# Efficient Circuits for Permuting and Mapping Packed Values Across Leveled Homomorphic Ciphertexts

Jelle Vos(✉), Daniël Vos, and Zekeriya Erkin

Cyber Security Group, Delft University of Technology, Delft, Netherlands
{J.V.Vos,D.A.Vos,Z.Erkin}@tudelft.nl

**Abstract.** Cloud services are an essential part of our digital infrastructure as organizations outsource large amounts of data storage and computations. While organizations typically keep sensitive data in encrypted form at rest, they decrypt it when performing computations, leaving the cloud provider free to observe the data. Unfortunately, access to raw data creates privacy risks. To alleviate these risks, researchers have developed secure outsourced data processing techniques. Such techniques enable cloud services that keep sensitive data encrypted, even during computations. For this purpose, fully homomorphic encryption is particularly promising, but operations on ciphertexts are computationally demanding. Therefore, modern fully homomorphic cryptosystems use packing techniques to store and process multiple values within a single ciphertext. However, a problem arises when packed data in one ciphertext does not align with another. For this reason, we propose a method to construct circuits that perform arbitrary permutations and mappings of such packed values. Unlike existing work, our method supports moving values across multiple ciphertexts, considering that the values in real-world scenarios cannot all be packed within a single ciphertext. We compare our open-source implementation against the state-of-the-art method implemented in HElib, which we adjusted to work with multiple ciphertexts. When data is spread among five or more ciphertexts, our method outperforms the existing method by more than an order of magnitude. Even when we only consider a permutation within a single ciphertext, our method still outperforms the state-of-the-art works implemented by HElib for circuits of similar depth.

**Keywords:** Secure outsourced data processing · Data packing · Fully homomorphic encryption · Applied cryptography

## 1 Introduction

Nowadays, organizations use cloud providers to outsource their data processing, easing deployment and allowing them to scale the architecture up and down when required [2]. While these organizations typically keep sensitive data in encrypted form at rest, they decrypt it when performing computations. Consequently, these organizations must fully trust the cloud providers, who can observe all sensitive data. To protect sensitive data while processing, researchers propose secure outsourced data processing solutions, which allow cloud providers to offer their services on data that they cannot see. In the settings of those proposals, organizations assume that the cloud provider performs the operations they ask them to, thus reducing privacy risks.

One possible approach that enables cloud providers to process sensitive data relies on fully homomorphic encryption (FHE) schemes. FHE allows anyone with the correct public key to perform computations on encrypted data without seeing it. In current schemes, one typically encrypts integers or real numbers, which can be manipulated through addition and multiplication. A subset of FHE schemes (such as BFV [8], BGV [4], and CKKS [6]) allows one to encrypt entire fixed-length vectors of integers or real numbers in one ciphertext through ciphertext packing. A limited number of additions and multiplications can be performed as element-wise operations between encrypted vectors, following the concept of single-instruction multiple-data (SIMD). As a result, operating on packed ciphertexts leads to significant speed-ups when there is a large set of data to be processed.

A problem arises when the data stored in two encrypted vectors do not align. For example, consider two ciphertexts that each hold a database relating to the incomes of a set of employees. One ciphertext holds their salary sorted by their first name, while another holds their yearly bonus sorted by their last name. An outsourced HR system might compute each employee's total income by adding the two together. However, directly adding the two ciphertexts together leads to a meaningless result. Instead, the HR system must align the data stored within one ciphertext with the other by permuting it.

FHE schemes that support ciphertext packing implement ciphertext rotations to allow one to align encrypted vectors. This primitive shifts the encrypted vector $x$ places towards the end while cycling the last $x$ encrypted numbers to the beginning. However, rotations alone are not enough to perform arbitrary permutations on encrypted vectors. Instead, it requires an intricate circuit that combines additions, multiplications, and rotations. We call these permutation circuits. Halevi & Shoup [11] conjecture that finding the optimal (i.e., fastest given a maximum multiplicative depth) is a hard problem.

Previous work has focused on generating permutation circuits that permute a single ciphertext. However, for applications in the real world, not all data may be stored in the same ciphertext due to size constraints or because the data has different origins. Therefore, with the current solutions, the problem of permuting across multiple ciphertexts requires splitting the entire permutation into multiple within-ciphertext permutations. We highlight this problem in Fig. 1. Solving this problem may also lead to improvements in the circuits for other applications, such as circuits that perform AES encryptions homomorphically.

In this work, we propose a new primitive that performs arbitrary mappings on values in ciphertexts and does so significantly cheaper than previous work

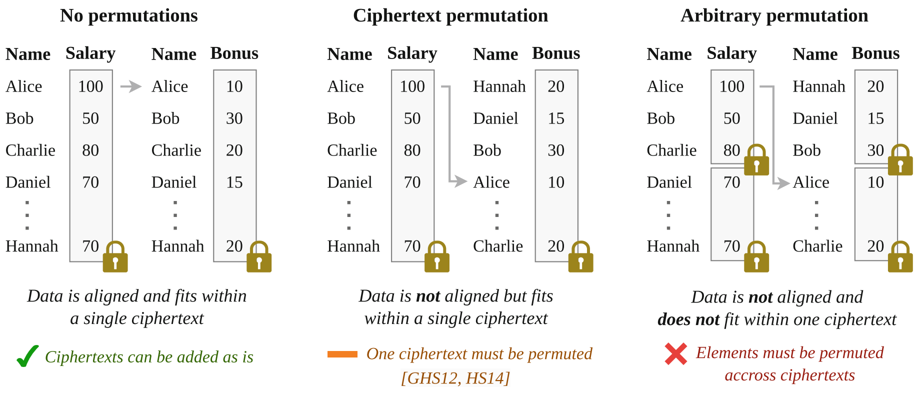
No permutations
Ciphertext permutation
Arbitrary permutation
Name Salary Name Bonus
Alice 100 Alice 10
Bob 50 Bob 30
Charlie 80 Charlie 20
Daniel 70 Daniel 15
Hannah 70 Hannah 20
Alice 100 Hannah 20
Bob 50 Daniel 15
Charlie 80 Bob 30
Daniel 70 Alice 10
Hannah 70 Charlie 20
Data is aligned and fits within a single ciphertext
✓ Ciphertexts can be added as is
Data is not aligned but fits within a single ciphertext
One ciphertext must be permuted [GHS12, HS14]
Data is not aligned and does not fit within one ciphertext
✗ Elements must be permuted accross ciphertexts

**Fig. 1.** If data is not aligned between two ciphertexts, one of the ciphertexts must be permuted. The existing methods work when data fits within one ciphertext, but when data spans multiple ciphertexts they must be adapted and lose performance rapidly.

regarding the computational effort required. These mappings are arbitrary in the sense that they may span multiple ciphertexts. Unlike previous methods which generate circuits for a chosen maximum multiplicative depth, our method focuses on a specific class of permutation circuits with a constant multiplicative depth. Still, we argue that our circuits' depth is reasonable for the complexity of the operation required. Our new primitive takes the burden off the implementor to create manual mapping circuits when data spans multiple ciphertexts. Its high efficiency brings secure outsourced computation one step closer to practice.

We summarize our contributions as follows:

- We propose a new method for efficiently performing arbitrary mappings on encrypted values in packed, leveled-homomorphic ciphertexts.
- We compare an open-source implementation of our method to HElib for performing permutations on single ciphertexts and show that it consistently outperforms HElib for circuits of similar multiplicative depth.
- We compare our implementation to an adjusted version of HElib to perform arbitrary permutations. We show that it outperforms HElib by more than an order of magnitude when the data is spread among five or more ciphertexts.

The remainder of this paper is structured as follows: In Sect. 2, we shortly explain operations in leveled homomorphic encryption, graph coloring, and the notation we use. In Sect. 3, we discuss related work. Next, in Sect. 4, we put forward our method for constructing mapping circuits, and in Sect. 5 we analyze its complexity. Finally, in Sect. 6 we compare our method against that implemented in HElib, after which we conclude in Sect. 7.

## 2 Preliminaries and Notation

In this section, we give a high-level explanation of the underlying techniques used in this paper. Table 1 contains a summary of the notation that we use.

**Table 1.** Summary of the symbols used in this work.

| Symbol | Definition |
|---|---|
| $\ell$ | Number of slots in the ciphertext |
| $n$ | Total number of elements to permute |
| $\pi(x)$ | Target for index x after permuting |
| $\mu(x)$ | Targets for index x after mapping |
| $P$ | Set of indices to permute (preimage) |
| $\chi$ | Chromatic number (minimum number of colors) |
| $\phi(\_)$ | Euler's totient function |
| $m$ | Order of cyclotomic polynomial |
| $p$ | Prime modulus defining the message space |
| $Q$ | Ciphertext modulus defining the ciphertext space |

### 2.1 Permutations and Mappings

We consider permutations and mappings of elements across vectors of length $n$. Here, we denote $P$ as the set of indices to map, which is short for the preimage. We say that element $x \in P$ is permuted to position $\pi(x)$ when considering permutations, or mapped to position $\mu(x)$ in the case of a mapping. Note that permutations are a restriction of mappings.

### 2.2 Graph Coloring

Graph coloring is one of Karp's original 21 NP-complete problems [12]. In this problem, we are given a loopless graph $G = (V, E)$ where we must assign a color to each vertex such that no two adjacent vertices share the same color. The minimum number of colors needed to be able to properly color $G$ is the chromatic number $\chi$. In this work, we translate the process of setting up an efficient homomorphic circuit for ciphertext mappings to the problem of graph coloring. While the problem is NP-complete in general, we can practically solve our instances here using algorithms such as DSATUR [5].

### 2.3 Leveled Homomorphic Encryption Schemes

This work specifically considers leveled homomorphic encryption schemes that support packing multiple elements into one ciphertext. Here, leveled refers to the fact that we can only perform operations up to a certain level before decryption is likely to fail. The level is typically indicated as the multiplicative depth of the arithmetic circuit. The reason for this is that the ciphertexts incorporate a small noise term that grows with each homomorphic operation. This is why we speak of the *remaining noise budget* of a ciphertext, which we express as the number of bits of the ciphertext that the growing noise can still consume before the ciphertext is no longer decryptable. When there is a need to perform circuits of arbitrary depth, one can use bootstrapping techniques [9]. In that case, we speak of *fully* homomorphic encryption. In our implementation, we only consider the BGV [4] cryptosystem implemented in HElib, without bootstrapping operations.

One can add, multiply and rotate the values encrypted in a ciphertext. Element-wise additions are cheap operations between two ciphertexts with only small noise growth. In this work, we do not multiply ciphertexts together but only multiplications with constants, which is more efficient and incurs less noise growth. We use these plaintext multiplications to isolate values from the ciphertext by creating a mask that is zero everywhere except for the places with the elements we need to isolate where it is 1. Rotations can be performed using automorphisms on the underlying ring. In this work, we only consider the case where those automorphisms cause one-dimensional rotations.

## 3 Related Work

To the best of our knowledge, the first work that studied permutations in leveled homomorphic ciphertexts was the work by Gentry et al. [10]. In separate work, the same authors use it to implement an AES circuit homomorphically, which requires shuffling the elements within a ciphertext. Before that, Damgård et al. [7] already used the underlying techniques within the context of secure multiparty computation to permute packed secret shares rather than ciphertexts. The underlying technique called Beneš networks [3] originates in the study of efficient routing networks, which send packets from a range of senders to a range of receivers under constraints, effectively executing permutations.

In 2014, Halevi & Shoup [11] reduced the problem of constructing efficient permutation circuits for leveled homomorphic ciphertexts as a new problem named the cheapest-shift-network problem. Here, a shift-network is a series of shifts (permutations), which can be executed using additions, plaintext multiplications, and rotations. Each next shift considers only the shift before it. Halevi & Shoup put forward a method to efficiently optimize the computational cost of such a circuit given a maximum multiplicative depth, and implement it in the HElib library.[1] At the time of writing, we are not aware of other libraries that implement ciphertext permutations.

In this work, we consider a type of circuit that not only considers the layer before it but also any other layer before that. We also extend it beyond the range of a single ciphertext. In this sense, it is less restricted than the method proposed by Halevi & Shoup. However, it is an open question of how to optimize such a circuit efficiently, so we introduce other restrictions to turn the problem into one of graph coloring. For example, the multiplicative depth of our circuits scales logarithmically with the number of slots in a ciphertext. In the remainder of this section, we go into detail about the solutions of Gentry et al. [10] and Halevi & Shoup [11] (summarized in Table 2) and explain how one can trivially but inefficiently extend them to perform arbitrary permutations and mappings.

[1] The HElib repository can be found at https://github.com/homenc/HElib.

**Table 2.** Comparison of permutation circuits generated by related work

| Operation | Compute | Noise | Ciphertext permutation | | | Arbitrary | |
|---|---|---|---|---|---|---|---|
| | | | Naive | HElib | Ours | HElib* | Ours |
| Rotation | Expensive | Cheap | $\ell$ | $4\log(\ell)-2$ | $\log^2(\ell)$ | $O(n^2)$ | $O(n)$ |
| Plaintext mult | Cheap | Moderate | $\ell$ | $4\log(\ell)-2$ | $O(\log^3(\ell))$ | $O(n^2)$ | $O(n^2)$ |
| Addition | Cheap | Cheap | $\ell$ | $2\log(\ell)-1$ | $O(\log^3(\ell))$ | $O(n^2)$ | $O(n^2)$ |
| Rotation keys | Severe | – | $\ell$ | $2\log(\ell)$ | $\log(\ell)$ | $2\log(\ell)$ | $\log(\ell)$ |

## 3.1 Naive Method for Permutations

A naive method for performing permutations within and across ciphertexts rotates each individual element to its target index and sums up the result. As mentioned before, elements can be isolated by multiplying them with a vector of zeroes and a 1 in the right index. This approach requires a plaintext multiplication, rotation, and addition for each of the $\ell$ slots in a ciphertext when performing a permutation within one ciphertext. Moreover, key generation will also be computationally expensive as one has to be able to perform each possible automorphism. Alternatively, one incurs an additional run time penalty for certain rotations by composing it from other rotations. Note that we can omit rotations of 0 and that there are scenarios where identical rotations can be rotated at the same time. Still, after these optimizations, the algorithm scales with $O(n)$ in the worst case.

## 3.2 'Collapsed' Beneš Networks for Permutations

Both the works by Gentry et al. [10] and Halevi & Shoup [11] rely on Beneš networks. Such a network has a butterfly structure, which contains $2\log(\ell)-1$ layers in the case of a ciphertext permutation. This structure makes it a shift-network that can be constructed efficiently in a recursive manner for all possible permutations. Elements are either rotated leftwards or rightwards in each layer by a given amount.

Gentry et al. use Beneš networks without any modifications, leading to a permutation circuit with a multiplicative depth that scales as $2\log(\ell)-1$. Each layer only does a power-of-two rotation, meaning that one must generate $2\log(\ell)$ rotation keys.

Halevi & Shoup modify Beneš networks into other valid shift networks by collapsing layers to reduce the multiplicative depth of the resulting circuit. As mentioned before, they implement this in the HElib library. In Table 2 we consider the case where there is no bound to the multiplicative depth of the circuit. Since each layer of the network requires 2 plaintext multiplications and rotations, the total number is $4\log(\ell)-2$ in the worst case.

## 3.3 Extending Permutation Circuits to Arbitrary Permutations

We remark that while previous works do not explicitly describe how to construct arbitrary permutations or mappings, they can be easily extended to do so. We shortly explain how the work Halevi & Shoup [11] can be extended as such by expressing the arbitrary permutation across multiple ciphertexts as a series of within-ciphertext permutations.

The key idea is that one can break a permutation across multiple ciphertexts into a set of permutations from each ciphertext to every other ciphertext. A similar trick can be used to perform mappings by first breaking it down into a set of arbitrary permutations. In the worst case, performing permutations in this way scales quadratically with the number of ciphertexts. When the elements are densely packed, we need a total of $\lceil \frac{n}{\ell} \rceil = O(n)$ ciphertexts. Here we consider $\ell$ to be constant. Consequently, the worst-case complexity for rotations, plaintext multiplications, and additions alike is $O(n^2)$.

## 4 Constructing Arbitrary Mapping Circuits

In this section, we propose our method for constructing circuits to perform arbitrary permutations and mappings. Since the construction only has to happen once for each permutation, it can be considered a one-time setup.

### 4.1 High-Level Insight

The most time-consuming operation in a permutation circuit is a ciphertext rotation. Therefore, it stands to reason to minimize the number of rotations. Conversely, we want to maximize the number of elements we rotate at once. At the same time, since we have to generate special rotation keys for every possible rotation magnitude, we want to keep the number of different rotations as low as possible. In our method, we restrict all rotations to be powers of two. As we discuss later, this simplification allows us to optimize our permutation circuit efficiently. It is also possible to restrict rotations to powers of three (or any other base), but this requires certain rotations to be decomposed into a larger number of consecutive power of three rotations.

Given a permutation, we construct a circuit that realizes it by decomposing the number of places that each element must move into its binary representation. If there is a 1 in place $x$ of the binary representation, we add the element to the set of elements that must be rotated by $2^x$. For simplicity, let us fix the order of rotations in the final circuit as $2^0 = 1, 2^1 = 2, 2^2 = 4, \ldots$. One can imagine this idea as vertically-stacked conveyor belts that sequentially turn at increasing rates, as seen in Fig. 2. In this figure, an element (pictured as a box) starts at index 1 and must end up at index 6. To do so, it must travel $5 = 101_2$ places rightwards, and therefore it enters the first and third conveyor belt, but not the second.

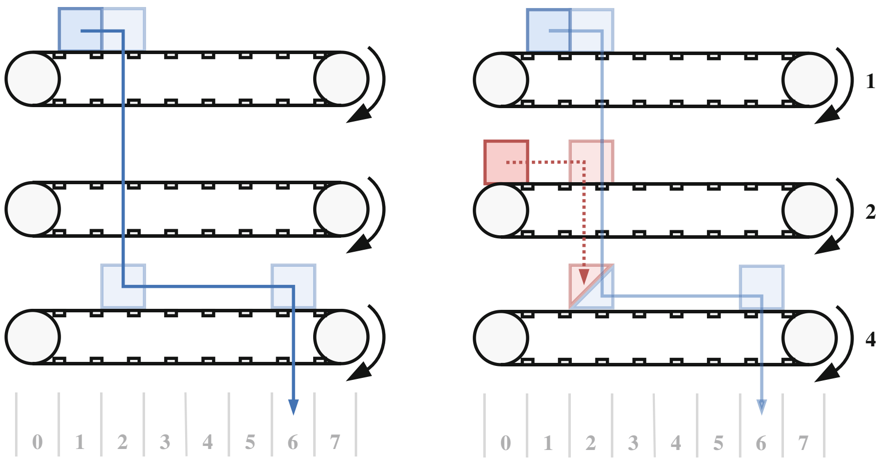

**Fig. 2.** Elements can be mapped to other locations by applying a sequence of rotations on them, as if on a conveyor belt. Multiple elements can exist on the same set of conveyor belts so long as they do not enter the same conveyor belt at the same location.

At first thought, the method described above seems to construct valid permutation circuits, but a problem arises when two elements must take the same place on the same conveyor belt. In an actual arithmetic circuit, this would add up the corresponding values of these elements, invalidating the permutation. In the right half of Fig. 2, we visualize this. There are two simple solutions to this problem. Firstly, one might change the order of the conveyor belts. For example, one might bring the third conveyor belt to the start. Another approach is to add a second independent set of conveyor belts. In our method, we use both approaches: We try several different random orderings of conveyor belts and use a graph coloring algorithm to distribute elements over multiple sets of conveyor belts in a way that elements do not collide. We use the minimum number of conveyor belts given a certain order of conveyor belts.

## 4.2 Assigning Elements to Sets of Conveyor Belts

To assign the elements to multiple sets of conveyor belts, we construct a graph where the vertices represent elements of the encrypted vector. The edges between them represent that the elements cannot coexist in the same conveyor belts. After performing a graph coloring, the color of a vertex represents the set of conveyor belts to which it is assigned. In the remainder of this subsection, we refer to a single conveyor belt as a rotation.

For a permutation $\pi$ with preimage $P$, we first create an undirected graph $G_\pi = (V, E)$, where $E = \emptyset$ and $V = P$. Then, for each element, we compute its position in the encrypted vector when it enters each rotation operation. If two elements $u, v \in P$ where $u \neq v$ enter the same rotation at the same position, we extend $E \leftarrow E \cup \{u, v\}$. This graph satisfies the property that any valid coloring represents a valid assignment. Figure 3 shows an example of such a graph and a possible coloring.

When we move beyond a permutation to a mapping $\mu$, we must consider that elements in the preimage may map to multiple positions in the final encrypted vector (replication), or multiple elements in the preimage may map to the same position (overlapping). Notice that overlapping elements do not necessarily have to be assigned to different sets of rotations and that the graph $G_\mu$ constructed as described above already adequately handles such situations. The reason is that overlapping elements in the final encrypted vector do not necessarily overlap in the encrypted vectors to which rotations are applied. This graph also adequately handles replications, as all outputs relating to the same input element are assigned to the same set of rotations. This means that even in the extreme case where one element of the input ciphertext is mapped to all positions of the output ciphertext, we only require one set of rotations.

After generating the graph, we use a dedicated graph coloring algorithm to color the vertices with the minimum number of colors required. In our implementation, we use the DSATUR algorithm [5], but any algorithm suffices.

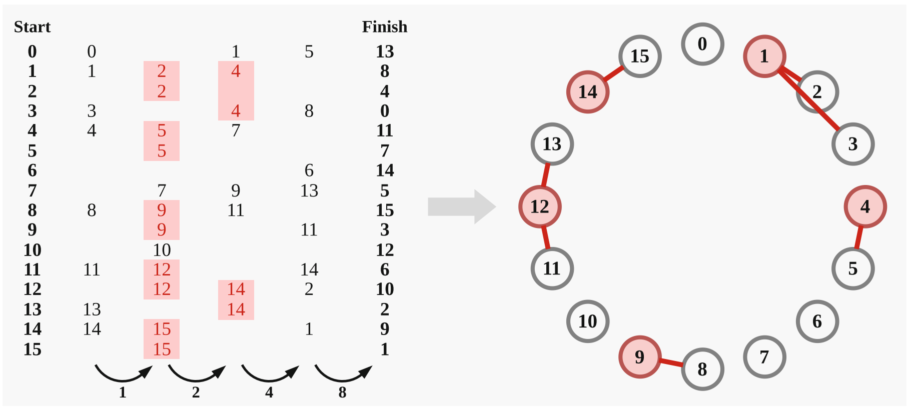

**Fig. 3.** Example of the graph generated for a within-ciphertext permutation of 16 slots. The graph contains edges between the elements that would collide with each other at any of the rotations. This graph can be colored with two colors, but larger ciphertexts, across-ciphertext permutations, and mappings typically require more colors. (Color figure online)

## 4.3 Determining the Order of Conveyor Belts

In the previous subsection, we did not explain how one should choose the order of the rotations. However, it follows that for the graph coloring to work, we require all sets of rotations to have the same order.

One approach is to fix the rotation order for every mapping. For example, $1, 2, 4, \ldots$. While this ordering performs well for random permutations and mappings, as we show in Sect. 6, one might try different orderings to avoid running into the worst-case behavior. In our implementation, we test multiple random orderings to find the one resulting in the graph that can be colored with the least colors. In our experiments, we compare the performance of trying only one random ordering against trying ten random orderings, which we refer to as a *long setup*.

It remains an open problem to integrate this step with the previous step to efficiently find an ordering that results in the minimum number of sets of rotations.

### 4.4 Generating Circuits for Conveyor Belts

Given an assignment that maps each element to a set of rotations, we construct a separate circuit for each set. Consequently, in a multi-threaded setup, one can execute these circuits in parallel. This subsection describes how to construct a circuit for one set of rotations, given a specific ordering of rotations and a set of elements that will not collide.

First, we create a set of masks for all the elements that must be included in a single rotation. In other words, we create one mask for each of the input ciphertexts and one mask for each of the ciphertexts resulting from all previous rotations. Such a mask contains ones in the positions of elements that must remain and zeroes in the positions of elements that must be dropped. We then perform a plaintext multiplication between each ciphertext and the corresponding mask and sum up the results. The result is a ciphertext containing all the relevant encrypted values, which we subsequently rotate.

Note that there are several places where we can prune this circuit to prevent performing meaningless computations. For example, if we do not need to consider any values from a ciphertext, the corresponding mask would be empty (i.e., filled with zeroes). Moreover, we do not need to perform any summations if there is only one relevant ciphertext. We implement both of these optimizations, but we stress that more pruning is still possible. For example, by keeping track of which positions in each ciphertext actually contain values rather than zeroes, one can discard multiplications that mask all values in a ciphertext.

In the worst case, an element must be shifted $11\ldots11_2 = \ell - 1$ places in the encrypted vector. The resulting circuit then has a multiplicative depth of $1+\log_2 \ell$ consecutive plaintext multiplications. When it comes to the asymptotic run time, each circuit only requires $\log_2 \ell$ rotations and, therefore, a total of $O(\log_2 \ell)$ plaintext multiplications and additions.

## 5 Performance Estimates and Bounds for Special Mappings

In this section, we analyze the complexity of the circuits constructed by our method.

### 5.1 Permutations

In the case of permutations within a single ciphertext, the chromatic number $\chi$ of the graph that our method constructs to assign elements to sets of rotations is bound by $\log \ell$. We prove this in the following theorem:

**Theorem 1.** *It takes at most $\chi = K - 1$ colors to color graph $G_\pi$ representing the collisions of permutation $\pi$ with preimage $P$.*

*Proof.* It suffices to show that any element $x \in P$ can only collide with at most $\log_2(\ell) - 1$ other elements at one position. In that case, $x$ and the other elements are all connected via an edge and must all be assigned a different color. For brevity, we denote $K = \log_2(\ell)$.

Let us express an upper bound for the maximum number of elements at a single position after $r$ rotations as a function $M(r)$. At the first rotation, the maximum number of overlaps is $M(1) = 1$, because the encrypted vector has no overlaps. At every rotation after that, the maximum number of overlaps is that of the previous rotation, plus one element that was already in this position, so $M(i) = M(i-1) + 1$. This only holds for $i = 2, \ldots, K-1$, however, because at the $K$th rotation, the result must not have any overlaps given that $\pi$ is a permutation. So, $M(K) = 0$. Our function $M$ is undefined for any other values.

We reach the maximum number of overlapping elements at any rotation at $M(K-1) = K - 1$. In fact, this upper bound overestimates the number of overlapping elements, because, after $r$ rotations, the overlapping elements can only move to $2^{K-r}$ remaining positions, so $K-1$ overlapping at $M(K-1)$ cannot satisfy a valid permutation.

As a result, we require at most $\log(\ell)$ sets of $\log(\ell)$ rotations. Also notice that in the case of arbitrary rotations, the number of rotations required is $O(n)$, when $\ell$ is kept constant. This is because even in the worst case where each of the $n$ elements to be permuted is assigned to a separate set of rotations, the relation is linear. However, this situation should be seen as an upper bound because when the number of elements grows, the sets of rotations become more densely packed in the average case. The number of plaintext multiplications and additions scale quadratically with the number of rotations because before the $x$th rotation there can be additions and multiplications with the prior $x - 1$ resulting ciphertexts.

### 5.2 Bounded Rotation Magnitude

The number of rotations that one element occupies is exactly the number of ones in the binary representation of the distance it must move. This number, which is called the Hamming weight, is $\frac{1}{2}\ell$ on average for random permutations. However, if the distance that elements move is bound or the Hamming weight of the distances is low, we expect to pack more elements within one set of rotations.

## 6 Results

In this section, we analyze the performance of our open-source implementation[2] and compare it against HElib. To facilitate a fair comparison, we execute our circuits with HElib's implementation of BGV. Note, however, that any FHE library can execute the resulting circuits with minimal engineering effort.

We perform three sets of experiments, which are increasingly generic. We start by comparing the performance of permutations within a single ciphertext to HElib. Then, we extend HElib to perform arbitrary permutations across multiple ciphertexts and compare the implementation against our work. Finally, we analyze the run time performance of our implementation when performing arbitrary mappings for increasing degrees of overlapping and replication.

Table 3 contains the parameters we used for our experiments. We choose the order of the cyclotomic polynomial $m = 2^x$ for some $x$, following the homomorphic encryption standard [1]. Since the number of slots $\ell = \frac{\phi(m)}{\mathrm{ord}(p)}$, we want the plaintext modulus $p$ to have a low order modulo $m$. On the other hand, when $\ell$ is large, the depth of our circuits might cause the noise in the ciphertexts to grow too large. So, we choose the highest $\ell$ for which the ciphertexts are still decryptable while selecting the lowest $p$ that satisfies it. We provide the number of bits in the modulus chain $\log_2 Q$, which we maximized while satisfying 128 bits of security as specified by the homomorphic encryption standard [1].

**Table 3.** BGV parameters used in the experiments

| | Order $m$ | Modulus $p$ | $\log_2 Q$ | Slots $\ell$ | HElib's depth |
|---|---|---|---|---|---|
| Small | $2^{13} = 8192$ | 31 | 111 | $2^4 = 16$ | 4 |
| Medium | $2^{14} = 16384$ | 127 | 213 | $2^6 = 64$ | 7 |
| Large | $2^{15} = 32768$ | 5119 | <440 | $2^6 = 64$ | 9 |

We executed all our experiments on a Unix machine with 16 virtual Intel® Xeon® Cascade Lake CPUs at 3100 MHz and 64 GB of memory. However, we only executed our experiments on a single thread. While our technique would work on any leveled homomorphic RLWE-based ciphertexts, we used the BGV cryptosystem in our experiments. Since the actual contents of the ciphertexts do not influence the performance in our experiments, we choose repeated encryptions of $0, \ldots, p-1$.

### 6.1 Within-ciphertext Permutations

Since HElib's permutation circuits aim to perform permutations on single ciphertexts, we compare its performance with that of our method. We test performance on the same 50 randomly-generated permutations. In Fig. 4 we show the mean

[2] The repository can be found at https://github.com/jellevos/perm_map_circuits.

run time to perform such a permutation, not considering the setup time, which is considerably smaller. Notice that our method outperforms HElib in each scenario. Moreover, while we execute the separate sets of rotations consecutively in these experiments, one can execute them on separate threads for an even larger speed-up. On the other hand, unlike HElib, our method does not allow the user to specify a maximum circuit depth, so this is only a suitable alternative when the ciphertext's noise budget is large enough.

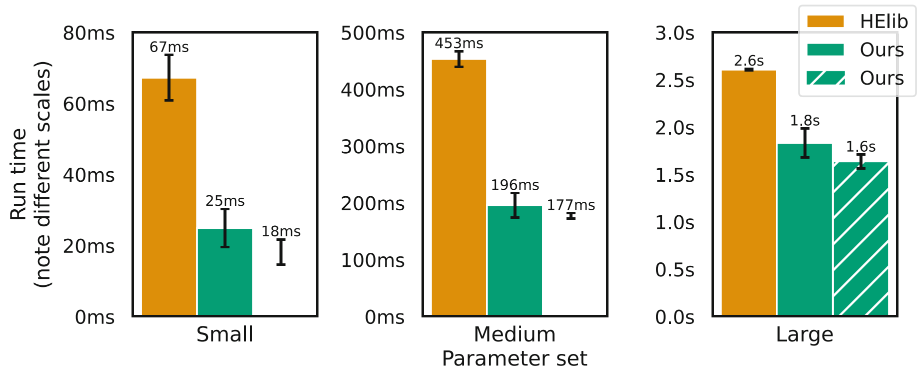

**Fig. 4.** While our circuits are not specifically made for permutations within ciphertexts, they outperform HElib in execution time for a similar noise budget by a factor 1.4× for large parameters up to 2.7× for small parameters. The error bars denote the standard deviation.

In our experiments, we aimed for the remaining noise budgets between our method and HElib's method to be similar, as displayed in Table 4. To do so, we set the depth bound for HElib's permutation circuit as displayed in the rightmost column of Table 3.

**Table 4.** Average remain noise budget of the resulting ciphertext expressed in bits. Here, higher is better, but we selected the parameters for both works to perform similarly.

| | Small | Medium | Large |
|---|---|---|---|
| HElib | 11.72 | 10.14 | 26.86 |
| Ours | 5.38 | 22.24 | 29.68 |
| Ours (long setup) | 5.46 | 22.34 | 29.76 |

## 6.2 Arbitrary Permutations

Next, we evaluate the performance when the number of ciphertexts we permute across grows. We measure the execution time for each number of ciphertexts over 20 random permutations, disregarding our long-setup method. We present the results in Fig. 5. The experiment supports the worst-case complexities that predict HElib's method to scale quadratically and our method linearly regarding the number of ciphertext rotations, which make up the most expensive operation. The improvement in run time is significant, exceeding an order of magnitude starting from as little as five ciphertexts.

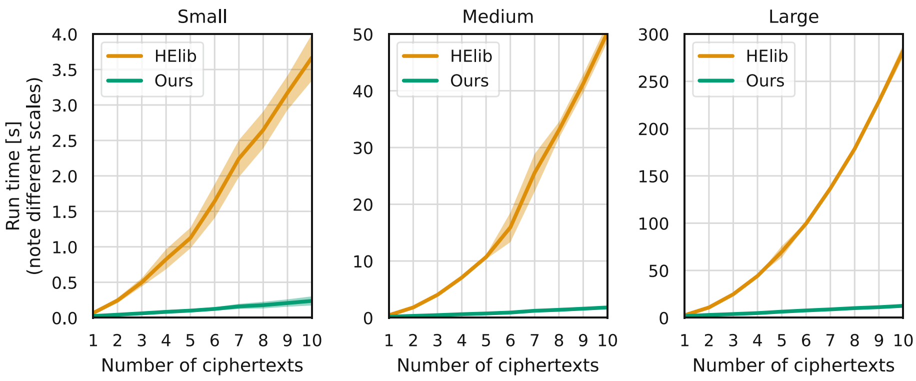

**Fig. 5.** Execution time for random permutations among a growing number of ciphertexts. The experiment confirms that the execution time of HElib scales quadratically, while our approach scales linearly. The shaded area represents the 99% confidence interval.

## 6.3 Arbitrary Mappings

Finally, we evaluate the setup and execution time required for performing arbitrary mappings using our method. We do not consider HElib's method for these experiments, which is prohibitively expensive when the overlap or replication degree exceeds 1. Our experiment considers random mappings across eight ciphertexts, which we generate by creating a set of possible targets and distributing them among the indices of each ciphertext, taking into account the overlap and replication constraints. We present the results in Fig. 6. In this figure, the upper left corner is an arbitrary permutation, and the leftmost column represents injective mappings (replications). Notice that the small and medium parameters finish in the order of seconds, even when elements in the output are allowed to overlap with three other elements. Also, notice that both the setup time and execution time only significantly increase when *both* the overlap and replication degree.

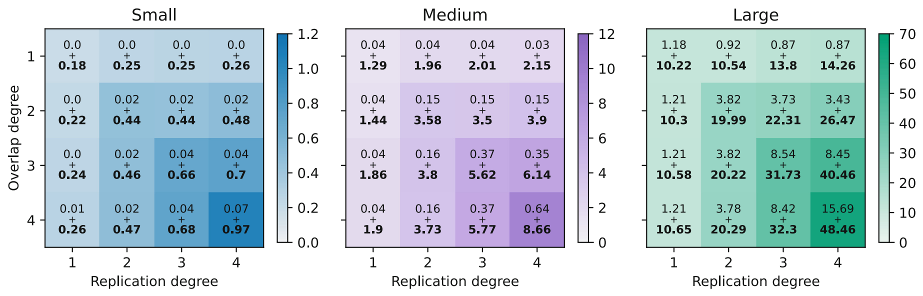

**Fig. 6.** Total time in seconds of arbitrary mappings for increasing overlap and replication degrees. The bold number is the execution time, while the time above is the setup time. Notice that the times hardly increase when only one of the parameters grows and that the setup time becomes non-negligible for higher replication and overlap degrees.

## 7 Conclusion

To the best of our knowledge, this work proposes the first efficient method for constructing mapping circuits across multiple ciphertexts. We experimentally show that our method consistently outperforms the algorithm in HElib, given a ciphertext that supports a large enough multiplicative depth.

Still, open questions remain:

1. Future work can optimize the generated circuits by pruning parts of the circuit. For example, there is no need to isolate elements using a plaintext multiplication when the ciphertext already only contains those elements.
2. Future work might look for an optimization algorithm that separately optimizes the order of rotations.
3. In our current method, all sets of rotations contain all power-of-two rotations, but one might construct shallower circuits by considering using only a subset of those rotations. Such a method would require a different optimization algorithm, however.

With our new primitive, one can construct efficient permutation circuits for permuting elements within a single ciphertext and across multiple ciphertexts. Where previous methods scale quadratically with the number of elements to permute, our method scales linearly regarding the total number of rotations to perform. Our method is concretely efficient when previous work becomes prohibitively expensive.

**Acknowledgement.** We would like to thank Neil Yorke-Smith for his great help with our optimization algorithm.

## References

1. Albrecht, M., et al.: Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada (November 2018)
2. Armbrust, M., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
3. Beneš, V.E.: Optimal rearrangeable multistage connecting networks. Bell Syst. Tech. J. **43**(4), 1641–1656 (1964). https://doi.org/10.1002/j.1538-7305.1964.tb04103.x
4. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. IACR Cryptol. ePrint Arch. p. 277 (2011). https://eprint.iacr.org/2011/277
5. Brélaz, D.: New methods to color the vertices of a graph. Commun. ACM **22**(4), 251–256 (1979). https://doi.org/10.1145/359094.359101
6. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 409–437. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_15
7. Damgård, I., Ishai, Y., Krøigaard, M.: Perfectly secure multiparty computation and the computational overhead of cryptography. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 445–465. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13190-5_23
8. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptol. ePrint Arch. p. 144 (2012), https://eprint.iacr.org/2012/144
9. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009, pp. 169–178. ACM (2009). https://doi.org/10.1145/1536414.1536440
10. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_28
11. Halevi, S., Shoup, V.: Algorithms in HElib. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8616, pp. 554–571. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44371-2_31
12. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA, pp. 85–103. The IBM Research Symposia Series, Plenum Press, New York (1972). https://doi.org/10.1007/978-1-4684-2001-2_9