and Barni, M. (2013) 'Encrypted signal processing for privacy protection: conveying the utility of homomorphic encryption and multiparty computation', *IEEE Signal Processing Magazine*, January, Vol. 30, No. 1, pp.82–105 [online] http://ieeexplore.ieee.org/document/6375935/.

Nateghizad, M., Erkin, Z. and Lagendijk, R.L. (2016) 'Efficient and secure equality tests', *2016 IEEE International Workshop on Information Forensics and Security (WIFS)*, December, Abu Dhabi, United Arab Emirates, pp.1–6, IEEE [online] http://ieeexplore.ieee.org/document/7823915/.

Paillier, P. (1999) 'Public-key cryptosystems based on composite degree residuosity classes', *Proceedings of Eurocrypt 1999, Lecture Notes in Computer Science*, Vol. 1592, pp.223–238, Springer-Verlag [online] http://citeseer.ist.psu.edu/article/paillier99publickey.html.

Saha, T.K. and Koshiba, T. (2016) 'Private equality test using ring-LWE somewhat homomorphic encryption', *2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE)*, December, Nadi, Fiji, pp.1–9, IEEE [online] http://ieeexplore.ieee.org/document/7941933/.

Smart, N.P. and Vercauteren, F. (2014) 'Fully homomorphic SIMD operations', *Designs, Codes and Cryptography*, April, Vol. 71, No. 1, pp.57–81 [online] http://link.springer.com/10.1007/s10623-012-9720-4.

Veugen, T. (1991) *Some Mathematical and Computational Aspects of Electronic Cash*, Master Thesis, Eindhoven University of Technology, The Netherlands.

Veugen, T. (2012) 'Improving the DGK comparison protocol', *2012 IEEE International Workshop on Information Forensics and Security (WIFS)*, December, Costa Adeje, Tenerife, Spain, pp.49–54, IEEE [online] http://ieeexplore.ieee.org/document/6412624/.

Veugen, T. (2014) 'Encrypted integer division and secure comparison', *International Journal of Applied Cryptography*, Vol. 3, No. 2, p.166 [online] http://www.inderscience.com/link.php?id=62738.

Veugen, T. (2018) 'Secure comparison through simple bit operations', in Spreeuwers, L. and Goseling, J. (Eds.): *2018 Symposium on Information Theory and Signal Processing in the Benelux*, May, University of Twente, Enschede, The Netherlands, pp.203–206, IEEE.

A Vector addition chain

We can efficiently compute $[X]=\prod_{i=1}^{n}[x_i]^{\mu_i \frac{M}{m_i}}$ with a vector addition chain (Veugen, 1991).

A.1 Vector of width two

When computing $x_1^{e_1} \cdot x_2^{e_2}$, for certain positive integers x_1, x_2, e_1 and e_2, the straightforward approach is to first compute the two exponentiations separately, and then multiply them together. This would roughly require 1 + 1.5 $\log_2 e_1$ + 1.5 $\log_2 e_2$ multiplications.

However, this can be improved as follows (Veugen, 1991). Assuming $e_1 < e_2$, one can compute $e_2 \div e_1$ and e_2 mod e_1, such that $e_2 = (e_2 \div e_1) \cdot e_1 + (e_2 \bmod e_1)$. Then, by rewriting $x_1^{e_1} \cdot x_2^{e_2}$ as $(x_1 \cdot x_2^{e_2 \div e_1})^{e_1} \cdot x_2^{e_2 \bmod e_1}$, the sizes of the exponents e_1 and e_2 have been reduced to the sizes of $(e_2 \div e_1) \cdot e_1$ and e_2 mod e_1. The same trick can be applied to the reduced exponents, yielding a recursive solution. The average number of multiplications of this recursive solution can be determined as 1.6 $\log_2 e_2$, which is only slightly more than the average number of multiplications for computing only $x_2^{e_2}$.

A.2 Divide and conquer

Using the algorithm from the previous subsection, we propose a 'divide and conquer' approach for batching. Given integers i and j, $1 \le i \le j \le n$, define $[i, j]$ as the set of integers $\{i, i+1, \ldots, , j\}$. Further define, given integer set S,

$$\Xi_S = \prod_{i \in S} \left([x_i]^{\mu_i}\right)^{\prod_{s \in S, s \ne i} m_s}.$$

In order to compute $\Xi_{[1,n]}$, we propose to first compute $\Xi_{[1,(n+1)\div 2]}$ and $\Xi_{[(n+1)\div 2+1,n]}$, and then combine the two:

$$\Xi_{[1,n]} = \left(\Xi_{[1,(n+1)\div 2]}\right)^{\prod_{s \in [1,(n+1)\div 2]} m_s} \cdot \left(\Xi_{[(n+1)\div 2+1,n]}\right)^{\prod_{s \in [(n+1)\div 2+1,n]} m_s}.$$

The combining of these two integers requires a vector addition chain of width two, for which we can apply the previous recursive solution.

A.3 Number of multiplications

We can determine the average number of multiplications to compute one batch, i.e., to compute $\Xi_{[1,n]}$ from $[x_i]$, $1 \le i \le n$. Let σ be the maximal number of bits of the numbers m_i, $1 \le i \le n$. Let $A(n)$ be the average number of multiplications to compute a batch of n encrypted integers. Using the previous subsections, we can derive the following recurrent equation:

$$A(n) = 2 \cdot A(n/2) + 1.6 \cdot (n/2)\sigma.$$

For $n = 2$, we have $A(2) = 1.6 \cdot 2\sigma$. Therefore, if n is a power of two, we have $A(n) = (1 + \log_2 n) \cdot 0.8 \cdot \sigma n$. If n is not a power of two, the interval $[1, n]$ will at some point in the recurrence relation not split up in two exactly equal parts, but our formula for $A(n)$ will still be a good approximation.

B Combining secure comparison protocols

Let x and y be two privately held integers, $0 \le x, y < 2^{\ell+2}$. Suppose we have securely computed $\epsilon = (x \bmod 2^{\ell} < y \bmod 2^{\ell})$, and want to extend this result to $\delta = (x < y)$. The following relation between δ and ϵ is easily derived.

$$\delta = \left(x_{(\ell+1)}x_{(\ell)} = y_{(\ell+1)}y_{(\ell)}\right)\cdot\epsilon + \left(x_{(\ell+1)}x_{(\ell)} < y_{(\ell+1)}y_{(\ell)}\right)$$

In this expression, $x_{(\ell+1)}x_{(\ell)}$ denotes the binary representation of a two-bit number. To compute δ, we first compute sharings of $e_{\ell+1} = (x_{(\ell+1)} = y_{(\ell+1)})$ and $e_\ell = (x_{(\ell)} = y_{(\ell)})$, like in our secure comparison protocol, by local additions. The same for $d_{\ell+1} = (x_{(\ell+1)} < y_{(\ell+1)})$ and $d_\ell = (x_{(\ell)} < y_{(\ell)})$, costing one sharing multiplication each. These shared bits are then used to compute $\delta = e_{\ell+1} \cdot e_\ell \cdot \epsilon + d_{\ell+1} + e_{\ell+1} \cdot d_\ell = d_{\ell+1} + e_{\ell+1} \cdot (\epsilon \cdot e_\ell + d_\ell)$, which requires two secure multiplications of sharings, for a total of four secure multiplications of shares.