Verified non-recursive calculation of Beneš networks applied to Classic McEliece

Wrenna Robson[1] and Samuel Kelly[2]

[1] University of Bristol, Bristol, UK, wrenna.robson@bristol.ac.uk
[2] University of Bristol, Bristol, UK, sam.j.kelly.2020@bristol.ac.uk

Abstract. The Beneš network can be utilised to apply a single permutation to different inputs repeatedly. We present novel generalisations of Bernstein's [Ber20] formulae for the control bits of a Beneš network and from them derive an iterative control bit setting algorithm. We provide verified proofs of our formulae and prototype a provably correct implementation in the Lean language and theorem prover. We develop and evaluate portable and vectorised implementations of our algorithm in the C programming language. Our implementation utilising Intel's Advanced Vector eXtensions 2 feature reduces execution latency by 29% compared to the equivalent implementation in the libmceliece software library on the Intel Ultra 7 165U CPU.

Keywords: Computer aided cryptography, formal methods, efficient software implementation, permutation networks

1 Introduction

The secure and efficient implementation of permuting and sorting is a fundamental challenge in contexts such as privacy-preserving cloud computing [EGT10, OSF+16, DDCO17], the mitigation of side-channels Trusted Execution Environments (TEEs) [HOW22, SJG23], Fully Homomorphic Encryption (FHE) [GHS12, HKC+21], Multi-Party Computation (MPC) [HEK12, MS13, KS14], and post-quantum cryptography [BBC+20, ABC+22a].

Problem statement. In this work, we address the problem of efficiently applying a single secret permutation to multiple arrays in constant-time software. Notably, the constant-time requirement implies that one cannot access memory indices that depend on the permutation to apply the permutation. This rules out naive implementations. Whilst there are well-established methods for applying a single secret permutation to a single array, e.g., via sorting networks, the repeated use of these general methods to apply a single known permutation leaves potential perfomance gains on the table.

Repeated application of a single secret permutation is required for decapsulation in the Classic McEliece Key Encapsulation Mechanism (KEM). Addressing this challenge prompted [BCS13], [Cho17], and [Ber20] to shed new light upon the theories of sorting networks [Bat68, Knu98] and permutation networks [Ben65, Lei14] and their high-assurance cryptographic implementation. Our work further extends this body of research.

Motivation. We motivate our work through the challenges faced by the Classic McEliece team. Each decapsulation requires one to apply a secret permutation π to a q-bit string. Note that the obvious approach of using π to index into the string could leak information about the secret through the cache. They initially solve this problem in [BCS13] by adapting a common technique for obliviously applying a single permutation to a single input: the decorate-sort-undecorate idiom (see Subsection 2.3) with Batcher's odd-even

sorting network [Bat68]. Briefly, if one records the $(m^2+m+4)2^{m-1}-1$ comparator swaps applied by this sorting network to the permutation on 2^{m+1} elements π^{-1}, then 'replaying' this sorting network, with its comparators swapped for binary switches, is equivalent to applying π. Since a binary switch, or conditional flip, can be implemented in constant-time using only 4 bit operations, this is a simple constant-time solution. However, this solution is not optimal: all permutations can be realised through fewer binary switches.

The Beneš network [Ben64, Ben65, Ben75], or rearrangeable permutation network, is an arrangement of binary switches, connected by transmission links, that can be configured to realise arbitrary permutations to the input data passed through it. A binary switch has two inputs and two outputs. If the *control bit* associated with a given switch is set to 1, the positions of the two inputs are swapped and output. If the control bit is 0, the inputs are left in-place and output. The Beneš network realising a permutation on 2^{m+1} elements has $2m+1$ layers of 2^m switches. For the Classic McEliece parameter set where 2^{13} are permuted, this reduces the binary switches required to apply the permutation by two-thirds compared to the method described above. The Classic McEliece team specify the uniform random ordering of the field $\mathbb{F}_q$ as a sequence of control bits for the 'in-place' Beneš network [ABC+22b].

Utilising the Beneš network in decapsulation, however, adds the calculation of the control bits to the key generation phase. This is a complicated and relatively expensive computation with a large design space of algorithms and implementation characteristics to choose from. Bernstein implements the Nassimi-Sahni [NS82] fast parallel variant of the Waksman-Stone canonical 'looping' algorithm [Wak68]. This implementation is adopted in the libmceliece software implementation of Classic McEliece [Cho25]. However, as Bernstein notes, *"it takes effort to see why [some] algorithms work"* and why some do, or do not, give the same control bits as each other [Ber25b]. The 'looping' algorithm is recursive: one sets the control bits of the outermost layers so that the inner layers act separately on two half-size disjoint permutations. Bernstein verifies this step with the HOL Light proof assistant [Har96] by providing formulae for the control bits of the outer layers [Ber20]. He proves that these formulae, when applied to an arbitrary permutation, do give outer layer permutations which when factored out leave a parity-preserving permutation.

By factoring this parity-preserving permutation into two permutations on odd and even inputs, Bernstein can proceed inductively. However, the Classic McEliece specification requires the control bits to be stored and applied following the in-place representation of the Beneš network (see Subsection 2.1). Therefore, the calculated control bits must be reordered to match this in-place form. There is a gap here. Bernstein does not prove that the specification's in-place application of the reordered control bits will indeed realise the permutation from which those control bits were calculated. Whilst a careful reader can check the reordering is valid and that the reference implementation of it is correct, this disconnect produces an algorithm that requires the delicate passing of position tracking variables between recursive calls.

The recursive construction also hampers the algorithm's implementation in the Single-Instruction Multiple-Data (SIMD) paradigm. The libmceliece software library provides an implementation designed for higher performance on Intel and AMD CPUs by utilising the Advanced Vector eXtensions 2 (AVX2) feature. The use of AVX2 accelerates several primitives in the Classic McEliece KEM, such as matrix and vector operations in $\mathbb{F}_q$. However, in the calculation of the control bits, the majority of recursive subcalls act on permutations with data sizes smaller than the width of AVX2's 256-bit ymm registers, resulting in suboptimal usage of the available hardware.

Contributions. In this paper, we further investigate the calculation of the control bits for the Beneš network. We present novel formulae for the control bits of a Beneš network, and use them to derive an iterative switch setting algorithm. We develop and evalu-

ate portable and AVX2-enabled implementations of our algorithm. The latter reduces execution latency by 29% compared to the equivalent AVX2-enabled implementation in libmceliece version 20250507 on an Intel Ultra 7 165U CPU.

We also prove our novel formulae formally correct. We provide pen-and-paper proofs in this paper, but also provide proofs of our theorems in the Lean programming language and theorem prover [MU21]. In fact, this formalisation pre-dated the innovation of the formulae themselves: that is to say, the process of their discovery arose from formal exploration.

The original work [Rob25] on which this paper builds its implementation aimed originally to re-implement and slightly extend the work of [Ber20], as part of a larger monograph on the challenges in producing verified high-assurance cryptography. However, challenges encountered during that process led to the realisation that it might be possible to perform the careful and more extensive generalisation on which our work rests and which we present below. Formalisation does not have to be an adjunct to discovery — this was a happy example in which it was the foundation of it.

There are caveats and limitations to our formalisation. We provide details on the precise claims we believe are safely formally supported in Subsection 4.4. Notably, our C implementations use a notion of interlaced sorting networks around which there are some verification gaps, though we do provide a simple argument which evidences the validity of rearranging sorting networks to leverage SIMD instructions — key to the speed-ups we mention above.

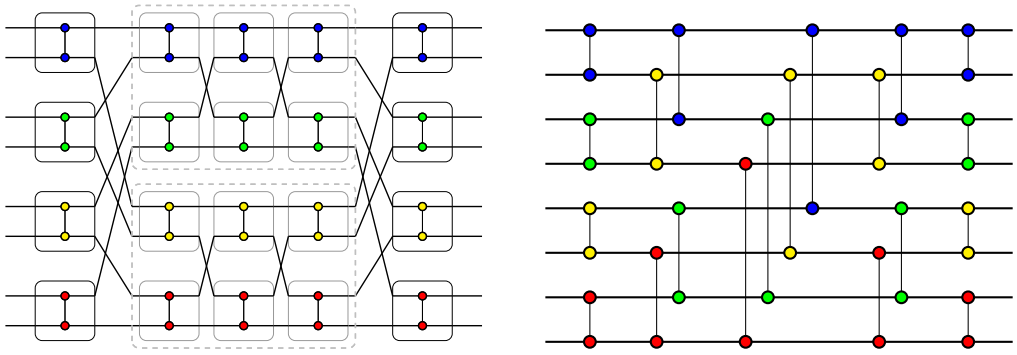
Structure. In Section 2, we provide background on Beneš networks, notation, and the recursive calculation of control bits. In Section 3, we present our novel iterative formulae and algorithm and explain how we use interlaced sorting networks to perform efficient parallel calculation of the control bits. In Section 4, we present our proofs and discuss their verification in Lean. In Section 5, we describe the implementation of our iterative algorithm and evaluate its performance. All associated material (e.g., source code relating to all software implementations and formally verified proofs of our key result) is available at `https://github.com/bristolcrypto/controlbits/` under the open source MIT license.

2 Preliminaries

2.1 Beneš networks

History. The first rearrangeable permutation networks were presented by Beneš [Ben64, Ben65, Ben75]. Later work by Waksman [Wak68], showed that a single switch could be spared at every step of the recursive construction of a Beneš permutation network; networks with this optimisation are often called Waksman networks. Chang and Melham show that Beneš networks can be extended to permute non-power of two sized lists [CM97], with Beauquier and Darrot showing the same result for Waksman networks specifically [BD99].

The canonical algorithm for calculating the control bits is the 'looping' algorithm of Waksman-Stone [Wak68]. In this work, Waksman presents a constructive proof, which he accredits to Stone, that the Beneš network is capable of applying any permutation on 2^{m+1} elements. Stone's proof leads directly to the aforementioned algorithm. The approach is to calculate the outermost stages of switches (via a 'looping' procedure) such that, after applying these two outer stages, the inner permutation of size 2^{m+1} is partitioned into two permutations of size 2^m; one then continues recursively. Fast parallel variants of this algorithm were introduced by Lev-Pippenger-Valiant [LPV81] in 1981, Nassimi-Sahni [NS82] in 1982, and Lee-Liew [LL96] in 1996. These parallel algorithms exploit a common idea, allowing them to compute the control bits of an outer layer in a number



(a) Shuffled Beneš Network. (b) In-place Beneš Network.

Figure 1: Representations of Beneš Networks with $n = 8$ inputs.

of steps logarithmic to the input permutation's size. Bernstein verifies the correctness of Nassimi-Sahni's algorithm and shows that it can be implemented obliviously using sorting networks [Ber20]. Our algorithm is an iterative variant of this 'looping' algorithm.

A different approach to calculating the control bits was introduced by Lee [Lee85, Lee87]. This approach is distinct from the others as it does not calculate pairs of outer layers. Instead, it iteratively sets the control bits a single layer at a time from left to right. The construction divides the layers into two halves. The control bits in the first half are set so that the control bits in the second half can be set by a trivial switch setting algorithm. This algorithm was recently used as a basis for oblivious sorting and shuffling in the online/offline model [SJG23].

Visual representations of Beneš networks. Algorithms for calculating and storing control bits are often framed inside a particular visual representation of the Beneš network. Many of these representations are rearrangements of each other. We depict two common representations in Figure 1. The inputs follow the transmission links which pass from left to right. A switch between two transmission links is represented by a vertical line with circles at its end points. The colours on the switches are to help visualise the rearrangements. We use the terms 'in-place' and 'shuffled' to match Bernstein's usage in [Ber20] and 'modified' to match the usage in [SJG23]. The shuffled Beneš network is depicted in Figure 1a. The in-place representation, depicted in Figure 1b, can be realised by 'straightening out' the transmission links: fix the positions of the outer switches of Figure 1a and align the end points of the outermost switches of the recursive halves to them.

The control bits are typically stored in an order that corresponds with a given visual representation. For each of the representations, one stores the control bits ordered in layers from left to right and within layers from top to bottom. The Classic McEliece specification stores the control bits according to the in-place representation. This complicates the recursive libmceliece implementation as it calculates the control bits according to a depth-first traversal of the decomposed halves of the shuffled representation, whilst simultaneously storing them in their in-place representation's order.

From an implementation perspective, there is no need to strictly follow any given visual representation of the Beneš Network. One is free to rearrange both the memory addresses of the elements to be compared and the order of application of conditional swaps (within each layer). When implementing the Beneš network in hardware, one may wish to minimise the wire length and choose the shuffled Beneš network. In software, the in-place and modified representations are chosen to remove the perfect shuffle transmission links between layers. The representation chosen determines the order of memory accesses and has a large impact on performance. For example, Sasy et al. find their implementation of Lee's algorithm with the modified representation is *'one to two orders of magnitude faster'*

than the implementation of Nassimi-Sahni's algorithm in libmceliece with the in-place representation. Yet, when the same algorithm is ran with the modified representation, the difference is *'negligible'* [SJG23, Sec. 7.1]. Further, Chou's optimised software implementation of control bit application swaps between the in-place and modified representations in order to always apply 128 conditional swaps in parallel [Cho17].

2.2 The Lean theorem prover

Lean is a programming language and interactive theorem prover [dMU21], based originally on the same foundation as the Rocq [BC04] theorem prover. Lean aspires to be a first-class programming language in its own right, rather than a theorem prover first with programming capability. Mathlib [mC20] is Lean's main mathematics library: it is comparable to Rocq's MathComp. In recent years Lean has enjoyed unusually high usage by leading mathematicians [ABLM20, Sch22, Tao24], which reflects the strong community that has built up around it.

2.3 Notation and definitions

Notation. We denote the exclusive-or, or XOR, operation with $\oplus$. Let $n \in \mathbb{N}$ be a nonnegative integer and $\mathbb{N}_{<n}$ denote the set of nonnegative integers less than n. Let V, U be arrays of length n. We denote the i-th entry of an array V by $\mathsf{V}[i]$. We use the following slice notation $\mathsf{V}[l : u : s]$ to denote the collection of elements in V of the form $\mathsf{V}[l + i \cdot s]$ for $0 \leq i < u/s$. We define the array $\min(\mathsf{U}, \mathsf{V})$ such that $\min(\mathsf{U}, \mathsf{V})[i] = \min(\mathsf{U}[i], \mathsf{V}[i])$. The array given by concatenating arrays V and U is given by $\mathsf{V} \| \mathsf{U}$. We denote by S_n the group of permutations on $\mathbb{N}$ which are fixed for all inputs $x \geq n$. Equivalently, this is the symmetric group on n elements $\{0, \ldots, n-1\}$. Transpositions are cycles that swap two elements, denoted (i, j). We denote the commutator of two group elements g, h as follows: $[g, h] := g \cdot h \cdot g^{-1} \cdot h^{-1}$. For a permutation $\pi \in S_n$, we define $\mathsf{Vec}(\pi)$ to be the array of length n such that $\mathsf{Vec}(\pi)[i] = \pi(i)$. If an array A represents a permutation, i.e., $\mathsf{A} = \mathsf{Vec}(\pi)$ for some π, then we define the reverse conversion $\pi = \mathsf{Perm}(\mathsf{A})$. We denote the identity permutation by $\mathbb{I}_n \in S_n$ and its corresponding array $\mathsf{id}_n = \mathsf{Vec}(\mathbb{I}_n)$. An involution $g \in S_n$ satisfies $g^2 = \mathbb{I}_n$. Finally, given a permutation $\pi \in S_n$ acting on an element x, we denote the minimum member of the cycle of π containing x as $\mathrm{cyclemin}(\pi)(x) = \min\{\pi^k(x) \mid k \in \mathbb{Z}\}$. Observe that for any π, $\mathrm{cyclemin}(\pi)(\pi(x)) = \mathrm{cyclemin}(\pi)(x)$.

Bit operations. Given a nonnegative integer x, we access the k-th bit of x by the function $\mathrm{bit}_k : \mathbb{N} \to \{0, 1\}$. The least significant bit is given by $\mathrm{bit}_0(x)$ — this is the parity bit of x.

Definition 1. We define the following operations (here given arithmetically rather than as bit operations):

$$\mathrm{insertbit}_k(b, x) := 2^{k+1} \left\lfloor \frac{x}{2^k} \right\rfloor + \left(b \cdot 2^k\right) + \left(x \bmod 2^k\right).$$
$$\mathrm{removebit}_k(x) := 2^k \left\lfloor \frac{x}{2^{k+1}} \right\rfloor + \left(x \bmod 2^k\right).$$

The function $\mathrm{insertbit}_k(b, x) : \{0, 1\} \times \mathbb{N} \to \mathbb{N}$ inserts the bit b into bit position k of x, leaving the k-th least significant bits in-place and left shifting the most significant bits above the inserted bit. Conversely, the function $\mathrm{removebit}_k : \mathbb{N} \to \mathbb{N}$ removes the k-th bit of an integer, shifting the higher bits down. Observe that for all x, $\mathrm{insertbit}_k(\mathrm{bit}_k(x), \mathrm{removebit}_k(x)) = x$.

Definition 2. We define $\mathrm{flip}_k : \mathbb{N} \to \mathbb{N}$ as the map defined by $\mathrm{flip}_k(x) := x \oplus 2^k$: the involutive permutation that flips the k-th bit of its input.

Note that for all x, $\mathrm{flip}_k(x) = \mathrm{insertbit}_k(1 \oplus \mathrm{bit}_k(x), \mathrm{removebit}_k(x))$ and that $\mathrm{flip}_k(x) \equiv x \mod 2^k$. As such, when $k \leq m$, $\mathrm{flip}_k \in S_{2^{m+1}}$. In this case we can view it as the product of 2^m transpositions of the form $(\mathrm{insertbit}_k(0, j), \mathrm{insertbit}_k(1, j))$ for $0 \leq j < 2^m$.

Definition 3. We define $\mathrm{cflip}_k : \{0,1\} \times \mathbb{N} \to \mathbb{N}$ as the map given by $\mathrm{cflip}_k(b, x) := x \oplus b \cdot 2^k$. This is a conditional version of flip_k.

Note that for fixed b, $\mathrm{cflip}_k(b, \cdot) = (\mathrm{flip}_k)^b$ — it is also a permutation. We have $\mathrm{cflip}_k(b, x) = \mathrm{insertbit}_k(b \oplus \mathrm{bit}_k(x), \mathrm{removebit}_k(x))$ and $\mathrm{bit}_k(\mathrm{cflip}_k(b, x)) = b \oplus \mathrm{bit}_k(x)$.

2.4 Oblivious application of permutations

Given an array A and permutation π, the permuted array whose i-th entry is $\mathsf{A}[\pi(i)]$ is denoted by $\mathsf{A} \triangleleft \pi$. In Classic McEliece the permutation is secret, thus, to avoid the risk of leaking information about π through side-channels such as the cache, we never compute $\mathsf{A} \triangleleft \pi$ by using π to index into A. We instead apply permutations obliviously. For the particular case where π is a conditional transposition $(i, j)^b$, where $b \in \{0, 1\}$ is secret but i and j are not, we define the primitive $\textsc{oFlip}(\mathsf{A}, i, j, b)$. Using a short sequence of bit operations or instructions such as Intel's `cmovg`, we can obliviously swap elements $\mathsf{A}[i]$ and $\mathsf{A}[j]$ just when the bit $b = 1$. Therefore, we have $\textsc{oFlip}(\mathsf{A}, i, j, b) = \mathsf{A} \triangleleft (i, j)^b$. This primitive is utilised to apply the Beneš network.

For an arbitrary permutation π, we can calculate $\mathsf{A} \triangleleft \pi^{-1}$ with the decorate-sort-undecorate idiom and a data-oblivious sorting algorithm (for which, in this paper, we use sorting networks). The process is as follows: From A, form the decorated array B of tuples, whose i-th member is $(\mathsf{Vec}(\pi)[i], \mathsf{A}[i])$. Sort B in-place lexicographically. Undecorate B, returning the array whose i-th element is equal to the right element of the i-th tuple of B which is $\mathsf{A} \triangleleft \pi^{-1}$. We can calculate $\mathsf{A} \triangleleft \pi$ by first applying this process with $\mathsf{A} = \mathsf{id}$, calculating $\mathsf{id} \triangleleft \pi^{-1} = \mathsf{Vec}\left(\pi^{-1}\right)$ and, then, using this to calculate $\mathsf{A} \triangleleft (\pi^{-1})^{-1} = \mathsf{A} \triangleleft \pi$.

The decorate-sort-undecorate idiom is preferred for constant-time sampling of permutations in Classic McEliece, PERK [ABB+24], and [Beu20] as it is much more efficient than constant-time Fisher-Yates shuffling (see [BCMP24]).

2.5 The layers of the in-place Beneš network

The Classic McEliece specification applies the control bits in the in-place Beneš network representation [ABC+22a]. For a size 2^{m+1} array A with control bits C, we sequentially apply $\textsc{oFlip}(\mathsf{A}, \mathrm{insertbit}_k(0, j), \mathrm{insertbit}_k(1, j), \mathsf{C}[j])$ for $0 \leq j < 2^m$. Algorithm 1 defines this operation. Note that we can also view this as a family of permutations applied to the indicies: the product of (conditional) transpositions $(\mathrm{insertbit}_k(0, j), \mathrm{insertbit}_k(1, j))^{C[j]}$. We call this permutation family layer_k: it is involutive, because all of the transpositions commute with one another (as $\mathrm{insertbit}_k$ is injective, there are no overlapping values) and as such when applied twice they all cancel to the identity.

Definition 4. We define the permutation family $\mathrm{layer}_k : \{0,1\}^{2^m} \to S_{2^{m+1}}$ as

$$\mathrm{layer}_k(\mathsf{C}) := \prod_{j=0}^{2^m-1} (\mathrm{insertbit}_k(0, j), \mathrm{insertbit}_k(1, j))^{C[j]}.$$

We then have $\textsc{oLayer}(\mathsf{A}, \mathsf{C}, k) = \mathsf{A} \triangleleft \mathrm{layer}_k(\mathsf{C})$: $\textsc{oLayer}$ contains precisely the sequence of conditional transpositions that defines layer_k. It is convenient to have a closed form for the application of layer_k to a particular value.

Lemma 1. *For all x,* $\mathrm{layer}_k(\mathsf{C})(x) = \mathrm{cflip}_k\left(\mathsf{C}[\mathrm{removebit}_k(x)], x\right)$

Algorithm 1 Obliviously apply one layer of the in-place Beneš network.

function oLayer(A: 2^{m+1} element array, C: array of 2^m control bits, k: gap size)
  for $j = 0$ to $2^m - 1$ inclusive **do**
    $\mathsf{A} \leftarrow \text{oFlip}(\mathsf{A}, \text{insertbit}_k(0, j), \text{insertbit}_k(1, j), \mathsf{C}[j])$
  end for
  return A
end function

Algorithm 2 Apply the control bits of an in-place Beneš network.

function cbapply(A: 2^{m+1} element array to be permuted, C: array of $(2m+1)2^m$ control bits)
  for $j = 0$ to $2m$ inclusive **do**
    $k \leftarrow \min(j,\ 2m - j)$
    $\mathsf{A} \leftarrow \text{oLayer}(\mathsf{A},\ \mathsf{C}[j \cdot 2^m : (j+1) \cdot 2^m : 1],\ k)$
  end for
  return A
end function

Proof. In the product of permutations in the definition of $layer_k$, each transposition appears exactly once. We have $x = \text{insertbit}_k(\text{bit}_k(x), \text{removebit}_k(x))$: as such it is exactly the transposition with $j = \text{removebit}_k(x)$ that x appears in. In this cases, the j-th bit of x is flipped just when $\mathsf{C}[\text{removebit}_k(x)]$ is set: i.e. the result is $\text{insertbit}_k(\mathsf{C}[\text{removebit}_k(x)] \oplus \text{bit}_k(x), \text{removebit}_k(x))$. This is exactly $\text{cflip}_k\,(\mathsf{C}[\text{removebit}_k(x)], x)$. □

The in-place Beneš network on 2^{m+1} elements is the product of $2m+1$ such layer applications, each layer represented by an array of 2^m bits, thus to represent the entire network requires an array of size $(2m+1) \cdot 2^m$. Algorithm 2 describes, given a size 2^{m+1} array A and an array of $(2m+1) \cdot 2^m$ control bits C, how to right-apply the permutation represented by the control bits C to the permutation A. This is done by sequentially applying the layers contained in C. This can be viewed as a permutation on the indices.

2.6 Recursive calculation of control bits

One of our primary conditions is a careful generalisation of the proofs contained in [Ber20], which the author himself describes as "*4 pages of definition-theorem-proof in brutalist style*" [Ber25b]. We favour a similarly brutalist style, as it naturally translates from our verified Lean implementation. However, to aid intuition, in this subsection we motivate this formal approach with a brief description of the algorithms of Waksman-Stone [Wak68] and Nassimi-Sahni [NS82], on which Bernstein's work is based. These algorithms choose the control bits for the outermost layers, denoted by arrays F and L, in order to factor our permutation $\pi \in S_{2^{m+1}}$ into the outer layers and a parity-preserving inner permutation ρ, i.e $\pi = \text{layer}_0(\mathsf{F}) \cdot \rho \cdot \text{layer}_0(\mathsf{L})$.

The 'looping' algorithm. The proof presented in [Wak68] leads directly to the 'looping' algorithm for determining the outer layers. To demonstrate this procedure, consider an example using the following permutation $\pi \in S_8$, given by $\pi = (0, 6, 7, 5)(1, 2, 4)$.

Figure 2 is a standard depiction of this procedure. For the network to realise π, we must connect each input x on the left with its output $\pi(x)$ on the right. We are allowed to set the switches in the outermost layers and form arbitrary connections within ρ_0 and ρ_1. We begin the 'looping' procedure at output $0 = \pi(5)$ and arbitrarily set $\mathsf{F}[0] := 0$, i.e., set the switch straight by Waksman's switch saving scheme. We must then connect output 0 to its corresponding input $\pi^{-1}(0) = 5$, requiring us to pass through ρ_0 and then

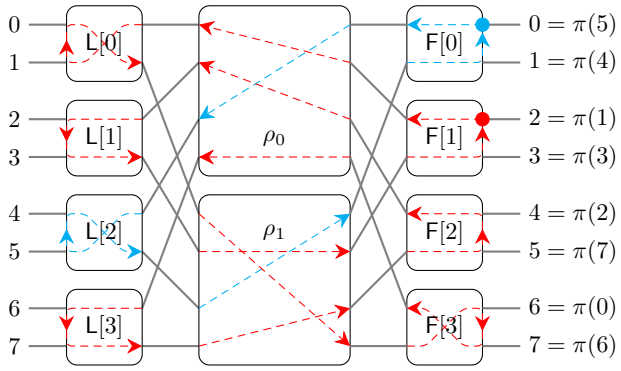


Figure 2: Example Calculation of the 'Looping' Algorithm.

set $\mathsf{L}[2] := 1$. This, in turn, connects input 4 to ρ_1 and we finish the loop at the switch we began. The blue path represents this loop. We now begin another loop, depicted in red, at the least unset output $2 = \pi(1)$, setting switch $\mathsf{F}[1] := 0$ again by Waksman's switch saving scheme. We continue looping backwards and forwards until the loop is complete. If not all of the control bits are set, we begin another loop at the least unset switch. Two proofs of the validity of this approach can be found in [Wak68, Lei14].

Cycle minima calculation. Nassimi and Sahni observe that this approach of following cycles sequentially is parallelisable [NS82]. Instead of starting the loops at the least unset control bit and completing the entire cycle before continuing on, one can start the loop from an arbitrary output and record the minimum value of the cycle. The switch at this value must have been set to zero by Waksman's switch saving scheme. Nassimi and Sahni show that one can set all the control bits in terms of the parity of these cycle minima. Whilst calculating every cycle multiple times appears wasteful, the cycles are sufficiently structured so that, in each cycle, we need only take the minima over a logarithmic number of cycle indices to obtain the minima over the entire cycle. The parallelisation results in an algorithm that is natural to implement in constant-time. Calculating every cycle in parallel requires one to simply apply the permutation represented by an array to itself. This is done with the decorate-sort-undecorate idiom. Conversely, the naive sequential following of loops accesses memory dependent on secrets. This is a challenge faced in the constant-time implementation of Lee's switch setting algorithm. This algorithm has a similar loop-like process but does not benefit from the logarithmic speed up when similarly parallelised. A 'scramble-then-compute' approach to implementing the sequential following of loops in constant-time is presented in [SJG23].

We calculate $\text{cyclemin}([\pi, \text{flip}_k])$ using the algorithm $\textsc{FastCycleMin}(\pi, k)$, taken in this form from Bernstein but based on Nassimi-Sahni's work. (Note that FastCycleMin calculates the cycle minima associated with a commutator, not the original permutation).

Recursive algorithm description. Algorithm 3 describes the recursive algorithm utilised by the Classic McEliece implementers, deriving from Bernstein's formulae as we describe below, for calculating the control bits for the in-place Beneš network. Observe that the recursive control flow requires a non-trivial usage of variables `pos` and `step` to ensure the control bits are stored in their in-place representation's positions. The first call to CBRecursion must set `pos` to 0 and `step` to 1. The two recursive calls operate on overlapping regions of memory (though there are no actual race conditions).

As mentioned, Bernstein defines formulae representing the control bit settings as they are given by Nassimi-Sahni, from which this algorithm is derived. Specifically, if $m = 0$ there is just a single control bit which can be read off from the parity of the first element of $\mathsf{Vec}(\pi)$. For $m > 0$, Bernstein then provides formulae for the control bits

Algorithm 3 Recursively calculate the control bits for an in-place Beneš network.

Require: Array P such that $\mathsf{P}[x] = \pi(x)$ for some $\pi \in S_{2^{m+1}}$. A global array C of size $(2m+1)\cdot 2^m$ for the control bits.

function CBRECURSION(P, C, $\mathtt{pos}$, $\mathtt{step}$)
  if $m = 0$ **then** ▷ Base case
    $\mathsf{C}[\mathtt{pos}] \leftarrow \mathsf{P}[0]$
    return
  end if
  $\mathsf{C}_{\min} \leftarrow$ FASTCYCLEMIN($\mathsf{Perm}(\mathsf{P})$, 0)
  for $j = 0$ to $2^m - 1$ inclusive **do**
    $\mathsf{F}[j] \leftarrow \mathrm{bit}_0(\mathsf{C}_{\min}[2j])$ ▷ $\mathsf{F}[j] = \mathrm{cyclemin}([\pi, \mathrm{flip}_0])(2j) \mod 2$.
  end for
  $\mathsf{P} \leftarrow$ OSORT($\mathrm{layer}_0(\mathsf{F})$, OSORT($\mathsf{id}_n$, P, 0), 0) ▷ Set $\mathsf{P}[x] = \mathrm{layer}_0(\mathsf{F})(\pi(x))$
  for $j = 0$ to $2^m - 1$ inclusive **do**
    $\mathsf{L}[j] \leftarrow \mathrm{bit}_0(\mathsf{P}[2j])$ ▷ $\mathsf{L}[j] = \mathrm{layer}_0(\mathsf{F})(\pi(x)) \mod 2$.
  end for
  $\mathsf{C}[\mathtt{pos} : \mathtt{pos} + 2^m \cdot \mathtt{step} : \mathtt{step}] \leftarrow \mathsf{F}$
  $\mathsf{C}[\mathtt{pos} + m \cdot 2^{m+1} \cdot \mathtt{step} : \mathtt{pos} + (2m+1) \cdot 2^m \cdot \mathtt{step} : \mathtt{step}] \leftarrow \mathsf{L}$
  $\mathsf{P} \leftarrow$ OLAYER(P, L, 0) ▷ Set $\mathsf{P}[x] = \mathrm{layer}_0(\mathsf{F})(\pi(\mathrm{layer}_0(\mathsf{L})(x)))$
  for $j = 0$ to $2^m - 1$ inclusive **do**
    $\mathsf{P}_0[j],\ \mathsf{P}_1[j] \leftarrow \left\lfloor \frac{\mathsf{P}[2j]}{2} \right\rfloor,\ \left\lfloor \frac{\mathsf{P}[2j+1]}{2} \right\rfloor$
  end for
  CBRECURSION(P_0, C, $\mathtt{pos} + 2^m \cdot \mathtt{step}$, $2 \cdot \mathtt{step}$)
  CBRECURSION(P_1, C, $\mathtt{pos} + (2^m + 1) \cdot \mathtt{step}$, $2 \cdot \mathtt{step}$)
end function

of the two outer layers F and L. He defines $\mathsf{F}[j] := \mathrm{cyclemin}([\pi, \mathrm{flip}_0])(2j) \mod 2$ and $\mathsf{L}[j] := \mathrm{layer}_0(\mathsf{F})(\pi(2j)) \mod 2$. These formulae are derived from the 'looping' procedure of Waksman-Stone by representing it as the commutator element $[\pi, \mathrm{flip}_0]$. (Note this represents following the loops clockwise, instead of anticlockwise as in Figure 2 above).

He then defines a permutation $\rho = \mathrm{layer}_0(\mathsf{F}) \cdot \pi \cdot \mathrm{layer}_0(\mathsf{L})$ which he proves is parity-preserving. He is then able to define $\rho_0, \rho_1 \in S_{2^m}$ such that they satisfy $\rho(2x) = 2\rho_0(x)$ and $\rho(2x+1) = 2\rho_1(x)+1$. Proceeding inductively, he can calculate control bits for these permutations. The resulting control bits are not in the order required for in-place application, due to this recursive construction, and they must be reassembled via interlacing (or, equivalently, the two recursive steps must operate on overlapping regions of memory).

3 Formulae for the in-place Beneš network

3.1 Interlaced sorting networks

Sorting networks are the default practical choice for implementing data-oblivious sorting. They are fixed, data-independent sequences of compare-and-exchange modules, or comparators, designed to sort a fixed number of inputs [Bat68, Knu98]. The Bitonic [Bat68], Odd-Even Merge [Bat68], and Pairwise [Par92] constructions all require asymptotically $\mathcal{O}(n \log^2 n)$ comparators, and remain the most practical for efficient implementation [Ebb15]. Bitonic is favoured in SIMD implementations for its regular structure [Bra17, BGK21, Ber24].

The input to the k-th iteration of our iterative algorithm (Subsection 3.2) is a permutation $\pi \in S_{2^{m+1}}$ satisfying $\pi(x) \equiv x \mod 2^k$ for all x. This is a highly structured permutation: π is independently permuting 2^k interlaced subsets of size 2^{m+1-k}. Therefore, when utilising the decorate-sort-undecorate idiom to apply this permutation, we do not need to apply a full sorting network on 2^{m+1} elements as many of the comparators are redundant. We can instead utilise a comparator network created from 2^k interlaced

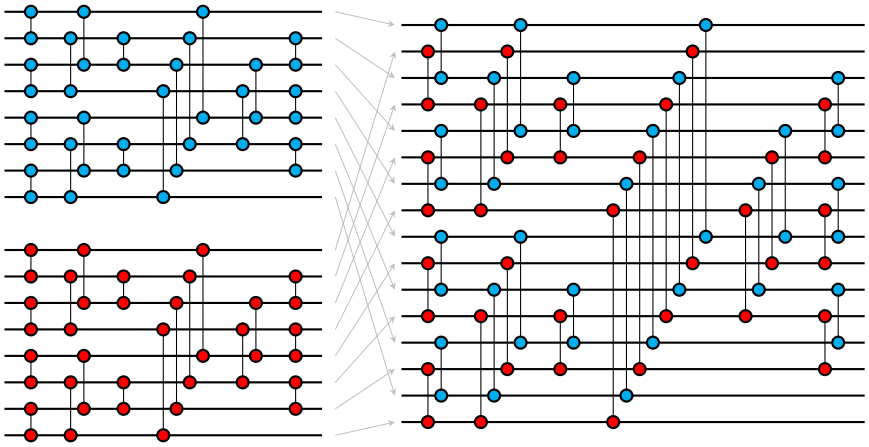


Figure 3: Interlacing two instances of Batcher's Odd-Even Merge sorting network on $n = 8$ elements.

sorting networks on 2^{m+1-k} elements. A visualisation of an interlaced sorting network for any array A on 16 elements satisfying $\mathsf{A}[x] \equiv x \mod 2$ is given in Figure 3.

For $\pi \in S_{2^{m+1}}$, we define $\textsc{oSort}(\mathsf{A}, \mathsf{Vec}(\pi), k)$ to denote the primitive that performs this interlaced decorate-sort-undecorate process. This performs 2^k simultaneous sorts of size 2^{m+1-k}: when $\pi(x) \equiv x \mod 2^k$ for all x, this indeed performs the decorate-sort-undecorate procedure as described above, calculating $\mathsf{A} \triangleleft \pi^{-1}$.

3.2 Iterative calculation of control bits

The first iteration of the loop of our iterative Algorithm 4 calculates the same outer layers and parity-preserving permutation using the same steps as Algorithm 3. The sole difference is that we choose not to decompose the parity-preserving permutation. Instead, we leave it in-place and adapt the calculations for the outer layers. Consider the decomposition map giving P_0 and P_1 in line 18 of Algorithm 3. When passed into the recursive subcalls, we study the cycle minima of their commutators with flip_0. Yet, if we never decomposed them, we could equivalently consider the cycle minima of the commutator of P and flip_1. Similarly, the sorting networks required in the subcalls to apply $\textsc{oSort}$ to P_0 and P_1 can be interlaced, as in Figure 3, to give a comparator network that sorts P. Combining the two cycle minima calculations into one also implies we calculate all of the second layer's control bits before calculating any control bits for deeper layers. Therefore, we can apply the layers represented by these control bits, thus factorising our parity-preserving permutation again. In the same fashion that P_0 and P_1 will become parity preserving permutations, our in-place P, after being factorised, will satisfy $\mathsf{P}[x] \equiv x \mod 4$.

Our iterative algorithm is the natural extension of these observations. Upon completing the k-th iteration, we obtain a factorised permutation in array P satisfying $\mathsf{P}[x] \equiv x \mod 2^k$. Algorithm 4 describes the function $\textsc{cbiterative}$. Definition 5 below gives our formulae for the control bits, which correspond to the algorithm. The theorems and corollaries that follow in Section 4 are our main results which show the correctness of these formulae. Note that Bernstein's formulae and ours agree when $k = 0$ — the distinction is between what happens next.

There are three primary differences between Algorithm 3 and Algorithm 4. Firstly and primarily, instead of proceeding by decomposing our permutation into two smaller permutations, we modify our permutation in-place and change the parameter of the operation we perform upon it. Consequently, the call to $\textsc{FastCycleMin}$ takes its second parameter depending on k: we discuss this in Subsection 4.2, but essentially we need to account for

Algorithm 4 Iteratively calculate the control bits for an in-place Beneš network.

Require: Array P such that $\mathsf{P}[x] = \pi(x)$ for some $\pi \in S_{2^{m+1}}$.

function CBITERATIVE(P)
  Initialise array C of size $(2m+1) \cdot 2^m$ for storing control bits.
  for $k = 0$ to $m-1$ inclusive **do**
    $\mathsf{C}_{\min} \leftarrow$ FASTCYCLEMIN($\mathsf{Perm}\,(P)$, k)
    for $j = 0$ to $2^m - 1$ inclusive **do**
      $\mathsf{F}[j] \leftarrow \mathrm{bit}_k(\mathsf{C}_{\min}[\mathrm{insertbit}_k(0, j)])$
    end for
    $\mathsf{P} \leftarrow$ OSORT($\mathrm{layer}_k(\mathsf{F}), $OSORT($\mathsf{id}_n, \mathsf{P}, k), k$)
    for $j = 0$ to $2^m - 1$ inclusive **do**
      $\mathsf{L}[j] \leftarrow \mathrm{bit}_k(\mathsf{P}[\mathrm{insertbit}_k(0, j)])$
    end for
    $\mathsf{C}[k \cdot 2^m : (k+1) \cdot 2^m : 1] \leftarrow \mathsf{F}$
    $\mathsf{C}[(2m-k) \cdot 2^m : (2m-k+1) \cdot 2^m : 1] \leftarrow \mathsf{L}$
    $\mathsf{P} \leftarrow$ OLAYER($\mathsf{P}, \mathsf{L}, k$)
  end for
  for $j = 0$ to $2^m - 1$ inclusive **do**
    $\mathsf{L}[j] \leftarrow \mathrm{bit}_m(\mathsf{P}[j])$
  end for
  $\mathsf{C}[m \cdot 2^m : (m+1) \cdot 2^m : 1] \leftarrow \mathsf{L}$
  return C
end function

the fact that as our permutations are increasingly structured, we need to do less work in calculating cycle minima. Thirdly, the control bits are in our algorithm calculated and stored in their natural order, removing the need for variables `pos` and `step` or indeed any kind of global array. Writing happens only to adjacent slices of memory.

At the end of the k-th iteration, P then stores $\mathrm{middle}_k(\pi)$ as defined below. Instead of an m-th iteration, however, we can actually read off the final layer of control bits from P. This corresponds to the base case of Bernstein's recursion in which he similarly can just read off a result.

The proofs of these theorems and corollaries are presented in Section 4. The analogy of Corollary 3 for CBRECURSION is exactly that which Bernstein does not prove in his work. We do have a proof of this, but we do not present this proof in this paper. However, it is in our Lean formalisation which extends Bernstein's work. We do not provide an explicit proof that CBITERATIVE($\mathsf{Vec}\,(\pi)$) calculates the same control bits that CBRECURSION($\mathsf{Vec}\,(\pi)\,, \mathsf{C}, 0, 1$) sets C to, but the careful reader can verify for themselves that they do. Note that, for instance, statement *(ii)* of Theorem 1 exactly corresponds via interlacing to Bernstein's property that the first bit of each layer he calculates is 0.

Definition 5. Let k, m be nonnegative integers and $\pi \in S_{2^{m+1}}$ be an arbitrary permutation. We define $\mathsf{F}_k(\pi), \mathsf{L}_k(\pi), \mathrm{first}_k(\pi), \mathrm{last}_k(\pi) \in \{0,1\}^{2^m}$ and $\rho_k(\pi), \mathrm{middle}_k(\pi) \in S_{2^{m+1}}$ as follows:

(i) $\mathsf{F}_k(\pi)[i] := \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])(\mathrm{insertbit}_k(0, i))\right)$.

(ii) $\mathsf{L}_k(\pi)[i] := \mathrm{bit}_k\left(\mathrm{layer}_k\left(\mathsf{F}_k(\pi)\right)\left(\pi\left(\mathrm{insertbit}_k(0, i)\right)\right)\right)$.

(iii) $\rho_k(\pi) := \mathrm{layer}_k\left(\mathsf{F}_k(\pi)\right) \cdot \pi \cdot \mathrm{layer}_k\left(\mathsf{L}_k(\pi)\right)$.

(iv) $\mathrm{middle}_0(\pi) := \pi$ and $\mathrm{middle}_{k+1}(\pi) := \rho_k(\mathrm{middle}_k(\pi))$.

(v) $\mathrm{first}_k(\pi) := \mathsf{F}_k(\mathrm{middle}_k(\pi))$.

(vi) $\mathrm{last}_k(\pi) := \mathsf{L}_k(\mathrm{middle}_k(\pi))$.

Theorem 1. *Let k, m be nonnegative integers and $\pi \in S_{2^{m+1}}$ be an arbitrary permutation. Then $\pi = \mathrm{layer}_k(\mathsf{F}_k(\pi)) \cdot \rho_k(\pi) \cdot \mathrm{layer}_k(\mathsf{L}_k(\pi))$. Moreover if, for all x, $\pi(x) \equiv x \mod 2^k$, then:*

(i) *The first* 2^k *bits in* F_k *are* 0.

(ii) $\rho_k(\pi)(x) \equiv x \mod 2^{k+1}$.

Theorem 2. *Let* k, m *be nonnegative integers and* $\pi \in S_{2^{m+1}}$ *be an arbitrary permutation. Then:*

(i) $\text{middle}_k(\pi) = \text{layer}_k(\text{first}_k(\pi)) \cdot \text{middle}_{k+1}(\pi) \cdot \text{layer}_k(\text{last}_k(\pi))$.

(ii) *The first* 2^k *bits of* $\text{first}_k(\pi)$ *are zero. In particular, when* $m \leq k$*, every entry of* $\text{first}_k(\pi)$ *is zero.*

(iii) $\text{middle}_k(\pi)(x) \equiv x \mod 2^k$. *In particular,* $\text{middle}_{m+1} = \mathbb{I}_{2^{m+1}}$.

Corollary 1.

$$\pi = \text{layer}_0(\text{first}_0(\pi)) \cdots \text{layer}_{m-1}(\text{first}_{m-1}(\pi)) \cdot \text{layer}_m(\text{last}_m(\pi)) \cdots \text{layer}_0(\text{last}_0(\pi)).$$

Corollary 2.

$$\textsc{cbiterative}(\mathsf{Vec}\,(\pi)) = \text{first}_0(\pi) \| \cdots \| \text{first}_{m-2}(\pi) \| \text{last}_{m-1}(\pi) \| \cdots \| \text{last}_0(\pi).$$

Corollary 3.

$$\mathsf{A} \triangleleft \pi = \textsc{cbapply}(\mathsf{A}, \textsc{cbiterative}(\pi)).$$

4 Proofs and verification

In this section, we generalise Bernstein's definitions and proofs in [Ber20] by generalising his results and formulae. It is only implicit in Bernstein's work that the control bits that result from this inductive process can be correctly re-interpreted by Algorithm 2.

Some care is needed in the proofs, and this has motivated our careful setting up of the relevant definitions in the previous section. Certain inferences which are natural and obvious in Bernstein's work — where he considers only the $k = 0$ case — are still analogously true in our generalisation, but are less clearly true to the casual observer. To support our work, alongside these pen and paper proofs we have produced a formalisation of them in the Lean theorem prover, as Bernstein formalised his own proofs. We present below our written proofs, generalised from Bernstein's results.

4.1 Results on the commutator

Let m be a nonnegative integer, and $\pi \in S_{2^{m+1}}$ an arbitrary permutation. We consider the commutator $[\pi, \text{flip}_k] = \pi \cdot \text{flip}_k \cdot \pi^{-1} \cdot \text{flip}_k$. (Recall that flip_k is an involution and so is its own inverse). Note that $[\pi, \text{flip}_k] \in S_{2^{m+1}}$, as it is the product of elements of $S_{2^{m+1}}$. Further, if π is such that, for all x, $\pi(x) \equiv x \mod 2^k$, then we also have $[\pi, \text{flip}_k](x) \equiv x \mod 2^k$ for all x. As flip_k is not only an involution but also one without any fixed points, we can derive some useful lemmas using elementary properties of the commutator (mirroring Bernstein's approach: all of our proofs are generalisations of his).

Lemma 2. *Suppose* $\pi \in S_{2^{m+1}}$ *for nonnegative* m*. For all integers* t*, we have*

$$\text{flip}_k \cdot [\pi, \text{flip}_k]^t = [\pi, \text{flip}_k]^{-t} \cdot \text{flip}_k.$$

In other words, applying the commutator t *times and then* flip_k*, is equivalent to applying* flip_k *and then applying the inverse of the commutator* t *times.*

Proof. Observe that without loss of generality we may take $t \geq 0$ (if it is negative we can simply multiply both sides of the equation by flip_k on the left and right and move to a positive case). As such, we can induct on t. Clearly when $t = 0$ the statement holds. Suppose in the inductive case that t is of the form $s+1$, where s is a nonnegative integer. Then we have:

$$\begin{aligned}
\mathrm{flip}_k \cdot [\pi, \mathrm{flip}_k]^{s+1} &= \mathrm{flip}_k \cdot [\pi, \mathrm{flip}_k]^s \cdot [\pi, \mathrm{flip}_k] \cdot \mathrm{flip}_k \cdot \mathrm{flip}_k \\
&= [\pi, \mathrm{flip}_k]^{-s} \cdot \mathrm{flip}_k \cdot [\pi, \mathrm{flip}_k] \cdot \mathrm{flip}_k \cdot \mathrm{flip}_k \\
&= [\pi, \mathrm{flip}_k]^{-s} \cdot \mathrm{flip}_k \cdot \pi \cdot \mathrm{flip}_k \cdot \pi^{-1} \cdot \mathrm{flip}_k \cdot \mathrm{flip}_k \cdot \mathrm{flip}_k \\
&= [\pi, \mathrm{flip}_k]^{-s} \cdot \mathrm{flip}_k \cdot \pi \cdot \mathrm{flip}_k \cdot \pi^{-1} \cdot \mathrm{flip}_k \\
&= [\pi, \mathrm{flip}_k]^{-s} \cdot [\pi, \mathrm{flip}_k]^{-1} \cdot \mathrm{flip}_k \\
&= [\pi, \mathrm{flip}_k]^{-(s+1)} \cdot \mathrm{flip}_k.
\end{aligned}$$

□

Lemma 3. *Suppose $\pi \in S_{2^{m+1}}$ for nonnegative m. Let $i, j \in \mathbb{Z}$. Then for all x,*

$$[\pi, \mathrm{flip}_k]^i(x) \neq \mathrm{flip}_k([\pi, \mathrm{flip}_k]^j(x)) = [\pi, \mathrm{flip}_k]^{-j}(\mathrm{flip}_k(x)).$$

In other words, for any nonnegative integer $x < 2^m$, the cycles of x and $\mathrm{flip}_k(x)$ under the commutator are disjoint.

Proof. The equality in our statement holds by Lemma 2, so we need only prove the inequality. Fix x. Suppose for a contradiction there exists some i, j which in fact satisfy $[\pi, \mathrm{flip}_k]^i(x) = \mathrm{flip}_k \cdot [\pi, \mathrm{flip}_k]^j(x)$. We can multiply both sides by flip_k to reverse the roles of i and j, so without loss of generality we may take $i \geq j$. We choose such a i, j so that $i - j$ is minimal. If $i = j$ then we would immediately have x as a fixed point of flip_k, which is impossible. So $i \geq j + 1$ Suppose $i = j + 1$. Then we would have:

$$\begin{aligned}
\pi^{-1}([\pi, \mathrm{flip}_k]^{j+1}(x)) &= \pi^{-1}([\pi, \mathrm{flip}_k]([\pi, \mathrm{flip}_k]^j(x))) \\
&= \mathrm{flip}_k(\pi^{-1}(\mathrm{flip}_k([\pi, \mathrm{flip}_k]^j(x)))) \\
&= \mathrm{flip}_k(\pi^{-1}([\pi, \mathrm{flip}_k]^{j+1}(x))).
\end{aligned}$$

But, again, flip_k has no fixed points, so this cannot be so. We must have $i \geq j + 2$. Then:

$$\begin{aligned}
[\pi, \mathrm{flip}_k]^{i-1}(x) &= [\pi, \mathrm{flip}_k]^{-1}([\pi, \mathrm{flip}_k]^i(x)) \\
&= [\pi, \mathrm{flip}_k]^{-1}(\mathrm{flip}_k([\pi, \mathrm{flip}_k]^j(x))) \\
&= \mathrm{flip}_k([\pi, \mathrm{flip}_k]^{j+1}(x)).
\end{aligned}$$

So $(i - 1, j + 1)$ are also such a pair. We have $i \geq j + 2$, so $i - 1 \geq j + 1$. However, $(i - 1) - (j + 1) = (i - j) - 2$. This contradicts the minimality of $i - j$. □

Lemma 4. *Suppose $\pi \in S_{2^{m+1}}$ for nonnegative m and $x < 2^{m+1}$. Suppose $S \subseteq \mathbb{N}_{<2^{m+1}}$ satisfies: (1) For all $y \in S$, $\mathrm{flip}_k(y) \in S$, and (2) For all $i \in \mathbb{Z}$, $[\pi, \mathrm{flip}_k]^i(x) \in S$. Then, the period of x under the permutation $[\pi, \mathrm{flip}_k]$ is at most $\frac{|S|}{2}$.*

Proof. Fix x. We have x, $\mathrm{flip}_k(x) \in S$. Let $S_0 = \{[\pi, \mathrm{flip}_k]^i(x) \mid i \in \mathbb{Z}\}$ and $S_1 = \{[\pi, \mathrm{flip}_k]^i(\mathrm{flip}_k(x)) \mid i \in \mathbb{Z}\}$. We have $S_0 \subseteq S$ by assumption. We also have $S_1 \subseteq S$: $[\pi, \mathrm{flip}_k]^i(\mathrm{flip}_k(x)) = \mathrm{flip}_k([\pi, \mathrm{flip}_k]^{-i}(x))$ by Lemma 2, and as S is invariant under flip_k and contains S_0, it follows that it contains S_1. As such $|S_0 \cup S_1| \leq |S|$. We can also use this observation to note that flip_k is surjective and maps S_0 into S_1. As flip_k is globally injective, it will also be injective on this restriction. Thus, it defines a bijection between them and so $|S_0| = |S_1|$. Moreover, by Lemma 3, the cycles of x and $\mathrm{flip}_k(x)$ are disjoint, i.e. S_0 and S_1 are disjoint. Thus, $|S_0 \cup S_1| = |S_0| + |S_1|$. It follows that $2 \cdot |S_0| = |S_0| + |S_1| = |S_0 \cup S_1| \leq |S|$, and hence the period of x, $|S_0|$, is at most $\frac{|S|}{2}$. □

Lemma 5. *Suppose $\pi \in S_{2^{m+1}}$ for nonnegative m. Suppose $k \leq m$ is such that for all x, we have $\pi(x) \equiv x \mod 2^k$. Then for any x, the period of x under the permutation $[\pi, \text{flip}_k]$ is at most 2^{m-k}.*

Proof. Clearly if $2^{m+1} \leq x$ or $x < 0$ then the period of x is 1, which is certainly at most 2^{m-k}. Hence, without loss of generality, assume $x < 2^{m+1}$. Let $S_x = \left\{y \mid 0 \leq y < 2^{m+1} \wedge y \equiv x \bmod 2^k\right\}$. In other words, S_x is the set of nonnegative integers which can be represented in $m+1$ bits and which share their bottom k bits with x.

Now, if $y \in S_x$, it is easy to see that $\text{flip}_k(y) \in S_x$. Moreover, for all i, $[\pi, \text{flip}_k]^i(x) \in S_x$: $[\pi, \text{flip}_k]$ is invariant on the bottom k bits, and is in $S_{2^{m+1}}$, so it preserves the defining property of S_x. Observe that $|S_x| = 2^{(m+1)-k}$, because there are $(m+1)-k$ "free" bits. It follows from Lemma 4 that the period of x is at most 2^{m-k} as required. □

Lemma 6. *Let $x, y \in \mathbb{Z}$. Suppose $x \equiv y \mod 2^k$, $x \leq \text{flip}_k(y)$, and $y \leq \text{flip}_k(x)$. Then either $y = x$ or $y = \text{flip}_k(x)$.*

Proof. As $x \equiv y \mod 2^k$, we have $y = x + 2^k \cdot a$ for some integer a. Without loss of generality, suppose $x \leq y$. Then $0 \leq a$, and $x \leq \text{flip}_k(x)$, so in fact $\text{flip}_k(x) = x + 2^k$. As such, $x + 2^k \cdot a \leq x + 2^k$, so $a \leq 1$. If $a = 0$, we have $y = x$. If $a = 1$, $y = \text{flip}_k(x)$. □

Lemma 7. *Suppose $\pi \in S_{2^{m+1}}$ for nonnegative m. Suppose $k \leq m$ is such that for all x, $\pi(x) \equiv x \mod 2^k$. Then it follows that for all $x \leq 2^{m+1}$,*

$$\text{cyclemin}([\pi, \text{flip}_k])(\text{flip}_k(x)) = \text{flip}_k(\text{cyclemin}([\pi, \text{flip}_k])(x)).$$

Proof. Let m, π, x be as in the theorem statement. Let j, s be integers such that:

$$\text{cyclemin}([\pi, \text{flip}_k])(x) = [\pi, \text{flip}_k]^j(x).$$
$$\text{cyclemin}([\pi, \text{flip}_k])(\text{flip}_k(x)) = [\pi, \text{flip}_k]^s(\text{flip}_k(x)).$$

Using minimality, Lemma 2, Lemma 3, and the congruence property of π, we have:

$$\begin{aligned}
[\pi, \text{flip}_k]^s(\text{flip}_k(x)) &\leq [\pi, \text{flip}_k]^{-j}(\text{flip}_k(x)) \\
&= \text{flip}_k([\pi, \text{flip}_k]^j(x)). \\
[\pi, \text{flip}_k]^j(x) &\leq [\pi, \text{flip}_k]^{-s}(x) \\
&= \text{flip}_k([\pi, \text{flip}_k]^s(\text{flip}_k(x))). \\
[\pi, \text{flip}_k]^j(x) &\equiv [\pi, \text{flip}_k]^s(\text{flip}_k(x)) \mod 2^k. \\
[\pi, \text{flip}_k]^j(x) &\neq [\pi, \text{flip}_k]^s(\text{flip}_k(x)).
\end{aligned}$$

The result follows by Lemma 6. □

4.2 The cycle minima calculation

A naive calculation of the cycle minima is costly. However as Bernstein notes from Nassimi-Sahni, the operation can be performed in parallel. Note this corollary of Lemma 5.

Corollary 4. *Let $\pi \in S_{2^{m+1}}$ be such that $\pi(x) \equiv x \mod 2^k$ for all x. Then:*

$$\text{cyclemin}([\pi, \text{flip}_k]) = \min\{x, [\pi, \text{flip}_k](x), [\pi, \text{flip}_k]^2(x), \ldots, [\pi, \text{flip}_k]^{2^{m-k}-1}(x)\}.$$

Proof. By Lemma 5, the period of any x under $[\pi, \text{flip}_k]$ is at most 2^{m-k}. Thus the minimum defined by the right-hand side is indeed the minimum over the whole cycle. □

Theorem 3. *Let $\pi \in S_{2^{m+1}}$ be such that $\pi(x) \equiv x \mod 2^k$ for all x. Then Algorithm 5,* FastCycleMin, *has* $\text{cyclemin}([\pi, \text{flip}_k])(x) = \text{FastCycleMin}(\pi, k)[x]$ *for all $x < 2^{m+1}$.*

Algorithm 5 Calculate cycle minima of $[\pi, \mathrm{flip}_k]$ where $\pi(x) \equiv x \mod 2^k$ for all x.

function FastCycleMin(π, k)
  $\mathsf{C}_{\min} \leftarrow \mathsf{id}_n$ ▷ Initialise cyclemin array.
  $\mathsf{P} \leftarrow \mathsf{Vec}(\pi \cdot \mathrm{flip}_k)$ ▷ Implemented as $\mathsf{P}[x] = \pi(x \oplus 2^k)$.
  $\mathsf{Q} \leftarrow \mathsf{Vec}(\mathrm{flip}_k \cdot \pi)$ ▷ Implemented as $\mathsf{Q}[x] = \pi(x) \oplus 2^k$.
  for i in $0, 1, \ldots, m-k-1$ **do**
    $\mathsf{P},\ \mathsf{Q} \leftarrow \textsc{oSort}(\mathsf{P}, \mathsf{Q}, k),\ \textsc{oSort}(\mathsf{Q}, \mathsf{P}, k)$ ▷ P is $[\pi, \mathrm{flip}_k]^{2^i}$, Q is $[\pi, \mathrm{flip}_k]^{-2^i}$
    $\mathsf{CP} \leftarrow \textsc{oSort}(\mathsf{C}_{\min}, \mathsf{Q}, k)$ ▷ $\mathsf{CP}[x] = \mathsf{C}_{\min}\left[[\pi, \mathrm{flip}_k]^{2^i}(x)\right]$
    $\mathsf{C}_{\min} \leftarrow \min\left(\mathsf{C}, \mathsf{CP}\right)$
  end for return $\mathsf{C}_{\min}$
end function

Proof. For every use of oSort, we will be dealing with a permutation that has the required congruence property (because π does), and so we may indeed conclude that the interlaced sorting network are correctly calculating $\mathsf{A} \triangleleft \rho$ for various A and ρ. We argue inductively via loop invariants that $\mathsf{C}_{\min}[x] = \min\{x, [\pi, \mathrm{flip}_k](x), [\pi, \mathrm{flip}_k]^2(x), \ldots, [\pi, \mathrm{flip}_k]^{2^i-1}(x)\}$ before the i-th iteration. Before the loop begins, $\mathsf{C}_{\min} \leftarrow \mathsf{id}_n$ (as required), $\mathsf{Perm}(\mathsf{P}) \leftarrow \pi \cdot \mathrm{flip}_k$, and $\mathsf{Perm}(\mathsf{Q}) \leftarrow \mathrm{flip}_k \cdot \pi$. On each iteration, we set $\mathsf{Perm}(\mathsf{P}) \leftarrow \mathsf{Perm}(\mathsf{P}) \cdot \mathsf{Perm}(\mathsf{Q})^{-1}$ and $\mathsf{Perm}(\mathsf{Q}) \leftarrow \mathsf{Perm}(\mathsf{Q}) \cdot \mathsf{Perm}(\mathsf{P})^{-1}$.

It follows that when $i = 0$, $\mathsf{Perm}(\mathsf{P})$ is set to $[\pi, \mathrm{flip}_k]$ and $\mathsf{Perm}(\mathsf{Q})$ is set to $[\pi, \mathrm{flip}_k]^{-1}$. Thereafter on the i-th iteration, $\mathsf{Perm}(\mathsf{P})$ is set to $[\pi, \mathrm{flip}_k]^{2^i}$ and $\mathsf{Perm}(\mathsf{Q})$ is set to $[\pi, \mathrm{flip}_k]^{-2^i}$. It then follows that CP is set such that $\mathsf{CP}[x] = \mathsf{C}_{\min}[[\pi, \mathrm{flip}_k]^{2^i}(x)]$, using the values of $\mathsf{C}_{\min}$ before the loop, and then $\mathsf{C}_{\min}$ is set to the minimum of its current values and the values in CP. Using the inductive hypothesis, this will be exactly the minimum of $\min\{x, \ldots, [\pi, \mathrm{flip}_k]^{2^i-1}(x)\}$ and $\min\{[\pi, \mathrm{flip}_k]^{2^i}(x), \ldots, [\pi, \mathrm{flip}_k]^{2^{i+1}-1}(x)\}$. This minimum is indeed equal to the cycle minima we require by Corollary 4. □

4.3 Proving our theorems and corollaries

Recall that for all i, $\mathrm{insertbit}_k(\mathrm{bit}_k(i), \mathrm{removebit}_k(i)) = i$. We use this, and the formalism we have set up, to prove two crucial lemmas without too much trouble.

Lemma 8. *Suppose $\pi \in S_{2^{m+1}}$, and for all x, $\pi(x) \equiv x \mod 2^k$. Then*

$$\mathrm{bit}_k\left(\mathrm{layer}_k(\mathsf{F}_k(\pi))(i)\right) = \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])(i)\right).$$

Proof.

$$\begin{aligned}
\mathrm{bit}_k\left(\mathrm{layer}_k(\mathsf{F}_k(\pi))(i)\right) &= \mathrm{bit}_k\left(\mathrm{cflip}_k(\mathsf{F}_k(\pi)[\mathrm{removebit}_k(i)], i)\right) \\
&= \mathrm{bit}_k(i) \oplus \mathsf{F}_k(\pi)[\mathrm{removebit}_k(i)] \\
&= \mathrm{bit}_k(i) \oplus \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])[\mathrm{insertbit}_k(0, \mathrm{removebit}_k(i))]\right).
\end{aligned}$$

If $\mathrm{bit}_k(i) = 0$, we are done. If $\mathrm{bit}_k(i) = 1$, then we use Lemma 7:

$$\begin{aligned}
&= 1 \oplus \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])[\mathrm{flip}_k(i)]\right) \\
&= 1 \oplus \mathrm{bit}_k\left(\mathrm{flip}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])[i]\right)\right) \\
&= 1 \oplus 1 \oplus \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])[i]\right).
\end{aligned}$$

□

Lemma 9. *Suppose $\pi \in S_{2^{m+1}}$, and for all x, $\pi(x) \equiv x \mod 2^k$. Then*

$$\mathrm{bit}_k\left(\mathrm{layer}_k(\mathsf{L}_k(\pi))(i)\right) = \mathrm{bit}_k\left(\left(\mathrm{layer}_k(\mathsf{F}_k(\pi))\right)(\pi(i))\right).$$

Proof.

$$\begin{aligned}\mathrm{bit}_k\left(\mathrm{layer}_k(\mathsf{L}_k(\pi))(i)\right) &= \mathrm{bit}_k\left(\mathrm{cflip}_k(\mathsf{L}_k(\pi)[\mathrm{removebit}_k(i)], i)\right)\\ &= \mathrm{bit}_k(i) \oplus \mathsf{L}_k(\pi)[\mathrm{removebit}_k(i)]\\ &= \mathrm{bit}_k(i) \oplus \mathrm{bit}_k\left(\mathrm{layer}_k\left(\mathsf{F}_k(\pi)\right)\left(\pi\left(\mathrm{insertbit}_k(0, \mathrm{removebit}_k(i))\right)\right)\right).\end{aligned}$$

If $\mathrm{bit}_k(i) = 0$, we are done. If $\mathrm{bit}_k(i) = 1$, then we use Lemma 8, Lemma 7, the fact that $\mathrm{cyclemin}(\rho)(x) = \mathrm{cyclemin}(\rho)(\rho(x))$, and the the relation $[\pi, \mathrm{flip}_k] \cdot \mathrm{flip}_k \cdot \pi \cdot \mathrm{flip}_k = \pi$.

$$\begin{aligned}&= 1 \oplus \mathrm{bit}_k\left(\mathrm{layer}_k\left(\mathsf{F}_k(\pi)\right)\left(\pi\left(\mathrm{flip}_k(i)\right)\right)\right)\\ &= 1 \oplus \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])\left(\pi\left(\mathrm{flip}_k(i)\right)\right)\right)\\ &= \mathrm{bit}_k\left(\mathrm{flip}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])\left(\pi\left(\mathrm{flip}_k(i)\right)\right)\right)\right)\\ &= \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])\left(\mathrm{flip}_k\left(\pi\left(\mathrm{flip}_k(i)\right)\right)\right)\right)\\ &= \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])\left([\pi, \mathrm{flip}_k] \cdot \mathrm{flip}_k \cdot \pi \cdot \mathrm{flip}_k\right)(i)\right)\\ &= \mathrm{bit}_k\left(\mathrm{cyclemin}([\pi, \mathrm{flip}_k])\left(\pi\left(i\right)\right)\right)\\ &= \mathrm{bit}_k\left(\mathrm{layer}_k\left(\mathsf{F}_k(\pi)\right)\left(\pi\left(i\right)\right)\right).\end{aligned}$$

□

We are now able to prove our central claims.

Proof of Theorem 1. As $\mathrm{layer}_k(\mathsf{F}_k(\pi))$ and $\mathrm{layer}_k(\mathsf{L}_k(\pi))$ are both involutions, it immediately follows from the definition of ρ_k that indeed $\pi = \mathrm{layer}_k(\mathsf{F}_k(\pi)) \cdot \rho_k(\pi) \cdot \mathrm{layer}_k(\mathsf{L}_k(\pi))$. Now let us also assume that for all x, $\pi(x) \equiv x \mod 2^k$.

Suppose $i < 2^k$. $\mathrm{insertbit}_k(0, i) = i$ because inserting a 0 at the k-th bit doesn't actually change the value. $[\pi, \mathrm{flip}_k]$ can only send i to numbers which are congruent to it modulo 2^k, and i is the smallest such number, so the minimum of the cycle of i under $[\pi, \mathrm{flip}_k]$ is also i. The k-th bit of i is 0. It follows that $\mathsf{F}_k(\pi)[i] = 0$. This proves the first claim.

We now want to show that $\rho_k(\pi)(x) \equiv x \mod 2^{k+1}$ for all x. Simply by definition of ρ_k, and the congruence property of π, this congruence equation is true modulo 2^k, i.e. the bits below the k-th bit of both sides match. Hence it is sufficient to verify that $\mathrm{bit}_k\left(\rho_k(\pi)(x)\right) = \mathrm{bit}_k(x)$. We can see this by using Lemma 9.

$$\begin{aligned}\mathrm{bit}_k\left(\rho_k(\pi)(x)\right) &= \mathrm{bit}_k\left(\mathrm{layer}_k\left(\mathsf{F}_k(\pi)\right)\left(\pi\left(\mathrm{layer}_k(\mathsf{L}_k(\pi))\left(x\right)\right)\right)\right)\\ &= \mathrm{bit}_k\left(\mathrm{layer}_k\left(\mathsf{L}_k(\pi)\right)\left(\mathrm{layer}_k(\mathsf{L}_k(\pi))\left(x\right)\right)\right)\\ &= \mathrm{bit}_k\left(\mathrm{layer}_k(\mathsf{L}_k(\pi)) \cdot \mathrm{layer}_k(\mathsf{L}_k(\pi))\left(x\right)\right)\\ &= \mathrm{bit}_k(x).\end{aligned}$$

□

Proof of Theorem 2. This is a straightforward inductive application of Theorem 1. That $\mathrm{middle}_k(\pi) = \mathrm{layer}_k(\mathrm{first}_k(\pi)) \cdot \mathrm{middle}_{k+1}(\pi) \cdot \mathrm{layer}_k(\mathrm{last}_k(\pi))$ is just the first claim of Theorem 1. It is also certainly true that $\mathrm{middle}_0(\pi)(x) \equiv x \mod 2^0$ (everything is equal modulo 1)! Proceeding inductively, assuming $\mathrm{middle}_k(\pi)(x) \equiv x \mod 2^k$, we can apply the conditional conclusions of Theorem 1: because we have $\mathrm{first}_k(\pi) = \mathsf{F}_k(\mathrm{middle}_k(\pi))$, the first 2^k bits of $\mathrm{first}_k(\pi)$ are zero, and because $\mathrm{middle}_{k+1}(\pi) = \rho_k(\mathrm{middle}_k(\pi))$, we have $\mathrm{middle}_{k+1}(\pi)(x) \equiv x \mod 2^{k+1}$. The particular cases we note in the theorem are thereafter clearly true. □

Proof of Corollary 1. This corollary is true simply by using the facts that $\mathrm{middle}_{m+1} = \mathbb{I}_{2^{m+1}}$, $\mathrm{layer}_m(\mathrm{first}_m(\pi)) = \mathbb{I}_{2^{m+1}}$, and $\mathrm{middle}_k(\pi) = \mathrm{layer}_k(\mathrm{first}_k(\pi)) \cdot \mathrm{middle}_{k+1}(\pi) \cdot \mathrm{layer}_k(\mathrm{last}_k(\pi))$, and expanding inductively from $\pi = \mathrm{middle}_0(\pi)$. □

Proof of Corollary 2. We have Theorem 3, which justifies that we really are calculating the right cycle minima on each iteration of the loop. Similarly, the interlaced sorts will

happen correctly because of the congruence properties we have shown. Our layers in each iteration are set more or less directly from the above formulae, so that on the i-th loop they are set to $\text{first}_i(\pi)$ and $\text{last}_i(\pi)$, and the value of the permutation each loop is set such that before the i-th loop, $\mathsf{Perm}(\mathsf{P}) = \text{middle}_k(\pi)$, and after the i-th loop it is $\mathsf{Perm}(\mathsf{P}) = \text{middle}_{k+1}(\pi)$. We do not bother performing the full loop when $i = k$, as we know that $\text{middle}_{m+1}(\pi)$ and $\text{first}_m(\pi)$ are trivial: moreover note that $\text{insertbit}_m(0, i)$ for $i < 2^m$, and so $\text{last}_m(\pi)[i] = \text{bit}_k(\text{middle}_m(\pi)(i))$, justifying the final layer that we set. □

Proof of Corollary 3. This proof is essentially straightforward: the definition of Algorithm 2 is one of applying the control bits in a long sequence in 2^m-sized slices, which is exactly what we have concatenated, and Algorithm 1 precisely applies the sequence of conditional transpositions that make up each application of layer_k. As such, it is essentially immediate from Corollary 1 and Corollary 2 that the claim is true. □

4.4 Formal verification in Lean

Almost all of the theorems, lemmas and corollaries above have been formalised and proven in the Lean theorem prover, and are presented in the repository accompanying this paper. This generalises the HOL Light verification performed in [Ber20] from the $k = 0$ case to all k. The principal verification gap is that our Lean implementation of Algorithm 5 applies permutations directly rather than via oSort, as it does not aim to be constant-time. Our formal proof of Theorem 3 therefore only partially verifies correctness of FastCycleMin, as it does not make use of oSort. Corollary 2 and Corollary 3 inherit this caveat. The repository does contain a separate proof that the reordering of comparators in our interlaced sorting networks is valid, and we believe the formal gaps are therefore covered by our written account here.

The repository also contains a proof, not presented in this paper, that the control bits produced by the recursive method of [Ber20], when applied layer-by-layer via Algorithm 2, recover the original permutation. This closes the gap noted in the introduction. Our iterative approach renders this result unnecessary, but it may be of independent interest.

5 Implementation and evaluation

Overview. We provide two implementations of Algorithm 4 in the C programming language. The implementation cbiterative is designed for portability, whereas the implementation cbiteravx demonstrates the vectorisability of the algorithm using instructions from Intel's AVX2 feature. We evaluate our implementations and compare their performance against cbrecursion: the implementation of Algorithm 3 in libmceliece version 20250507 [Cho25]. All the implementations take as input an array of 2^{m+1} 16-bit integers representing a permutation, and write, to a pre-allocated buffer, the $(2m+1) \cdot 2^m$ control bits for the in-place Beneš network realising said permutation. Both our implementations follow constant-time coding best practices: we do not (1) branch on secret values, (2) make memory accesses dependent on secret values, or (3) pass secret values as input to variable time instructions. In Classic McEliece, the permutation is secret and its size is public. In the portable implementation, we utilise the cryptoint library [Ber25a] to reduce the risk of the compiler optimising away our precautions. In cbiteravx, we utilise only instructions in the Intel Data-Operand Independent Timing (DOIT) list [Cor25]. We verify that our implementations satisfy constant-time best practices (1) and (2) using Valgrind's memcheck tool [NS07].

Intel AVX2. Intel's AVX2 provides 16 256-bit ymm registers for integer SIMD instructions. These registers offer divisions into lanes of size 128, 64, 32, 16, and 8 bits. A *swizzle* instruction is an intra-register permutation. Data movement across the two 128-bit lanes is both limited and relatively expensive. We identify instructions by their assembly mnemonic. We loosely term an algorithm SIMD-friendly if its scalar implementation is similar to its SIMD implementation. Algorithm 4 is SIMD-friendly; to utilise the ymm registers in the implementation of cbiteravx, we need only unroll the loops in the cbiterative implementation. On the other hand, the deepest subcalls (which are the majority) of Algorithm 3 act on less data than a single ymm register will hold: this algorithm is not SIMD-friendly.

5.1 Implementation strategy

Permutations. The permutation in Classic McEliece acts on n elements, where n is either 2^{12} or 2^{13} depending on parameter set, and is initially represented by an array of 16-bit integers. The challenge of the implementation is to apply OSORT as little as possible because sorting large arrays is expensive. Conversely, OFLIP can be implemented with 3 XOR instructions and a mask or instructions such as `cmovg`. To implement OSORT, we represent the tuple array as an array of 32-bit integers formed by concatenating the two 16-bit halves. Sorting these 32-bit integers is equivalent to sorting the tuples with the upper half as the key. Recall that to apply π to A on the right requires *two* sorts: one sort to obtain π^{-1}, and one to obtain $\mathsf{A}[\pi]$. The implementations of cbiterative and cbrecursion repeatedly utilise this primitive to calculate the cycle minima and to left-apply $\text{layer}_k(\mathsf{F}_k(\pi))$ to π. The C implementations require storage space for two temporary arrays of n 32-bit integers for these cycle calculations. We call these A and B.

Storing the control bits. The control bits are stored as a bit array. Towards the base cases of cbrecursion, less than 8 bits are set. To set individual bits in a byte, Bernstein bit-shifts and XORs them into a zero-initialised array. The iterative algorithm can avoid these smaller cases and place a byte of control bits at a time by unrolling the loop. We found this to have an negligible performance effect in the scalar implementations, aside from avoiding the call to memset to zero-initialise the array. The cbiteravx implementation, however, sets 32 control bits per memory store, leading to noticeable performance gains.

Vectorisation. The primary challenge in the cbiteravx implementation is extracting the control bits. After having calculated the cycle minima for iteration k in Algorithm 4, the i-th control bit (for the current layer) exists in the k-th bit of the $\text{insertbit}_k(0, i)$-th element of B. We must extract these control bits out of our ymm registers and into their condensed representation as an array of bits. This can be done with the movemask instructions `vpmovmsk{b, ps, pd}`. However, only `vpmovmskb`, which extracts the most significant bit of each byte in a ymm register into a 32-bit integer, is on the DOIT list. To extract 32 control bits per iteration, each loop must load 64 elements from B. Therefore, cbiteravx only takes as input permutations with more than 64 elements. Indexing in bits, the i-th control bit is in bit $k + 32 \cdot \text{insertbit}_k(0, i)$ of B and must be placed in bit $8i + 7$ of a ymm register in order to obtain the control bits for the in-place Beneš network. For values of $k \leq 2$, this rearrangement requires special cases as the $\text{insertbit}_k(0, i)$ addend changes their positions by awkward multiples of 32 bits.

In the same loop that we extract the control bits, we also apply them to the identity to obtain $\text{layer}_k(\mathsf{F})$ for the calculation of $\text{layer}_k(\mathsf{F}) \cdot \pi$. After having calculated the cycle minima for iteration k, B is as-above and the i-th element of array A has the value i in its most significant half and the value $\pi^{-1}(i)$ in its least significant half. We must prepare A to contain π^{-1} in its most significant half and F in its least significant half for sorting.

To swap halves, we use the `vpshufb` instruction. Applying the control bits is simple. We first copy each control bit at index $\text{insertbit}_k(0, i)$ to their counterpart at $\text{insertbit}_k(1, i)$. As the control bit is in the k-th bit of each 32-bit integer, we can apply it to the identity in A with a single XOR. Note that a ymm register holds 8 32-bit integers, thus for $k \leq 3$, both $\text{insertbit}_k(0, i)$ and $\text{insertbit}_k(1, i)$ are in the same register. This requires us to use specific swizzle instructions for the cases $k \leq 2$, again complicating the implementation and requiring a large amount of unrolling.

5.2 Experiments and Evaluation

We evaluate our implementations and compare their performance against the implementation of cbrecursion in libmceliece version 20250507 [Cho25]. These results are summarised in Table 1 and Table 2. We then evaluate the impact of our most performant implementation on the Classic McEliece key generation implementations available on the SUPERCOP database [BL24]. These results are summarised in Table 3.

As the performance of all three control bit algorithm implementations is strongly determined by the underlying sorting network, we benchmark the three control bit implementations across six different sorting network implementations. The first is portable4, a scalar implementation of Batcher's odd-even sorting network from the djbsort library [Ber24]. The second is bitonic, an AVX2 implementation of Batcher's Bitonic sorting network. The third is djbsort, which is essentially Batcher's Bitonic sorting network with its wires permuted for SIMD-friendliness. The AVX2-enabled implementations utilise Bitonic, despite it requiring more comparisons than Odd-Even Merge, because it is easy to capture its incredibly regular structure with SIMD instructions [Ebb15]. Each has an interlaced variant, named iportable4, ibitonic, and ipermsort respectively.

Experimental environment. All benchmarks are performed on a single Efficiency Core (E-core) of an Intel (R) Core (TM) Ultra 7 165U CPU (Meteor Lake) running Ubuntu 24.04.3 LTS. We use clang version 18.1.3 and gcc version 13.3.0. We utilise the same compiler options as the libmceliece implementation. Table 1 presents the benchmarks for the portable implementations compiled with flags `-O2`, `-fwrapv`, and `-fPIC`. Table 2 presents the benchmarks for the AVX-enabled implementations compiled with extra flags for Intel intrinsics such as `-mavx2` and with optimisations such as `-mtune=meteorlake`. The device has an L1D cache of size 352KiB, L1I cache of size 640KiB, an L2 cache of size 6MiB and an L3 cache of size 12MiB.

We record the median clock cycles required to calculate control bits for an array of 2^{13} 16-bit integers (the most common permutation size in the Classic McEliece parameter sets) using the read performance-monitoring counters (`rdpmc`) instruction. In order to reduce timing variations, we follow the guidance recommended on the Supercop website [BL24], i.e., disabling Turbo Boost and underclocking. In order to avoid Hyperthreading, we benchmark only on the E-cores, which do not offer Hyperthreading. However, extra care must be taken when benchmarking on the E-cores because they operate at a lower frequency than the 'nominal' frequency reported by the Time-Stamp Counter. Therefore, benchmarking our software with the `rdtsc` instruction results in an over-estimate of the cycle counts. To provide accurate cycle counts, we benchmark our software using the `rdpmc` instruction via the libcpucycles API [Ber26].

We calculate the median from 63 measurements. For $n = 63$, it is highly likely that the interquartile range was less than 1% of the median for every test. We compile and benchmark with both gcc and clang, we record and report the faster of the two. The implementations require several initialisation steps, such as generating the permutation, temporary storage, and output storage. We initialise all these data structures, and then take all measurements in a tight loop. We then verify the correctness of the control bits and record the timings. We chose this experimental set-up to prioritise reproducibility,

Table 1: **Performance of portable implementations.** Latency (in median clock cycles) required to calculate control bits for a permutation of 2^{13} integers, taken over $n = 63$ samples, with an acceleration factor relative to cbrecursion with portable4. Both are compiled with `-O2` optimisations.

Sorting Implementation	Control Bit Implementation			
	cbrecursion		cbiterative	
portable4	132969723	1.00×	236477896	1.78×
iportable4	N/A		129255737	0.97×

Table 2: **Performance of AVX2-enabled implementations.** Latency (in median clock cycles) required to calculate control bits for a permutation of 2^{13} integers, taken over $n = 63$ samples, with an acceleration factor relative to cbrecursion with djbsort. Both are compiled with `-O2`, `-mavx2`, `-mtune=meteorlake` optimisations.

Sorting Implementation	Control Bit Implementation					
	cbrecursion		cbiterative		cbiteravx	
bitonic	24686875	1.15×	39171869	1.82×	36880919	1.71×
djbsort	21534000	1.00×	35644119	1.66×	33366053	1.55×
ibitonic	N/A		20329779	0.94×	17980542	0.83×
ipermsort	N/A		17610844	0.82×	15393579	0.71×

one should note that the key-generation phase of Classic McEliece requires only *one* call to the control bits implementation and thus the cache will be in a different state to our benchmarks.

Results summary. Table 1 and Table 2 present the median clock cycles required to perform a single control bit calculation on 2^{13} elements over varying control bit implementations and sorting network implementations. The primary factor influencing the performance of the implementations is the performance of the sorting network utilised. A simple sampling-based analysis with gprof [GKM82] suggests over 97% of execution time is spent sorting in both cbiterative and cbrecursion when they are paired with a scalar sorting network implementation. In Subsection 5.3, we enumerate and evaluate the strategies we use to implement sorting networks.

Both the recursive and iterative algorithms perform the same operations, their difference lies in the order in which they apply those operations. The performance increases in Table 2 are primarily driven by exploiting the data-parallelism exposed by Algorithm 4 with Intel's AVX2 feature. Our most optimised AVX2-enabled implementation (cbiteravx with ipermsort) calculates the control bits with an execution latency 0.71× that of the current AVX2-enabled libmceliece implementation (cbrecursion with djbsort).

In contrast, our scalar implementations cannot exploit this exposed data-parallelism. Whilst our cbiterative with iportable4 implementation does observe a small reduction in execution latency of 3% when compared with the current implementation in libmceliece (cbrecursion with portable4), this is only due to the following implementation detail: the current implementation of libmceliece applies $\text{layer}_0(\mathsf{L})$ to the right of $\text{layer}_0(\mathsf{F})(\pi)$ by using the OSORT primitive instead of the faster OLAYER primitive. When cbrecursion with portable4 is adapted to utilise OLAYER instead of OSORT, the performance difference is negligible and aforementioned reduction in execution latency drops to 0%.

Impact on Classic McEliece key generation. Table 3 presents the performance impact of our algorithm on key pair generation over the AVX2-enabled SUPERCOP implementations of the Classic McEliece "f" parameter sets. The "f" parameter sets are optimised for faster key generation through their utilisation of a semi-systematic matrix form. The `348864f` parameter set applies a permutation on 2^{12} elements and the others apply a

Table 3: **Impact on Classic McEliece key generation.** Comparison of cycles (measured with the stabilised second quartile) required to generate a Classic McEliece key pair when utilising cbrecursion with djbsort and cbiteravx with ipermsort.

Implementation	cbrecursion and djbsort	cbiteravx and ipermsort	Difference	Acceleration factor
mceliece348864f/avx	36786408	34050556	2735852	0.926×
mceliece460896f/avx	112098167	105162063	6936104	0.938×
mceliece6960119f/avx	211273369	204049988	7223381	0.966×
mceliece6688128f/avx	236102874	228852327	7250547	0.969×
mceliece8192128f/avx	265150943	257819345	7331598	0.972×

permutation on 2^{13} elements. Across the latter sets, we observe a constant decrease of around 7 million clock cycles. The differing acceleration factors are due to the relative weightings of the control bit calculation in the different parameter sets.

The values reported in Table 3 are obtained using the SUPERCOP toolkit [BL24] in the same experimental environment and conditions described above. To measure the latency of key generation in a KEM, SUPERCOP performs 3 rounds of 32 measurements of the clock cycles required to generate a key pair. In each round, SUPERCOP reports the stabilised second quartile [Ber25c] of the 32 measurements. The stabilised second quartile is more robust than the median in the presence of the many rejection-samplings in Classic McEliece's key generation. We report the minimum stabilised second quartile observed from the three rounds in order to minimise the impact of operating system interupts when benchmarking.

5.3 Sorting networks

Interlaced sorting networks. The recursive algorithms observe a natural advantage: each recursive subcall sorts array sizes descending in powers of two. Conversely, *every* call to the sorting network in the iterative implementations must sort an array of full size. Across all standard sorting networks, the iterative implementations are at least 1.42× slower than cbrecursion. When the iterative algorithms utilise interlaced sorting networks, we observe an interesting effect: the recursive decomposition introduces performance increases *until* the number of elements to sort becomes too small to vectorise. The majority of calls from cbrecursion to djbsort are on array sizes that are too small for djbsort to vectorise. In contrast, the full-size arrays that the iterative variant sorts become increasingly structured after every iteration, requiring fewer comparators in increasingly SIMD-friendly locations. We find cbiteravx with ibitonic requires an execution latency 0.73× that of cbrecursion with bitonic.

Permuting sorting networks. The instructions `vpminsd` and `vpmaxsd` apply 8 comparators in parallel *if* all the data elements we want to compare are at the same indices of two different registers. To apply the comparators of distance less than 8, which is the majority in Bitonic, we require heavy usage of swizzle instructions. However, one can reduce the amount of swizzles required by permuting the horizontal wires of the network to ensure the majority of comparators are between addresses a distance of 8 apart. We measure the performance effect of permuted sorting networks on control bit algorithms by providing non-permuted bitonic and ibitonic, and their respective permuted variants djbsort and ipermsort. We find cbrecursion with bitonic is 1.15× slower than cbrecursion with djbsort. Similarly, we find cbiteravx with ibitonic is 1.17× slower than cbiteravx with ipermsort.

Interlaced adaptations and validity. To implement a scalar interlaced sorting network requires minimal adaptations to its standard variant. For example, obtaining iportable4 from portable4 requires altering the terminating clause of a single for loop. The vectorised implementations require considerably more care as the comparisons between addresses of length less than 8 require sequences of swizzle instructions.

Bernstein exerts considerable effort into verifying that djbsort [Ber24] is correct. He provides a verifier which takes the compiled library, symbolically executes and converts it into a sequence of compare-and-exchange operations with the angr tool [SWS+16]. He then uses a 2-way merging argument to verify that the extracted sequence is a sorting network. We do not use symbolic execution to map compiled code into a comparator network to be verified. However, we do have a logical argument that justifies our network choices. We utilise the notion of *equivalence up to wire relabelling*, which was originally used to reduce the search space in the study of minimal sorting network enumeration [BZ14, CCFFSK14, BCCF+17], to increase the SIMD-friendliness of sorting networks by permuting the comparators for comparisons of index widths 1, 2, and 4 (which are the most common and most expensive) to comparators of widths 2^m, 2^{m-1}, and 2^{m-2} (which are the least common and inexpensive). If one permutes the wires of a sorting network by a permutation π, then the composition of this permuted sorting network and subsequently permuting by π correctly sorts all input data. We provide a verified proof of this in the accompanying artifact. We constructed ipermsort with the goal of removing small-distance comparisons by choosing similar bit-swapping permutations and following this argument. When $k \geq 3$, there are no small sorts to remove. When $k = 0$, we use djbsort. When $k = 1, 2$, the k least significant bits are sorted and we construct permutations sending elements that differ in these bits to elements which differ in their most significant bits.

Acknowledgements

We would like to thank Daniel Page and François Dupressoir for their support, insights, and feedback throughout this project. We also thank the anonymous reviewers and the shepherd for their feedback and suggestions that greatly improved this work. Wrenna Robson's research was supported by the EPSRC Centre for Doctoral Training in Cyber Security for the Everyday (EP/SO2187/1), which is funded by the EPSRC, the UK Government, and Royal Holloway, University of London. She received further funding from the Safety Critical Harsh Environment Micro-processing Evolution (SCHEME) project (10065634) which is funded by UKRI. Samuel Kelly's research was funded by the EPSRC Cyber Secure Everywhere Centre for Doctoral Training (EP/Y035313/1).

References

[ABB+24] Najwa Aaraj, Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Victor Dyseryn, Andre Esser, Thibauld Feneuil, Philippe Gaborit, Mukul Kulkarni, Victor Mateu, Marco Palumbi, Lucas Perin, Matthieu Rivain, Jean-Pierre Tillich, and Keita Xagawa. PERK. Technical report, National Institute of Standards and Technology, 2024. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-2-additional-signatures.

[ABC+22a] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions.

[ABC+22b] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wang Wen. Classic McEliece Design Rationale, 2022. https://classic.mceliece.org/mceliece-rationale-20221023.pdf.

[ABLM20] Jeremy Avigad, Kevin Buzzard, Robert Y Lewis, and Patrick Massot. Mathematics in Lean, 2020. Available at https://leanprover-community.github.io/mathematics_in_lean3/ Accessed on 19th January 2026.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307314, New York, NY, USA, 1968. Association for Computing Machinery. https://doi.org/10.1145/1468075.1468121.

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. https://doi.org/10.1007/978-3-662-07964-5.

[BCCF+17] Daniel Bundala, Michael Codish, Luís Cruz-Filipe, Peter Schneider-Kamp, and Jakub Závodný. Optimal-depth sorting networks. *Journal of Computer and System Sciences*, 84:185–204, 2017. https://doi.org/10.1016/j.jcss.2016.09.004.

[BCMP24] Alessandro Budroni, Isaac A. Canales-Martínez, and Lucas Pandolfo Perin. SoK: Methods for sampling random permutations in post-quantum cryptography. Cryptology ePrint Archive, Report 2024/008, 2024. https://eprint.iacr.org/2024/008.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In GuidoBertoniandJean-SébastienCoron, editor, *CHES 2013*, volume 8086 of *LNCS*, pages 250–272. Springer,Berlin,Heidelberg, August 2013. https://doi.org/10.1007/978-3-642-40349-1_15.

[BD99] Bruno Beauquier and Eric Darrot. On Arbitrary Waksman Networks and their Vulnerability. Technical Report RR-3788, INRIA, October 1999. https://doi.org/10.1142/S0129626402000999.

[Ben64] Václav E. Beneš. Permutation groups, complexes, and rearrangeable connecting networks. *Bell System Technical Journal*, 43(4):1619–1640, 1964. https://doi.org/10.1002/j.1538-7305.1964.tb04102.x.

[Ben65] Václav E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Mathematics in science and engineering : a series of monographs and textbooks. Academic Press, 1965. ISBN: 978-0-12-087550-4.

[Ben75] Václav E. Beneš. Applications of group theory to connecting networks. *Bell System Technical Journal*, 54(2):407–420, 1975. https://doi.org/10.1002/j.1538-7305.1975.tb02844.x.

[Ber20] Daniel J. Bernstein. Verified fast formulas for control bits for permutation networks. Cryptology ePrint Archive, Report 2020/1493, 2020. https://eprint.iacr.org/2020/1493.

[Ber24] Daniel J. Bernstein. djbsort: Intro. Website, 2024. Available at https://sorting.cr.yp.to/index.html. Accessed on 5th Januray 2025.

[Ber25a] Daniel J. Bernstein. The cryptoint library. *cr.yp.to Website*, 2025. Available at https://cr.yp.to/papers.html Accessed on 19th December 2025.

[Ber25b] Daniel. J Bernstein. Papers with computer-checked proofs. cr.yp.to Website, 2025. Available at https://cr.yp.to/papers.html Accessed on 19th December 2025.

[Ber25c] Daniel. J Bernstein. Robust statistics for rejection-sampling timings. cr.yp.to Website, 2025. Available at https://cr.yp.to/papers.html Accessed on 7th April 2026.

[Ber26] Daniel. J Bernstein. libcpucycles. cr.yp.to Website, 2026. Available at https://cpucycles.cr.yp.to/index.html Accessed on 13th April 2026.

[Beu20] Ward Beullens. Sigma protocols for MQ, PKP and SIS, and Fishy signature schemes. In AnneCanteautandYuvalIshai, editor, *EUROCRYPT 2020,Part III*, volume 12107 of *LNCS*, pages 183–211. Springer,Cham, May 2020. https://doi.org/10.1007/978-3-030-45727-3_7.

[BGK21] Mark Blacher, Joachim Giesen, and Lars Kühne. Fast and Robust Vectorized In-Place Sorting of Primitive Types. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.SEA.2021.3.

[BL24] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT Benchmarking of Cryptographic Systems. Website, 2024. Available at https://bench.cr.yp.to Accessed 11 December 2025.

[Bra17] Berenger Bramas. A Novel Hybrid Quicksort Algorithm Vectorized using AVX-512 on Intel Skylake. *International Journal of Advanced Computer Science and Applications*, 8(10), 2017. http://dx.doi.org/10.14569/IJACSA.2017.081044.

[BZ14] Daniel Bundala and Jakub Závodnỳ. Optimal sorting networks. In *International Conference on Language and Automata Theory and Applications*, pages 236–247. Springer, 2014. https://doi.org/10.1007/978-3-319-04921-2_19.

[CCFFSK14] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 186–193. IEEE, 2014. https://doi.org/10.1109/ICTAI.2014.36.

[Cho17] Tung Chou. McBits revisited. In WielandFischerandNaofumiHomma, editor, *CHES 2017*, volume 10529 of *LNCS*, pages 213–231. Springer,Cham, September 2017. https://doi.org/10.1007/978-3-319-66787-4_11.

[Cho25] Tung Chou. libmceliece: Intro. libmceliece website, 2025. Available at https://lib.mceliece.org/index.html Accessed on 19th December 2025.

[CM97] Chihming Chang and Rami Melhem. Arbitrary size Benes networks. *Parallel Processing Letters*, 07, 05 1997. https://doi.org/10.1142/S0129626497000292.

[Cor25] Intel Corp. Data Operand Independent Timing Instruction Set Architecture (ISA) guidance. Website, 2025. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html Accessed on 19th January 2026.

[DDCO17] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. *PoPETs*, 2017(3):21, July 2017. https://doi.org/10.1515/popets-2017-0026.

[dMU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. https://doi.org/10.1007/978-3-030-79876-5_37.

[Ebb15] Kris Vestergaard Ebbesen. On the practicality of data-oblivious sorting. Master's thesis, Aarhus Universitet, 2015.

[EGT10] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, page 1322, New York, NY, USA,

2010. Association for Computing Machinery. https://doi.org/10.1145/1869790.1869796.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In DavidPointchevalandThomasJohansson, editor, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 465–482. Springer,Berlin,Heidelberg, April 2012. https://doi.org/10.1007/978-3-642-29011-4_28.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120126, June 1982. https://doi.org/10.1145/872726.806987.

[Har96] John Harrison. HOL Light: A tutorial introduction. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 1996. https://dl.acm.org/doi/10.5555/646184.682934.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. TheInternetSociety, February 2012.

[HKC+21] Seungwan Hong, Seunghong Kim, Jiheon Choi, Younho Lee, and Jung Hee Cheon. Efficient sorting of homomorphic encrypted data with k-way sorting network. Cryptology ePrint Archive, Report 2021/551, 2021. https://eprint.iacr.org/2021/551.

[HOW22] William L. Holland, Olga Ohrimenko, and Anthony Wirth. Efficient oblivious permutation via the waksman network. In YujiSugaandKouichiSakuraiandXuhuaDingandKazueSako, editor, *ASIACCS22*, pages 771–783. ACMPress, May / June 2022. https://doi.org/10.1145/3488932.3497761.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998. https://dl.acm.org/doi/book/10.5555/280635.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In PalashSarkarandTetsuIwata, editor, *ASIACRYPT 2014,Part II*, volume 8874 of *LNCS*, pages 506–525. Springer,Berlin,Heidelberg, December 2014. https://doi.org/10.1007/978-3-662-45608-8_27.

[Lee85] Kyungsook Yoon Lee. On the rearrangeability of $2(\log 2n) - 1$ stage permutation networks. *IEEE Trans. Comput.*, 34(5):412–425, May 1985. https://doi.org/10.1109/TC.1985.1676581.

[Lee87] Kyungsook Yoon Lee. A new Benes network control algorithm. *IEEE Transactions on Computers*, C-36(6):768–772, 1987. https://doi.org/10.1109/TC.1987.1676970.

[Lei14] Tom F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*. Morgan Kaufmann, 2014. https://dl.acm.org/doi/10.5555/119339.

[LL96] Tony T. Lee and Soung-Yue Liew. Parallel routing algorithms in Benes-Clos networks. In *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*, volume 1, pages 279–286 vol.1, 1996. https://doi.org/10.1109/INFCOM.1996.497904.

[LPV81] Gavriela Freund Lev, Nicholas Pippenger, and Leslie G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C-30(2):93–100, 1981. https://doi.org/10.1109/TC.1981.6312171.

[mC20] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367381, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3372885.3373824.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In ThomasJohanssonandPhongQ.Nguyen, editor, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer,Berlin,Heidelberg, May 2013. https://doi.org/10.1007/978-3-642-38348-9_33.

[MU21] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 1215, 2021, Proceedings*, page 625635, Berlin, Heidelberg, 2021. Springer-Verlag. https://doi.org/10.1007/978-3-030-79876-5_37.

[NS82] Nassimi and Sahni. Parallel algorithms to set up the Benes permutation network. *IEEE Transactions on Computers*, C-31(2):148–154, 1982. https://doi.org/10.1109/TC.1982.1675960.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89100, June 2007. https://doi.org/10.1145/1273442.1250746.

[OSF+16] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In ThorstenHolzandStefanSavage, editor, *USENIXSecurity2016*, pages 619–636. USENIXAssociation, August 2016. https://dl.acm.org/doi/10.5555/3241094.3241143.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 02(02n03):205–211, 1992. https://doi.org/10.1142/S0129626492000337.

[Rob25] Wrenna Robson. *Finding meaning in cryptographic verification*. PhD thesis, Royal Holloway, University of London, 2025.

[Sch22] Peter Scholze. Liquid Tensor Experiment. *Exp. Math.*, 31(2):349–354, 2022. https://doi.org/10.1080/10586458.2021.1926016.

[SJG23] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Waks-on/waks-off: Fast oblivious offline/online shuffling and sorting with waksman networks. In WeizhiMengandChristianDamsgaardJensenandCasCremersandEnginKirda, editor, *ACMCCS2023*, pages 3328–3342. ACMPress, November 2023. https://doi.org/10.1145/3576915.3623133.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016. https://doi.org/10.1109/SP.2016.17.

[Tao24] Terence Tao. Machine assisted proof. *Notices of the American Mathematical Society, to appear*, 2024.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968. https://doi.org/10.1145/321439.321449.