Secure Branching Program Evaluation

Jonas Janneck[1], Anas Boudi[2], Anselme Tueno[1], Matthew Akram[1]

[1] SAP SE
firstname.lastname@sap.com
[2] Zama
anas.boudi@zama.ai

August 18, 2022

Abstract

We address the problem of privately evaluating a branching program on encrypted data. This scenario is a 2-party protocol consisting of a server and a client. The server privately holds a branching program which is a representation of a boolean function using a directed acyclic graph. The client holds a secret input to the branching program. The goal of the computation is to evaluate the client's input on the server program such that only the result is revealed to the client, and nothing is revealed to the server. To solve this problem Ishai-Paskin introduced a public-key encryption scheme that is based on Damgård-Jurik additively homomorphic encryption and has the property, that given a branching program P and an encryption c of an input y, it is possible to efficiently compute a succinct ciphertext c' corresponding to $P(y)$. The entire computation is done by the server relying on the fact that Damgård-Jurik scheme has length-flexible ciphertexts which allows multiplications between ciphertexts of different sizes under the same encryption key. Although the decryption of the Damgård-Jurik scheme is theoretically efficient, the size of c' and the decoding time depend on the depth of the branching program. In this paper, we propose a new scheme where the input is instead encrypted using fully homomorphic encryption and discuss different variants and optimizations. The entire computation is also done by the server but the size of the resulting ciphertext is independent of the depth of the program. We implement Ishai-Paskin and our scheme and show that the running time of our scheme is an order of magnitude smaller.

1 Introduction

A branching program (BP) is a representation of a boolean function using a directed acyclic graph. Depending on the function to be evaluated, we can sometimes represent a BP using a directed tree. The result of an evaluation can then be obtained by traversing the tree for a chosen input. For example, special

cases of BPs are ordered binary decision diagrams or finite automata.
In this paper we consider a 2-party protocol. There is a server holding a BP and there is a client holding an input to the program. The goal is for the client to obtain the result of the evaluation of the servers BP on the client's input, without the server learning anything about the client's input, or the client learning anything about the server's BP (aside from the result of its query). Secure BPs can be used in applications such as Private Information Retrieval or the evaluation of 2-DNF formulas or degree-2 polynomials.

One approach is to use homomorphic encryption to ensure both parties' privacy, by letting the server evaluate the BP on encrypted inputs. By representing the BP as a tree, the output can be computed by aggregating all possible paths. We must therefore be able to apply multiplication as well as addition on encrypted data. The Ishai-Paskin protocol [20] solves this problem using the Damgård-Jurik cryptosystem [12]. Although the Damgård-Jurik scheme only allows for homomorphic additions, it is length-flexible and can therefore be used for multiplications between ciphertext spaces of different sizes. However, a significant drawback is that the length of the ciphertext is increasing in the tree's depth.

Instead, we present an approach relying on fully homomorphic encryption (FHE). This reduces the query time and moreover, provides a fixed-length ciphertext output, independent of the tree's depth. In contrast to partially homomorphic schemes, for many FHE schemes, we must take the multiplicative depth into account. For many FHE schemes, an expensive bootstrapping procedure must be invoked after a given number of multiplications. To prevent this, we can avoid bootstrapping operations, by reducing the multiplicative depth of ciphertexts. This can be achieved by simplifying the tree representation of our function to a smaller directed acyclic graph. We also present some optimizations of the basic protocol as well as some extensions and generalizations to the basic case. We implement our approach using the BGV scheme [6] and TFHE [10]. Using the gate-by-gate bootstrapping mode of TFHE, the multiplicative depth is not relevant since a (comparably efficient) bootstrapping operation is executed after every computation. Evaluating the three variants, we observe that for higher depths of the BP (greater than five) our implementation using BGV is faster than Ishai-Paskin's protocol. The implementation of our protocol using TFHE outperforms both other variants and is especially about 400 times faster than Ishai-Paskin.

2 Related Work

Our work is related to multiparty computation (MPC) where multiple parties want to securely compute a public function on their private data such that only the result is revealed [4,8,11,14,15,17,22,34]. A special case is 2-PC with exactly two computing parties. Two party computation is used in private function evaluation (PFE) where one of the party (the server) holds a private function and the second party (the client) holds a private input, with to goal to evaluate

the client's input on the server function and reveal the result only to the client [24, 25]. There are different techniques used to design 2-PC protocols, including garbled circuit (GC) [3], secret sharing [27] and homomorphic encryption (HE) [6, 10, 16]. Using HE for PFE, the client holds a pair of encryption keys, and the server evaluates the client's encrypted input on the private function and sends an encrypted result back to the client. HE can be additive, meaning that only addition is possible on encrypted data [12, 13, 26]. Using additive HE with a specific property called length-flexibility [13], Ishai and Paskin proposed an encryption scheme for evaluating branching programs (BP) on encrypted data [20]. A special case of BPs are decision trees, which are used in the context of PFE, where the server holds a private decision tree and the client wants to securely learn the classification of its private input [2,5,7,21,23,28,29,31,33]. In [29], Tueno et al. used FHE to evaluate decision trees on encrypted data. Our work relies on this to evaluate branching program using FHE instead of additive HE. The idea of evaluating a BP on encrypted data, has been used for integer comparison, where the server represents the comparison with a given value as a binary tree and evaluates it securely on the client's input [30].

3 Preliminaries

3.1 Homomorphic Encryption

Homomorphic encryption (HE) allows computations on ciphertexts by generating an encrypted result whose decryption matches the result of a function on the plaintexts [6, 10, 16].

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

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

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

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

We define $\boxplus$, as the homomorphic addition of two cipherexts and $\boxdot$, as the homomorphic multiplication of two ciphertexts.

3.2 Damgård-Jurik Scheme

The Damgård-Jurik cryptosystem [12] is a generalization of the Paillier cryptosystem [26]. It operates in the multiplicative group $\mathbb{Z}^*_{n^{s+1}}$ for a predefined s. The special case $s = 1$ is equivalent to the Paillier cryptosystem. The group defined above can be divided into two groups, a cyclic group G of order n^s and a group H isomorphic to $\mathbb{Z}^*_n$, such that $\mathbb{Z}^*_{n^{s+1}} = G \times H$.

For key generation, we choose $n = p \cdot q$ as an RSA modulus and $g \in \mathbb{Z}^*_{n^{s+1}}$ such that $g \equiv (1+n)^j x \mod n^{s+1}$ for a known j which is coprime to n and $x \in H$. Let $\lambda = lcm(p-1, q-1)$ and choose d such that $d \mod n \in \mathbb{Z}^*_n$ and $d \equiv 0 \mod \lambda$. The values n and g function as the public key, and d as the private key. The plaintext space of the scheme is $\mathbb{Z}_{n^s}$. The encryption of an element $i \in \mathbb{Z}_{n^s}$ is $E(i) = g^i r^{n^s} \mod n^{s+1}$ for a randomly chosen $r \in \mathbb{Z}^*_{n^{s+1}}$. Decrypting a ciphertext c can be done by computing $c^d \mod n^{s+1}$.

The following two properties are of particular interest for Ishai-Paskin's application. The scheme is additively homomorphic, since multiplying two ciphertexts corresponds to an addition of the underlying plaintexts. We can also multiply ciphertexts by plaintext constants, i.e. given a ciphertext $[\![m]\!] \in \mathbb{Z}^*_{n^{s+1}}$ and a plaintext $k \in \mathbb{Z}_{n^s}$, we can can compute $[\![m]\!]^k \equiv g^{km} r'^{n^s} \equiv [\![km]\!] \mod \mathbb{Z}^*_{n^{s+1}}$. The encryption scheme is also length-flexible. For $s = i$, we have a plaintext space $\mathbb{Z}^*_{n^i}$ and ciphertext space $\mathbb{Z}^*_{n^{i+1}}$. This ciphertext space is the plaintext space of a scheme with $s = i + 1$ as illustrated in Figure 1. We can therefore use the multiplication with plaintext constants to implement a multiplication of ciphertexts if we consider ciphertexts from different schemes, i.e. with different parameters s.

s	Plaintext	$\rightarrow$	Ciphertext
$s = 1$	$\mathbb{Z}_n$	$\rightarrow$	$\mathbb{Z}_{n^2}$
$s = 2$	$\mathbb{Z}_{n^2}$	$\rightarrow$	$\mathbb{Z}_{n^3}$
$s = 3$	$\mathbb{Z}_{n^3}$	$\rightarrow$	$\mathbb{Z}_{n^4}$
$\vdots$	$\vdots$	$\vdots$	

Figure 1: Length-flexibility of Damgård-Jurik

4 Definitions

In this section, we introduce the data-structure we use in order to represent and evaluate BP's, as well as the security goal for a secure evaluation procedure.

4.1 Branching programs

A branching program (BP) is a directed acyclic graph that can be used to represent a boolean function.

Definition 4.1 (Branching Program [20])**.** *A* branching program P *over input variables* $x_1, \ldots, x_n$*, input domain* I*, and output domain* O *is a tuple*

$$P = (G = (V, E), v_0, T, \psi_V, \psi_E)$$

with:

- $G = (V, E)$ *a directed acyclic graph with* V *the set of nodes and* E *the set of edges and* $\Gamma(v)$ *the children set of a node* v*,*
- v_0 *an initial node with indegree 0,*
- $T \subseteq V$ *set of terminal or leaf nodes with outdegree 0,*
- $\psi_V : V \to \{1, \ldots, n\} \cup O$ *a node labeling function assigning an output value to terminal nodes and a variable index to inner nodes* $V \backslash T$*,*
- $\psi_E : E \to \wp(I)$[1] *an edge labeling function such that every edge is mapped to a non-empty set and for every node* v *the sets labeling the edges to nodes in* $\Gamma(v)$ *form a partition of* I*.*

Even though, the formal definition of a BP is very generic, we start with a simpler instantiation and only consider binary BPs, i.e. $I = O = \{0, 1\}$, which we represent as binary trees. In later chapters, we also extend our approach to generalized variants.

We further recursively assign a unique index to every node. The root has index 0 and for an inner node x_i with index i, its left child has the index $2i + 1$ and its right child has the index $2i + 2$. Each leaf v' is assigned a boolean output $O_{v'}$. The edge to the left child of any inner node is then labeled with 0, and the edge to the right child with the label 1.

By iteratively unifying equivalent paths, we can obtain a smaller binary tree, that represents the same function. This is referred to as *pruning* the tree. Figure 2 illustrates a simple branching program with a root labeled x_0, an inner node labeled x_1 and three leaves with output O_1, O_2 and O_3.

[1] $\wp(I)$ denotes the power set of I.

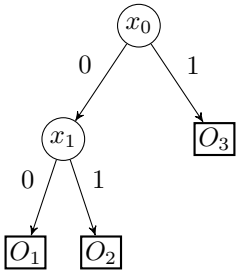


Figure 2: Example of a Binary Tree of Depth 2

The *size* of a BP is given by its number of inner nodes. For example, the BP in Figure 2 has two inner nodes and three leaves; thus, its size is 2. The *depth* of a node corresponds to the distance between it and the root. The depth of a binary tree is then the maximum of the leaves' depth.

The nodes of a tree can be separated into disjoint layers, where the j'th layer L_j consists of all the nodes with distance $j+1$ from the root. An inner node $x_i \in L_j$ therefore has its children in L_{j+1}. For example the complete binary tree in Figure 3 below represents the same boolean function as the one above.

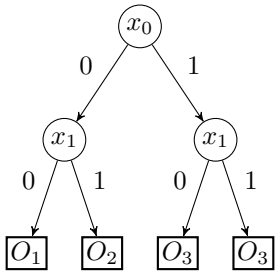


Figure 3: Example of a Complete Binary Tree of Depth 2

4.2 Branching Program Evaluation

The following definition describes the evaluation if a BP.

Definition 4.2 (BP Evaluation)**.** *Let $P = (G = (V, E), v_0, T, \psi_V, \psi_E)$ be a BP. The output of P on inputs $x_1, \ldots, x_n$ can be extracted by following the unique path from v_0 to a terminal node v_l such that every node v and its successor node v' fulfill $x_{\psi_V(v)} \in \psi_E(v, v')$. Then, the output is $P(x_1, \ldots, x_n) = \psi_V(v_l)$.*

In other words, we traverse the graph in a very natural manner. For every node we check the corresponding input value which indicates which edge we have to take next. Following such a path we end up with a leaf label which is the final result of the computation.

4.3 Data Structure

The data structure we use to represent a BP is a binary tree consisting of inner nodes and leaves. Each inner node has two child nodes and leaves have no children. There is a single node with no parent node, that is called the *root*. Let v be a node in the tree. We define a node data structure Node consisting of the following attributes.

- v.parent: a value representing the pointer to the parent node,
- v.left: a value representing the pointer to the left child node,
- v.right: a value representing the pointer to the right child node,
- v.lEdge: a bit representing the *edge label* to the left child node,
- v.rEdge: a bit representing the *edge label* to the right child node,
- v.cLabel: a value representing a *node label*,
- v.level: an integer representing the *node level* in the tree,
- 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. 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 leaves and undefined for inner nodes. The level v.level is 1 for the root, 2 for the child nodes of the root and so on. The cost attribute v.cost is computed during tree evaluation. Note that this data structure represents the basic case, where we represent our function as a binary tree. We further generalize this structure in Section 7, for more complex scenarios.

4.4 Security Goal

The protocol consists of a server holding the branching program P, and a client holding an input y. We assume that the client's input and the program's input consist of μ−bit strings. The ideal functionality $\mathcal{F}_{\mathsf{BP}}$ takes y from the client and P from the server, computes $P(y)$, and outputs it to the client without leaking information to the server, as described in Figure 4.

We rely on simulation-based security for deterministic functionalities in the two party setting [19]. To this end, we define:

Definition 4.3 (Probability Ensemble)**.** *A* probability ensemble $X = X(a, \lambda)_{a,\lambda}$ *is an infinite sequence of random variables indexed by* $a \in \{0,1\}^*$ *and* $\lambda \in \mathbb{N}$.

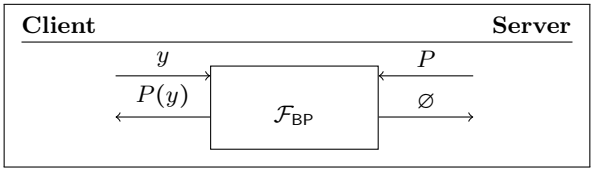


Figure 4: Illustration of a secure BP evaluation

Definition 4.4 (Computational Indistinguishability)**.** *Let $a \in \{0,1\}^*$ and $\lambda \in \mathbb{N}$. Two probability ensembles $X = X(a,\lambda)_{a,\lambda}$ and $Y = Y(a,\lambda)_{a,\lambda}$ are* computationally indistinguishable*, or $X \overset{c}{\equiv} Y$, if for every non-uniform probabilistic-polynomial time (PPT) algorithm D there exists a negligible function μ such that for every a and λ it holds*

$$|Pr[D(X(a,\lambda)) = 1] - Pr[D(Y(a,\lambda)) = 1]| \leq \mu(\lambda).$$

We further define $\text{VIEW}_i^{\Pi}(a,b,\lambda)$ as the view of party i during the execution of protocol Π on inputs a and b with security parameter λ. This is the party's input and all received messages.

A protocol correctly implements a BP evaluation if after the computation the output $P(y)$ to the client is correct. A protocol securely implements the BP evaluation $\mathcal{F}_{\mathsf{BP}}$ if the client learns only the result $P(y)$ and nothing else, and the server learns nothing.

Definition 4.5 (Semi-Honest Security for BPs)**.** *A protocol Π_{BP} securely computes $\mathcal{F}_{\mathsf{BP}}$ in the presence of semi-honest adversaries if the output is correct and*

- *there exists a PPT algorithm $\mathcal{S}_1$ that simulates the server's view $\mathsf{View}_1^{\Pi_{\mathsf{BP}}}$ given only P such that:*

$$\{\mathcal{S}_1(P,\varnothing)\}_{P,y,\lambda} \overset{c}{\equiv} \{\mathsf{View}_1^{\Pi_{\mathsf{BP}}}(P,y,\lambda)\}_{P,y,\lambda}, \tag{1}$$

- *there exists a PPT algorithm $\mathcal{S}_2$ that simulates the client's view $\mathsf{View}_2^{\Pi_{\mathsf{BP}}}$ given only y and $P(y)$ such that:*

$$\{\mathcal{S}_2(y,P(y))\}_{P,y,\lambda} \overset{c}{\equiv} \{\mathsf{View}_2^{\Pi_{\mathsf{BP}}}(P,y,\lambda)\}_{P,y,\lambda}. \tag{2}$$

5 Ishai-Paskin's protocol

Evaluating an encrypted BP is slightly more complicated than evaluating a non-encrypted BP. In order to evaluate an encrypted BP, we need to evaluate the paths from the root to each leaf individually. This is done by comparing the i'th input bit to the i'th edge label, and then multiplying all of the results together. We refer to this as the *cost* of the path. If all of the input bits are equal to

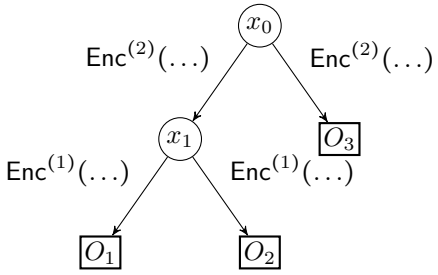


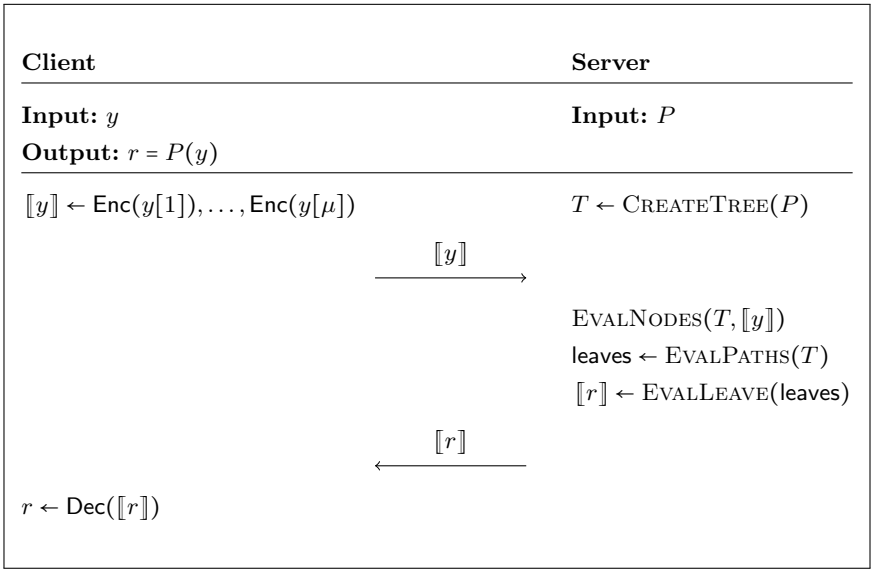
Figure 5: Example for Ishai-Paskin BP Structure

the respective edge labels, then all of the comparison bits are equal to 1, and the cost of the path is 1. If at least one of the input bits is not equal to the respective edge label, then at least one of the comparison bits is equal to 0, and the cost of the path is 0. On any input, there is therefore exactly one path, from the root to a leaf, of cost 1. To then obtain the output of the BP, we multiply the cost at each leaf, with its label, and then sum up all of these results. Since exactly one leaf has non-zero cost, we obtain only the label of that leaf as a result.

The protocol of Ishai and Paskin is based on the same BP structure we described in Section 4. For simplicity, we only describe the binary case, i.e. where every inner node has exactly two children. The protocol of Ishai and Paskin [20] evaluates the BP bottom up. They start with the labels of the leaves and evaluate the deepest inner nodes given an input. This is theoretically done by an Oblivious Transfer (OT) query returning the respective leaf which is indicated by the associated input. Specifically, if we have a left child with label l_0 and a right child with label l_1, we query the label l_{y_i} where y_i is the input bit at level i. They implement such an OT query by using the Damgård-Jurik cryptosystem as described in Section 3.2.

In the following, we describe some details of our own implementation of Ishai-Paskin's protocol. The client encrypts its input bits with an encryption instance depending on the level in the BP. Starting with $s = 1$ for the deepest input level, and with an increasing s for upper levels, for s the encryption parameter for Damgård-Jurik, see Section 3.2. By $\mathsf{Enc}^{(s)}$, we denote an encryption with parameter s. The server can then homomorphically compute decision bits on each of the branches indicating which branch is taken by the input. We aggregate all the decision bits by evaluating the tree bottom up as described above. This needs a multiplication of a decision bit with the label of the previous level and an addition afterwards. While the addition is a basic property of the Damgård-Jurik scheme, the multiplication follows from the length-flexibility and the choice of different encryption instances based on the considerations in Section 3.2. The structure is illustrated in Figure 5.

The problem of such a procedure is that the ciphertext size is increasing during the process and depends on the BP depth. On the one hand, this leads to higher communication cost between server and client since they have to send



Protocol 6: Protocol Overview

larger ciphertexts. On the other hand, the computation time strongly increases with the use of schemes with a larger parameter s because we are operating in the group $\mathbb{Z}^*_{n^{s+1}}$. Our solution, on the other hand, relies on Fully Homomorphic Encryption (*FHE*) which comes with a compactness property, i.e. the ciphertext size is bounded and independent of the evaluation that is applied.

6 Our Protocol

Our protocol consists of an offline and an online phase. In the offline phase, the server and client initialize their settings; the server creates a tree based on its program and the the client encrypts its input. In the online phase, the client sends the bit-wise encrypted input to the server, which evaluates the input homomorphically and sends the encrypted result to the client. Figure 6 gives an overview of the protocol.

6.1 Algorithms

We now introduce some subroutines that our protocol will make use of. We then present a detailed overview of the implementation of our protocol based on leveled FHE (Section 6.3) and bootstrapping FHE (Section 6.4).

Initialization. The initialization consists of a one time key generation. The client generates an appropriate triple $(\mathsf{pk}, \mathsf{sk}, \mathsf{ek})$ of public, private and evalua-

tion keys for a homomorphic encryption scheme. Then, the client sends $(\mathsf{pk}, \mathsf{ek})$ to the server. For each computation, the client encrypts its input and sends it to the server. As explained before, the actual computation on the binary tree is done only by the server. The following steps describe the computation done by the server, starting by creating the binary tree.

Creating the Binary Tree. We build a tree based on the BP as defined in Section 4.3. Depending on the input, the tree might be complete, i.e. each leaf has the same depth, or it might already be pruned. To analyze the same data structure for the same input, we prune every tree as described in the following. If for a node v the right and the left child have the same cLabel, we assign node v their cLabel and remove the children. Especially for a small output range, e.g. $\{0, 1\}$, pruning the tree comes with a significant reduction of the complexity of the tree structure. This step can be pre-computed since it does not depend on the client. Moreover, the server can use the representation of the BP several times.

Computing Decision Bits. The client sends each input y bitwise encrypted. Let $\overline{y} = y[1], \ldots, y[\mu]$ be the corresponding bit string and $[\![\overline{y}]\!] = [\![y[1]]\!], \ldots, [\![y[\mu]]\!]$ the corresponding ciphertexts. The server computes the decision bits at each inner node v by comparing each $[\![y[i]]\!]$ against the edge labels of node v. This comparison is a bit equality test that returns $[\![1]\!]$ if the two bits are equal and $[\![0]\!]$ otherwise. We could therefore implement it using the Boolean XNOR gate. However, since the server knows the labels of the branches, it is easier to apply a Boolean XOR gate to the negated label. This is equivalent to addition modulo 2. Thus, we obtain $[\![y[i]]\!] \boxplus 1$ for the left branch and analogously just $[\![y[i]]\!]$ for the right branch. The computation of decision bits is illustrated in Algorithm 7.

Note that the decision bits are the same for every node on the same level. We can therefore reduce the complexity by computing the decision bits only once per level and using it for any node at this level.

Aggregating Decision Bits. For each leaf v, the server aggregates the comparison bits along the path from the root to v. This is done using homomorphic multiplication of the decision bits along the path. The aggregated result is stored at the leaf of the corresponding path. We implement it using a queue and traversing the tree in BFS order as illustrated in Algorithm 8.

Finalizing. After aggregating the decision bits along the paths, each leaf v stores either $[\![v.\mathsf{cost}]\!] = [\![0]\!]$ or $[\![v.\mathsf{cost}]\!] = [\![1]\!]$, whereby there is a unique leaf with $[\![v.\mathsf{cost}]\!] = [\![1]\!]$ and all other leaves have $[\![v.\mathsf{cost}]\!] = [\![0]\!]$. Then, the server aggregates the costs at the leaves by computing for each leaf v the value $[\![v.\mathsf{cost}]\!] \boxdot [\![v.\mathsf{cLabel}]\!]$ and summing up the results of all leaves. This computation is illustrated in Algorithm 9.

```
1:  function EvalNodes(root, [[ȳ]])
2:      let Q be a new queue
3:      Q.enqueue(root)
4:      parse [[ȳ]] to [[y[1]]], ..., [[y[μ]]]
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          if v.left ≠ null then
8:              [[v.left.cost]] ← [[y[v.level]]] ⊞ 1
9:              Q.enqueue(v.left)
10:         if v.right ≠ null then
11:             [[v.right.cost]] ← [[y[v.level]]]
12:             Q.enqueue(v.right)
```

Algorithm 7: Evaluating Nodes by Computing Decision Bits

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

Algorithm 8: Evaluating Path by Aggregating Decision Bits

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

Algorithm 9: Evaluating Leaves by Summing up the Costs at Leaves

6.2 Security

The correctness of the protocol is given since our construction only allows the cost of the corresponding leaf to be 1. Hence, the result of the protocol is exactly the leaf label which is the desired program output. Privacy is shown in the proof of the following theorem.

Theorem 6.1. *If the underlying encryption scheme is IND-CPA secure, Protocol 6, called* Π*, securely computes functionality* $\mathcal{F}_{\mathsf{BP}}$ *in the presence of a semi-honest adversary.*

Proof. Correctness is given by previous elaborations.

For privacy, we start with the simulator of the server, $\mathcal{S}_1$. The view of the server only consists of μ encrypted bits. Therefore, the simulator only encrypts μ random bits and sends it to the server. Since the encryption scheme is IND-CPA secure, there is no non-negligible probability to distinguish the μ encrypted random values from the actual messages. Hence, the simulator is indistinguishable from the actual view and it holds $\{\mathcal{S}_1(P, \varnothing)\}_{P,y,\lambda} \stackrel{c}{\equiv} \{\mathsf{View}_1^{\Pi}(P, y, \lambda)\}_{P,y,\lambda}$.

The simulator of the client obtains y and $P(y)$. The client's view consists of the resulting ciphertext, i.e. $[\![P(y)]\!]$. Hence, the simulator encrypts the result $P(y)$ such that the simulated values decrypt to the correct result and the decrypted values cannot be distinguished. However, the ciphertexts itself also have to be indistinguishable to obtain a correct simulation. Using an FHE scheme, one can distinguish a fresh ciphertext from a ciphertext with operations applied since they have a different noise level. To this end, the simulator can apply as many additions and multiplications as the original protocol. For each addition, the simulator adds 0, for each multiplication it multiplies the result by 1 to obtain a ciphertext with the same noise level as the real protocol. Thus, the simulator cannot be distinguished from the real client's view. □

6.3 Leveled FHE

One way to implement our basic protocol is to rely on a leveled FHE scheme. We now present some specifications and refinements compared to the basic algorithms described so far.

Batch Multiplication. For leveled FHE, the multiplicative depth of the program to be evaluated is relevant because the parameters of the scheme have to be chosen accordingly. A larger multiplicative depth leads to larger parameters with higher computational overhead. We therefore adapt Algorithm 8, in order to decrease the multiplicative depth required to evaluate paths, at the cost of increasing the total number of multiplications. The idea is based on the private evaluation of decision trees from Tueno et al. [29].

Given the ciphertexts $a_1, \ldots, a_n$, define the function g as

$$g(\{a_1, \ldots, a_n\}) \coloneqq \begin{cases} a_1, & \text{if } n = 1, \\ g(\{a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}\}) \boxdot g(\{a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n\}, & \text{if } n > 1 \end{cases}$$

We refer to the application of g onto a given set of ciphertexts as *batch multiplication*. This is useful, since it allows us to decrease the multiplicative depth of this series of multiplications by the following theorem.

Theorem 6.2. *The operation $g(\{a_1, \ldots, a_n\})$ has a multiplicative depth of $\lceil \log(n) \rceil$.*

Proof. We proceed with this proof inductively. For $n = 1$, no multiplication is performed, and thus the multiplicative depth of $g(a_1)$ is zero. For $n \geq 2$ we know that $g(\{a_1, \ldots, a_n\}) = g(\{a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}\}) \boxdot g(\{a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n\}$, and hence the multiplicative depth is $\max\{\lfloor \log \frac{n}{2} \rfloor, \lceil \log \frac{n}{2} \rceil\} + 1$ by our inductive assumption. We therefore see that the multiplicative depth of $g(\{a_1, \ldots, a_n\})$ is $\lceil \log \frac{n}{2} \rceil + 1 = \lceil \log n \rceil$. □

Analysis. Using batch multiplication, we can reduce the multiplicative depth of evaluating a path from d to $\log(d)$. However, this does increase the number of multiplications required to evaluate the tree from $\mathcal{O}(2^d)$ to $\mathcal{O}(\log(d) \cdot 2^{d \log(d)})$. A detailed description of how this is achieved can be found in Appendix A.

6.4 (Bootstrapping) FHE

For FHE schemes capable of bootstrapping we distinguish between two variants.

There are schemes which apply a bootstrapping operation when the noise of the ciphertext is too high [16]. Since a multiplication usually adds a lot of noise compared to an addition, it might be wise to reduce the multiplicative depth to avoid expensive bootstrapping operations. In this case, we could apply batch multiplication, presented in Section 6.3. However, this procedure leads to an increased number of multiplications which leads to a trade-off between the number of multiplications and the number of bootstrapping operations. The most efficient choice therefore depends on the parameters of the chosen scheme.

Other variants of FHE schemes use gate-by-gate bootstrapping [9]. They apply a bootstrapping operation after each gate evaluation without considering the actual noise. In this scenario, it is not reasonable to reduce the multiplicative depth but to reduce the overall homomorphic operations. In the following, we focus on this case.

	Const. XOR	Const. AND	Hom. XOR	Hom. AND
Node Evaluation	d	-	-	-
Path Evaluation	-	-	-	$2^{d+1}-2$
Leaves Aggregation	-	2^d	2^d-1	-
Total	$\boldsymbol{d}$	$\boldsymbol{2^d}$	$\boldsymbol{2^d-1}$	$\boldsymbol{2^{d+1}-2}$
Total Hom. Gates			$\boldsymbol{2^{d+1}+2^d-3}$	

Table 1: An overview of the number of required operations in Bootstrapping FHE mode. The symbol "-" indicates that there is no such operation in this step.

Analysis. Since we do not apply any further optimizations to improve the running time, we can use the algorithms of our basic protocol from Section 6, since they already optimize the number of homomorphic operations. We consider the worst case scenario, i.e. a complete tree of depth d. An overview of the required number of each operation can be found in Table 1. Note that, due to pruning and further optimizations, the worst case is very unlikely to happen (e.g. with a probability of 2^{-d} for binary outputs). If we rely on a binary scheme, we can represent an addition in $\mathbb{Z}_2$ by an XOR -gate. A multiplication is given by an AND-gate. For the following description, addition/XOR and multiplication/AND are used for the same operation.

1. To evaluate decision nodes, we need one constant XOR operation. Since all nodes on one level have the same value, we need d constant XOR operations.
2. The path evaluation needs two homomorphic AND operations per inner node. That is $2^{d+1}-2$ homomorphic AND gates.
3. Leaves aggregation needs one constant AND operation per leaf and additionally homomorphic XOR operations to sum up all the leaf results. This yields 2^d constant AND gates and 2^d-1 homomorphic XOR gates.

This is in total $2^{d+1}+2^d-3$ homomorphic gates.

7 Generalizations and Optimization

In this section, we present optimizations to our basic protocol to achieve a more efficient solution.

7.1 Reducing Computation Cost

Algorithm 7 is very efficient since there is exactly one homomorphic evaluation at each node. We can even save this operation by computing it on plaintext at the client's side. Instead of sending $[\![y[1]]\!], \ldots, [\![y[\mu]]\!]$, the client sends

$[\![y[1]]\!], [\![\neg y[1]]\!], \ldots, [\![y[\mu]]\!], [\![\neg y[\mu]]\!]$. This doubles the number of ciphertexts the client has to send for each input. The server can now evaluate each node without any homomorphic operation by labeling the left branch with $[\![y[i]]\!]$ as before and the right branch with $[\![\neg y[i]]\!]$. This is very similar to the procedure from [20].

7.2 Reducing Communication Cost

We can further reduce the communication costs using pre-computation. For an input y the client sends a random ciphertext $[\![-r]\!]$ in the offline phase. In the online phase, the client can send $y + r$ which is not an FHE ciphertext and thus much more efficient. Here, "+" is the operation on the underlying group, e.g. $\oplus$ for binary representations. The security is still guaranteed based on the one-time-pad. Then, the server can compute $[\![y + r]\!] \boxplus [\![-r]\!]$ to obtain the actual input.

7.3 Non-binary Trees

We can generalize our approach using a branching program by relying on an m-ary tree instead of a binary tree. Now, we have to parse the input element by base m and create a tree with m children for each inner node. The client has to encrypt values from $\{0, \ldots, m-1\}$ which reduces the number of ciphertexts that have to be sent. Moreover, the computation time on the server can be reduced as well, since the server can parallelize the execution of independent sub-trees. The only difference in the evaluation of an m-ary tree is the node evaluation since we have to take m branches into account. We present two possibilities to evaluate the nodes of an m-ary tree.

Precomputed Booleans. In order to get the correct input bit, the client could precompute it on plaintext. This idea is based on the construction from Section 7.1. Instead of an encrypted input digit $[\![y_i]\!]$, the client sends the encrypted bits $[\![y_i == 0]\!], \ldots, [\![y_i == (m-1)]\!]$. For node i and branch j the server can take the j-th encrypted boolean as the decision bit on that branch. This procedure of course neglects the advantage of sending less ciphertexts. However, compared to the binary tree and the extension from Section 7.1 the amount of data being sent does not increase significantly.

Polynomial Evaluation. Another variant is to evaluate a polynomial on the server's side to get the decision bit of a branch. For the i-th node and the j-th branch we can homomorphically evaluate the polynomial

$$P_j = \prod_{k=0, k \neq j}^{m-1} \frac{[\![y_i]\!] - k}{c_j},$$

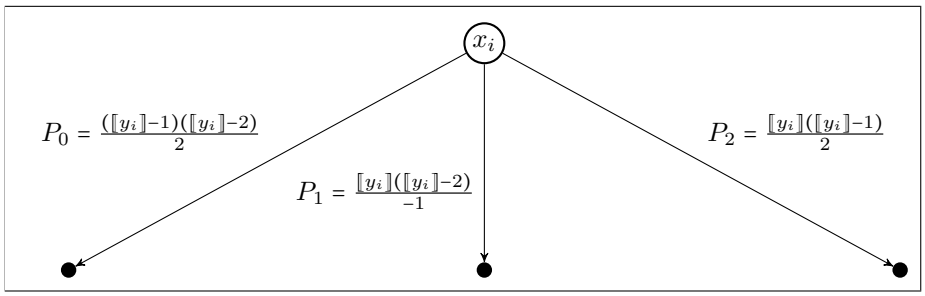


Figure 10: Example of polynomial evaluation for $m = 3$ at node x_i

for a constant c. If y_i is any value but j, the product is an encryption of 0. We choose c such that we obtain an encryption of 1 in the other case:

$$c_j = \prod_{k=0,k\neq j}^{m-1} (j-k).$$

An example is presented in Figure 10.

This procedure increases the multiplicative depth of a decision bit but it is not worse than the basic case since we reduce the tree depth at the same time by using m-ary trees instead of binary trees.

We now investigate how to efficiently evaluate these polynomials. The goal is to evaluate one polynomial on each of m branches. In the following, we ignore the constant term to normalize the polynomial since it can be easily precomputed and multiplied in the online phase using only one constant multiplication. Hence, we consider polynomial P_j for branch j:

$$P_j(y) = \prod_{i=0,i\neq j}^{m-1} (y-i).$$

All polynomials have degree $n-1$. For the first branch, we obtain

$$P_0(y) = \prod_{i=1}^{m-1} (y-i).$$

We can evaluate it in the offline phase to obtain the coefficient form, i.e. an array of n coefficients starting with the highest degree x^{m-1}:

$$(c_{m-1}, \ldots, c_0).$$

Some observations:

- $c_{m-1} = 1$ for every polynomial
- $c_0 = 0$ for every polynomial except the first (P_0)

Let now $(c_{m-1}, \ldots, c_0)$ be the coefficient form of polynomial P_0. We can then precompute the coefficient forms of $P_1, \ldots, P_{m-1}$. For polynomial P_j, we iteratively compute

$$(c^j_{m-1}, \ldots, c^j_0) = (1, c_{m-2} + jc_{m-1}, c_{m-3} + j(c_{m-2} + jc_{m-1}), \ldots).$$

In a more compact form, we compute the i-th coefficient of P_j (c^j_i) based on P_0:

$$c^j_i = c_i + jc^j_{i+1},$$

starting with $c^j_{m-1} = 1$.

In the online phase, we only need the powers of y which might be computationally expensive. Then, we can use the coefficient forms to evaluate the polynomial by using constant multiplications and additions.

Overall effort per tree level (online phase):

- Powers from $y^2, y^3, \ldots, y^{m-1}$ (ciphertext powers)
- $m \cdot (m-2) = m^2 - 2m$ constant multiplications
- $m \cdot (m-2) + 1 = m^2 - 2m + 1$ additions (+1 since the first polynomial has a constant term)

We can even decrease the computational complexity further. So far, we implicitly assumed the computation to be done over some field (we assumed c_j to have an inverse element). If we introduce some further constraints, there occur some quite practical consequences. The output of the polynomial is either 0 or 1 and the input is at most m. Then, it is sufficient to compute over $\mathbb{Z}_m$ which is a field if we assume m to be prime.[2] The improvement can be illustrated by considering polynomial P_0 which simplifies to

$$P_0(y) = y^{m-1} - 1 \mod m$$

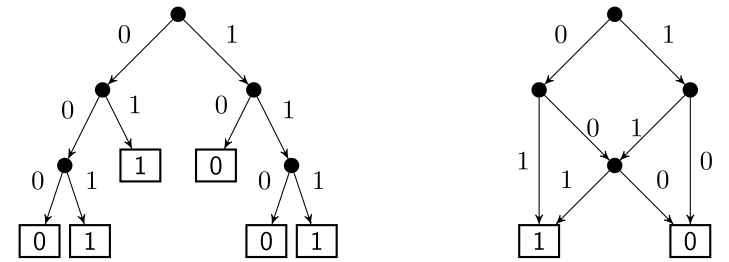
which saves many computations.

Finally, we can introduce a trade-off between communication and computation. The most expensive operations are ciphertext-ciphertext multiplications. For evaluating the polynomials for multiple branches, these are the ciphertext powers to compute. To save some computation time, the client could precompute these powers. This can be done in the offline phase and especially on plaintexts before the encryption step.

7.4 BDD

Our protocol is based on the evaluation of a BP. So far, we represented the BP by a simple tree structure but there are more efficient ways of evaluation, e.g. a binary decision diagram (BDD). This is a special case of a BP which applies to our construction. As for BPs there are a lot of different notations for BDDs. In

[2] Otherwise we could take the smallest prime larger than m.



(a) Pruned tree for $X = \{1, 2, 3, 7\}$ (b) BDD for $X = \{1, 2, 3, 7\}$

Figure 11: Visualization of pruned tree and BDD for $X = \{1, 2, 3, 7\}$

contrast to our previous construction, a BDD can have more than one parent node. We need some slight modifications to our evaluation procedure since we now have nodes with in-degree greater than 1. See Figure 11 for an example with $X = \{1, 2, 3, 7\}$. It is easy to see that the BDD consists of less edges and less nodes which leads to a smaller evaluation effort. Evaluating the pruned tree, we need at least 10 multiplications and 5 additions (aggregating bottom up or using path prefixes). For the BDD, we only need 8 multiplications and 4 additions.

In the following, we present changes in the protocol's subroutines to allow the creation and evaluation of BDDs. The algorithms for node evaluation (Algorithm 7) and leaf evaluation (Algorithm 9) can be used without any changes for BDDs.

Creating the BDD. Our main goal is to represent the data structure as compact as possible. Therefore, we are considering reduced BDDs. A BDD is reduced if none of the reduction rules can be applied [32]. The reduction rules are [32]:

1. Eliminate nodes with isomorphic children
2. Merge isomorphic subgraphs

In our case, the first rule corresponds to pruning the tree. Isomorphic subgraphs means that there is a bijection between the nodes of the subgraphs such that adjacent nodes in a subgraph are also adjacent to the corresponding nodes in the other subgraph. An example for the second rule has been shown in Figure 11. The leftmost subgraph and the rightmost subgraph are isomorphic since in our data structure nodes on the same level are labeled with the same input.

For a general approach, we start with the tree, which is already a special case of a BDD, from our basic protocol and use established algorithms to efficiently reduce it [32]. As an addition of the data structure from Section 6, the `parent` attribute is now a list of pointer values since a node can have several parents. Note that this computation can be done in the offline phase and on plaintexts.

```
1:  function EvalPaths(root)
2:      let Q be a queue
3:      let leaves be a queue
4:      Q.enqueue(root)
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          [[tmp]] ← [[0]]
8:          for all p ∈ v.parent do
9:              [[tmp]] ← [[tmp]] ⊞ [[v.cost]] ⊡ [[p.cost]]
10:         [[v.cost]] ← [[tmp]]
11:         if v.isLeaf() then
12:             leaves.enqueue(v)
13:         else
14:             if v.left ≠ null then
15:                 Q.enqueue(v.left)
16:             if v.right ≠ null then
17:                 Q.enqueue(v.right)
18:     return leaves
```

Algorithm 12: Evaluating Paths (BDD Case)

Aggregating Decision Bits. The aggregation along each path works similar but we must take care of nodes with more than one predecessor. Instead of evaluating the decision bits of edges to a node's children, we evaluate the decision bits of edges to a node's parents and aggregate them. The result is depicted in Algorithm 12.

7.5 Third-Party Computation

Our protocol is built for a client-server-scenario in which the server holds the larger set and has a lot of computation power. In case the latter is not given, the server might outsource the computational effort to a third party. With some slight modifications we are able to extend our protocol such that the server can outsource the main computation to a third party in a secure way. As the client, the server encrypts its program and the third party can evaluate our protocol on an encrypted input y and an encrypted tree built on P. To this end, the server builds the tree and encrypts edge labels and leaf labels. Since we want to guarantee the server's privacy, we must create the tree in a generic way such that the resulting structure does not leak anything about the server's set. The server sends these encrypted values and the client sends the encrypted input to the third party which applies the same evaluation as before except an encrypted node evaluation algorithm. In contrast to the plaintext case, the evaluation has to be done homomorphically. The modification can be found in Algorithm 13

```
1:  function EvalNodes(root, [[ȳ]])
2:      let Q be a new queue
3:      Q.enqueue(root)
4:      parse [[ȳ]] to [[y[1]]], ..., [[y[μ]]]
5:      while Q.empty() = false do
6:          v ← Q.dequeue()
7:          if v.left ≠ null then
8:              [[v.left.cost]] ← [[y[v.level]]] ⊞̄ [[v.lEdge]]      ▷ Encrypted label
9:              Q.enqueue(v.left)
10:         if v.right ≠ null then
11:             [[v.right.cost]] ← [[y[v.level]]] ⊞̄ [[v.rEdge]]     ▷ Encrypted label
12:             Q.enqueue(v.right)
```

Algorithm 13: Evaluating Nodes (Encrypted Case)

where lines 8 and 11 changed compared to the basic protocol.

8 Implementation and Evaluation

In this section, we present implementation details and describe our evaluation where we compare the running time of the different implementations of our protocol with each other and with the baseline implementation of Ishai-Paskin's protocol.

8.1 Implementation

We implemented Ishai-Paskin's protocol as a baseline implementation. It is based on libscapi [18], which is a library containing the Damgård-Jurik scheme [12].

For our protocol, we implemented two variants. The first variant is based on Section 6.3 implementing the leveled BGV scheme [6] from HElib library [1]. The second variant is based on Section 6.4 and uses TFHE as a gate-by-gate bootstrapping scheme [9, 10].

8.2 Experimental Setup

We compare protocols with only one interaction between the two parties. The client sends the encrypted input to the server and receives the encrypted output. The computation is done on the server's side. This is why we only measure the overall computation time on the server. We further measure the communication in bytes sent from the client to the serer and from the server to the client.

The protocols are evaluated for depths 3 through 12 on random inputs. For our variants, we computed the average over 30 runs. Due to the running time of Ishai-Paskin's protocol, we averaged over only 20 runs for depth 6, 7 and 8

and over only 10 runs for depth 9. To evaluate the effect of our optimizations from Section 7, we also compare the running times of different optimization procedures.

All experiments were performed on an AWS instance equipped with 24 virtual cores of an Intel Xeon scalable processor with up to 4GHz and 192 GB of RAM running Ubuntu 20.04.

8.3 Evaluation Results

The relevant data for our experiments is presented in Table 2.

	Ishai-Paskin	Ours (TFHE)			Ours (BGV)		
		Naive	Pruning	BDD	Naive	Pruning	BDD
$d = 3$	7.44635	0.6013	0.5697	0.6384	0.1394	0.1375	0.2649
$d = 4$	18.9638	0.6576	0.6204	0.7467	0.2952	0.2663	0.4322
$d = 5$	43.2954	0.7215	0.7238	0.9057	0.5121	0.4698	0.6895
$d = 6$	93.6150	0.8764	0.9030	1.0971	1.0856	0.9254	1.3019
$d = 7$	195.7310	1.2544	1.2551	1.4327	2.0830	1.6626	1.7867
$d = 8$	402.0605	1.9530	1.9509	1.7813	4.5693	3.5818	2.4503
$d = 9$	815.8435	3.3567	3.3624	2.3534	7.8346	5.7843	2.9849
$d = 10$	-	6.1514	4.8179	3.3270	17.2331	12.3072	11.9572
$d = 11$	-	11.6349	8.8790	4.8329	36.1534	25.0358	18.3435
$d = 12$	-	22.8414	16.9379	6.8911	82.0635	56.3668	28.7130

Table 2: Running Time (s) for All Schemes

Our experimental evaluation concludes that for all tested tree depths, the evaluation of both BGV and TFHE comes with less overhead, as well as better scalability when compared to the Ishai-Paskin protocol. This is visualized in Figure 14. Here we see that BGV is between 16 times faster than the Ishai-Paskin protocol on the smallest instances and 31 times faster on the largest instances. TFHE, is marginally slower than BGV on the smallest instances, but eventually becomes faster due to better scalability. Notably, TFHE and HElib seem to scale similarly on small examples, but as instances get bigger we are forced to select less optimal parameters in HElib to accomodate for a larger multiplicative depth. Since picking less optimal parameters makes every operation a little bit slower, this accumilates into a massive hit in performace, as seen when going from depth 9 to depth 10. On the largest instance (depth 12), TFHE outperforms BGV by about 4 times.

We also tested the performance gained using pruning, as well as the combination of pruning and the evaluation of the BDD representation of the tree. These are visualized in Figures 15 and 16. We see, that both our TFHE implementation as well as our BGV implementation benefit from pruning. Moreover, since pruning has a proportionally larger effect on larger trees, we see an increase in performance gained by BGV and TFHE as the depth of the evaluated tree increases. Note however, that evaluating the BDD representation of a tree requires a deeper multiplicative depth. This forces us to select less optimal parameters to allow for a deeper multiplicative depth, much earlier that when

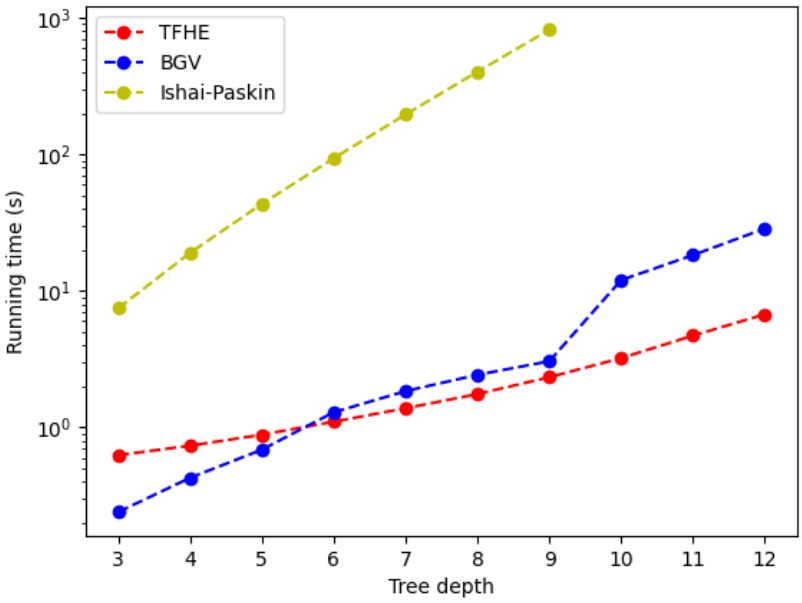


Figure 14: Comparison of Running Time

evaluating the trees directly. This means that when using BGV, our performance when using the BDD representation can become comparable to what we obtain using only pruning.

More interestingly, the combination of using pruning and the BDD representation, leads initially to slower times. This is due in part, to the fact that evaluating a BDD is more complex than evaluating a pruned tree. The performance gained by by decreasing the size of the instance is therefore negated by the increase in evaluation overhead on smaller trees. As the tree size increases, we see that this is no longer the case, and that using the BDD representation can provide a substantial performance gain over just pruning.

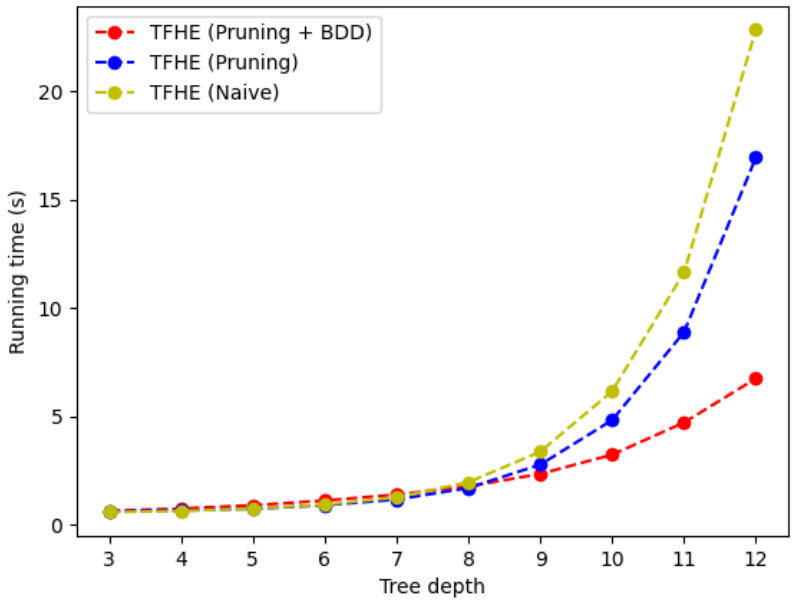


Figure 15: Influence of Optimizations for TFHE

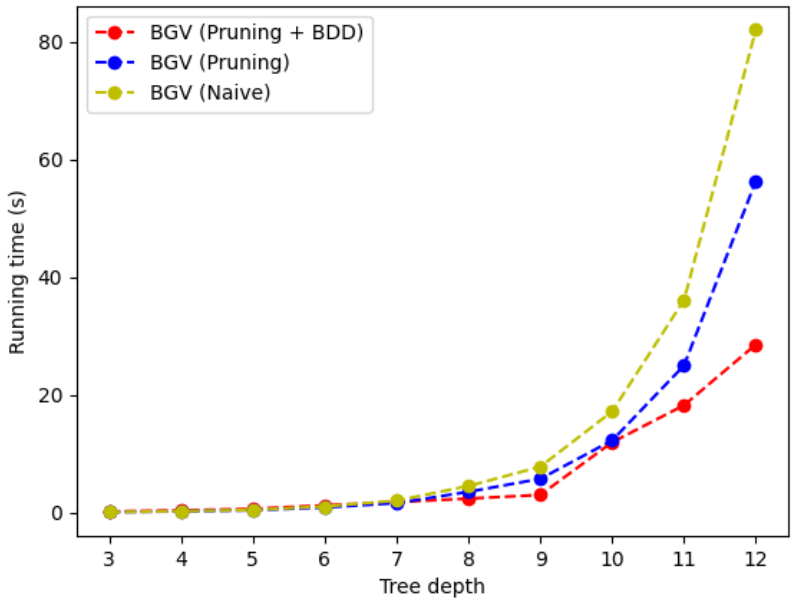


Figure 16: Influence of Optimizations for BGV

In terms of space efficiency, Figure 17 presents the number of bytes sent from the server to the client. We see that TFHE uses constant ciphertext-size, and therefore becomes marginally more space efficient than BGV and Ishai-Paskin. The Ishai-Paskin protocol uses ciphertexts of a similar order-of-magnitude to those used in TFHE but becomes less space efficient the deeper the tree becomes, since ciphertext size is dependence on the tree depth in the Ishai-Paskin protocol. BGV uses cipherexts that are much larger than TFHE and Ishai-Paskin. We also see a sudden increase in ciphertext size, that corresponds to the selection of new parameters in order to accomodate for larger multiplicative depths. We can conclude, that the space efficiency of both Ishai-Paskin and BGV scale poorly in comparison to the TFHE implementation.

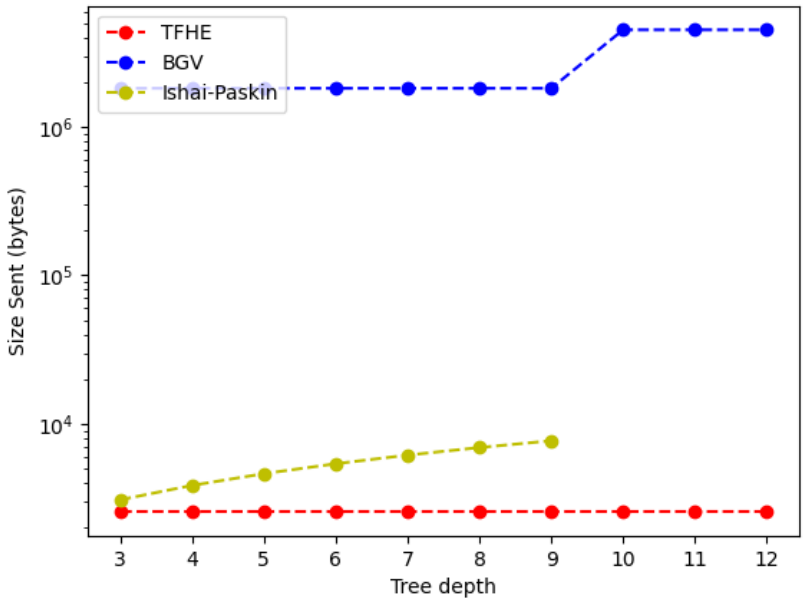


Figure 17: The number of bytes sent from the server to the client, i.e. the size of a single ciphertext.

This trend can also be seen in the communication cost from the client to the server. Figure 18 presents the number of bytes received by the server from the client. We see that again, TFHE is much more efficient that both BGV and Ishai-Paskin. We also see, that both BGV as well as TFHE scale better than Ishai-Paskin.

As we previously noticed, when using BGV our performance when using the BDD representation can become comparable to what we obtain using only pruning. In such situations, it might be even be wiser to only use pruning, since sub-optimal parameters also greatly increase ciphertext size. This can be seen in Figure 19 and Figure 20, where the selection of new parameters greatly increases the communication cost between both parties.

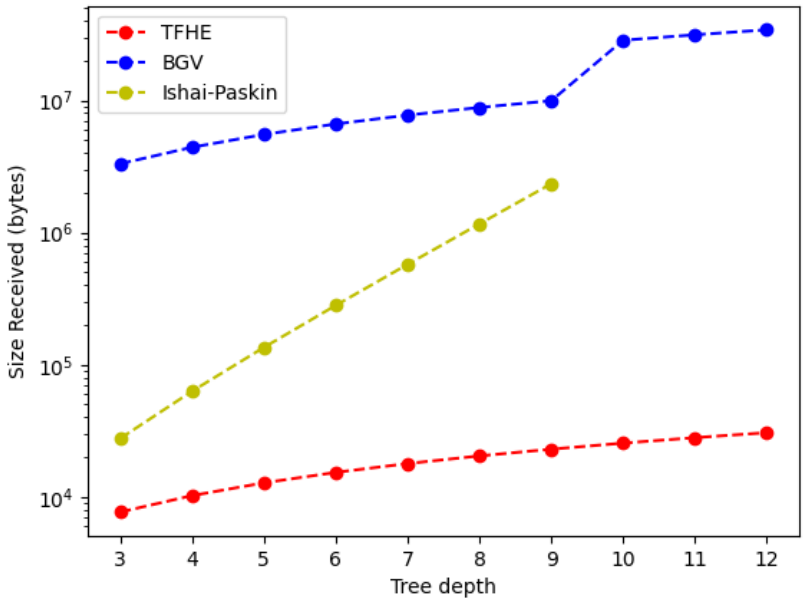


Figure 18: The numer of bytes sent from the client to the server.

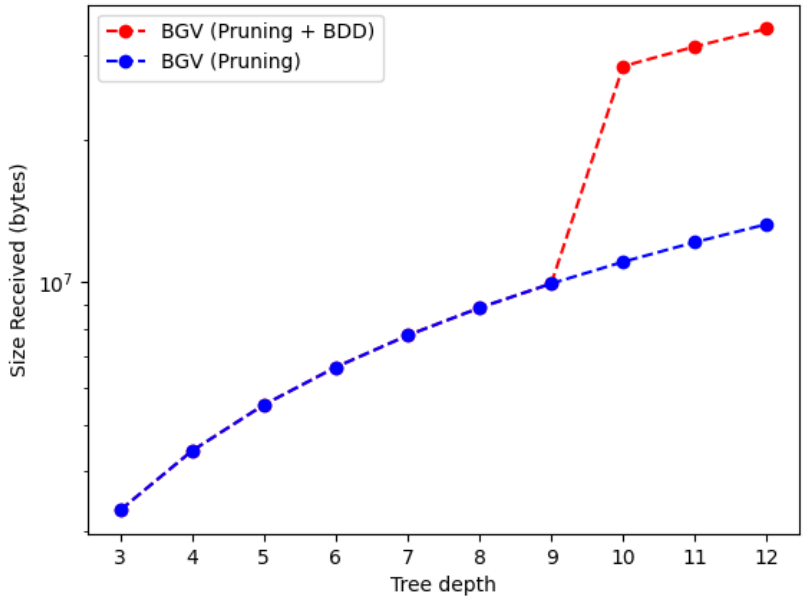


Figure 19: The number of bytes received by the server from the client in BGV variants.

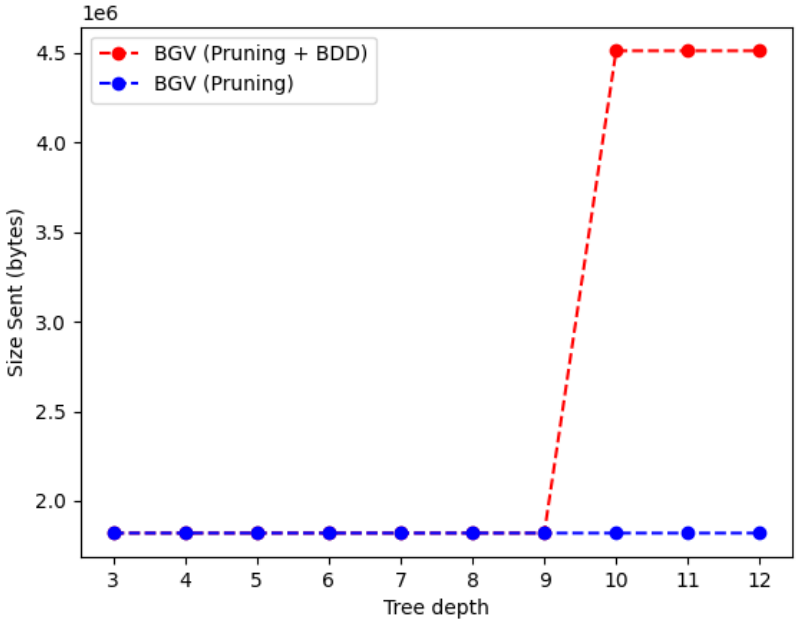


Figure 20: The number of bytes sent from the server to the client in BGV variants, i.e. the size of one ciphertext.

References

[1] Helib, October 2021. https://github.com/homenc/HElib.

[2] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider. Secure evaluation of private linear branching programs with medical applications. In *ESORICS*, pages 424–439, Berlin, Heidelberg, 2009. Springer-Verlag.

[3] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *CCS*, CCS '12, pages 784–796, 2012.

[4] 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.

[5] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

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

[7] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. Privacy-preserving remote diagnostics. In *CCS*, pages 498–507, New York, NY, USA, 2007. ACM.

[8] D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19, New York, NY, USA, 1988. ACM.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[11] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, pages 280–299, 2001.

[12] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC '01*, PKC '01, pages 119–136, London, UK, UK, 2001. Springer-Verlag.

[13] I. Damgård and M. Jurik. A length-flexible threshold cryptosystem with applications. In *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*, pages 350–364, 2003.

[14] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS '13*, pages 1–18, 2013.

[15] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO '12*, pages 643–662, 2012.

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

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

[18] B. I. C. R. Group et al. libscapi: The secure computation api, 2016.

[19] 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.

[20] Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *Theory of Cryptography Conference*, pages 575–594. Springer, 2007.

[21] M. Joye and F. Salehi. Private yet efficient decision tree evaluation. In *DBSec*, volume 10980 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2018.

[22] M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS '16*, pages 830–842, 2016.

[23] Á. Kiss, M. Naderpour, J. Liu, N. Asokan, and T. Schneider. Sok: Modular and efficient private decision tree evaluation. *PoPETs*, 2019(2):187–208, 2019.

[24] V. Kolesnikov and T. Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC*, pages 83–97, 2008.

[25] P. Mohassel, S. S. Sadeghian, and N. P. Smart. Actively secure private function evaluation. In *ASIACRYPT*, pages 486–505, 2014.

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

[27] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[28] 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.

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

[30] A. Tueno and J. Janneck. A method for securely comparing integers using binary trees. *IACR Cryptol. ePrint Arch.*, page 1646, 2021.

[31] A. Tueno, F. Kerschbaum, and S. Katzenbeisser. Private evaluation of decision trees using sublinear cost. *PoPETs*, 2019(1):266–286, 2019.

[32] I. Wegener. *Branching programs and binary decision diagrams: theory and applications*. SIAM, 2000.

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

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

A Analysis Multiplication DAG

In this section, we analyse the number of multiplications required to evaluate a complete tree using batch multiplication. First we show that the required number of multiplications is bound by $\log(d) \cdot 2^{d \log(d)}$, where d is the depth of the tree, and then we give an explicit formula for the required number of multiplications.

Theorem A.1. *The number of multiplications required to evaluate a complete tree using batch multiplication is bound by* $\log(d) \cdot 2^{d \log(d)}$*, where* d *is the depth of the tree.*

Proof. We prove this fact by induction. To do this, we need a recursive formula for the number of multiplications required to evaluate the tree. For large trees, the evaluation of the tree using batch multiplication goes as follows. The tree is split at level $\lfloor \frac{d}{2} \rfloor$. The tree above the split is of depth $\lfloor \frac{d}{2} \rfloor$, and the trees below the split are of depth $\lceil \frac{d}{2} \rceil$, with the leaves of the upper tree forming the roots of the lower trees (since we multiply edge labels and not node labels). Each of the trees is evaluated separately, and then the value at each leaf in the upper tree is multiplied with each of the values at the leaves below it. There are $2^{\lfloor \frac{d}{2} \rfloor}$ leaves in the tree above the split. For each of those leaves, there is exactly one tree below it, which is of depth $\lceil \frac{d}{2} \rceil$. Since each leaf of each lower tree is multiplied with one leaf from the upper tree, we get an extra 2^d multiplications. Let $p(d)$ be a function that gives us the number of multiplications required to evaluate a tree of depth d using batch multiplication. We now know that

$$p(d) = p\left(\left\lfloor \frac{d}{2} \right\rfloor\right) + 2^{\lfloor \frac{d}{2} \rfloor} \cdot p\left(\left\lceil \frac{d}{2} \right\rceil\right) + 2^d,$$

for $d \geq 2$. By inspection, we obtain the initial value condition $p(1) = 0$.

We now only need to bound this function from above, by $\log(d) \cdot 2^{d \log(d)}$. We do this inductively for all $d \geq 8$ (we require large d in our proof), and by inspection for all smaller d.

Our base case is simple. Evaluating the function p between 1 and 8 yields the values $0, 4, 16, 36, 100, 208, 432$, and 868. The reader can verify that for these values, p is in fact bound by $\log(d) \cdot 2^{d \log(d)}$.

Our induction step then goes as follows. We make the distinction between odd and even d. For even d, we know

$$\begin{aligned} p(d) &= p\left(\left\lfloor \frac{d}{2} \right\rfloor\right) + 2^{\lfloor \frac{d}{2} \rfloor} \cdot p\left(\left\lceil \frac{d}{2} \right\rceil\right) + 2^d \\ &\underset{B.C.}{\leq} \log\left(\frac{d}{2}\right) 2^{\frac{d}{2} \log(\frac{d}{2})} + 2^{\frac{d}{2}} \cdot \log\left(\frac{d}{2}\right) \cdot 2^{\frac{d}{2} \log(\frac{d}{2})} + 2^d \\ &= \log(d) \left(2^{\frac{d}{2} \log(\frac{d}{2})} + 2^{\frac{d}{2}} \cdot 2^{\frac{d}{2} \log(\frac{d}{2})}\right) - 2^{\frac{d}{2} \log(\frac{d}{2})} - 2^{\frac{d}{2}} \cdot 2^{\frac{d}{2} \log(\frac{d}{2})} + 2^d \end{aligned}$$

For any $d \geq 8$, $log_2(d/2) \geq 2$ and thus $2^{\frac{d}{2}\log(d/2)} \geq 2^d$. Hence we obtain

$$\begin{aligned}
&\log(d)\left(2^{\frac{d}{2}\log(\frac{d}{2})} + 2^{\frac{d}{2}} \cdot 2^{\frac{d}{2}\log(\frac{d}{2})}\right) - 2^{\frac{d}{2}\log(\frac{d}{2})} - 2^{\frac{d}{2}} \cdot 2^{\frac{d}{2}\log(\frac{d}{2})} + 2^d \\
&\leq \log(d)\left(2^{\frac{d}{2}\log(\frac{d}{2})} + 2^{\frac{d}{2}} \cdot 2^{\frac{d}{2}\log(\frac{d}{2})}\right) \\
&= \log(d)\left(2^{\frac{d}{2}\log(\frac{d}{2})} + 2^{\frac{d}{2}\log(d)}\right) \\
&\leq \log(d)\left(2^{\frac{d}{2}\log(d)} + 2^{\frac{d}{2}\log(d)}\right).
\end{aligned}$$

For any number r that is at least 2, we know that $r + r \leq r \cdot r$. Hence,

$$\begin{aligned}
&\log(d)\left(2^{\frac{d}{2}\log(d)} + 2^{\frac{d}{2}\log(d)}\right) \\
&\leq \log(d)\left(2^{\frac{d}{2}\log(d)} \cdot 2^{\frac{d}{2}\log(d)}\right) \\
&= \log(d)\left(2^{d\log(d)}\right).
\end{aligned}$$

For odd $d > 8$, the argumentation is analogous. We obtain

$$\begin{aligned}
&p\left(\left\lfloor \frac{d}{2} \right\rfloor\right) + 2^{\lfloor \frac{d}{2} \rfloor} \cdot p\left(\left\lceil \frac{d}{2} \right\rceil\right) + 2^d \\
&\underset{B.C.}{\leq} \log(\frac{d-1}{2}) 2^{\frac{d-1}{2}\log(\frac{d-1}{2})} + 2^{\frac{d-1}{2}} \cdot \log(\frac{d+1}{2}) \cdot 2^{\frac{d+1}{2}\log(\frac{d+1}{2})} + 2^d \\
&\leq \log(d+1)\left(2^{\frac{d-1}{2}\log(\frac{d-1}{2})} + 2^{\frac{d-1}{2}} \cdot 2^{\frac{d+1}{2}\log(\frac{d+1}{2})}\right) \\
&\quad - 2^{\frac{d-1}{2}\log(\frac{d-1}{2})} - 2^{\frac{d-1}{2}} \cdot 2^{\frac{d+1}{2}\log(\frac{d+1}{2})} + 2^d, \\
&\leq \log(d)\left(2^{\frac{d-1}{2}\log(\frac{d-1}{2})} + 2^{\frac{d-1}{2}} \cdot 2^{\frac{d+1}{2}\log(\frac{d+1}{2})}\right) \\
&\leq \log(d)\left(2^{\frac{d-1}{2}\log(\frac{d-1}{2}) + \frac{d+1}{2}\log(d+1)}\right) \\
&\leq \log(d)\left(2^{(d-1)\log(d-1) - \frac{d-1}{2} + \log(d+1)}\right) \\
&\leq \log(d)\left(2^{(d-1)\log(d-1)}\right) \\
&\leq \log(d)\left(2^{d\log(d)}\right)
\end{aligned}$$

By induction, it follows that for all $d \in \mathbb{N}$, $p(d) \leq \log(d) \cdot 2^{d\log(d)}$. □