A Method for Securely Comparing Integers using Binary Trees

Anselme Tueno
SAP SE
anselme.tueno@sap.com

Jonas Janneck
SAP SE
jonas.janneck@sap.com

December 16, 2021

Abstract

In this paper, we propose a new protocol for secure integer comparison which consists of parties having each a private integer. The goal of the computation is to compare both integers securely and reveal to the parties a single bit that tells which integer is larger. Nothing more should be revealed. To achieve a low communication overhead, this can be done by using homomorphic encryption (HE). Our protocol relies on binary decision trees that is a special case of branching programs and can be implemented using HE. We assume a client-server setting where each party holds one of the integers, the client also holds the private key of a homomorphic encryption scheme and the evaluation is done by the server. In this setting, our protocol outperforms the original DGK protocol of Damgård et al. and reduces the running time by at least 45%. In the case where both inputs are encrypted, our scheme reduces the running time of a variant of DGK by 63%.

1 Introduction

Multi-party computation (MPC) is a cryptographic technique that allows several parties to compute a function on their private inputs without revealing any information other than the function's output [1, 13, 18, 19, 21]. A classic example in the literature is the so-called Yao's Millionaire's problem introduced in [38]. Two millionaires are interested in knowing which of them is richer without revealing their actual wealth. Formally, let there be two parties P_1, P_2 with private input x, y respectively. The goal of the computation is to compute and reveal $b = [x \geq y]$ to the parties and nothing else. This is illustrated in Figure 1.

Integer comparison is one of the basic arithmetic operations in computer programming and algorithm design. Secure integer comparison is therefore necessary in many privacy-preserving computations. In machine learning, private integers must be compared securely while evaluating classifiers such as decision trees or neural networks. In secure enterprise benchmarking, key performance

indicators are securely compared to determine how companies perform compared to their competitors. In secure auction, bids are privately compared to determine the winner. Secure integer comparison has application in different privacy-preserving analytics.

In the following, the party with input x is the client and the party with input y is the server. The idea of our solution consists of having the server construct a binary tree that represents y. Then, the client encrypts x using a homomorphic encryption scheme and let the server evaluate on the tree representing y. Finally, the client receives the result of the computation and decrypts it. Depending on the use case, the client can send the result to the server or they could both get a share of the final result. Furthermore, there are two variants of the protocol. The first (basic) variant utilizes the input of the server in plaintext, the second variant requires both inputs to be encrypted. We compare our results of the first variant to the original DGK protocol [11] and reduce the running time by 45%. Compared to an optimization of the DGK protocol proposed by Veugen [36], we can reduce the running time by about 10% for the first variant. However, for the second variant where both inputs are encrypted we achieve a reduction of more than 63%.

The remainder of the paper is structured as follows. We review preliminaries in Section 2 before defining correctness and security of the functionality in Section 3. We describe our protocol and its algorithms in Section 4 and present some extensions in Section 5. A detailed complexity analysis can be found in Section 6. Section 7 and 8 give details about our implementation, evaluation results and applications. In Section 9, we present related work before concluding our work in Section 10.

2 Preliminaries

Homomorphic encryption (HE) allows computations on ciphertexts by generating an encrypted result whose decryption matches the result of a function on the plaintexts [5, 16].

HE Algorithms. A HE scheme consists of the following algorithms:

- $\mathsf{pk}, \mathsf{sk}, \mathsf{ek} \leftarrow \mathsf{KGen}(\lambda)$: This probabilistic algorithm takes a security parameter λ and outputs public, private, and evaluation keys pk, sk, and ek.
- $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$: This algorithm takes pk and a message m and outputs a ciphertext c. We will use $[\![m]\!]$ as a shorthand notation for $\mathsf{Enc}(\mathsf{pk}, \mathsf{m})$.
- $\mathsf{c} \leftarrow \mathsf{Eval}(\mathsf{ek}, f, \mathsf{c}_1, \ldots, \mathsf{c}_n)$: This algorithm takes ek, an n-ary function f and n ciphertexts $\mathsf{c}_1, \ldots \mathsf{c}_n$ and outputs a ciphertext c.
- $\mathsf{m}' \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{c})$: This deterministic algorithm takes sk and a ciphertext c and outputs a message m'.

We require IND-CPA security and the following correctness conditions. Given any set of n plaintexts $\mathsf{m}_1, \ldots, \mathsf{m}_n$, it must hold for any $\mathsf{pk}, \mathsf{sk}, \mathsf{ek}$:

- $\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_i)) = \mathsf{Dec}(\mathsf{sk}, [\![\mathsf{m}_i]\!]) = \mathsf{m}_i$,
- $\mathsf{Dec}(\mathsf{sk}, \mathsf{Eval}(\mathsf{ek}, f, [\![\mathsf{m}_1]\!], \ldots, [\![\mathsf{m}_n]\!])) = \mathsf{Dec}(\mathsf{sk}, [\![f(\mathsf{m}_1, \ldots, \mathsf{m}_n)]\!])$.

In practice, a homomorphic encryption defines two basic operations for addition and multiplication that can then be used to compute larger functionalities.

FHE Operations. An FHE scheme defines both operations (addition and multiplication). For all plaintexts $\mathsf{m}_1, \mathsf{m}_2$, we define the following operations and introduce shorthand notations:

- Addition: $\textsc{Add}([\![\mathsf{m}_1]\!], [\![\mathsf{m}_2]\!]) = [\![\mathsf{m}_1]\!] \boxplus [\![\mathsf{m}_2]\!] = [\![\mathsf{m}_1 + \mathsf{m}_2]\!]$,
- Constant Addition : $\textsc{AddCons}([\![\mathsf{m}_1]\!], \mathsf{m}_2) = [\![\mathsf{m}_1]\!] \boxplus \mathsf{m}_2 = [\![\mathsf{m}_1 + \mathsf{m}_2]\!]$,
- Multiplication: $\textsc{Mul}([\![\mathsf{m}_1]\!], [\![\mathsf{m}_2]\!]) = [\![\mathsf{m}_1]\!] \boxdot [\![\mathsf{m}_2]\!] = [\![\mathsf{m}_1 \cdot \mathsf{m}_2]\!]$,
- Constant Multiplication : $\textsc{MulCons}([\![\mathsf{m}_1]\!], \mathsf{m}_2) = [\![\mathsf{m}_1]\!] \boxdot \mathsf{m}_2 = [\![\mathsf{m}_1 \cdot \mathsf{m}_2]\!]$.

Additively HE. If the scheme supports only addition, then it is called *additively HE (AHE)*. Schemes such as Paillier [30] or Elliptic Curve ElGamal [22] are additively homomorphic and have the following properties for all integer plaintexts $\mathsf{m}_1, \mathsf{m}_2$ and bit plaintexts $a, b \in \{0, 1\}$:

- Addition: $\textsc{Add}([\![\mathsf{m}_1]\!], [\![\mathsf{m}_2]\!]) = [\![\mathsf{m}_1]\!] \boxplus [\![\mathsf{m}_2]\!] = [\![\mathsf{m}_1 + \mathsf{m}_2]\!]$,
- Constant Multiplication: $\textsc{MulCons}([\![\mathsf{m}_1]\!], \mathsf{m}_2) = [\![\mathsf{m}_1]\!] \boxdot \mathsf{m}_2 = [\![\mathsf{m}_1 \cdot \mathsf{m}_2]\!]$,
- Xor: $\textsc{Xor}([\![a]\!], b) = \textsc{Add}([\![b]\!], \textsc{MulCons}([\![a]\!], (-1)^b)) = [\![a \oplus b]\!]$.

Note that we use the same shorthand notation for FHE and AHE. The actual implementation depends on the underlying scheme.

Somewhat, Leveled and Fully HE. If the scheme supports addition and multiplication, but for a limited number of times, then it is somewhat homomorphic (SHE). When arbitrary computation can be performed on encrypted data, then the encryption scheme is fully homomorphic (FHE). Because FHE requires the so-called bootstrapping that is computationally expensive, it is sometime useful to use leveled FHE for efficiency. Leveled FHE can evaluate only computation up to a given circuit depth that is fixed by the encryption keys. In the following, we will use only the term FHE for fully homomorphic encryption and leveled fully homomorphic encryption.

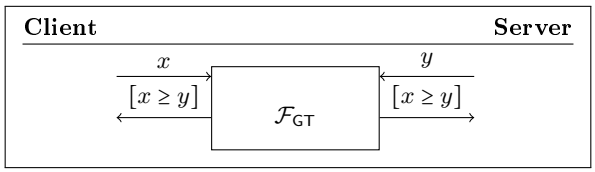


Figure 1: Illustration of the GT functionality

3 Definitions

Setting. The protocol consists of a server holding an input y, and a client holding an input x . We assume that both inputs consist of μ−bit integers and μ is public. The ideal functionality $\mathcal{F}_{\mathsf{GT}}$ takes y from the server and x from the client. It computes and outputs a bit $b = [x \geq y]$ to the parties such that $b = 1$ if $x \geq y$ and $b = 0$ otherwise. The functionality is illustrated in Figure 1. In the following, we build our protocol on the case where only the client gets an output $b = [x \geq y]$. It can be easily extended to a symmetric scenario if the server chooses a random $b_s \in \{0,1\}$ and homomorphically computes $b_c = b \oplus b_s$ before sending the result to the client. Then, the client decrypts the result and both parties holding shares b_c, b_s respectively and can reconstruct the result $b = b_c \oplus b_s$. In some larger settings, it might be required to return only these shares of b to the parties, preventing them to learn intermediate result.

Security and Correctness. A protocol correctly implements the GT functionality if after the computation the output is correct, i.e., $b = 1$ if $x \geq y$ and $b = 0$ otherwise. Besides correctness parties must learn only what they are allowed to. To formalize this, we need the following definition [18]. Two probability distribution ensembles $\{X_i\}_{i\in\{0,1\}^*}, \{Y_i\}_{i\in\{0,1\}^*}$ are *computational indistinguishable* (denoted by $\stackrel{c}{\equiv}$) if for every probabilistic polynomial-time (PPT) algorithm D, every positive polynomial $p(\cdot)$ and all sufficiently long $w \in \{0,1\}^*$ it holds that $|Pr[D(X_w, w) = 1] - Pr[D(Y_w, w) = 1]| < 1/p(|w|)$. In other words, there is no algorithm that can distinguish between the distributions.

In multi-party protocols the *view* of a party consists of its input and the sequence of messages that it has received during the protocol execution [18]. The protocol is said to be secure if for each party, one can construct a simulator that, given only the input of that party and the output, can generate a distribution that is computationally indistinguishable to the party's view. We focus on the semi-honest security model in which parties follow the protocol but may try to learn more information from its execution.

A protocol securely implements the GT functionality $\mathcal{F}_{\mathsf{GT}}$ in the semi-honest model if each party learns only its output and nothing else. In particular, there must exists simulators $\mathsf{Sim}_{\mathsf{C}}^{\mathsf{gt}}$ and $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{gt}}$ that simulate the client and the server given only their input and output to the protocol. Let Π_{GT} denote a protocol that securely implements $\mathcal{F}_{\mathsf{GT}}$, and let $\mathsf{View}_{\mathsf{P}}^{\Pi_{\mathsf{GT}}}(x, y)$ denote the view of party

P during the protocol, then the following hold:

- there exists a PPT algorithm $\mathsf{Sim}_{\mathsf{S}}^{\mathsf{gt}}$ that simulates the server's view $\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{GT}}}$ given only y and $[x \geq y]$ such that:

$$\{\mathsf{Sim}_{\mathsf{S}}^{\mathsf{gt}}(y, [x \geq y])\}_{x,y \in \{0,1\}^*} \quad \stackrel{c}{\equiv} \quad \{\mathsf{View}_{\mathsf{S}}^{\Pi_{\mathsf{GT}}}(x, y)\}_{x,y \in \{0,1\}^*},$$

- there exists a PPT algorithm $\mathsf{Sim}_{\mathsf{C}}^{\mathsf{gt}}$ that simulates the client's view $\mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{GT}}}$ given only x and $[x \geq y]$ such that:

$$\{\mathsf{Sim}_{\mathsf{C}}^{\mathsf{gt}}(x, [x \geq y])\}_{x,y \in \{0,1\}^*} \quad \stackrel{c}{\equiv} \quad \{\mathsf{View}_{\mathsf{C}}^{\Pi_{\mathsf{GT}}}(x, y)\}_{x,y \in \{0,1\}^*}.$$

4 Our Protocol

Our protocol relies on a branching program that is represented as a binary tree. We therefore start by defining our data structure. Then, we describe how our algorithms use this data structure to implement the functionality.

4.1 Data Structure

The data structure is a binary tree consisting of inner nodes and terminal nodes. Each inner node has two child nodes and terminal nodes have no child nodes. There is a node with no parent node that is called root node. Let v be a node in the tree. We define a node data structure Node consisting of the following attributes:

- $v.\mathsf{parent}$: a value representing the pointer to the parent node,
- $v.\mathsf{left}$: a value representing the pointer to the left child node,
- $v.\mathsf{right}$: a value representing the pointer to the right child node,
- $v.\mathsf{lEdge}$: a bit representing the *edge label* to the left child node,
- $v.\mathsf{rEdge}$: a bit representing the *edge label* to the right child node,
- $v.\mathsf{cLabel}$: a value representing a *node label*,
- $v.\mathsf{cost}$: an integer representing the cost on the path from the root.

The pointer to the parent node $v.\mathsf{parent}$ is initially null and points to the respective parent node, when the child node is created. This pointer remains null for the root node. The pointers to the child nodes $v.\mathsf{left}, v.\mathsf{right}$ are initially null, and point to the respective nodes if they are created. The edge labels to the child nodes $v.\mathsf{lEdge}, v.\mathsf{rEdge}$ are 0 on the left and 1 on the right. The node label $v.\mathsf{cLabel}$ is 0 or 1 for terminal nodes and undefined for inner nodes. The cost attribute $v.\mathsf{cost}$ is computed during evaluation of the tree.

4.2 Algorithms

Our scheme works for both AHE and FHE but must be implemented differently. To simplify the description of our scheme, we therefore introduce the symbol β to differentiate between AHE and FHE. Namely, if the encryption scheme is FHE then $\beta = 1$ otherwise $\beta = 0$. For an integer x, we use $\bar{x} = x[1], \ldots, x[\mu]$ to denote the corresponding bit representation, where $x[\mu]$ is the most significant bit, and we use $[\![\bar{x}]\!] = [\![x[1]]\!], \ldots, [\![x[\mu]]\!]$ to denote the bitwise encryption of $\bar{x}$.

Initialization. The Initialization consists of a one time key generation. The client generates an appropriate triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek})$ of public, private and evaluation key for a homomorphic encryption scheme. Then, the client sends $(\mathsf{pk}, \mathsf{ek})$ to the server. For each computation, the client just encrypts its input and sends it to the server.

Creating the Binary Tree. Let y be the server input with bit-length μ. The server starts by creating a binary tree representing y. The basic idea consists of creating a binary tree representing all bit strings of length μ. Then the leaf of the path that represents y and the leaves of all paths right to the path of y are labelled with 1 (i.e., $v.\mathsf{cLabel} = 1$). The leaves of the paths left to y are labelled with 0 (i.e., $v.\mathsf{cLabel} = 0$). Finally, we can prune all subtrees that are labelled with the same bit. That is, if an inner node v has two child nodes labelled with the same bit b, we remove the child nodes of v from the tree and transform v into a leaf node labelled with b, (i.e., $v.\mathsf{cLabel} = b$). However, we can avoid the pruning by traversing the tree a single time with the bits of y. If $y[i] = 1$, we insert a leaf node on the left with $\mathsf{cLabel} = 0$, and a new node on the right, then we traverse to the right. If $y[i] = 0$, we insert a leaf node on the right with $\mathsf{cLabel} = 1$ and a new node on the left, then we traverse to the left. Note that inserting a leaf node is only required if we are using FHE. For AHE, the traversal works similarly as above except that no leaf node is inserted left from the traversed path. Therefore, the created tree contains only paths, that can be evaluated to zero, i.e., paths labelled with integers that are larger or equal to y. The creation of the binary tree is illustrated in Algorithm 1.

For example, assume that $\mu = 3$, then Figure 2 illustrates the binary trees of 2, 3, and 5 if the scheme is FHE and Figure 3 illustrates the binary trees of 2, 3, and 5 if the scheme is AHE.

Computing Decision Bits. Let x be the input of the client. The client sends x bitwise encrypted. That is, the client computes the bit representation $\bar{x} = x[1], \ldots, x[\mu]$ and then sends the corresponding ciphertext $[\![\bar{x}]\!] = [\![x[1]]\!], \ldots, [\![x[\mu]]\!]$ to the server. The server computes the decision bits at each inner node v by comparing each $[\![x[i]]\!]$ against the edge labels of node v. This is represented by a comparison operation comp. For FHE, it is implemented as a bit *equality* test that returns $[\![1]\!]$ if the two bits are equal and $[\![0]\!]$ otherwise. For AHE, it is implemented as an *inequality* test that returns $[\![0]\!]$ if the two bits are equal and $[\![1]\!]$ otherwise. The operation can be computed by at least one

Algorithm 1 Creating the Binary Tree for an Integer y.

```
1:  function CreateTree(y)
2:      let root be a new node
3:      let curr be an empty node
4:      let y[1], ..., y[μ] be the bit string of y
5:      curr ← root
6:      for i = μ downto 1 do
7:          if y[i] = 1 then
8:              let v be a new node
9:              curr.right ← v
10:             if β = 1 then                                  ▷ Only FHE
11:                 let v be a new node
12:                 v.cLabel ← 1 − β
13:                 curr.left ← v
14:             curr ← curr.right
15:         else
16:             let v be a new node
17:             curr.right ← v
18:             if β = 1 then                                  ▷ Only FHE
19:                 v.cLabel ← β
20:             let v be a new node
21:             curr.left ← v
22:             curr ← curr.left
23:     if β = 1 then                                          ▷ Only FHE
24:         curr.cLabel ← β
25:     root.cost ← [[β]]
26:     return root
```

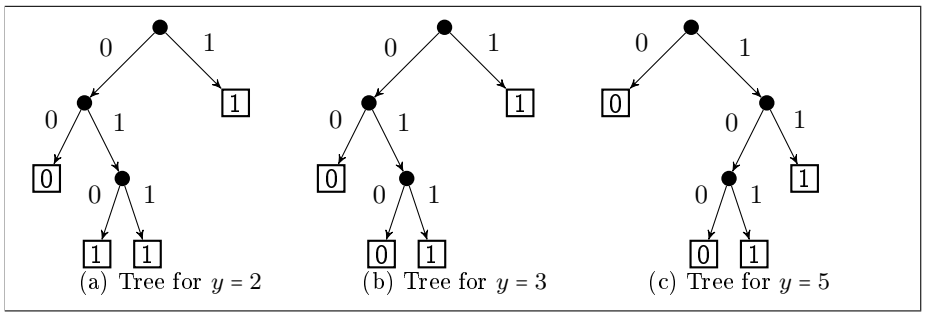


Figure 2: Tree Creation for FHE ($\beta = 1$) and $\mu = 3$

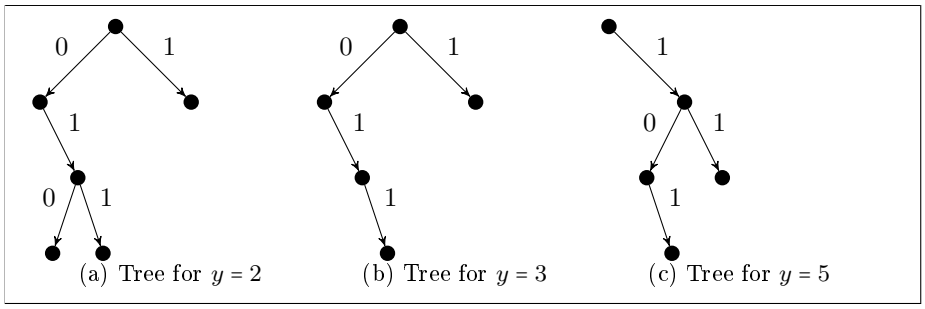


Figure 3: Tree Creation for AHE ($\beta = 0$) and $\mu = 3$

NOT gate. For example, consider the FHE case in which we have to achieve an equality test. If the edge label is 1, we just take the client's input $[\![x[i]]\!]$ as the comparison result. If it is 0, we compute $[\![\neg x[i]]\!]$.

The computation of decision bits is illustrated in Algorithm 2.

Aggregating Decision Bits. For each leaf node v, the server aggregates the comparison bits along the path from the root to v For FHE this is done using homomorphic multiplication of the decision bits. For AHE, it is done using homomorphic addition of the decision bits. To unify the depiction of our algorithms as much as possible, we introduce a new notation for aggregating the decision bits: DecBitAgg. It refers to the homomorphic multiplication in the FHE case and to the homomorphic addition in the AHE case. The aggregated result is then stored at the leaf node of the corresponding path. We implement it using a queue and traversing the tree in BFS order as illustrated in Algorithm 3. Note that this computation can be improved using path prefixes, i.e. for two paths having the same prefix, the prefix is evaluated once.

Evaluating leaves. The evaluation of the leaves depends on the scheme as well. For FHE, after aggregating the decision bits along the paths to the leaf

Algorithm 2 Computing Decision Bits

```
1:  function EvalNodes(root, ⟦x̄⟧)
2:      parse ⟦x̄⟧ to ⟦x[1]⟧, ..., ⟦x[μ]⟧
3:      v ← root
4:      for i = μ downto 1 do
5:          if v.left ≠ null then
6:              ⟦v.left.cost⟧ ← comp(⟦x[i]⟧, v.lEdge)
7:          if v.right ≠ null then
8:              ⟦v.right.cost⟧ ← comp(⟦x[i]⟧, v.rEdge)
9:          if v.right ≠ null and v.right.IsLeaf() = false then
10:             v ← v.right
11:         else
12:             v ← v.left
```

Algorithm 3 Aggregating Decision Bits

```
1:  function EvalPaths(root)
2:      let Q be a queue
3:      let leaves be a queue
4:      Q.enqueue(root)
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          if v.left ≠ null then
8:              ⟦v.left.cost⟧ ← DecBitAgg(⟦v.left.cost⟧, ⟦v.cost⟧),
9:              if v.left.IsLeaf() then
10:                 leaves.enqueue(v.left)
11:             else
12:                 Q.enqueue(v.left)
13:         if v.right ≠ null then
14:             ⟦v.right.cost⟧ ← DecBitAgg(⟦v.right.cost⟧, ⟦v.cost⟧),
15:             if v.right.IsLeaf() then
16:                 leaves.enqueue(v.right)
17:             else
18:                 Q.enqueue(v.right)
19:     return leaves
```

Algorithm 4 Evaluating leaves using FHE

```
1: function EvalLeaves(leaves)
2:     [[b]] ← [[0]]
3:     for each v ∈ leaves do
4:         [[b]] ← [[b]] ⊞ ([[v.cost]] ⊡ [[v.cLabel]])
5:     return [[b]]
```

Algorithm 5 Evaluating leaves using AHE

```
1: function EvalLeaves(leaves)
2:     let c be an array of ciphertexts
3:     for i = 1 to leaves.size() do
4:         c[i] ← randomize(leaves.get(i).cost)        ▷ Randomize the ciphertext
5:     for i = leaves.size() to μ do
6:         c[i] ← genRandCtxt()                        ▷ Generate a random ciphertext
7:     return permute(c)
```

nodes, each leaf node v stores either $[\![v.\mathsf{cost}]\!] = [\![0]\!]$ or $[\![v.\mathsf{cost}]\!] = [\![1]\!]$. Moreover, there is a unique leaf with $[\![v.\mathsf{cost}]\!] = [\![1]\!]$ and all other leaves have $[\![v.\mathsf{cost}]\!] = [\![0]\!]$. Then, the server aggregates the costs at the leaves by computing for each leaf v the value $[\![v.\mathsf{cost}]\!] \boxdot [\![v.\mathsf{cLabel}]\!]$ and summing all the results of all leaves. This computation is illustrated in Algorithm 4.

For AHE, after aggregating the decision bits along the paths to the leaves nodes, each leaf node v stores either $[\![v.\mathsf{cost}]\!] = [\![0]\!]$ or $[\![v.\mathsf{cost}]\!] = [\![r]\!]$, for a random plaintext r. Moreover, there is at most one leaf with $[\![v.\mathsf{cost}]\!] = [\![0]\!]$ and all other leaves have $[\![v.\mathsf{cost}]\!] = [\![r]\!]$, for a random r. Note that for $y \neq 0$ the number of paths is smaller or equal to μ. The server randomizes all ciphertexts, chooses other random ciphertexts permutes the list and sends it to the client. These operations are implemented to guarantee the server's privacy. Randomization and permutation of ciphertexts prevents leakage of any information about y that is not intended. The generation of additional ciphertexts prevents leakage of the tree structure and therefore, potential information about y as well. Note that we exclude the case of randomly generating a ciphertext which decrypts to zero. The computation is illustrated in Algorithm 5.

Decrypting the Result. The client decrypts the result of the evaluation. For FHE, it is a single encrypted bit. For AHE, the evaluation result consists of μ ciphertexts among which exactly one encrypts 0 and the remaining ones encrypt random plaintext. The client uses Algorithm 6 to decrypt and learn the final result.

Putting It All Together. As illustrated in Protocol 4, the whole computation is performed by the server. The server first creates a tree representation of its input y as illustrated in Algorithm 1. Then, the client sends the encrypted

Algorithm 6 Decrypting

```
1:  function DECRYPT(result)
2:      if β = 1 then                                   ▷ FHE case
3:          parse result to ⟦b⟧
4:          b ← Dec(⟦b⟧)
5:          return b
6:      else                                            ▷ AHE case
7:          parse result to c[1], ..., c[μ]
8:          for i = 1 to μ do
9:              m ← Dec(c[i])
10:             if m = 0 then
11:                 return 1
12:         return 0
```

bit representation $[\![\bar{x}]\!]$ of its input x and the server sequentially runs the Algorithms 2, 3 and 4/5 described above. The server sends an encrypted result, which the client can decrypt to learn the final comparison bit $b = [x \geq y]$.

Lemma 4.1 *Let y and x be integers of length μ. If the encryption scheme is correct, then the comparison protocol is correct.*

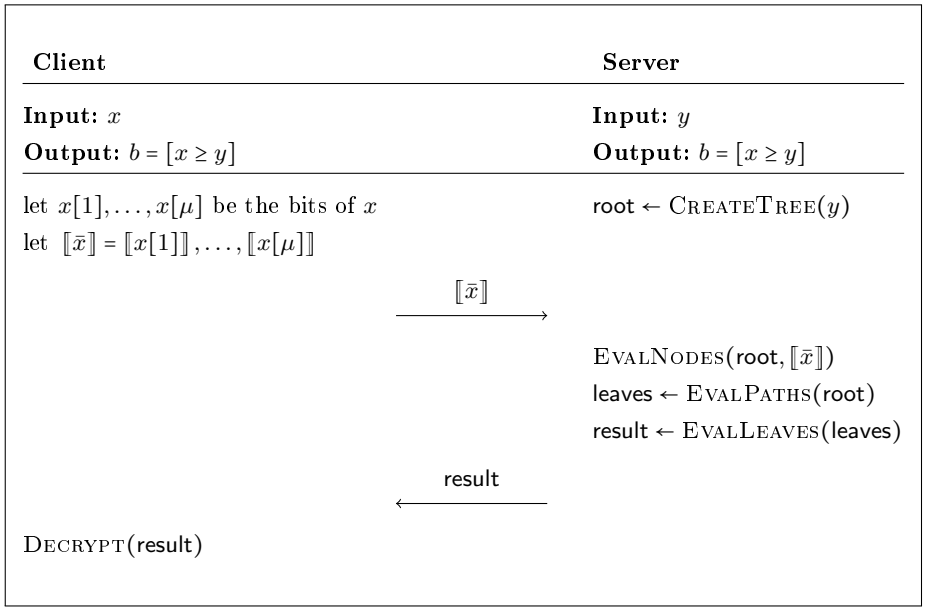
Proof. In the tree of y, there is a single path that is labeled with a prefix of x. Evaluating the nodes on this path and aggregating the results produces a bit 1 (if FHE), resp. 0 (if AHE). On all other paths, at least one edge is labelled with a bit that is different to the bit of x at the same position such that the evaluation of the path produces a bit 0 (if FHE), resp. an integer $r \neq 0$ (if AHE). ■

Theorem 4.2 *Let y and x be integers of length μ. If the encryption scheme is IND-CPA secure, then the comparison protocol is secure in the semi-honest model.*

Proof (sketch). The client only encrypts its own input and decrypts the final result which for FHE is a single bit, and for AHE a randomly ordered list of μ ciphertexts among which at most one encrypts 0 and the remaining ones encrypt each a random plaintext. The server, on the other hand, computes on IND-CPA ciphertexts. Constructing the simulators therefore consists of on simply generating corresponding random strings for each protocol message except for the actual results. ■

5 Extension

In the previous section, we discuss the basic idea of our scheme. Now we want to discuss how the basic scheme can be extended to different use cases.



Protocol 4: The Basic Protocol

5.1 Handling Comparison to Zero for AHE

Recall that if the encryption is AHE, then Algorithm 1 creates a tree containing only paths, that can be evaluated to zero, i.e., paths labelled with integers that are larger or equal to y. If $y = 0$ then the created tree has $\mu + 1$ leaves, since everything is larger or equal to zero. But the server is supposed to send back μ ciphertexts to the client. That is, we still want the parties to perform the computation such that nothing more than the comparison bit is revealed. We notice that for all values smaller than $2^{\mu-1}$ (i.e., the most significant bit is 0), x traverses the tree of $y = 0$ to the left. To handle the case $y = 0$, the server, therefore, replaces the first encrypted bit of x by a ciphertext of 0 and omits the rightmost path of the tree in the evaluation.

5.2 Shared Output Bit

In 2-party comparison like DGK [11], it is usual to share the comparison bit between the client and server. That is, if b is the comparison bit, then the server gets b_s and the client gets b_c such that $b = b_c \oplus b_s$. In our scheme, the server can randomly choose between computing GT (e.g., $[x \geq y]$) or LT (e.g., $[x \leq y]$) functionality. The server, therefore, flips a random coin b_s and computes GT if $b_s = 0$, otherwise it computes LT. We will describe later how to compute LT.

5.3 Handling Encrypted Inputs

So far we assume that only x is encrypted. In this section, we consider the case where both inputs are encrypted. In this scenario, the server has to run the comparison of two encrypted inputs with the help of the client (or another server) which has the decryption key. It is assumed that the inputs x and y do not belong to any party and must remain private. After the computation, the server learns the encrypted comparison bit. If the encryption is FHE, the server can perform the computation on its own. However, in the AHE case, the client must help the server to learn the encrypted result.

To guarantee the privacy of both inputs, the protocol has to be evaluated on ciphertexts only. However, the tree structure reveals a lot of information about y why we have to use a general representation of the tree to avoid any leakage. We start with a few formal definitions.

Definition 5.1 (Comparison Tree) *A* comparison tree *or* cmp-tree *for an integer y is a binary tree where edges and leaves are labelled with 0 or 1 such that for every integer x, traversing the tree along a path labelled with the bits of x (starting with the most significant bit of x) reaches a leaf labelled with 1 if $x \geq y$ and 0 otherwise.*

Note that for secure comparison the bits are encrypted such that we do not actually traverse the tree y, but evaluate it on x as explained in the previous section. Therefore, when we say x traverses the tree of y, we mean that there is a single path where x evaluates to 1 (if FHE), resp. 0 (if AHE), and on all other paths x evaluates to 0 (if FHE), resp. to an $r \neq 0$ (if AHE).

While Definition 5.1 describes a cmp-tree, it can be built as follows. Let μ be the input bit-length of y. First build a binary tree representing all bit strings of length μ, i.e. left edges are labelled with 0 and right edges are labelled with 1. Then, there is path p representing y, label the leaf of p and the leaves of all paths right to p with 1. Finally, label the leaves of all path left to p with 0. Such a tree construction is illustrated in Figure 5 for $y = 2$ and $y = 5$. Note that the trees from Figure 5 are unnecessarily to large as there are inner node whose child nodes are both leaves labelled with the same value. Such resulting sub-trees can be pruned without changing the semantic of the cmp-tree. We next formally define pruned cmp-tree.

We first recall the depth of a binary tree.

Definition 5.2 (Depth of a Tree) *For a binary tree, we define the* depth *of the tree as the length (i.e., number of edges) of the longest path. The* depth of a node *is the number of edges between this node and the root node. Let d be the depth of the binary tree, a* deepest inner node *is a node whose child nodes are both leaves with depth d.*

Definition 5.3 (Pruned Cmp-tree) *A comparison tree for an integer y is* full-pruned *if there is no inner node whose children are both leaves with the*

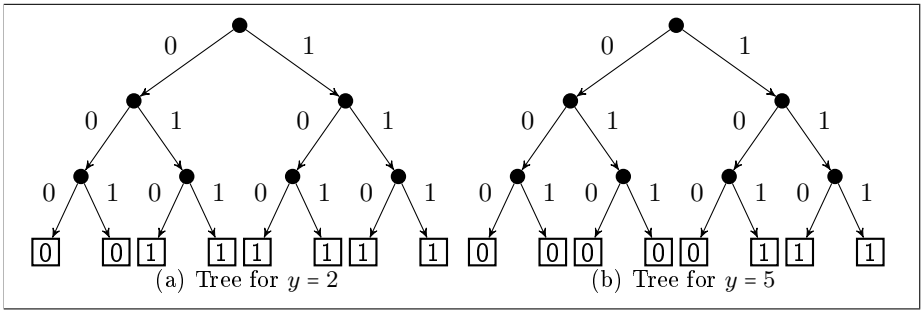


Figure 5: Comparison Tree Creation for input bit-length $\mu = 3$

same label. A cmp-tree for an integer y is half-pruned *if its depth is the bit-length of y and for each non deepest inner node exactly one child node is a leaf.*

Note that a half-pruned cmp-tree is not necessarily full-pruned. For example if the bit-length is $\mu = 3$, then half-pruned tree of 4 is not full-pruned. In this case, the full-pruned tree is only the root with 2 leaves. In the following, we will rather consider half-pruned tree since the structure is similar for every input. The half-pruned tree for integer y can be built as follows. Traverse the non-pruned cmp-tree from Definition 5.1 along the path of y. At each level, replace the non-traversed subtree by a leaf node. Let p be the path representing y. Label the leaf of p and the leaves of all paths right to p with 1. Label the leaves of all path left to p with 0. By using Algorithm 1 with $\beta = 1$, this can be done without first generating the full cmp-tree.

Now, we want to introduce a structure of the tree based on the input size but independent of the actual inputs. We first define further notation. Recall that we use the symbol $\beta = 1$ if the encryption scheme is FHE and $\beta = 0$ if the encryption scheme is AHE. For a bit $b \in \{0,1\}$, we now define the function $F_\beta(b) = \beta + (-1)^\beta \cdot b$. Note that the function F_β does not have to be evaluated homomorphically, as its only purpose is to simplify the notation. For an encrypted bit $[\![b]\!]$, we have $F_\beta([\![b]\!]) = [\![b]\!]$ if the encryption is AHE and $F_\beta([\![b]\!]) = [\![1-b]\!]$ if the encryption is FHE with arithmetic encoding or $F_\beta([\![b]\!]) = [\![1+b]\!]$ for binary encoding, as the addition is modulo two.

Definition 5.4 (Normal Cmp-Tree) *Let y be an integer of length μ. A* normal Cmp-tree *of y is a binary tree with the following structure:*

- *there is a leftmost path p of length μ which is labelled with the bits of y,*
- *the deepest inner node of path p has a left leaf node labelled with β,*
- *each inner node of path p has a right child leaf node,*
- *for each inner node, let b be the label on the left edge, then the label on the right edge is $1-b$ and the label on the right child leaf node is $F_\beta(b)$.*

While Algorithm 1 generates a half-pruned cmp-tree for y assuming the bits are given in plaintext, the normal cmp-tree can be built even if the input bits are homomorphically encrypted. For each encrypted bit b of the input string, one can homomorphically compute the encrypted inverse bit $1-b$ and build the cmp-tree. The generation of the normal cmp-tree is described in Algorithm 7.

Note that in contrast to Algorithm 1, left and right edges are not per default labeled with 0 and 1, but they are assigned according to the definition of the normal cmp-tree. This makes the normal cmp-tree a general structure which is independent of the actual input y. The input only influences the labels of the edges but not the tree structure itself.

Although both algorithms have the same complexity, Algorithm 7 is shorter and simpler. An example of normal cmp-trees is illustrated in Figure 6. Before using the normal form defined above, we need to prove that it is indeed a cmp-tree, by showing that the normal cmp-tree has the same number of nodes (inner nodes and leaves) as the half-pruned cmp-tree and that they can be transferred into each other.

Lemma 5.5 *Let y be an integer of length μ. The half-pruned tree of y has $\mu+1$ leaves and μ inner nodes.*

Proof. The depth of the tree is obviously μ. A complete tree with depth μ has $2^\mu+2^\mu-1$ nodes. While constructing the half-pruned cmp-tree as explained above, we start from the root with depth 0 and stop at a node with depth $\mu-2$, since node with depth $\mu-1$ have only leaves as child nodes. In each step at depth $h \in \{0,\ldots,\mu-2\}$, we replace a subtree, that has $2^{\mu-h}-1$ nodes, with a leaf. That is, at depth $h \in \{0,\ldots,\mu-2\}$, we remove $2^{\mu-h}-2$ nodes. Then the numbers of nodes remaining in the tree is:

$$S = 2^\mu + 2^\mu - 1 - \sum_{h=0}^{\mu-2}(2^{\mu-h}-2) = 2^\mu + 2^\mu - 1 - \left(\sum_{h=0}^{\mu-2} 2^{\mu-h} - \sum_{h=0}^{\mu-2} 2\right).$$

Now we have:

$$\sum_{h=0}^{\mu-2} 2^{\mu-h} = \sum_{h=0}^{\mu} 2^{\mu-h} - (2^1+2^0) = 2^\mu\left(\sum_{h=0}^{\mu} 2^{-h}\right) - 3 = 2^\mu\left(2-2^{-\mu}\right) - 3 = 2^{\mu+1} - 4.$$

We also have $\sum_{h=0}^{\mu-2} 2 = 2(\mu-1) = 2\mu-2$. Therefore we can compute $S = 2^\mu+2^\mu-1-(2^{\mu+1}-4-2\mu+2) = -1+2+2\mu = 2\mu+1$. By construction, there are μ inner nodes and hence $\mu+1$ leaves. ■

Note that among the $\mu+1$ leaves, at most μ leaves are labelled with 1 (if FHE), resp. 0 (if AHE). For the AHE case, if n is the number of these leaves, then exactly the paths corresponding to them are created in Algorithm 1, evaluated and sent back (with $\mu-n$ random ciphertexts) to the client.

Definition 5.6 *Two cmp-trees are equivalent if they represent the same value, have the same depth and the same number of leaf nodes and inner nodes.*

Lemma 5.7 *The normal cmp-tree of y and a half-pruned cmp-tree of y are equivalent.*

Proof. Given an arbitrary y of length μ. By construction, the normal cmp-tree has depth μ since the leftmost path is the longest one. The same holds for the definition of a half-pruned cmp-tree.

For the normal cmp-tree, there are μ inner nodes on the leftmost path (including the root node). Since every node's right child is a leaf node, there are exactly μ inner nodes. Moreover, we have $\mu + 1$ leaf nodes because every inner node has exactly one child leaf node (the right child) except the deepest inner node where both children are leaf nodes.

Lemma 5.5 shows that a half-pruned tree has the same number of nodes and therefore they are equivalent. ■

Theorem 5.8 *Let y and x be integers of length μ. If the encryption scheme is correct, then the comparison protocol is correct.*

Proof. By Lemma 5.7, we already know that the normal-cmp tree and a half-pruned cmp-tree are equivalent. Moreover, we can transfer one representation into another without changing the result.

By definition, a normal cmp-tree is half-pruned. It remains to show that it is also a cmp-tree. We assume that the encryption scheme is FHE. The case for AHE is similar. If x and y are equal, then x traverses the normal cmp-tree of y on the path as y itself.

Otherwise, x and y have a common prefix that labels a path from the root to a node v with depth h such that y traverses the tree to the left of v while x traverses to the right of v. By construction of the normal cmp-tree, the left edge from v is labelled with the bit $y[h]$, while the right edge is labelled with the bit $1 - y[h]$. Moreover, the right child node of v is a leaf vr labelled with $1 - y[h]$. If $y[h] = 0$ (resp. $y[h] = 1$), then vr is labelled with 1 (resp. 0) and the path to vr evaluates to 1 (resp. 0). On all other paths at least one edge label differs from the bit of x at the same position such that the path evaluates to 0. This is sufficient to conclude whether $x \geq y$ or not.

For the other direction, we have to transfer a half-pruned cmp-tree into a normal cmp-tree. We start at the root node. If the left child node is not a leaf, we proceed with the left child. If not, we switch the below sub-trees and proceed with the left child which is now a leaf node. We repeat this procedure until we reach the tree's depth. The resulting structure fulfills all the requirements of a normal cmp-tree and still represents the same value y. ■

With the normal cmp-tree we have a structure independent of the actual tree which allows the server to compute on ciphertexts without learning anything of input x. This structure is also equivalent to the structure we used for our basic protocol and yields correct results such that we can apply nearly the same routines. The only difference is in the computation of decision bits since the server does not know the edge labels in plaintext. Therefore, we have to apply an inequality/equality test on ciphertexts. For FHE we need an inequality test

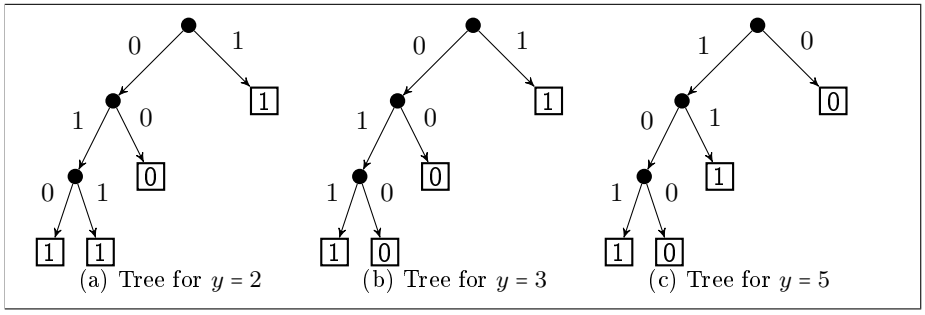


Figure 6: Normal Cmp-Tree for input bit-length $\mu = 3$ and $\beta = 1$

Algorithm 7 Creating a Normal Cmp-Tree for an Integer y.

```
1:  function CreateNormalCmpTree([[y]])
2:      let root be a new node
3:      parse [[y]] to [[y[1]]], ..., [[y[μ]]]
4:      curr ← root
5:      for i = μ downto 1 do
6:          curr.rEdge ← Not([[y[i]]])
7:          let vr be a new node
8:          vr.cLabel ← F_β([[y[i]]])
9:          curr.right ← vr
10:         curr.lEdge ← [[y[i]]]
11:         let vl be a new node
12:         curr.left ← vl
13:         curr ← curr.left
14:     curr.cLabel ← [[β]]
15:     return root
```

which can be implemented using an FHE Xnor gate. For AHE we must perform an equality test which can be implemented using an AHE Xor gate.

5.4 Handling Encrypted Inputs under AHE

The handling of encrypted inputs described above works only for FHE. The reason is that the Xor-operation for AHE encrypted bits requires one bit to be in the clear. In this section, we describe how to extend the previous section to handle the case for AHE. We assume the client sends two encrypted inputs $[\![x]\!]$ and $[\![y]\!]$ to the server. The server creates the normal cmp-tree of $[\![y]\!]$ using Algorithm 7, evaluates the encrypted input $[\![x]\!]$ on the tree and sends back a result that only the client can decrypt. However, the encrypted result is not an encrypted bit, but a set of μ ciphertexts.

The computation needs two basic bit-operations, namely Not and Xor, that have to be simulated under AHE. Let $[\![a]\!]$ and $[\![b]\!]$ be two encrypted bits. We

compute the NOT-operation as $[\![\neg b]\!] = [\![1-b]\!] = \text{ADD}([\![1]\!], \text{MULCONS}([\![b]\!], -1))$. Then, we compute the XOR-operation as $[\![a \oplus b]\!] = [\![a-b]\!]$. Note that while the NOT-operation is correct, this is not always the case for the XOR-operation, namely we have $[\![0 \oplus 1]\!] = [\![-1]\!]$. We will handle this before aggregating the paths.

Recall that we have encrypted bits of x and y and we want to compute a comparison bit. First, using the encrypted input $[\![y]\!]$, we can build the normal cmp-tree as explained in Algorithm 7. This requires only the NOT-operation. Then we can evaluate the bits of $[\![x]\!]$ on the built tree. For that, we first have to apply the XOR-operations on the bits of $[\![x]\!]$ along the paths of the tree and then sum the result along the paths. Our goal is that, if $x \geq y$, then exactly one path will have all XOR-results equal 0 such that the sum along the path is also 0. The remaining paths will have at least one XOR-result that is different to 0 resulting in a sum different to 0.

Now we have the following problem: If the XOR-results of a path contain $[\![1]\!]$ and $[\![-1]\!]$ then this path too may sum to 0. To get rid of the problem, we multiply the XOR-result at level i (i.e., edges starting at a node with depth i) by 2^i before aggregating the results along the paths. Since 2^i is constant, the multiplication can be applied on an AHE ciphertext. The following lemma ensures that the sum on such a path is then always different to 0.

Lemma 5.9 *Let $(b_0, \cdots, b_l) \in \{-1, 0, 1\}^{l+1}$ such that there exist at least one $b_i \neq 0$ then it holds $\sum_{i=0}^{l} b_i \cdot 2^i = b_0 \cdot 2^0 + \cdots + b_l \cdot 2^l \neq 0$.*

Proof. W.l.o.g, we assume $b_l \neq 0$ otherwise we can set $l = l-1$. Now we compare $X = \left|\sum_{i=0}^{l-1} b_i 2^i\right|$ and $Y = |b_l 2^l| = 2^l$. Since $|b_i| \leq 1$, we obtain

$$X \leq \sum_{i=0}^{l-1} 2^i = 2^l - 1 < Y.$$

This concludes that Y strictly dominates X and therefore, the whole sum can never be 0. ■

For example, let $y = 2$, $x = 1$ and $\mu = 3$. Since we are evaluating $[x \geq y]$ under AHE, no path should evaluate to 0. However, the leftmost path of the cmp-tree of $y = 2$ is labelled with $(0, 1, 0, 0)$, where the last entry describes the label of the leaf node. Its evaluation on $(0, 0, 1)$ would result in $(0-0)+(0-1)+(1-0)+0 = 0$. By multiplying with powers of 2 as explained above, we have $(0-0)\cdot 2^0 + (0-1)\cdot 2^1 + (1-0)\cdot 2^2 + 0 = 0 - 2 + 4 + 0 = 2$, which is different to 0 as expected.

5.5 Less Than (LT) Comparison

The computation of the Less-Than (LT) function is similar by using the following definition that is the inverse of the normal cmp-tree.

Definition 5.10 (Inverse Normal Cmp-Tree) *Let y be an integer of length μ. An* inverse normal Cmp-tree *of y is a binary tree with the following structure:*

- *there is a rightmost path p of length μ which is labelled with the bits of y,*
- *the deepest inner node of path p has a left leaf node labelled with β,*
- *each inner node of path p has a left child leaf node,*
- *for each inner node, let b be the label on the right edge, then the label on the left edge is $1-b$ and the label on the left child leaf node is $1-F_\beta(b)$.*

While the inverse normal cmp-tree is defined with a right oriented structure (contrary to the left oriented structure of Definition 5.4), the inverse normal cmp-tree can be represented with a left oriented structure as well. The only difference is that all leaves except the leftmost one must be labelled with $1-F_\beta(b)$ as in Definition 5.10 instead of $F_\beta(b)$ as in Definition 5.4. Hence, the inverse normal cmp-tree can be constructed similar to Definition 5.4 with the exception that the last is updated as follow: for each inner node, let b be the label on the left edge, then the label on the right edge is $1-b$ and the label on the right child leaf node are $1-F_\beta(b)$.

6 Analysis

In the sections above, we proved already that the computation correctly returns 1 if $x \geq y$ and 0 otherwise. The computation is also secure as the server evaluates input encrypted under the client's public key. In this section, we therefore focus on the complexity analysis and count the number of homomorphic operations (addition and multiplication).

6.1 Number of Operations

We start by counting the number of operations depending on the main steps of the algorithm, namely: node evaluation, path evaluation, leaves aggregation. In the following, we use A_1, A_2, A_3 (resp. M_1, M_2, M_3) to denote the number of addition (resp. multiplication) operation in node evaluation, path evaluation, leaves aggregation and A_T (resp. M_T) for the total.

Node Evaluation. For node evaluation at each inner node, the algorithm performs exactly one NOT gate due to the fact that the left and right edges of an inner node are always labelled with opposite bits. For the encrypted case (Section 5.3), we need one NOT and one XOR. Hence, we have in total μ NOT-operations.

Path Evaluation. For path aggregation, the algorithm performs $\mu-1$ multiplications on the leftmost path and 2 multiplications on each right path except the rightmost path that requires only 1 multiplication. This result in total of $\mu-1+2\cdot(\mu-1)+1=3\mu-2$.

		Hom. Add.	Hom. Mult.
Binary Circuit	Node Eval.	2μ	-
	Path Eval.	-	$3\mu - 2$
	Leaves Aggr.	μ	-
	Ours (total)	$\mathbf{3\mu}$	$\mathbf{3\mu - 2}$
	Cheon et al.	$\mathbf{2\mu - 2}$	$\mathbf{2\mu - 3 + \frac{(\mu-1)\log(\mu-1)}{2}}$
Arithmetic Circuit	Node Eval.	2μ	μ
	Path Eval.	-	$3\mu - 2$
	Leaves Aggr.	μ	-
	Ours (total)	$\mathbf{3\mu}$	$\mathbf{4\mu - 2}$
	Cheon et al.	$\mathbf{2\mu - 2}$	$\mathbf{3\mu - 4 + \frac{(\mu-1)\log(\mu-1)}{2}}$

Table 1: Overview Number of Operations in FHE (Encrypted Case). "-" indicates that there is no such operation in this step.

Leaves Aggregation. In the case of FHE, the algorithm finally aggregates the $\mu + 1$ paths requiring μ additions.

6.2 Complexity for FHE

For FHE, we need to distinguish between binary and arithmetic circuit or encoding. An overview can be found in Table 1.

FHE Binary Circuit. For binary encoding, all operations are done modulo 2 such that XOR and NOT operations are implemented as an addition. As a result, we have $A_1 = \mu$ additions in node evaluations, no addition in path evaluation (i.e., $A_2 = 0$) and $A_3 = \mu$ additions during leaves aggregation resulting in a total of $A_T = \mu + 0 + \mu = 2\mu$ additions. For the encrypted case, we need an additional XOR at each node resulting in $A_1 = 2\mu$ additions and in a total of $A_T = 3\mu$ additions. Only path aggregation requires $M_2 = 3\mu - 2$ multiplications ($M_1 = M_3 = 0$), such that $M_T = 0 + (3\mu - 2) + 0 = 3\mu - 2$. As a comparison, the scheme of Cheon et al. [8] requires $2\mu - 2$ additions and $2\mu - 3 + \frac{(\mu-1)log(\mu-1)}{2}$ multiplications using a binary encoding.

FHE Arithmetic Circuit. For arithmetic encoding, the XOR operation $a \oplus b$ is homomorphically computed as $(a-b)^2$, such that each XOR operation requires 1 addition and 1 multiplication. The NOT operation $\neg b$ is computed as $1-b$. As a result, the node evaluation requires $A_1 = \mu$ additions. For the encrypted case, we need $A_1 = 2\mu$ additions and $M_1 = \mu$ multiplications. For path and leaves evaluation, we have $A_2 = 0$, $M_2 = 3\mu - 2$, $A_3 = \mu$, $M_3 = 0$. The evaluation of the tree, therefore, requires $A_T = \mu + 0 + \mu = 2\mu$ additions and $M_T = 0 + (3\mu - 2) + 0 =$

		Hom. Add.	Const. Mult.
Constant Case	Node Eval.	μ	-
	Path Eval.	$2\mu - 2$	-
	Leaves Aggr.	-	-
	Ours (total)	$\mathbf{3\mu - 2}$	-
	Veugen	$\mathbf{4\mu}$	-
Encrypted Case	Node Eval.	2μ	-
	Path Eval.	$3\mu - 1$	2μ
	Leaves Aggr.	-	-
	Ours (total)	$\mathbf{5\mu - 1}$	$\mathbf{2\mu}$
	Veugen	$\mathbf{\geq 5\mu + 3}$	$\mathbf{\geq 2/3\mu^2 + 3\mu + 2/3\mu}$

Table 2: Overview Number of Operations in AHE. "-" indicates that there is no such operation in this step.

$3\mu - 2$ multiplications. In the encrypted case, the total is $A_T = 3\mu$ additions and $M_T = 4\mu - 2$ multiplications.

Comparison to Previous Work. In [8], Cheon et al. assume binary encoding which means that their addition is modulo two, i.e., XOR operation. Using arithmetic encoding, the $2\mu - 2$ additions, therefore, require an addiitonal $\mu - 1$ homomorphic multiplications. As a result, the total number of multiplications in their scheme is $3\mu - 4 + \frac{(\mu-1)log(\mu-1)}{2}$. As our approach, they also achieve a logarithmic depth.

6.3 Complexity for AHE

For AHE, we need to distinguish between the encrypted case where both inputs are encrypted and the the constant case where only one input is encrypted. An overview can be found in Table 2. In both cases, XOR and NOT operations are realized using homomorphic addition. As a result, there are $A_1 = \mu$ homomorphic operations for node evaluation. In the encrypted case, we have $A_1 = 2\mu$ homomorphic operations.

Constant Case. Recall that in this case, we can omit the leaves (See Algorithm 1 and Figure 3). This results in $A_2 = 2\mu - 2$ operations for paths evaluations, i.e., $\mu - 1$ operations for evaluating the leftmost path and one operation for each of the $\mu - 1$ deepest right oriented paths. In total, our scheme requires $A_T = \mu + 2\mu - 2 = 3\mu - 2$ operations. As a comparison, in [11], the DGK scheme performs 5μ additions, μ constant multiplications which is equivalent to 2μ additions. In total, DGK has 7μ additions plus additional μ encryption

operation and μ modular inverse operations. Veugen [36] improved the DGK scheme by requiring only 4μ operations.

Encrypted Case. We now have $A_1 = 2\mu$ homomorphic operations for node evaluation, but $A_2 = 3\mu - 1$ operations for paths evaluation, as we have to consider the leaves. Additionally, we need 2μ constant multiplications to prevent the problem explained in Section 5.4. A constant multiplication $[\![\mathsf{m}]\!] \boxdot n$ requires in worst case $2 \log n$ homomorphic additions. For each level i, we perform in Section 5.4 two multiplications by 2^i resulting in $2 \sum_{i=2}^{\mu-1} i = \mathcal{O}(\mu^2)$ operations which dominates the number of operations for nodes and paths evaluation.

Comparison to Previous Work. As a comparison Veugen also proposed two extensions of the DGK scheme in the encrypted case: a statistical and perfect secure scheme. Both have 2 rounds, i.e., 4 moves between the parties (while our scheme contains still one round as the initial DGK). Both schemes require efficient decryption of a random plaintext and cannot be efficiently implemented using ECC ElGamal (see section on ElGamal). In their scheme the server computes $[\![z]\!] = [\![x - y + 2^\mu + r]\!]$, where the random value r blinds the difference and has length $\mu + 1 + \sigma$ bits for statistical security or r is an arbitrarily long plaintext for perfect security. The client gets and decrypts $[\![z]\!]$ and computes $y' = z \bmod 2^\mu$, while the server computes $x' = r \bmod 2^\mu$. For the statistical security case, the parties then run a normal DGK protocol to compare x' and y', which requires additional homomorphic operations to get the final result.

For the perfect security case, Veugen proposed a modified DGK protocol that is very complex and requires two times the same constant multiplication by 2^i as our scheme and additional operations, such as encryption, decryption and modular inversion, to get the final result. Since the scheme is very complex and the number of operations depends on the actual values, we use a complexity lower bound for a comparison with our scheme. Table 2 shows that our scheme is a significant improvement even to the lower bound of the optimized DGK protocol. As in [36], we assume a homomorphic multiplicative inversion to need $\frac{2}{3}e$ multiplications where e is the bit-length of the number which is a ciphertext in this case. For the inversions used by the modified DGK protocol, this is on average $\frac{\mu}{3}$ multiplications.

7 Implementation

In this section, we describe some implementation details and report on the experimental results of our implementations.

7.1 Optimized Implementation

Instead of implementing our scheme using a binary tree, we can rely on a simpler data structure by using a two dimensional array $a[(1, \ldots, \mu + 1), (1, 2, 3)]$ with $\mu + 1$ rows and three columns. The idea is illustrated in Figure 7 for $x = 1, y =$

$3, \mu = 3, \beta = 1$. The array is initialized with the cmp-tree of $y = 3$, where the first column stores the labels on the leftmost path. Column 2 and 3 store the right oriented paths from the first row to the last one. That is, the last row and the last column store leaf labels, where on the last row, only the first cell if filled.

The evaluation itself is illustrated in Algorithm 8. On each row, we store the equality of current bits of $x[i]$ and $y[i]$ in the first cell, its negation in the second, and $F_\beta(y[i])$ in the third cell. This corresponds to the computation of decision bits and the leaf node labels. Then, for the first row, we multiply the second and third cell and store the result in the third cell. For the remaining rows, we multiply $a[i, 1]$ and $a[i-1, 1]$ and store the result in $a[i, 1]$, which corresponds to the aggregation of the leftmost path. We also multiply $a[i-1, 1]$, $a[i, 2]$ and $a[i, 3]$ and store the result in $a[i, 3]$ corresponding to the evaluations of the paths with right child nodes. Because the first cell of the last row is 1 for FHE (resp. 0 for AHE), it does not change the result already computed so far on the leftmost path which is then just copied in the third cell of the last row. Finally, we have to evaluate the leaves. Summing up the third column yields the result for FHE. For AHE, the third column can be sent back to client permuted and with ciphertexts randomized.

For the FHE case, the multiplicative depth of the procedure is of relevance if the encryption scheme is leveled FHE. This is because a leveled FHE has a fixed parameter L such that circuits with depth at most L can be evaluated without bootstrapping. Therefore, we first evaluate the inner nodes as before by evaluating Xor operations, but use the multiplication with a direct acyclic graph described in [33]. This is illustrated in Figure 8 and consists of first computing a dependency list (DL) table for each element of the matrix (the middle table in Figure 8). The DL is a stack, represented as [) with bottom [and top) that contains cells' numbers along a multiplication path, i.e., the set of cells that must be multiplied together. In Figure 8, we have the following multiplication paths: $(1, 4, 7, 10), (1, 4, 8, 9), (1, 5, 6), (2, 3)$. For each path, we start with a list of nodes. First, we group the elements by pairs and add the first element to the second elements' DL. Then, we reduce the list by all elements that occur in any DL and repeat the procedure until there is only one element left. If a multiplication path consists of nodes a, b, c, d in this order, then the DLs are as follows: $[), [a), [), [b, c)$. Note that the computation of the DL table does not depend on the input but only on the tree structure. In fact, it can be computed once and given as input to the algorithm. While multiplying, we move from top to down and from left to right in the matrix and compute the aggregated result of each cell using its DL. For example, using the DLs $[), [a), [), [b, c)$, there is nothing to do for nodes a and c since their DLs are empty. For node b, we compute $b \leftarrow a \cdot b$. For node d, we first compute $d \leftarrow c \cdot d$ and then $d \leftarrow b \cdot d$. For a path of length k, this procedure reduces the multiplicative depth from k to $\log k$.

7.2 Choice of AHE Scheme

For AHE, we choose ElGamal encryption [12] that we implement as elliptic curve ElGamal (ECE) [22,23]. We briefly describe it in the following and refer to the literature for more details. Let $\mathbb{G}$ be an elliptic curve group over $\mathbb{F}(p^n)$ generated by a point P of prime order p. ECE consists of the following algorithms:

- Key Generation $\mathsf{pk}, \mathsf{sk} \leftarrow \mathsf{KGen}(\lambda)$: This algorithm randomly chooses $s \in \mathbb{Z}_p$ and outputs $\mathsf{sk} = s$ and $\mathsf{pk} = s \cdot P$ as private and public key.
- Encryption $\mathsf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$: This algorithm takes pk and a message m, then it chooses a random $r \in \mathbb{Z}_p$ and outputs the ciphertext $\mathsf{c} = (r \cdot P, \mathsf{m} \cdot P + r \cdot \mathsf{pk})$.
- Decryption $\mathsf{m} \leftarrow \mathsf{Dec}(\mathsf{sk}, \mathsf{c})$: This algorithm takes sk and a ciphertext $\mathsf{c} = (Q_1, Q_2)$, compute $Q = Q_2 - Q_1 \cdot \mathsf{sk}$ and returns the discrete logarithm of Q on $\mathbb{G}$.

The above scheme is indeed AHE. If $\mathsf{c}_1 = (r_1 \cdot P, \mathsf{m}_1 \cdot P + r_1 \cdot \mathsf{pk})$ and $\mathsf{c}_2 = (r_2 \cdot P, \mathsf{m}_2 \cdot P + r_2 \cdot \mathsf{pk})$ are ciphertexts of two plaintexts m_1 and m_2 then $\mathsf{c}_1 + \mathsf{c}_2 = ((r_1 + r_2) \cdot P, (\mathsf{m}_1 + \mathsf{m}_2) \cdot P + (r_1 + r_2) \cdot \mathsf{pk}))$ is a ciphertext of $\mathsf{m}_1 + \mathsf{m}_2$.

While the decryption requires the computation of the discrete logarithm, we stress that in our comparison protocol, computing the discrete logarithm is not necessary as in Algorithm 6, we are looking for a ciphertext of zero. A ciphertext of zero has the form $\mathsf{c} = (r \cdot P, r \cdot \mathsf{pk})$. Hence, checking if the random ciphertext $\mathsf{c} = (Q_1, Q_2)$ is encrypting 0, is efficiently done by computing $Q = Q_2 - Q_1 \cdot \mathsf{sk}$ and then checking if Q is the neutral element of $\mathbb{G}$ that for an elliptic curve is the point at infinity.

7.3 Setup Environment

For AHE, we implemented DGK [11], the optimized DGK by Veugen [36] and our scheme in Java. We instantiated AHE with ElGamal on elliptic curve as described above using curve `secp256r1`. We implemented our scheme in three variants: the naive implementation using tree representation (Section 4), the optimized implementation using array representation (Algorithm 8) and the encrypted case (Section 5). We note that Veugen [36] also proposed a protocol for the encrypted case, which is computationally more complex and no longer one round. For this reason, we did not implement it as it no longer fits with our basic protocol (Protocol 4) and our encrypted case is already theoretically better. It is one round and requires only a constant multiplication (by 2^i as explained above) per bit.

We evaluated our implementation according to our basic protocol which is a one round protocol. That is, the client encrypts its input and sends it to the server. The server evaluates and sends back encrypted result to the client. The client finally decrypts to learn the result. We therefore evaluated all implementations on a single Laptop with a 6-core Intel(R) Xeon(R) E-2176M CPU @ 2.70GHz and 32GB of RAM running Windows 10 Enterprise.

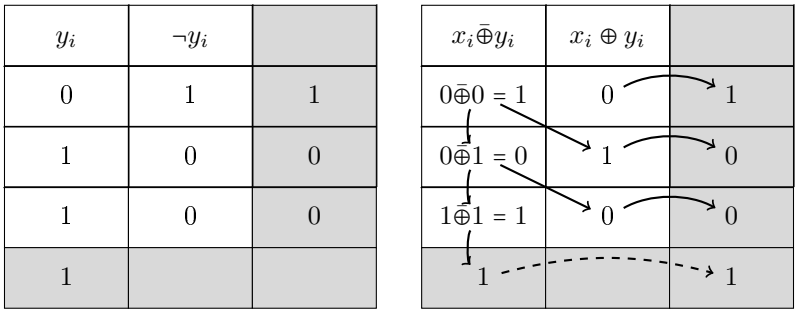


Figure 7: Illustration of the optimized implementation for $x = 1, y = 3, \mu = 3, \beta = 1$. The left table is an array representation of the cmp-tree of $y = 3$, where the first column represents the leftmost path, the second column represents the right oriented paths and the gray cells represent the leaves. The right table illustrates the evaluation of the cmp-tree of $y = 3$ on input $x = 1$. The arrows illustrate paths evaluation, where $a \to b$ means that a and b are aggregated and the result is stored in b. The final results of paths evaluation are stored in the third column, such that leave evaluation is performed by aggregating of the third column. The dashed arrow means that the aggregation result of the leftmost path is copied in the third column.

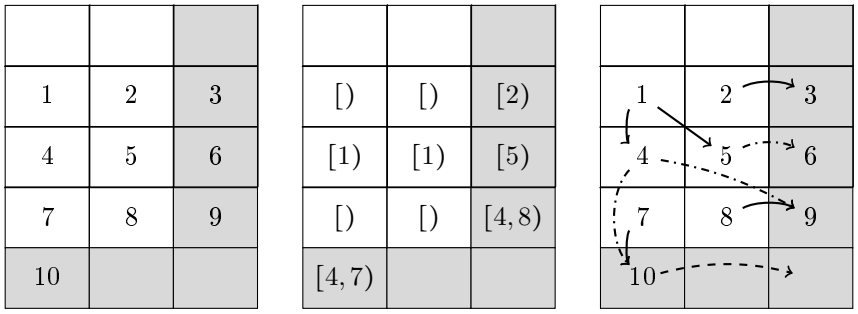


Figure 8: Illustration of the optimized implementation for $\mu = 3$ and levelled FHE. The left table is as in Figure 7 an array representation of the cmp-tree of $y = 3$, where the xor-results are omitted for simplicity, and the cells are numbered. The middle table illustrates dependency lists (DL), where [) stands for a stack with bottom [and top). The right table illustrates as in Figure 7 paths evaluation. The arrows are computed according to the DL, i.e., if cell b has a in its DL, then there is an arrow $a \to b$. The solid (resp. dashed-dotted) arrows have multiplicative depth 1 (resp. 2).

Algorithm 8 Efficient implementation

function EvalCmp($[\![\bar{x}]\!], [\![\bar{y}]\!]$)
  parse $[\![\bar{x}]\!]$ to $[\![x[1]]\!], \ldots, [\![x[\mu]]\!]$
  parse $[\![\bar{y}]\!]$ to $[\![y[1]]\!], \ldots, [\![y[\mu]]\!]$
  let $a[(1, \ldots, \mu+1), (1, \ldots, 3)]$ be a matrix of $(\mu+1) \times 3$ elements
  for $i = 1$ **to** μ **do**
    $a[i,1] \leftarrow \mathsf{comp}([\![x[\mu+1-i]]\!], [\![y[\mu+1-i]]\!])$
    $a[i,2] \leftarrow \text{NOT}(a[i,1])$
    if $\beta = 0$ **then** ▷ Encryption is AHE
      $a[i,1] \leftarrow \text{MulCons}(a[i,1], 2^{i-1})$ ▷ Section 5.4
      $a[i,2] \leftarrow \text{MulCons}(a[i,2], 2^{i-1})$
    $a[i,3] \leftarrow F_\beta([\![y[\mu+1-i]]\!])$
    if $i = 1$ **then**
      $a[i,3] \leftarrow \mathsf{DecBitAgg}(a[i,3], a[i,2])$
    else
      $a[i,1] \leftarrow \mathsf{DecBitAgg}(a[i,1], a[i-1,1])$
      $a[i,3] \leftarrow \mathsf{DecBitAgg}(a[i,3], \mathsf{DecBitAgg}(a[i,2], a[i-1,1]))$
  $a[\mu+1,3] \leftarrow a[\mu,1]$
  if $\beta = 1$ **then** ▷ Encryption is FHE
    return $\sum_{i=1}^{\mu+1} a[i,3]$
  else ▷ Encryption is AHE
    let $\mathsf{c}[1, \ldots, \mu+1]$ be an array
    for $i = 1$ **to** $\mu+1$ **do**
      $\mathsf{c}[i] \leftarrow \mathsf{randomize}(a[i,3])$
    return $\mathsf{permute}(\mathsf{c})$

		$\mu = 8$	$\mu = 16$	$\mu = 32$	$\mu = 64$	$\mu = 96$	$\mu = 128$
DGK [11]		27.54	48.20	101.24	180.21	267.48	374.48
Veugen [36]		16.12	29.13	60.50	109.87	165.38	232.16
Ours	Naive	16.32	28.80	59.90	104.06	152.37	215.86
	Optimized	14.31	25.85	56.13	101.87	152.63	207.36
	Encrypted	18.25	33.33	71.61	132.87	216.84	306.29

Table 3: Performance Comparison of Running Time in milliseconds

7.4 Results

For all three schemes (DGK, Veugen constant case and our scheme), the communication is the same, i.e., number of ciphertexts (μ ciphertexts from client and μ ciphertexts from server) sent times the length of a ciphertext. However, for the encrypted case, the scheme in [36] additionally requires Paillier encryption to encrypt large randomized plaintexts. This cannot be done with additive ElGamal, as decryption requires computing the discrete logarithm over a large domaine. Veugen's Scheme additionally sends few Paillier ciphertexts and requires 2 rounds instead of one in the encrypted case. The above is also true for client computation effort. In DGK, constant case Veugen and our scheme, the client encrypts μ plaintext bits and decrypts μ ciphertexts. We therefore focus our evaluation on the server computation.

To evaluate the running time, we generated random inputs x and y and compare them using each protocols at security level 128. We repeated the experiment 100 times and computed the average running time which is illustrated in Table 3, for input bit-length $\mu = 8, 16, 32, 64, 96, 128$. While Veugen scheme clearly performs better than the original DGK scheme, our naive implementation is most of the time better than Veugen scheme (except for the case $\mu = 8$) and our optimized implementation always better. Although our encrypted case requires additional constant multiplications per bits, it still performs better than the original DGK scheme. Veugen's scheme in the encrypted case additionally requires few Paillier ciphertext operations to the both client and server.

The scheme works as followed. The server holds $[\![x]\!]$ and $[\![y]\!]$ encrypted under Paillier with modulus $N = pq$, where p and q are large primes. The server chooses a random number r, such $0 \leq r < N$, computes $[\![z]\!] \leftarrow [\![x - y + 2^\mu + r]\!] = [\![x]\!] \cdot [\![y]\!]^{-1} \cdot [\![2^\mu + r]\!] \bmod N^2$ and sends it to client, which the client decrypts to get z. Then the server with $\alpha = r \bmod 2^\mu$ and the client with $\beta = z \bmod 2^\mu$ run a modified DGK protocol resulting in the parties learning shares δ_S and δ_C of the comparison bit $[\beta < \alpha]$. The client then encrypts and sends $[\![z/2^\mu]\!]$ and $[\![\delta_C]\!]$. The server finally computes the encrypted comparison bit for $[\beta < \alpha]$ as $[\![[\beta < \alpha]]\!] = \left[\!\left[1 + (-1)^{\delta_S} + (-1)^{1-\delta_S}\delta_C\right]\!\right]$ and the final comparison bit as $[\![[x < y]]\!] = [\![z/2^\mu]\!] \cdot ([\![r/2^\mu]\!] \cdot [\![[\beta < \alpha]]\!])^{-1} \bmod N^2$. All the operations described above are done under Paillier encryption. We implemented and evaluated them at security level 128 (i.e., bit-length of the modulus N is 3072) on a single laptop as described above. These extra Paillier operations require on average 600 milliseconds which almost double our running time for $\mu = 128$. Note that the network cost for the extra protocol round is not included.

8 Applications

Secure integer comparison is a fundamental building block in many multi-party computation protocols. In this section, we describe few applications where our scheme can be useful. The following description is of course not exhaustive and gives only an overview of applications.

Machine Learning. Many applications in machine learning require integer comparison. For example, a decision tree is a common and very popular classifier that consists of decision nodes, each marked with a test condition, and leaf nodes, each marked with a classification label. Each test condition is actually of a greater-than comparison between a threshold value and an attribute of the input to be classified. In private decision trees setting, a server holds a private decision tree and a client holds a private attribute vector. The goal is to classify the client's attribute vector using the server's decision tree model such that the result of the classification is revealed only to the client and nothing else is revealed neither to the client nor the server. Wu et al. [37] and Tai et al. [31] proposed a private decision tree protocol, that uses the DGK comparison and AHE. Based on the idea of Tai et al. [31], Tueno et al. [33] proposed a non-interactive protocol that uses the comparison scheme of Cheon et al. [8]. In [34], Tueno et al. proposed an application for range queries that uses search tree structure to implement order-preserving encryption (OPE). They overcame the limitation of private-key OPE by using garbled circuit or DGK comparison to traverse the search tree.

Benchmarking and Auction. In this case, the goal is to securely compute the k^{th}-ranked element in a distributed setting. That is, given n parties each holding a private integer, the problem is to securely compute the element ranked k (for a given k such that $1 \leq k \leq n$) among these n integers. The computation should reveal to the parties only the k^{th}-ranked element (or the index of party holding it) and nothing else. The computation of the k^{th}-ranked element has applications in benchmarking, where a company is interested in knowing how well it is doing compared to others, or in auctions where bidders are interested in knowing the highest bid. In fact, the DGK protocol were proposed with online auction as application [11]. Other work, including [3], [4], [35], have proposed protocols for computing the k^{th}-ranked element using the DGK comparison protocol. In [35], Tueno et al. also proposed a variant of their protocol based on SHE/FHE and using the comparison scheme of Cheon et al. [8].

9 Related Work

In his seminal paper [38], Yao introduced the millionaires' problem where two millionaires are interested in knowing which of them is richer without revealing their actual wealth. The underlying functionality takes two integers and returns

to the parties a comparison bit. Later, different alternatives of securely comparing integers have been proposed. In [25,26] Kolesnikov et al. proposed optimized boolean circuits that are tailored for garbled circuits. In [11], Damgård et al. proposed the so-called DGK protocol, where the client holding a private-public key pair sends its input bitwise encrypted using AHE. The server holding only the public key homomorphically computes $[\![z_1]\!], \ldots, [\![z_\mu]\!]$ where

$$z_i = s + x[i] - y[i] + 3 \sum_{j=i+1}^{\mu} (x[j] \oplus y[j]).$$

The variable s can be set either to 1 (for LT) or -1 (for GT) and allows secret-sharing the comparison bit. Finally, the server randomizes each ciphertext $[\![z_i]\!]$ and sends them back to the client in a random order. In [36], Veugen improved the DGK protocol and proposed different variants such as the case where both inputs are encrypted. In [27], Lin and Tzeng introduced another protocol using AHE, where the parties compute for each bit position so-called 0-encoding and 1-encoding. Client's encoding is sent encrypted to the server that evaluates by just performing homomorphic additions with its own encoding. This scheme has later been improved in [35]. Fischlin [14] also proposed a similar protocol but using Goldwasser-Micali encryption scheme [20]. Other protocols using AHE or a so-called arithmetic black-box[1] include [2, 6, 10, 15, 28, 29, 32]. In [7, 8], Cheon et al. proposed a comparison circuit that can be evaluated using SHE/FHE. They compute $[\![c_1]\!], \ldots, [\![c_\mu]\!]$ and return $[\![c_\mu]\!]$ as the comparison bit, where

$$\begin{cases} c_1 = (1 \oplus x[1]) \cdot y[1], \\ c_i = ((1 \oplus x[i]) \cdot y[i]) \oplus ((1 \oplus x[i] \oplus y[i]) \cdot c_{i-1}), \text{ for } i > 1. \end{cases}$$

Gentry et al. [17] proposed a scheme that reduces GT to zero testing using somewhat homomorphic encryption. Based on the observation of [28, 32] that the comparison of two strings can be reduced to comparing the first equal length sub-strings on which they differ, Couteau [9] proposed a scheme that uses exclusively oblivious transfer on short strings [24], instead of public-key primitives.

In summary, the schemes described above require access to the bit representation of the integers and perform operations on private data using garbled circuit, homomorphic encryption or additive secret sharing. Several schemes have a constant number of rounds but require a complexity that is at least linear in the bit-length of the integers [2, 8, 11, 14, 25–27]. The scheme of Garay et al. [15] and the scheme of Gentry et al. [17] have a logarithmic (in the input bit-length) number of rounds and a linear complexity, but in contrast to [2, 11, 14, 27] they output an encrypted comparison bit. The scheme of Couteau [9] has a log-logarithmic (in the input bit-length) number of rounds and a linear complexity, but outputs secret shares of the comparison bit.

[1] The arithmetic black-box (ABB) is an ideal functionality that performs basic arithmetic operations (i.e., addition and multiplication). While any party can in private specify input to the ABB, only a majority of parties can ask to perform any feasible computation and make (only) the result public. The ABB itself can be implemented using additive secret sharing or AHE.

10 Conclusion

We proposed a new protocol for secure integer comparison of two parties using the evaluation of binary trees. Our approach is based on HE and is a non-interactive solution which can be used for a broad range of applications or as a subroutine for larger protocols. We theoretically presented an FHE and an AHE mode with several extensions and optimizations and implemented both variants using improved data representations and evaluations to reduce the computational overhead.

References

[1] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, New York, NY, USA, 1988. ACM.

[2] I. F. Blake and V. Kolesnikov. Strong conditional oblivious transfer and computing on intervals. In *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 2004.

[3] E. Blass and F. Kerschbaum. Secure computation of the k^{th}-ranked integer on blockchains. *IACR Cryptology ePrint Archive*, 2019:276, 2019.

[4] E. Blass and F. Kerschbaum. BOREALIS: building block for sealed bid auctions on blockchains. In H. Sun, S. Shieh, G. Gu, and G. Ateniese, editors, *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, pages 558–571. ACM, 2020.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *ECCC*, 18:111, 2011.

[6] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *Proceedings of the 7th International Conference on Security and Cryptography for Networks*, SCN'10, pages 182–199, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] J. H. Cheon, M. Kim, and M. Kim. Search-and-compute on encrypted data. In *FC*, pages 142–159, 2015.

[8] J. H. Cheon, M. Kim, and K. E. Lauter. Homomorphic computation of edit distance. In *FC*, pages 194–212, 2015.

[9] G. Couteau. New protocols for secure equality test and comparison. In *ACNS*, volume 10892 of *Lecture Notes in Computer Science*, pages 303–320. Springer, 2018.

[10] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 285–304, 2006.

[11] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430, 2007.

[12] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[13] D. Evans, V. Kolesnikov, and M. Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 2018.

[14] M. Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In *CT-RSA*, pages 457–472, 2001.

[15] J. Garay, B. Schoenmakers, and J. Villegas. Practical and secure solutions for integer comparison. In *Proceedings of the 10th International Conference on Practice and Theory in Public-key Cryptography*, PKC'07, pages 330–342, Berlin, Heidelberg, 2007. Springer-Verlag.

[16] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, New York, NY, USA, 2009. ACM.

[17] C. Gentry, S. Halevi, C. S. Jutla, and M. Raykova. Private database access with he-over-oram architecture. In *ACNS*, volume 9092 of *Lecture Notes in Computer Science*, pages 172–191. Springer, 2015.

[18] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[19] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229, New York, NY, USA, 1987. ACM.

[20] S. Goldwasser and S. Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[21] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[22] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, Jan. 1987.

[23] N. Koblitz, A. Menezes, and S. A. Vanstone. The state of elliptic curve cryptography. *Des. Codes Cryptography*, 19(2/3):173–193, 2000.

[24] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2013.

[25] V. Kolesnikov, A. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, pages 1–20, 2009.

[26] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.

[27] H. Lin and W. Tzeng. An efficient solution to the millionaires' problem based on homomorphic encryption. In *ACNS*, pages 456–466, 2005.

[28] H. Lipmaa and T. Toft. Secure equality and greater-than tests with sublinear online complexity. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II*, ICALP'13, pages 645–656, Berlin, Heidelberg, 2013. Springer-Verlag.

[29] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Proceedings of the 10th International Conference on Practice and Theory in Public-key Cryptography*, PKC'07, pages 343–360, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99*, pages 223–238. Springer-Verlag, 1999.

[31] R. K. H. Tai, J. P. K. Ma, Y. Zhao, and S. S. M. Chow. Privacy-preserving decision trees evaluation via linear functions. In *ESORICS*, pages 494–512, 2017.

[32] T. Toft. Sub-linear, secure comparison with two non-colluding parties. In *Public Key Cryptography*, volume 6571 of *Lecture Notes in Computer Science*, pages 174–191. Springer, 2011.

[33] A. Tueno, Y. Boev, and F. Kerschbaum. Non-interactive private decision tree evaluation. In *Data and Applications Security and Privacy XXXIV - 34th Annual IFIP WG 11.3 Conference, DBSec 2020, Regensburg, Germany, June 25-26, 2020, Proceedings*, pages 174–194, 2020.

[34] A. Tueno and F. Kerschbaum. Efficient secure computation of order-preserving encryption. In *Proceedings of the 2020 on Asia Conference on Computer and Communications Security*, ASIACCS '20, 2020.

[35] A. Tueno, F. Kerschbaum, S. Katzenbeisser, Y. Boev, and M. Qureshi. Secure computation of the kth-ranked element in a star network. In *Financial Cryptography and Data Security (FC)*, 2020.

[36] T. Veugen. Improving the DGK comparison protocol. In *WIFS*, pages 49–54, 2012.

[37] D. J. Wu, T. Feng, M. Naehrig, and K. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, 2016.

[38] A. C. Yao. Protocols for secure computations. In *SFCS '82*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.