

A Method for Securely Comparing Integers using Binary Trees

Anselme Tueno
SAP SE
Germany
anselme.tueno@sap.com

Jonas Janneck
Ruhr University Bochum
Germany
jonas.janneck@rub.de

David Boehm
SAP SE
Germany
david.boehm@sap.com

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%.

KEYWORDS

homomorphic encryption, multi-party computation, integer comparison

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 [5, 21, 27–29]. A classic example in the literature is the so-called Yao's Millionaire's problem introduced in [53]. 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 [42, 45, 46, 51] or neural networks. In secure enterprise benchmarking [48], key performance indicators are securely compared to determine how companies perform compared to their competitors. In secure auction [8, 9], 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 [18] and reduce the running time by 45%. Compared to an optimization of the DGK protocol proposed by Veugen [49], we can reduce the running time by about 10% for the first variant. However, for the second variant in which both inputs are encrypted we achieve a reduction of more than 63%.

In total, our contributions are: We propose a new protocol for secure integer comparison based on binary trees. We present two instantiations of our protocol using FHE and AHE. We extend our protocol to handle shared output bit, encrypted inputs and less than comparison. We give a theoretical analysis including specifications for the two variants and their complexity implications. We implement and evaluate both variants.


The remainder of the paper is structured as follows. We review preliminaries in Section 2 and related work in Section 3. Section 4 defines correctness and security of the functionality. We describe our protocol and its algorithms in Section 5 and present some extensions in Section 6. Section 8 gives details about our implementation and evaluation results.

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 [12, 25].

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

- $pk, sk, ek \leftarrow \text{KGen}(\lambda)$: This probabilistic algorithm takes a security parameter λ and outputs public, private, and evaluation keys pk, sk , and ek .
- $c \leftarrow \text{Enc}(pk, m)$: This algorithm takes pk and a message m and outputs a ciphertext c . We will use $\llbracket m \rrbracket$ as a shorthand notation for $\text{Enc}(pk, m)$.
- $c \leftarrow \text{Eval}(ek, f, c_1, \dots, c_n)$: This algorithm takes ek , an n -ary function f and n ciphertexts c_1, \dots, c_n and outputs a ciphertext c .
- $m' \leftarrow \text{Dec}(sk, c)$: This deterministic algorithm takes sk and a ciphertext c and outputs a message m' .

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit <https://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Proceedings on Privacy Enhancing Technologies 2023(3), 469–487
© 2023 Copyright held by the owner/author(s).
<https://doi.org/10.56553/popets-2023-0092>

We require IND-CPA security and the following correctness conditions. Given any set of n plaintexts m_1, \dots, m_n , it must hold for any pk, sk, ek :

- $\text{Dec}(sk, \text{Enc}(pk, m_i)) = \text{Dec}(sk, \llbracket m_i \rrbracket) = m_i$,
- $\text{Dec}(sk, \text{Eval}(ek, f, \llbracket m_1 \rrbracket, \dots, \llbracket m_n \rrbracket)) = \text{Dec}(sk, \llbracket f(m_1, \dots, m_n) \rrbracket)$.

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 m_1, m_2 , we define the following operations and introduce shorthand notations:

- Addition: $\text{ADD}(\llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket) = \llbracket m_1 \rrbracket \boxplus \llbracket m_2 \rrbracket = \llbracket m_1 + m_2 \rrbracket$,
- Constant Addition : $\text{ADDCONS}(\llbracket m_1 \rrbracket, m_2) = \llbracket m_1 \rrbracket \boxplus m_2 = \llbracket m_1 + m_2 \rrbracket$,
- Multiplication: $\text{MUL}(\llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket) = \llbracket m_1 \rrbracket \boxtimes \llbracket m_2 \rrbracket = \llbracket m_1 \cdot m_2 \rrbracket$,
- Constant Multiplication: $\text{MULCONS}(\llbracket m_1 \rrbracket, m_2) = \llbracket m_1 \rrbracket \boxtimes m_2 = \llbracket m_1 \cdot m_2 \rrbracket$.

Additively HE. If the scheme supports only addition, then it is called *additively HE (AHE)*. Schemes such as Paillier [41] or Elliptic Curve ElGamal [33] are additively homomorphic and have the following properties for all integer plaintexts m_1, m_2 and bit plaintexts $a, b \in \{0, 1\}$:

- Addition: $\text{ADD}(\llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket) = \llbracket m_1 \rrbracket \boxplus \llbracket m_2 \rrbracket = \llbracket m_1 + m_2 \rrbracket$,
- Constant Multiplication: $\text{MULCONS}(\llbracket m_1 \rrbracket, m_2) = \llbracket m_1 \rrbracket \boxtimes m_2 = \llbracket m_1 \cdot m_2 \rrbracket$,
- Xor: $\text{XOR}(\llbracket a \rrbracket, b) = \llbracket a \oplus b \rrbracket$.

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.

3 RELATED WORK

In his seminal paper [53], Yao introduced the millionaires' problem and the first protocol to securely compare two integers. Later, different alternatives of securely comparing integers have been proposed. In [35, 36] Kolesnikov et al. proposed schemes that use garbled circuits. In [18], Damgård et al. proposed the so-called DGK protocol, where the parties must evaluate $z_i = s + x[i] - y[i] + 3 \sum_{j=i+1}^{\mu} (x[j] \oplus y[j])$ on the input bit encrypted with AHE. DGK has been improved by Veugen [49] and Joye and Salehi [31]. Similar protocols to the DGK, that rely on AHE, include Lin and Tzeng [37, 48], Fischlin [22], Blake and Kolesnikov [6], Garay et. al. [24]. Other protocols [13, 17, 38, 40, 43] and most recent ones [4, 20, 39]

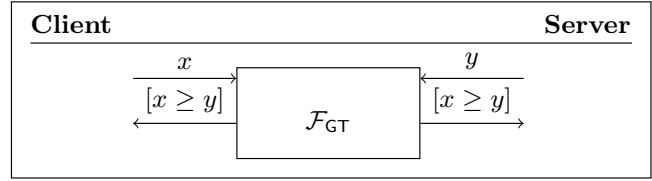


Figure 1: The GT functionality

are based on the arithmetic black-box model which is a powerful tool commonly used in MPC frameworks such as MP-SPDZ [32] or SCALE-MAMBA [1]. In [14, 15], Cheon et al. proposed a scheme where the following circuit is evaluated using SHE/FHE: $c_i = ((1 \oplus x[i]) \cdot y[i]) \oplus ((1 \oplus x[i] \oplus y[i]) \cdot c_{i-1})$, for $i > 1$. In summary, all these schemes require access to the bit representation of the integers. Some schemes have a constant number of rounds [6, 15, 18, 22, 35–37]. Other schemes have log-logarithmic number of rounds [16, 24, 26]. In Section 8, we analyze the closely related work in more detail and compare it to our approach. More recent work include [11, 30] that reduce to comparison of small integers, using a base $p > 2$ representation instead of the binary representation. Bourse et al. [11] Use a specific AHE construction based on Factoring and cannot support comparison of two encrypted input, and sharing of the output bit. We do require any specific AHE, but can implement our scheme using ECC ElGamal resulting in a faster scheme with smaller communication. Iliashenko and Zucca [30] use specific structure of BGV and BFV to design faster schemes. We do not assume a specific FHE scheme, have a smaller circuit depth and can compare more integers using SIMD.

4 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}_{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. This is formalized using the ideal/real world paradigm, where for each party there must exist 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 [27]. To formalize this, we need the following definition [27]. A function $\nu : \mathbb{N} \rightarrow \mathbb{R}$ is *negligible* if for every

positive polynomial $p(\cdot)$ there exists an ε such that for all $n \in \mathbb{N}$ with $n > \varepsilon$: $v(n) < 1/p(n)$. 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 [27]. 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}_{GT} in the semi-honest model if each party learns only its output and nothing else. In particular, there must exist simulators Sim_C^{gt} and Sim_S^{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}_{GT} , and let $\text{View}_P^{\Pi_{GT}}(x, y)$ denote the view of party P during the protocol, then the following hold:

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

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

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

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

5 OUR PROTOCOL

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

5.1 Intuition

Our key observation is that the comparison problem can be seen as a classification problem using a decision tree, where the tree is built on the server input and evaluated using the client input. That is, given a server's input y , a client's input x classifies as 1 if $x \geq y$ or 0 otherwise. The tree edges are labelled with bits such that the evaluation is done by traversing the tree along the path of x . However, we do not know the path representing x - since it is encrypted - and therefore have to evaluate all paths and aggregate the results.

Evaluating a path on encrypted inputs means computing bit equality between edge labels and the bits of x and adding or multiplying the results along the path. For example, let $y = 3$ whose tree is illustrated in Figure 2. We have the following paths $P_0 = (0, 0, -, 0), P_1 = (0, 1, 0, 0), P_2 = (0, 1, 1, 1), P_3 = (1, -, -, 1)$, where the last element in each represents the leaf label and missing

edges in shorter paths are represented with the symbol " - " that will be ignored. Let $x = 2$, i.e., binary vector $(0, 1, 0)$, we evaluate as follows: $([0 = 0], [0 = 1], -, 0), ([0 = 0], [1 = 1], [0 = 0], 0), ([0 = 0], [1 = 1], [1 = 0], 1), ([1 = 0], -, -, 1)$ resulting in $(1, 0, -, 0), (1, 1, 1, 0), (1, 1, 0, 1), (0, -, -, 1)$. By multiplying the elements in each vector $(1 \cdot 0 \cdot 0), (1 \cdot 1 \cdot 1 \cdot 0), (1 \cdot 1 \cdot 0 \cdot 1), (0 \cdot 1)$, we get $0, 0, 0, 0$ which finally results to 0 after adding all four zeros. Similarly for $x = 4$, i.e., $(1, 0, 0)$, we have $(0 \cdot 1 \cdot 0) + (0 \cdot 0 \cdot 1 \cdot 0) + (0 \cdot 0 \cdot 0 \cdot 1) + (1 \cdot 1) = 1$. Alternatively, we can instead label the leaves for $x' \geq y$ with 0, otherwise with 1, then compute bit inequality and check if at most one path sums to 0. That is, for $x = 2, y = 3$ we have $0+1+1 = 2, 0+0+0+1 = 1, 0+0+1+0 = 1, 1+0 = 1$; and for $x = 4, y = 3$ we have, $1 + 0 + 1 = 2, 1 + 1 + 0 + 1 = 3, 1 + 1 + 1 + 0 = 3, 0 + 0 = 0$.

In the tree, paths may have common prefixes which allows to avoid considering each path separately and thus reducing the number of operations. Moreover, if y is not encrypted, i.e., labels of tree edge are not encrypted, and we aggregate the paths multiplicatively (resp. additively) then it is enough to evaluate only paths that lead to a classification 1 (resp. 0), reducing the computation even further. This is because, if $y > x$ exactly one of those paths will aggregate to 1 (resp. 0), otherwise they will all aggregate to 0 (resp. non zero).

In our schemes, we compute the bit equality using XOR or XNOR. This is possible with AHE only if one of the bits is not encrypted and therefore requires the server to know y in the clear. Previous work solve this by adding a computation round which allows the parties to reduce the problem to the comparison of two random inputs, which preserve the order relation between the initial x and y . If x, y are encrypted, we compute the bit equality using a subtraction, without an extra round. This eventually introduces an error $(0-1 = -1)$ which we correct before aggregating the results on the path.

For (leveled) FHE, computing prefixes by simply multiplying values is inefficient, as the multiplicative depth matters. Our scheme keeps a logarithmic (instead of linear) multiplicative depth at the cost of a low overhead on the number of multiplications. We do this by pre-computing dependency lists for multiplication as in [45]. This pre-computation depends only on the bitlength (and not on the inputs), and its result is unique for a given bitlength. It can be computed once and offline and later used as a constant parameter in the online protocol.

5.2 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 consisting of the following:

- $v.\text{parent}$: a value representing the pointer to the parent node,
- $v.\text{left}$: a value representing the pointer to the left child node,
- $v.\text{right}$: a value representing the pointer to the right child node,
- $v.l\text{Edge}$: a bit representing the *edge label* to the left child node,
- $v.r\text{Edge}$: a bit representing the *edge label* to the right child node,
- $v.c\text{Label}$: a value representing a *node label*,

- $v.cost$: an integer representing the cost on the path from the root.

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

5.3 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 $\bar{x} = x[1], \dots, x[\mu]$ to denote the corresponding bit representation, where $x[\mu]$ is the most significant bit, and we use $\llbracket \bar{x} \rrbracket = \llbracket x[1] \rrbracket, \dots, \llbracket x[\mu] \rrbracket$ to denote the bitwise encryption of \bar{x} .

Initialization. The Initialization consists of a one time key generation. The client generates an appropriate triple (pk, sk, ek) of public, private and evaluation key for an HE scheme. Then, the client sends (pk, 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., $v.cLabel = 1$). The leaves of the paths left to y are labelled with 0 (i.e., $v.cLabel = 0$). 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., $v.cLabel = b$). 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 $cLabel = 0$, 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 $cLabel = 1$ 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 and 5 if the scheme is for FHE and Figure 3 illustrates the binary trees of 2, 3, and 5 if the scheme is for 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 $\bar{x} = x[1], \dots, x[\mu]$ and then sends the corresponding ciphertext $\llbracket \bar{x} \rrbracket = \llbracket x[1] \rrbracket, \dots, \llbracket x[\mu] \rrbracket$ to the server. The server computes the decision bits at each inner node v by comparing each

Algorithm 1 Creating the Binary Tree for Integer y .

```

1: let root be a new node
2: let curr be an empty node
3: let  $y[1], \dots, y[\mu]$  be the bit string of  $y$ 
4:  $curr \leftarrow root$ 
5: for  $i = \mu$  downto 1 do
6:   if  $y[i] = 1$  then
7:     let  $v$  be a new node
8:      $curr.right \leftarrow v$ 
9:     if  $\beta = 1$  then {Only FHE}
10:      let  $v$  be a new node
11:       $v.cLabel \leftarrow 1 - \beta$ 
12:       $curr.left \leftarrow v$ 
13:   end if
14:    $curr \leftarrow curr.right$ 
15: else
16:   let  $v$  be a new node
17:    $curr.right \leftarrow v$ 
18:   if  $\beta = 1$  then {Only FHE}
19:      $v.cLabel \leftarrow \beta$ 
20:   end if
21:   let  $v$  be a new node
22:    $curr.left \leftarrow v$ 
23:    $curr \leftarrow curr.left$ 
24: end if
25: end for
26: if  $\beta = 1$  then {Only FHE}
27:    $curr.cLabel \leftarrow \beta$ 
28: end if
29:  $root.cost \leftarrow \llbracket \beta \rrbracket$ 
30:
31: return root

```

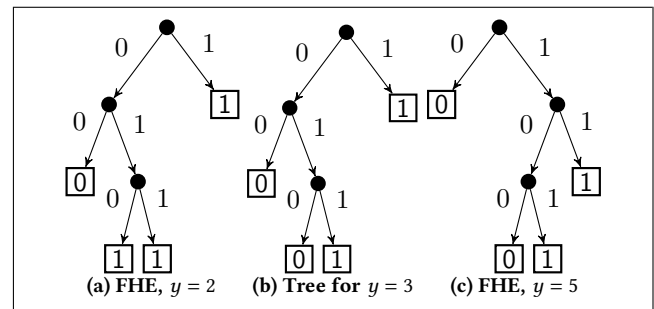


Figure 2: Tree Creation for FHE ($\beta = 1$) and $\mu = 3$

$\llbracket x[i] \rrbracket$ 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 $\llbracket 1 \rrbracket$ if the two bits are equal and $\llbracket 0 \rrbracket$ otherwise. For AHE, it is implemented as an inequality test that returns $\llbracket 0 \rrbracket$ if the two bits are equal and $\llbracket 1 \rrbracket$ otherwise. The operation can be computed by at least one 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 $\llbracket x[i] \rrbracket$ as the comparison result. If it is 0, we compute $\llbracket \neg x[i] \rrbracket$. The computation of decision

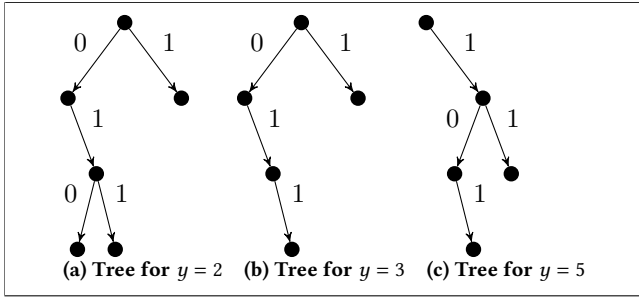


Figure 3: Tree Creation for AHE ($\beta = 0$) and $\mu = 3$

Algorithm 2 EVALNODES

```

Require: root,  $\llbracket \bar{x} \rrbracket$ 
1: parse  $\llbracket \bar{x} \rrbracket$  to  $\llbracket x[1] \rrbracket, \dots, \llbracket x[\mu] \rrbracket$ 
2:  $v \leftarrow$  root
3: for  $i = \mu$  downto 1 do
4:   if  $v.left \neq$  null then
5:      $\llbracket v.left.cost \rrbracket \leftarrow$  comp( $\llbracket x[i] \rrbracket, v.lEdge$ )
6:   end if
7:   if  $v.right \neq$  null then
8:      $\llbracket v.right.cost \rrbracket \leftarrow$  comp( $\llbracket x[i] \rrbracket, v.rEdge$ )
9:   end if
10:  if  $v.right \neq$  null and  $v.right.isLeaf() =$  false then
11:     $v \leftarrow v.right$ 
12:  else
13:     $v \leftarrow v.left$ 
14:  end if
15: end for
    
```

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: BitAgg. 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 nodes, each leaf node v stores either $\llbracket v.cost \rrbracket = \llbracket 0 \rrbracket$ or $\llbracket v.cost \rrbracket = \llbracket 1 \rrbracket$. Moreover, there is a unique leaf with $\llbracket v.cost \rrbracket = \llbracket 1 \rrbracket$ and all other leaves have $\llbracket v.cost \rrbracket = \llbracket 0 \rrbracket$. Then, the server aggregates the costs at the leaves by computing for each leaf v the value $\llbracket v.cost \rrbracket \boxtimes \llbracket v.cLabel \rrbracket$ and summing all the results of all leaves. This

Algorithm 3 EVALPATHS

```

Require: root
1: let  $Q$  be a queue
2: let leaves be a queue
3:  $Q.enqueue(root)$ 
4: while  $Q.empty() =$  false do
5:    $v \leftarrow Q.dequeue()$ 
6:   if  $v.left \neq$  null then
7:      $\llbracket v.left.cost \rrbracket \leftarrow$ 
       BitAgg( $\llbracket v.left.cost \rrbracket, \llbracket v.cost \rrbracket$ ),
8:     if  $v.left.isLeaf() =$  true then
9:       leaves.enqueue( $v.left$ )
10:    else
11:       $Q.enqueue(v.left)$ 
12:    end if
13:  end if
14:  if  $v.right \neq$  null then
15:     $\llbracket v.right.cost \rrbracket \leftarrow$ 
      BitAgg( $\llbracket v.right.cost \rrbracket, \llbracket v.cost \rrbracket$ ),
16:    if  $v.right.isLeaf() =$  true then
17:      leaves.enqueue( $v.right$ )
18:    else
19:       $Q.enqueue(v.right)$ 
20:    end if
21:  end if
22: end while
23:
24: return leaves
    
```

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 a cost which is either $\llbracket v.cost \rrbracket = \llbracket 0 \rrbracket$ or $\llbracket v.cost \rrbracket = \llbracket r \rrbracket$, where r is the number of 1s on a path. Moreover, there is at most one leaf with $\llbracket v.cost \rrbracket = \llbracket 0 \rrbracket$ and all other leaves have $\llbracket v.cost \rrbracket = \llbracket r \rrbracket$, for an unknown $r \in \{1, \dots, \mu\}$. Note that for $y \neq 0$ the number of paths is smaller or equal to μ (See Figure 3a, 3c). The server randomizes the encrypted costs at the leaves, chooses other random ciphertexts not encrypting zero, 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 indicating the comparison result. For AHE, the evaluation result consists of μ ciphertexts among which at most one encrypts 0 and the remaining ones encrypt random plaintexts. The comparison result is true iff there is an encryption of 0. The client uses Algorithm 6 to decrypt and learn the final result.

Algorithm 4 EVALLEAVES (FHE)

Require: leaves

```

1:  $\llbracket b \rrbracket \leftarrow \llbracket 0 \rrbracket$ 
2: for all  $v \in \text{leaves}$  do
3:    $\llbracket b \rrbracket \leftarrow \llbracket b \rrbracket \boxplus (\llbracket v.\text{cost} \rrbracket \boxtimes \llbracket v.\text{cLabel} \rrbracket)$ 
4: end for
5:
6: return  $\llbracket b \rrbracket$ 

```

Algorithm 5 EVALLEAVES (AHE)

Require: leaves

```

1: let  $c$  be an array of ciphertexts
2: for  $i = 1$  to  $\text{leaves.size}()$  do
3:    $c[i] \leftarrow \text{randomize}(\text{leaves.get}(i).\text{cost})$  [Randomize the ciphertext]
4: end for
5: for  $i = \text{leaves.size}()$  to  $\mu$  do
6:    $c[i] \leftarrow \text{genRandCtxt}()$  [Generate a random ciphertext]
7: end for
8:
9: return  $\text{permute}(c)$ 

```

Algorithm 6 Decrypting

Require: result

```

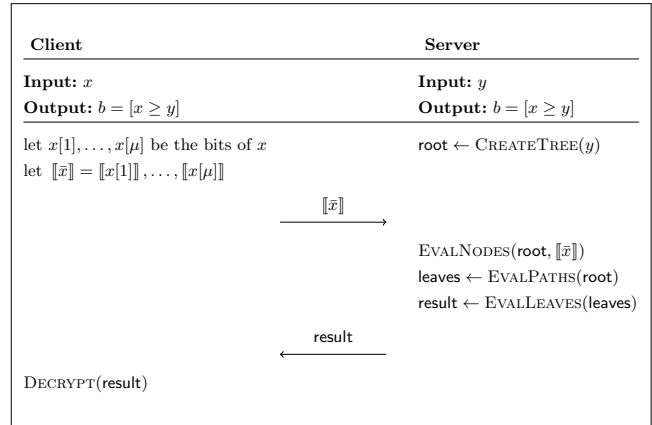
1: if  $\beta = 1$  then {FHE case}
2:   parse result to  $\llbracket b \rrbracket$ 
3:    $b \leftarrow \text{Dec}(\llbracket b \rrbracket)$ 
4:
5:   return  $b$ 
6: else {AHE case}
7:   parse result to  $c[1], \dots, c[\mu]$ 
8:   for  $i = 1$  to  $\mu$  do
9:      $m \leftarrow \text{Dec}(c[i])$ 
10:    if  $m = 0$  then
11:
12:      return 1
13:    end if
14:  end for
15:
16:  return 0
17: end if

```

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 bit representation $\llbracket \tilde{x} \rrbracket$ 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 $b = \lfloor x \geq y \rfloor$.

LEMMA 5.1. *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



Protocol 4: The Basic Protocol

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). \square

THEOREM 5.2. *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 (Appendix A). \square

6 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.

6.1 Handling Comparison to Zero for AHE in the Constant Case

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 labeled 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.

6.2 Shared Output Bit

In 2-party comparison like DGK [18], 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, chooses a bit b_s and computes GT if $b_s = 0$, otherwise it computes LT (See Section 6.5). Note that in both cases (GT vs. LT) the tree of y has the same structure, such that the server performs the same computation which is independent of $[x \geq y]$ and $[x \leq y]$ and hence does not leak which operation was evaluated. After this computation, the server returns μ ciphertexts to the client, among which at most one encrypts 0 (due to sum of zeros on the corresponding path). The remaining ciphertexts encrypt each a random plaintext (since there is at least a 1 on the corresponding path). The client can then extract its share b_c of the comparison bit. For further computation, the client can send back the ciphertext $\llbracket b_c \rrbracket$, from which the server can easily get the ciphertext $\llbracket b \rrbracket = \text{XOR}(\llbracket b_c \rrbracket, b_s)$ of the actual comparison bit. Note that this is not specific to our scheme, but it also works with other AHE-based comparison protocols.

6.3 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 6.1 (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 6.1 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 a path p representing y , label the leaf of p and the leaves of all paths right to p with 1. Label the leaves of all paths left to p with 0. Such a tree construction is illustrated in Figure 5 for $y = 2$. Note

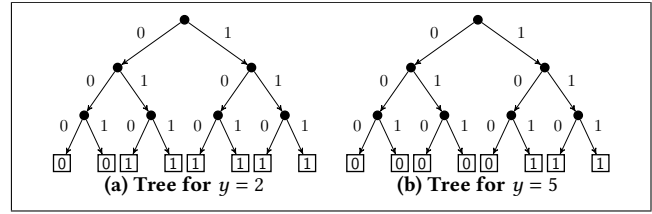


Figure 5: Comparison Tree for input bit-length $\mu = 3$

that the tree from Figure 5 is unnecessarily large as there are inner nodes 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 6.2 (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 6.3 (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 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 6.1 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 $\llbracket b \rrbracket$, we have $F_\beta(\llbracket b \rrbracket) = \llbracket b \rrbracket$ if the encryption is AHE and $F_\beta(\llbracket b \rrbracket) = \llbracket 1 - b \rrbracket$ if the encryption is FHE with arithmetic encoding or $F_\beta(\llbracket b \rrbracket) = \llbracket 1 + b \rrbracket$ for binary encoding, as the addition is modulo two.

DEFINITION 6.4 (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 6.5. *Let y be an integer of length μ . The half-pruned tree of y has $\mu + 1$ leaves and μ inner nodes.*

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 6.6. *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 6.7. *The normal cmp-tree of y and a half-pruned cmp-tree of y are equivalent.*

THEOREM 6.8. *Let y and x be integers of length μ . If the encryption scheme is correct, then the comparison protocol is correct.*

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 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.

6.4 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

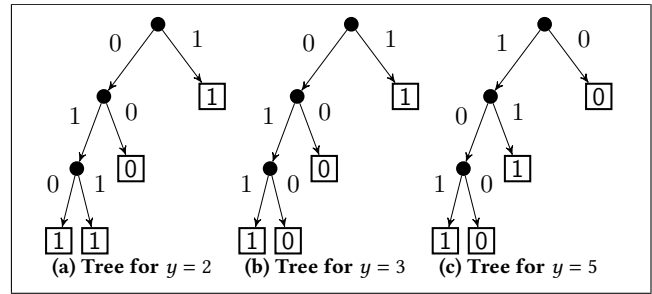


Figure 6: Normal Cmp-Tree for input bit-length $\mu = 3$ and $\beta = 1$

Algorithm 7 Creating Normal Cmp-Tree for y .

```

1: let root be a new node
2: parse  $\llbracket y \rrbracket$  to  $\llbracket y[1] \rrbracket, \dots, \llbracket y[\mu] \rrbracket$ 
3: curr  $\leftarrow$  root
4: for  $i = \mu$  downto 1 do
5:   curr.rEdge  $\leftarrow$  NOT( $\llbracket y[i] \rrbracket$ )
6:   let  $vr$  be a new node
7:    $vr.cLabel \leftarrow F_\beta(\llbracket y[i] \rrbracket)$ 
8:   curr.right  $\leftarrow vr$ 
9:   curr.lEdge  $\leftarrow \llbracket y[i] \rrbracket$ 
10:  let  $vl$  be a new node
11:  curr.left  $\leftarrow vl$ 
12:  curr  $\leftarrow$  curr.left
13: end for
14: curr.cLabel  $\leftarrow \llbracket \beta \rrbracket$ 
15:
16: return root

```

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 $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$ to the server. The server creates the normal cmp-tree of $\llbracket y \rrbracket$ using Algorithm 7, evaluates the encrypted input $\llbracket x \rrbracket$ 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 $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$ be two encrypted bits. We compute the NOT-operation as $\llbracket \neg b \rrbracket = \llbracket 1 - b \rrbracket = \text{ADD}(\llbracket 1 \rrbracket, \text{MULCONS}(\llbracket b \rrbracket, -1))$. Then, we compute the XOR-operation as $\llbracket a \oplus b \rrbracket = \llbracket a - b \rrbracket$. While the NOT-operation is correct, this is not always the case for the XOR-operation, namely we have $\llbracket 0 \oplus 1 \rrbracket = \llbracket -1 \rrbracket$. 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 $\llbracket y \rrbracket$, 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 $\llbracket x \rrbracket$ on the built tree. For that, we first have to apply the XOR-operations on the bits of $\llbracket x \rrbracket$ 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 $\llbracket 1 \rrbracket$ and $\llbracket -1 \rrbracket$ 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 6.9. *Let $(b_0, \dots, b_l) \in \{-1, 0, 1\}^{l+1}$ such that there exist at least one $b_i \neq 0$ then it holds $\sum_{i=0}^l b_i \cdot 2^i = b_0 \cdot 2^0 + \dots + b_l \cdot 2^l \neq 0$.*

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. First note that the paths are as follows (from left to right): $(0, 1, 0, 0)$, $(0, 1, 1, 0)$, $(0, 0, -, 1)$, $(1, -, -, 0)$, where the last element in each vector is the leaf label. Without Lemma 6.9, the evaluation of the leftmost path 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. For the other paths $(0, 1, 1, 0)$, $(0, 0, -, 1)$, $(1, -, -, 0)$ we will have respectively: $(0-0) \cdot 2^0 + (0-1) \cdot 2^1 + (1-1) \cdot 2^2 + 0 = 0 - 2 + 0 + 0 = -2$, $(0-0) \cdot 2^0 + (0-0) \cdot 2^1 + 1 = 0 + 0 + 1 = 1$, $(0-1) \cdot 2^0 + 0 = -1 + 0 = -1$. That is, on encrypted input we get $(\llbracket 2 \rrbracket, \llbracket -2 \rrbracket, \llbracket 1 \rrbracket, \llbracket -1 \rrbracket)$. The server will then homomorphically randomize the plaintexts and send a permuted vector, from which the client can deduce the comparison bit $b = 0$, since no ciphertext decrypts to zero.

6.5 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 6.10 (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 6.4), 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 6.10 instead of $F_\beta(b)$ as in Definition 6.4.

7 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).

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

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

7.1 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 6.3), 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 $\mu - 1 + 2 \cdot (\mu - 1) + 1 = 3\mu - 2$.

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

7.2 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 $M_T = 0 + (3\mu - 2) + 0 = 3\mu - 2$.

For some FHE schemes, it might come with performance improvements if we reduce the multiplicative depth. In our scheme, this only affects the path evaluation. The previous approach yields

a depth of μ but we can reduce the depth to $\log \mu$ by using an optimized implementation. For implementation details, we refer to Section 8 and Appendix B. The reduction of the multiplicative depth comes with an increasing number of multiplications. For the leftmost path, we need a multiplication for every pair, for every quadruple and so on. By the geometric sum formula, this is $\mu - 1$ multiplications. The splitting of the leftmost path creates a structure that divides the path into sections of powers of two. Therefore, the connection of right children of one section needs one additional multiplication for every child in that section. This results in a total number of multiplications of $\mu + \frac{\mu \log \mu}{2}$ for right children nodes. We provide a more detailed analysis in Appendix C.

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 NOR 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 $M_T = 0 + (3\mu - 2) + 0 = 3\mu - 2$ multiplications. In the encrypted case, the total is $A_T = 3\mu$ additions and $M_T = 4\mu - 2$ multiplications.

Comparison to Previous Work. Cheon et al. [15] also use a comparison protocol based on FHE and present a variant with logarithmic multiplicative depth. Their critical part is the iterative computation of a product with depth $\mu - 1$, i.e., given integers x_1, \dots, x_n , we want to compute products $P_i = \prod_{j=1}^i x_j$, for $i = 2, \dots, n$ while keeping the multiplicative depth logarithmic. They propose to compute the products using a recursive algorithm that builds a binary tree of products yielding sub products for positions which are power of 2. In the next step, they compute the missing products, i.e., for positions that are not power 2, based on a multiplication of the results for the power of two cases. We implemented this computation using an iterative algorithm described in Appendix D. For more details, we refer to the original work [15]. In total, the scheme of Cheon et al. requires $2\mu - 2$ additions and $2\mu - 3 + \frac{(\mu-1)\log(\mu-1)}{2}$ multiplications using binary encoding. Using arithmetic encoding, the $2\mu - 2$ additions require an additional $\mu - 1$ 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 but need less additions and slightly less multiplications. However, our optimized implementation uses a precomputation which is an advantage for the actual running time of the protocol, see further in Section 8.

7.3 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 NOR 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.

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

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

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 [18], 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 [49] 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 6.4. A constant multiplication $\llbracket m \rrbracket \boxtimes n$ requires in worst case $2 \log n$ homomorphic additions. For each level i , we perform in Section 6.4 two multiplications by 2^i resulting in $2 \sum_{i=2}^{\mu-1} i = O(\mu^2)$ 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 still has 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). The scheme works as follows. The server holds $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$ 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 $\llbracket z \rrbracket \leftarrow \llbracket x - y + 2^\mu + r \rrbracket = \llbracket x \rrbracket \cdot \llbracket y \rrbracket^{-1} \cdot \llbracket 2^\mu + r \rrbracket \pmod{N^2}$ and sends it to client, which the client decrypts to get z . For the statistical security case, the server with $\alpha = r \pmod{2^\mu}$ and the client with $\beta = z \pmod{2^\mu}$ 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 $\llbracket z/2^\mu \rrbracket$ and $\llbracket \delta_C \rrbracket$. The server finally computes the encrypted comparison bit for $[\beta < \alpha]$ as $\llbracket [\beta < \alpha] \rrbracket = \llbracket 1 + (-1)^{\delta_S} + (-1)^{1-\delta_S} \delta_C \rrbracket$ and the final comparison

bit as $\llbracket [x < y] \rrbracket = \llbracket [z/2^\mu] \rrbracket \cdot (\llbracket [r/2^\mu] \rrbracket \cdot \llbracket [\beta < \alpha] \rrbracket)^{-1} \bmod N^2$. All the operations described above are done under Paillier encryption.

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 [49], 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.

Another optimization of the DGK protocol has been published by Joye and Salehi [31]. Their scheme achieves about the same overhead as Veugen but prevents timing attacks. To this end, they use the hamming weight of one input and can furthermore half the number of ciphertexts being sent. However, to keep the security, they introduce another round to mask the input. Our protocol only needs one round but still transmits μ instead of $\mu/2$ ciphertexts.

8 EVALUATION AND IMPLEMENTATION

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

8.1 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 $a[(1, \dots, \mu + 1), (1, 2, 3)]$ 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 is filled.

The evaluation itself is illustrated in Algorithm 8 and Figure 7. On each row, we store $x[i] == y[i]$ in cell 1, its negation in cell 2, and $F_\beta(y[i])$ in cell 3. This corresponds to the computation of decision bits (cell 1 and 2) and the leaf node labels (cell 3). In Figure 7, the arrows illustrate paths evaluation, where $a_i \rightarrow a_j$ means that cells a_i and a_j are aggregated and the result is stored in cell a_j .

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 [45]. 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 queue, represented as $[]$ with back $[-$ and front $]$ 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

Algorithm 8 Efficient implementation

```

1: parse  $\llbracket \bar{x} \rrbracket$  to  $\llbracket x[1] \rrbracket, \dots, \llbracket x[\mu] \rrbracket$ 
2: parse  $\llbracket \bar{y} \rrbracket$  to  $\llbracket y[1] \rrbracket, \dots, \llbracket y[\mu] \rrbracket$ 
3: let  $a[(1, \dots, \mu + 1), (1, \dots, 3)]$  matrix
4: for  $i = 1$  to  $\mu$  do
5:    $xi \leftarrow \llbracket x[\mu + 1 - i] \rrbracket$ 
6:    $yi \leftarrow \llbracket y[\mu + 1 - i] \rrbracket$ 
7:    $a[i, 1] \leftarrow \text{comp}(xi, yi)$ 
8:    $a[i, 2] \leftarrow \text{NOT}(a[i, 1])$ 
9:   if  $\beta = 0$  then {AHE Section 6.4}
10:     $a[i, 1] \leftarrow \text{MULCONS}(a[i, 1], 2^{i-1})$ 
11:     $a[i, 2] \leftarrow \text{MULCONS}(a[i, 2], 2^{i-1})$ 
12:   end if
13:    $a[i, 3] \leftarrow F_\beta(\llbracket y[\mu + 1 - i] \rrbracket)$ 
14:   if  $i = 1$  then
15:      $a[i, 3] \leftarrow \text{BitAgg}(a[i, 3], a[i, 2])$ 
16:   else
17:      $a[i, 1] \leftarrow \text{BitAgg}(a[i, 1], a[i - 1, 1])$ 
18:      $z1 \leftarrow \text{BitAgg}(a[i, 2], a[i - 1, 1])$ 
19:      $a[i, 3] \leftarrow \text{BitAgg}(a[i, 3], z1)$ 
20:   end if
21: end for
22:  $a[\mu + 1, 3] \leftarrow a[\mu, 1]$ 
23: if  $\beta = 1$  then {Only FHE}
24:
25:   return  $\sum_{i=1}^{\mu+1} a[i, 3]$ 
26: else
27:   let  $c[1, \dots, \mu + 1]$  be an array
28:   for  $i = 1$  to  $\mu + 1$  do
29:      $c[i] \leftarrow \text{randomize}(a[i, 3])$ 
30:   end for
31:
32:   return  $\text{permute}(c)$ 
33: end if

```

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 bottom 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 reduces the multiplicative depth from k to $\log k$.

8.2 Setup Environment

For AHE, we implemented DGK [18], the optimized DGK by Veugen [49], the scheme of Joye and Salehi [31] and our scheme in Java. We instantiated AHE with ElGamal on elliptic curve as described in Appendix E using curve secp256r1. We implemented

y_i	$\neg y_i$	
0	1	1
1	0	0
1	0	0
1		

$x_i \oplus y_i$	$x_i \oplus y_i$	
$0 \oplus 0 = 1$	0	1
$0 \oplus 1 = 0$	1	0
$1 \oplus 1 = 1$	0	0
	1	1

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 $y = 3$. The right table illustrates the evaluation of the cmp-tree of $y = 3$ on input $x = 1$.

1	2	3
4	5	6
7	8	9
10		

()	()	[2]
[1]	[1]	[5]
()	()	[4, 8]
[4, 7]		

1	2	3
4	5	6
7	8	9
10		

Figure 8: Optimized implementation for $\mu = 3$ and leveled FHE. Left is the representation of $y = 3$, where xor-results are omitted, and cells are numbered. The middle table illustrates dependency lists (DL). Right is the paths evaluation.

our scheme in three variants: the naive implementation using tree representation (Section 5), the optimized implementation using array representation (Algorithm 8) and the encrypted case (Section 6). For the scheme of Joye and Salehi, we only implemented the basic comparison protocol without the additional masking round which would be required to achieve full security. We note that Veugen [49] 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.

For FHE, we compared our protocol with Cheon et al. [15]. The implementation uses the BGV scheme [12] from HELib [3].

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. The evaluation of the AHE implementations is done 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. Due to the higher running time and memory requirements of FHE schemes, the evaluation of FHE implementations is done on an AWS instance with a 24-core Intel(R) Xeon(R) Scalable processor (Skylake 8151) with up to 4.0 GHz and 192GB of RAM running Ubuntu 20.04 LTS.

8.3 Results

We present the result of our evaluation and compare our scheme to the related work. Since there are several schemes in the literature, we focus on schemes closely related to ours and discuss some recent

work in Appendix F. That is the comparison uses the binary decomposition, performs computation on homomorphically encrypted inputs but does not assume a specific AHE or FHE scheme.

AHE. For 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, Veugen 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 domain. Veugen’s scheme additionally sends few Paillier ciphertexts and requires two rounds instead of one in the encrypted case. Joye and Salehi have different communication because the server sends only half the number of ciphertexts as the other schemes. For the encrypted case, one might apply the same procedure as Veugen resulting in more expensive Paillier operations as well. The above is also true for the client computation effort. In DGK, constant case Veugen and our scheme, the client encrypts μ plaintext bits and decrypts μ ciphertexts. In Joye and Salehi, the decryption of the client is reduced by factor 2. Nevertheless, we focus our evaluation on the server computation but compare only to the basic scheme of Joye and Salehi without the additional round.

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 and Joye and Salehi scheme clearly perform better than the original DGK scheme, Joye and Salehi performs slightly better for small bit-lengths ($\mu \leq 32$), while Veugen performs better for large bit-lengths ($\mu \geq 64$). Our naive implementation is only better than the both for large bit-lengths ($\mu \geq 64$) and our optimized implementation is always better. Although our encrypted case requires additional constant multiplications per bits, it still performs better than the original DGK scheme. In the encrypted case of Veugen’s scheme and Joye and Salehi’s scheme client and server both require additional Paillier ciphertext operations.

We implemented and evaluated the Paillier operations in Veugen’s scheme 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.

		$\mu = 8$	$\mu = 16$	$\mu = 32$	$\mu = 64$	$\mu = 96$	$\mu = 128$
DGK [18]		34.12	67.43	109.91	189.16	273.18	354.22
Veugen [49]	Constant	20.05	39.79	66.86	121.68	169.98	222.25
	Encrypted	≥ 600					
Joye [31]		19.13	36.39	63.53	123.14	178.92	249.46
Ours	Naive	21.68	40.10	67.01	117.04	167.13	201.47
	Optimized	18.55	35.44	63.18	115.56	164.99	202.88
	Encrypted	24.23	45.41	81.90	151.21	230.54	296.27

Table 3: Comparison of Running Time in milliseconds for the AHE Implementation

FHE. To evaluate the running time of Cheon et al.’s protocol and our protocol, we use bit-lengths $\mu = 8, 16, 32, 64, 128$ and compute the average over 100 runs. The results are depicted in Table 4. The running time of Cheon et al. is slightly better which matches the theoretical considerations. However, we have two other advantages.

First, our optimized implementation only needs memory for the two-dimensional array, i.e. 3μ ciphertexts, since the multiplication procedure is based on a pre-computed plan and applied within the array. Cheon et al. also need 3 times μ ciphertexts but additional memory to compute μ products [15]. Second, the implementation of our protocol (Algorithm 8) by another party is much easier since the construction of the multiplication plan does not have to be implemented but the pre-computed results can be taken from a public source. Our FHE evaluation in Algorithm 12 is given as parameter the dependency list precomputed by Algorithm 11.

Cheon et al. and our scheme have three main steps. In the first step, we fill the array as explained in Figure 7 and Cheon et al. compute three arrays $a_i = x_i + 1, d_i = a_i \cdot y_i, z_i = a_i + y_i, 1 \leq i \leq \mu$. In the second step, we aggregate the array elements as in Figure 8 using a precomputed dependency list and Cheon et al. compute products $p_i = \prod_{j=i+1}^{\mu} z_j$ efficiently (i.e., reusing prefix results) while keeping the multiplicative depth logarithmic. This computation is illustrated in Appendix D and Algorithm 13. In the last step, we sum up all results in the third column and Cheon et al. compute $d_i = d_i \cdot p_i$ and finally $\sum_{i=1}^{\mu} d_i$.

In both cases the second step is the most complicated. However, the pre-computation of the dependency list makes our second step easy to implement and to evaluate, as it is done once and offline, such that for the online computation it is enough to implement and evaluate Algorithm 12. The product computation (Algorithm 13) in Cheon et al.’s scheme requires constructing a binary tree of product and process it recursively. This cannot be precomputed, must be implemented with the entire algorithm itself and requires extra memory for storing $\mu \log \mu$ ciphertexts during evaluation.

		$\mu = 8$	$\mu = 16$	$\mu = 32$	$\mu = 64$	$\mu = 128$
Plaintext	Cheon et al. [15]	0.0340	0.0569	0.0900	0.2000	1.2097
	Ours	0.0270	0.0431	0.0460	0.1080	0.2522
Encrypted	Cheon et al. [15]	4.9480	11.348	26.041	56.866	122.36
	Ours	5.1241	11.660	26.504	59.681	129.16
Memory	Cheon et al. [15]	235.33	356.51	595.99	1064.12	1993.18
	Ours	217.35	320.03	523.57	929.41	1728.05

Table 4: Comparison of Running Time for Plaintext (in ms) and Ciphertext (in s) Implementation, and the Memory (in MB)

9 APPLICATIONS

Integer comparison is a fundamental building block in many MPC protocols. In this section, we describe few applications where our scheme can improve the performance. We estimate this improvement to be proportional to the number of comparison operations required in the respective application. The following is of course not exhaustive and gives only an overview of applications.

Machine Learning (ML). ML classifiers are valuable tools in many areas such as healthcare, finance, spam filtering, intrusion detection, remote diagnosis, etc [50]. They usually require access to privacy-sensitive user’s data such as medical records, financial situation, location information, etc. On the one hand, the model itself may contain sensitive data. On the other hand, it may have been built on sensitive data. White-box and sometimes even black-box access to an ML model allows so-called *model inversion attacks* [23, 44, 52], which can compromise the privacy of the training data. Privacy-preserving techniques are therefore critically needed to protect the privacy of the model and user’s data. Many applications in ML require integer comparison. For example, a decision

tree (DT) is a common and very popular classifier that requires integer comparison to classify inputs. Some private DT schemes rely on DGK [42, 51] or on FHE [45]. In Appendix G, we evaluate the private DT scheme of Tai et al. [42] using our scheme and compared it to [18, 49]. In [46], 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 [18]. Privacy-preserving online auction was the world’s first large scale application and commercial use of MPC. A team around the DGK authors developed, in 2008, a double auction solution for the Danish industry, which allowed farmers (sugar beets producers) and Danisco (the only sugar beets processor) to compute a so-called market clearing price (price per unit of the commodity that is traded) in a privacy-preserving way [2, 10]. The main building block in this solution was integer comparison. Other work, including [9], [8], [48], have proposed protocols for computing the k^{th} -ranked element using the DGK comparison protocol. In [48], Tueno et al. also proposed a variant of their protocol based on SHE/FHE and using the comparison scheme of Cheon et al. [15].

Biometrics. Biometrics are used to authenticate or identify users. In an enrollment phase, biometric features are scanned and stored. During the authentication or identification phase, the same features are scanned again and compared with the stored ones. On the one hand, biometric images are never perfect, and therefore a match is determined by computing a proximity to the stored images. This implies that threshold comparisons are required in biometric systems [35]. On the other hand, biometric information is highly sensitive and subject to privacy issues due to possible misuse, lost or theft of biometric data. This gives rise to privacy-preserving biometric matching, that compute on biometric data without revealing sensitive information. Blanton and Gasti [7] proposed such a protocol for iris and fingerprint identification using the DGK protocol.

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.

ACKNOWLEDGMENTS

We thank the anonymous PoPETs reviewers for their constructive feedback on this paper. This research work was supported by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) in the project Trade-EVs II, FKZ:01MV20006A.

REFERENCES

- [1] Scale and mamba, November 2022. <https://github.com/KULeuven-COSIC/SCALE-MAMBA>.
- [2] Secure multiparty computation goes live, November 2022. <https://partisia.com/better-market-solutions/mpc-goes-live/>.
- [3] Helib, October 2021. <https://github.com/homenc/HELIB>.
- [4] A. Aly, K. Nawaz, E. Salazar, and V. Sucasas. Through the looking-glass: Benchmarking secure multi-party computation comparisons for relu's. In *Cryptography and Network Security - 21st International Conference, CANS 2022, Dubai, United Arab Emirates, November 13-16, 2022, Proceedings*, pages 44–67, 2022.
- [5] 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.
- [6] 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.
- [7] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, pages 190–209, 2011.
- [8] 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*, pages 558–571. ACM.
- [9] E. Blass and F. Kerschbaum. Secure computation of the k^{th} -ranked integer on blockchains. *IACR Cryptology ePrint Archive*, 2019:276, 2019.
- [10] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, pages 325–343, 2009.
- [11] F. Bourse, O. Sanders, and J. Traoré. Improved secure integer comparison via homomorphic encryption. In *Topics in Cryptology - CT-RSA 2020 - The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24-28, 2020, Proceedings*, pages 391–416, 2020.
- [12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *ECCC*, 18:111, 2011.
- [13] O. Catrina and S. De Hoogh. Improved primitives for secure multiparty integer computation. In *SCN'10*, pages 182–199. Springer-Verlag, 2010.
- [14] J. H. Cheon, M. Kim, and M. Kim. Search-and-compute on encrypted data. In *FC*, pages 142–159, 2015.
- [15] J. H. Cheon, M. Kim, and K. E. Lauter. Homomorphic computation of edit distance. In *FC*, pages 194–212, 2015.
- [16] 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.
- [17] I. Damgård, M. Fritzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC 2006*, pages 285–304, 2006.
- [18] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In *ACISP*, pages 416–430, 2007.
- [19] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, pages 10–18. Springer-Verlag, 1985.
- [20] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part II*, pages 823–852, 2020.
- [21] 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.
- [22] M. Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In *CT-RSA*, pages 457–472, 2001.
- [23] M. Fredrikson, S. Jha, and T. Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS*, pages 1322–1333, 2015.
- [24] J. Garay, B. Schoenmakers, and J. Villegas. Practical and secure solutions for integer comparison. In *PKC'07*, pages 330–342. Springer-Verlag, 2007.
- [25] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, New York, NY, USA, 2009. ACM.
- [26] 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.
- [27] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.
- [28] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229, New York, NY, USA, 1987. ACM.
- [29] 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.
- [30] I. Iliashenko and V. Zucca. Faster homomorphic comparison operations for BGV and BFV. *Proc. Priv. Enhancing Technol.*, 2021(3):246–264, 2021.
- [31] M. Joye and F. Salehi. Private yet efficient decision tree evaluation. In *BDSec*, pages 243–259. Springer, 2018.
- [32] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1575–1590, 2020.
- [33] N. Kobitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, Jan. 1987.
- [34] N. Kobitz, A. Menezes, and S. A. Vanstone. The state of elliptic curve cryptography. *Des. Codes Cryptography*, 19(2/3):173–193, 2000.
- [35] 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.
- [36] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.
- [37] H. Lin and W. Tzeng. An efficient solution to the millionaires' problem based on homomorphic encryption. In *ACNS*, pages 456–466, 2005.
- [38] H. Lipmaa and T. Toft. Secure equality and greater-than tests with sublinear online complexity. In *ICALP'13*, pages 645–656, 2013.
- [39] E. Makri, D. Rotaru, F. Vercauteren, and S. Wagh. Rabbit: Efficient comparison for secure multi-party computation. In *Financial Cryptography and Data Security - 25th International Conference, FC 2021, Virtual Event, March 1-5, 2021, Revised Selected Papers, Part I*, pages 249–270, 2021.
- [40] T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC'07*, pages 343–360, 2007.
- [41] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99*, pages 223–238. Springer-Verlag, 1999.
- [42] 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.
- [43] T. Toft. Sub-linear, secure comparison with two non-colluding parties. In *Public Key Cryptography*, volume 6571, pages 174–191. Springer, 2011.
- [44] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Stealing machine learning models via prediction apis. In *USENIX*, pages 601–618, 2016.
- [45] A. Tueno, Y. Boev, and F. Kerschbaum. Non-interactive private decision tree evaluation. In *DBSec*, pages 174–194, 2020.
- [46] A. Tueno and F. Kerschbaum. Efficient secure computation of order-preserving encryption. In *Proceedings of the 2020 Asia Conference on Computer and Communications Security, ASIACCS '20, 2020*.
- [47] A. Tueno, F. Kerschbaum, and S. Katzenbeisser. Private evaluation of decision trees using sublinear cost. *PoPETs*, 2019(1):266–286, 2019.
- [48] A. Tueno, F. Kerschbaum, S. Katzenbeisser, Y. Boev, and M. Qureshi. Secure computation of the k^{th} -ranked element in a star network. In *FC*, 2020.
- [49] T. Veugen. Improving the DGK comparison protocol. In *WIFS*, pages 49–54, 2012.
- [50] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.
- [51] D. J. Wu, T. Feng, M. Naehrig, and K. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, 2016.
- [52] X. Wu, M. Fredrikson, S. Jha, and J. F. Naughton. A methodology for formalizing model-inversion attacks. In *CSF*, pages 355–370, 2016.
- [53] A. C. Yao. Protocols for secure computations. In *SFCS '82, SFCS '82*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

A POSTPONED PROOFS

This section provides missing proofs for theorems and lemmas.

THEOREM 5.2. 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. For our scheme, we can consider settings where the comparison result is either revealed only to the client, or to both client and server, or secret-shared to both. Additionally, the server

can see its own input y or it can be encrypted as well. For simplicity, we consider the setting where the comparison result is revealed only to the client and the server input is not encrypted. In this case, security means that the client learns only the comparison bit and the server learns nothing. We then construct simulators $\text{Sim}_C^{\text{gt}}(x, b = [x \geq y])$, $\text{Sim}_S^{\text{gt}}(y, \emptyset)$, for the client and the server. The goal of the simulator is to generate a view that is indistinguishable from the party's view in the real protocol. That is, for each message m that a party sees, the simulator should be able to generate an indistinguishable message m' using only the information available to that party, e.g., input, output, public and private key.

Client Simulator FHE: In the real protocol, the client sees encryptions of its input bits $\bar{x} = x[1], \dots, x[\mu]$ and encryption of the comparison bit. The simulator $\text{Sim}_C^{\text{gt}}(x, b = [x \geq y])$ has access to input and output of the client and knows the public key. The simulation is trivial.

Server Simulator FHE: In the real protocol, the server sees encryptions of the client input bits $\bar{x} = x[1], \dots, x[\mu]$ and ciphertexts generated during the evaluation of the tree. The simulator $\text{Sim}_S^{\text{gt}}(y, \emptyset)$ has access to the server input and knows the public key, but not the private key. Hence, it can encrypt its own input bits. By assumption, the encryption scheme is IND-CPA secure, which means that each ciphertext is indistinguishable from a random value with the same length as the ciphertext. For all ciphertexts (either received from the client or resulting from the tree evaluation), the simulator just chooses a random element in the ciphertext space, i.e., encryption of a random plaintext.

Client Simulator AHE: The difference to the FHE case is that the client does not get an encrypted bit from the server, but a set of ciphertexts $c[1], \dots, c[\mu]$ where either all ciphertexts encrypt random plaintexts, or exactly one (at a random position) encrypts 0 while the remaining ones encrypt random plaintexts. That is, the simulation of the encrypted client bits $\bar{x} = x[1], \dots, x[\mu]$ is also trivial as before. Given the output b to the client, the simulator simulates the ciphertexts $c[1], \dots, c[\mu]$ as follows. Choose random elements $c'[1], \dots, c'[\mu]$ in the ciphertext space. If $b = 1$, choose a random position i between 1 and μ and replace $c'[i]$ by a ciphertext of 0.

Server Simulator AHE: The simulation is similar to the FHE case. \square

LEMMA 6.5. 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 $h \in \{0, \dots, \mu - 2\}$, we replace a subtree, that has $2^{\mu-h} - 1$ nodes, with a leaf. That is, at depth $h \in \{0, \dots, \mu - 2\}$, 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:

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

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. \square

LEMMA 6.7. 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 6.5 shows that a half-pruned tree has the same number of nodes and therefore they are equivalent. \square

THEOREM 6.8. Let y and x be integers of length μ . If the encryption scheme is correct, then the comparison protocol is correct.

PROOF. By Lemma 6.7, 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 . \square

LEMMA 6.9. Let $(b_0, \dots, b_l) \in \{-1, 0, 1\}^{l+1}$ such that there exist at least one $b_i \neq 0$ then it holds $\sum_{i=0}^l b_i \cdot 2^i = b_0 \cdot 2^0 + \dots + b_l \cdot 2^l \neq 0$.

Algorithm 9 COMPUTEPATHS

Require: μ

- 1: let paths be a matrix of $[0, \dots, \mu][1, \dots, \mu + 1]$ integers
- 2: **for** $i = 0$ **to** μ **do**
- 3: paths[0][i] $\leftarrow i \cdot 3 + 1$ {leftmost path}
- 4: **end for**
- 5: **for** $p = 1$ **to** $\mu - 1$ **do**
- 6: **for** $i = 0$ **to** $\mu - (p + 1)$ **do**
- 7: paths[p][i] $\leftarrow i \cdot 3 + 1$ {common prefixes with leftmost path}
- 8: **end for**
- 9: paths[p][$\mu - p + 1$] $\leftarrow (\mu - p) \cdot 3 + 2$ {second column}
- 10: paths[p][$\mu - p + 2$] $\leftarrow (\mu - p) \cdot 3 + 3$ {third column}
- 11: **end for**
- 12: paths[μ][2] $\leftarrow 2$ {rightmost path}
- 13: paths[μ][3] $\leftarrow 3$
- 14:
- 15: **return** paths

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. \square

B ALGORITHMS FOR FHE INSTANTIATION

In this section, we describe in detail our leveled FHE instantiation that uses the pre-computation of the dependency lists for the multiplication to keep a logarithmic multiplicative depth. We recall that this pre-computation depends only on the input length μ and can be computed a single time and stored for future evaluations of integer comparison. The pre-computation consists of the following three basic steps:

- Computation of the paths. Using the Illustration in Figure 8, we use a table representation of the tree as explained above and compute the paths: (1, 4, 7, 10), (1, 4, 8, 9), (1, 5, 6), (2, 3). This computation is illustrated in Algorithm 9.
- Computation of the dependency lists for each single path. We compute the dependency lists for each path using Algorithm 10.
- Computation of the dependency lists for the whole tree. This is done using Algorithm 11.

To evaluate the integer comparison of two encrypted inputs, we use Algorithm 12. A detailed analysis of the number of multiplications needed is provided in the next section.

C ANALYSIS NUMBER OF MULTIPLICATIONS

In this section, we analyse in detailed the number of multiplications for our leveled FHE Instantiation. To ease the understanding of the derivation of the number of multiplications, we present an example of bit-length 8 and apply the optimized implementation described

Algorithm 10 COMPUTEDL

Require: path, up, low, dlist

- 1: **if** up \geq low **then**
- 2:
- 3: **return**
- 4: **end if**
- 5: $\eta \leftarrow \text{low} - \text{up}$
- 6: mid $\leftarrow 2^{|\eta|-1} - 1 + \text{up}$
- 7: COMPUTEDL(path, up, mid, dlist)
- 8: COMPUTEDL(path, mid + 1, low, dlist)
- 9: $x \leftarrow \text{path}[\text{mid}]$
- 10: **if** dlist[low].contain(x) = false **then**
- 11: dlist[low].enqueue(x)
- 12: **end if**

Algorithm 11 COMPUTEALLDL

Require: μ

- 1: let dlmatrix be a matrix of $[1, \dots, \mu + 1][1, \dots, 3]$ queues
- 2: paths \leftarrow COMPUTEPATHS(μ)
- 3: **for** $p = 0$ **to** μ **do**
- 4: let dlist be an array of queues
- 5: COMPUTEDL(paths[p], 0, paths[p].size() - 1, dlist)
- 6: **if** $p = 0$ **then**
- 7: **for** $i = 0$ **to** paths[p].size() - 1 **do**
- 8: $x \leftarrow \text{paths}[p][i]$
- 9: $r \leftarrow (x \div 3) + 1$
- 10: $c \leftarrow ((x - 1) \bmod 3) + 1$
- 11: dlmatrix[r][c] \leftarrow dlist[i]
- 12: **end for**
- 13: **else**
- 14: $l \leftarrow \text{paths}[p].\text{size}()$
- 15: **if** $p < \mu$ **then**
- 16: $x \leftarrow \text{paths}[p][l - 2]$
- 17: $r \leftarrow (x \div 3) + 1$
- 18: $c \leftarrow ((x - 1) \bmod 3) + 1$
- 19: dlmatrix[r][c] \leftarrow dlist[$l - 2$] {second column}
- 20: **end if**
- 21: $x \leftarrow \text{paths}[p][l - 1]$
- 22: $r \leftarrow (x \div 3) + 1$
- 23: $c \leftarrow ((x - 1) \bmod 3) + 1$
- 24: dlmatrix[r][c] \leftarrow dlist[$l - 1$] {third column}
- 25: **end if**
- 26: **end for**
- 27:
- 28: **return** dlmatrix

in Section 8. The starting point is the array representation depicted in Figure 9.

Leftmost Path. The number of multiplications on the leftmost path are illustrated in Figure 10a. Our algorithm divides each path at half and repeats this procedure with the sub-paths until every node is connected. Going the other way around, this leads to one multiplication for each pair (In the example: 1 to 4, 7 to 10, 13 to 16 and 19 to 22), one multiplication for each quadruple (In the example:

Algorithm 12 FHE Evaluation with DL

```

Require:  $[\tilde{x}], [\tilde{y}], \mu, \text{dlmatrix}$ 
1: parse  $[\tilde{x}]$  to  $[x[1]], \dots, [x[\mu]]$ 
2: parse  $[\tilde{y}]$  to  $[y[1]], \dots, [y[\mu]]$ 
3: let  $a[(1, \dots, \mu + 1), (1, \dots, 3)]$  be a  $(\mu + 1) \times 3$ -matrix
4: for  $i = \mu$  downto 1 do
5:    $a[i, 1] \leftarrow \text{comp}([x[i]], [y[i]])$ 
6:    $a[i, 2] \leftarrow \text{NOT}(a[i, 1])$ 
7:    $a[i, 3] \leftarrow \text{NOT}([y[i]])$ 
8: end for
9: for  $i = \mu$  downto 1 do
10:  for  $j = 1$  to 3 do
11:   while  $\text{dlmatrix}[i][j].\text{empty}() = \text{false}$  do
12:     $x \leftarrow \text{dlmatrix}[i][j].\text{dequeue}()$ 
13:     $r \leftarrow x \div 3$ 
14:     $c \leftarrow x \bmod 3$ 
15:     $a[i][j] \leftarrow a[i][j] \cdot a[r][c]$ 
16:  end while
17: end for
18: end for
19:  $a[\mu + 1, 3] \leftarrow a[\mu, 1]$ 
20:
21: return  $\sum_{i=1}^{\mu+1} a[i, 3]$ 
    
```

4 to 10 and 16 to 22) and to one multiplication for each 8-tuple (In the example: 10 to 22) and so on.

In general, we have $\frac{\mu}{2^i}$ times a 2^i -tuple and the maximal number of such a tuple is $\log \mu$ where \log is the logarithm to base 2. Summing up all the multiplications, we obtain

$$\sum_{i=1}^{\log \mu} \frac{\mu}{2^i} = \sum_{i=0}^{\log(\mu)-1} 2^i = \mu - 1$$

by applying the geometric sum formula.

Right Children. For the representation of the multiplications on the paths to the right leaves, we refer to Figure 10b. Note that we removed some of the numbers and arrows to ease the understanding of the analysis. We split the analysis for the right children into two parts.

For each row we have exactly one edge connecting the second with third column, that is μ multiplications.

The multiplications from the first with the third column are more complicated. To analyze their total number, we consider the labels on the leftmost path and make use of the following observation. A label has at least one arrow to the right because it is on the paths to right leaves which have their last edge label in the second column and their leaf label in the third column. If it is multiplied with the following label on the leftmost path (first column), we do not have to consider it further since it is in the multiplication chain for any right child (third column) below. Therefore, such a label has only one multiplication with right children, for example 1, 7 or 19 in the example (marked with solid arrows). Based on the analysis of the leftmost path, there are $\mu/2$ such labels. For every second label, we have at least two multiplications to the right because its information is lost in the following row since we do not multiply

1	2	3			[2]
4	5	6	[1]		[1, 5]
7	8	9			[4, 8]
10	11	12	[4, 7]		[4, 7, 11]
13	14	15			[10, 14]
16	17	18	[13]		[10, 13, 17]
19	20	21			[10, 16, 20]
22	23	24	[10, 16, 19]		[10, 16, 19, 23]

Figure 9: Optimized Array Representation for bit-length $\mu = 8$

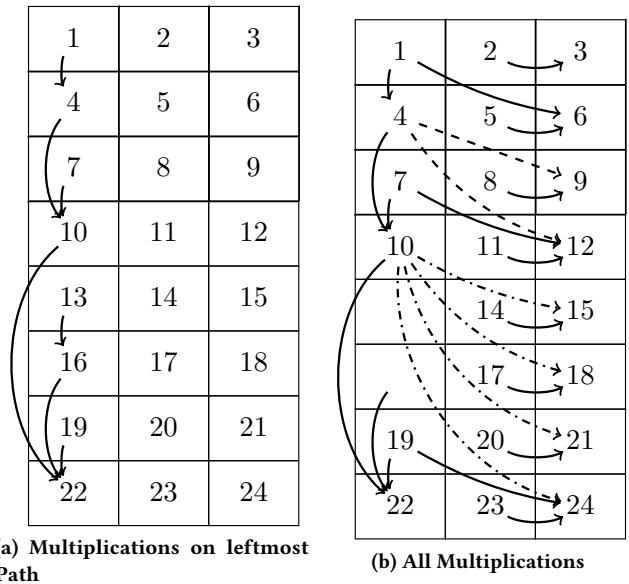


Figure 10: Illustration of Multiplications for bit-length $\mu = 8$

with it. In the example, it holds for 4 and is marked with a dashed line. The next step is 10 which is needed for the following 4 rows and marked with a dash-dotted arrow.

Following this procedure, we need 2^{i-1} multiplications of a label with “order” i . Order refers to the procedure of 2^i -tuples from the leftmost path. Now, we can count the number of multiplications analogously to the leftmost path and obtain

$$\sum_{i=1}^{\log \mu} 2^{i-1} \frac{\mu}{2^i} = \frac{\mu \log \mu}{2}.$$

Adding the multiplications from second to third column, this amounts to

$$\mu + \frac{\mu \log \mu}{2}.$$

D ITERATIVE PRODUCTS COMPUTATION

In this section, we describe our implementation of the product computation for Cheon et al.'s scheme. We start by recalling the task to be computed. Given integers z_1, \dots, z_n , we want to compute products $P_i = \prod_{j=1}^i z_j$, for $i = 2, \dots, n$ with logarithmic multiplicative depth. Cheon et al. [15] propose to compute the products using a recursive algorithm that builds a binary tree of products. That is, we start with z_1, \dots, z_n , compute products $z'_1 = z_1 \cdot z_2, z'_2 = z_3 \cdot z_4, \dots$ for the first level and continue this recursively until there is a single element left in the last level. At each level, if the number of elements is odd, then the last element is left alone. This yields products for positions which are a power of 2. In the next step, we compute the missing products, i.e., for positions i that are not a power of 2, based on a multiplication of the results for the power of two cases and the power of 2 decomposition of i .

Our implementation described in Algorithm 13 stores the binary tree of products in a matrix $a[][]$ with $\log(n) - 1$ rows and $n/2$ columns, where row i contains $n/2^i$ non null elements, for $i = 1, \dots, \log(n) - 1$. Then, the first column contains the products P_i for i a power of 2.

For the remaining positions i not a power of 2, we decompose i in its power of 2: s_1, \dots, s_k such that $s_j = 2^{e_j}, i = s_1 + \dots + s_k$ and $s_1 > \dots > s_k$. Then, we use each s_j to select a corresponding product Q_j from the matrix computed above. We observe that if $s_j = 1$ then $Q_j = x[i]$. Otherwise, we compute the row r and column c of the product corresponding to Q_j , i.e., $Q_j = a[r][c]$ as follows:

- we remark that if $s_j = 2^e$, then the corresponding product lies in the row $r = e$ of the matrix.
- for the column, if $j = 1$, then it is the first column, i.e., $c = 1$ otherwise we compute the column as $c = \lceil t/s_j \rceil - 1$ where $t = s_1 + \dots + s_j$.

Finally, we collect all the products (Q_1, \dots, Q_k) and apply again a multiplication with logarithmic depth to compute $Q_1 \cdot \dots \cdot Q_k$ using the binary tree of products as above.

E CHOICE OF AHE SCHEME

For AHE, we choose ElGamal encryption [19] that we implement as elliptic curve ElGamal (ECE) [33, 34]. 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 $pk, sk \leftarrow \text{KGen}(\lambda)$: This algorithm randomly chooses $s \in \mathbb{Z}_p$ and outputs $sk = s$ and $pk = s \cdot P$ as private and public key.
- Encryption $c \leftarrow \text{Enc}(pk, m)$: This algorithm takes pk and a message m , then it chooses a random $r \in \mathbb{Z}_p$ and outputs the ciphertext $c = (r \cdot P, m \cdot P + r \cdot pk)$.
- Decryption $m \leftarrow \text{Dec}(sk, c)$: This algorithm takes sk and a ciphertext $c = (Q_1, Q_2)$, compute $Q = Q_2 - Q_1 \cdot sk$ and returns the discrete logarithm of Q on \mathbb{G} .

The above scheme is indeed AHE. If $c_1 = (r_1 \cdot P, m_1 \cdot P + r_1 \cdot pk)$ and $c_2 = (r_2 \cdot P, m_2 \cdot P + r_2 \cdot pk)$ are ciphertexts of two plaintexts m_1 and m_2 then $c_1 + c_2 = ((r_1 + r_2) \cdot P, (m_1 + m_2) \cdot P + (r_1 + r_2) \cdot pk)$ is a ciphertext of $m_1 + m_2$.

Algorithm 13 Iterative Products Computation

Require: array $z = z_1, \dots, z_n$

- 1: let P be an array of $n - 1$ elements
- 2: let $a[(1, \dots, \log(n) - 1), (1, \dots, n/2)]$ be a matrix
- 3: **for** $i = 1$ **to** $\log(n)$ **do**
- 4: **for** $j = 1$ **to** $n/2^i; j = j + 2$ **do**
- 5: **if** $j + 1 < n$ **then**
- 6: $r \leftarrow i$
- 7: $c \leftarrow j/2$
- 8: **if** $i = 1$ **then**
- 9: $a[r][c] \leftarrow z[j] \cdot z[j + 1]$
- 10: **else**
- 11: $a[r][c] \leftarrow a[i - 1][j] \cdot a[i - 1][j + 1]$
- 12: **end if**
- 13: **end if**
- 14: **end for**
- 15: **end for**
- 16: **for** $i = 2$ **to** n **do**
- 17: **if** $i = 2^e$ **then**
- 18: $P[i] \leftarrow a[e][0]$
- 19: **else**
- 20: let $i = s_1 + \dots + s_k$ such that $s_j = 2^{e_j}$, and $s_1 > \dots > s_k$
- 21: let Q be an array of $[1, \dots, k]$ of integers
- 22: **for** $j = 1$ **to** k **do**
- 23: **if** $s_j = 1$ **then**
- 24: $Q[j] \leftarrow z[i]$
- 25: **else**
- 26: $r \leftarrow \log(s_j)$
- 27: **if** $j = 1$ **then**
- 28: $c \leftarrow 1$
- 29: **else**
- 30: $c \leftarrow \lceil t/s_j \rceil - 1$
- 31: **end if**
- 32: $Q[j] \leftarrow a[r][c]$
- 33: **end if**
- 34: **end for**
- 35: $P[i] \leftarrow Q_1 \cdot \dots \cdot Q_k$
- 36: **end if**
- 37: **end for**
- 38:
- 39: **return** P

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

F MOST RECENT WORK

Most recent work include protocols [4, 20, 39] based on the arithmetic black-box (ABB) model. The ABB model allows addition

on encrypted inputs and constant multiplication. It can be implemented using secret sharing or AHE. Some ABB-based protocols ([17] and the improved version [39]) have some resemblance with our scheme as they rely on the bit decomposition of the inputs, and compute equality of bits and aggregate them. Other schemes rely on the extraction of the most significant bit of the difference of both inputs [4, 20].

When implemented with secret sharing, ABB-schemes have a very fast local computation and allow unconditional security. They support several parties (more than 2), where parties have symmetric role in the protocol, i.e. each party perform the same computation on secret-shared inputs. However, they also require an offline phase to generate so-called Beaver triples; they require secure channel to avoid share reconstruction by an eavesdropper; they usually run in asymptotically multiple rounds, which can affect the overall performance.

Our scheme is secure as long as the underlying encryption scheme is secure. We support only 2 parties, where each has a specific role: one party (server or evaluator) evaluates the tree and the other one (client or decryptor) decrypts the result. We do not need an offline phase, but only a one-time setup phase, where the decryptor generates a pair of private and public key and publishes the public key. We do not require a secure channel as each message is already encrypted. We have exactly a single round, where the client sends its encrypted input to the server and gets an encrypted result. An overview comparison is summarized in Table 5.

	[4, 20, 39]	Ours
Primitive	ABB	HE
Security	unconditional	computational
# of Parties	multiple	two
Role of Parties	symmetric	asymmetric
Offline Phase	yes	no
Secure Channel	yes	no
# Rounds	multiple	1
Crypto	symmetric	asymmetric

Table 5: Summary of Comparison to ABB-based Schemes

G EVALUATION OF DECISION TREES

A decision tree (DT) 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 a GT or LT comparison between a threshold value and an attribute of the input to be classified. In a private DT setting, a server holds a private tree model and a client holds a private attribute or feature vector. The goal is to classify the client’s input using the server’s 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. [51] and Tai et al. [42] proposed a private DT protocol, that uses the DGK comparison and AHE.

We implemented the private decision tree protocol of Tai et al. [42] in Java. Our implementation is not optimized and focuses on the main computation (GT comparison of features and thresholds, and paths aggregation), i.e., we ignored e.g. zero-knowledge proof, compression of elliptic curve points or network communication. We then evaluated the decision tree protocol instantiating the GT comparison with DGK [18], Veugen [49] and our scheme and observed a performance result comparable to Table 3. Basically, one

can expect an improvement proportional to the number of comparisons. The result averaged over 100 runs is shown in Table 6 for a decision tree with depth $d = 17$, number of decision nodes $m = 58$ and number of features $n = 57$, which corresponds to the parameters of the Spambase dataset as in [42, 45, 47, 51]. We used bitlength $\mu = 8, 16, 32, 64, 128$ for the features and thresholds.

	$\mu = 8$	$\mu = 16$	$\mu = 32$	$\mu = 64$	$\mu = 128$
DGK [18]	3084.70	4234.50	7218.10	11981.80	27707.70
Veugen [49]	1994.00	2557.90	4525.40	8100.80	20254.50
Ours	1875.90	2472.00	4052.30	7512.80	17663.60

Table 6: Performance Comparison of Running Time (in ms) for PDTE