Can Symmetric Cryptography Be Liberated from the Von Neumann Style?

Jean-Baptiste Menant[1], Yves Ndiaye[2], and
Pierre-Évariste Dagand[2,3]

[1]École Normale Supérieure de Lyon
[2]Université Paris Cité, IRIF
[3]CNRS

In defiance of Hinchliffe's rule, this article sets out to demonstrate that its title can be answered by the word "yes". We show that modern symmetric ciphers can be modeled through a small set of algebraic structures (Boolean algebra, Naperian and circulant functors). We reveal that these ciphers exhibit some interesting compositional structure at the type-level. This enables systematic code transformation, known in the cryptographic folklore as "bitslicing" and "fixslicing". Our work rests on a Coq development providing the specification of two ciphers, Skinny and Gift, deriving a bitsliced and fixsliced implementation for each and proving their correctness with respect to their specifications.

1 Introduction

Anyone who has ever read a cryptography paper knows this for a fact: cryptographers are at the top of the academic thought chain. In the archetypal symmetric cryptography paper, there is a "Section 2" scene in which cryptographers are bravely waving their most potent finite field. In a latter "Section 4" scene, the very same cryptographers are seen extolling the virtues of some nifty Arm Cortex assembly instruction, which takes center stage in their hand-crafted, manually optimized implementation. No wonder we jealously stash them away in "zone à régime restrictive"! The rumor goes that, behind these padlocked doors, they have developed a set of domain-specific techniques to deliver high-throughput software implementation of symmetric ciphers. This has been slowly gnawing at compiler writers, whose job security is now threatened [1, 2]. Luckily for us, every now and then, someone inadvertently let the genies out of their Klein bottle and we get to read about the subtle art of "bitslicing" [3] and "fixslicing" [4, 5].

The present work aims at casting these two notions in the language of functional programming. We shall see that bitslicing corresponds to a run-off-the-mill *data* representation change (in effect, a matrix transposition). But this is only half of the work: our ability to systematically transform the *code* operating on such representation rests on suitably polymorphic definitions: parametricity at its best! Fixslicing, on the other hand, will be rationalized through equational reasoning over purely functional terms. Identifying commuting expressions is of key importance there: programming with algebraic structures (rather than untamed arrays of bits) will be instrumental!

In the long run, our ambition is to offer cryptographers the conceptual apparatus to better illuminate their "Section 4" scene. Each and every symmetric cryptosystem should not have to exhibit a bitsliced or fixsliced implementation: by just *specifying* the cryptosystem in the right framework (abstract yet suitably operational), one should get the guarantee that the design can be bitsliced and fixsliced. In fact, one would boldly claim that a compiler could automatically apply these transformations and produce optimized code, starting from an high-level description of the cipher. This is already the case for bitslicing [6], adding support for fixslicing is next on the list.

As intimidating as cryptographers are, the essence of modern symmetric cryptographic primitives can in fact be distilled in a few paragraph. A primitive can be understood as a purely functional programs. It takes two inputs: the "cleartext" and a list of "round subkeys" (which are derived from a single key through a non-performance critical process called "key schedule" that we shall ignore here). The output is the "ciphertext". The cleartext, the ciphertext and each individual round subkeys are processed as "blocks" of binary data. The block size is of the order of a hundred bits (typically, 64, 128 or 256 bits).

Looking at the process of turning the cleartext into a ciphertext, one realizes that cryptographers have been programming in the State monad all along: ciphers are described as operating on a "state", initialized to be the cleartext and read off at the end as the ciphertext. This state is updated by a "round" function that is repeated a given number of times (of the order of ten iterations). The exact number of iterations is a trade-off between latency (less iterations means faster computation) and security (more iterations makes for more costly cryptanalysis effort). The round function itself is built compositionally from 3 components: a key mixing layer (making the encryption process reversible), a confusion layer (smoothing local correlations among the inputs) and a diffusion layer (spreading correlations across the whole output block).

To avoid side-channel attack (for example, based on timing [7]) over software implementations, this pipeline is implemented as a purely combinational circuit. In particular, we are forbidden from performing control-flow operations on secret data, leaving only statically-bounded loops. We are also forbidden from performing memory access based on the secret data, leaving only register spilling.

Following the NSA involvement in the design of the DES confusion layer [8], cryptographers have been advocating for "nothing-up-my-sleeve" designs (which does not make magic tricks impossible, just harder [9]). For example, the confusion layer of the AES standard is based on the usual multiplicative inverse over the finite field induced by an innocent-looking polynomial. To support such design, the state modern ciphers operate on is taken to represent a certain matrix (oftentimes, a 2D or 3D array of bits): the cipher is not specified over a flat sequence of individual bits but over a n-dimensional array of bits.

Moreover, ciphers are now, by and large, designed to be run efficiently on software. The era of hardware-based cryptography is long gone, even for embedded platforms [10]. To achieve high-throughput, ciphers are typically designed so as to admit a bitsliced implementation. Bitslicing enables a form of parallel execution *within* machine words, relying sometimes explicitly on the availability of vectorized instructions [11, 12]. We thus gain access to a form of CPU-level "scale out" to larger machine words, in particular through single-instruction multiple-data (SIMD) instructions. Doubling the size of machine words yields nearly twice the throughput, without any effort. Evidence tends to suggest that cryptographers obtain bitsliced designs through sheer intellectual might. It is in fact the sole purpose of the traditional "Section 4" scene to exhibit a bitsliced witness in all the gory details.

What would it take to turn this art form into an engineering principle? To answer, we make the following contributions:

- we identify the algebraic structure necessary to model modern symmetric cryptographic primitives and, in particular, their state (Section 2). Doing so, we delineate a mathematical language of software circuits, rooted in the categorical notion of functor to account for data containers ;
- we give a formal account of bitslicing as a data representation change supported by

suitably polymorphic definitions (Section 3). As expected when it comes to switching between data representations, parametricity plays a key role to justify the equivalence of the resulting programs ;

- we give a formal account of fixslicing as a whole-program transformation (Section 4). We show how a fixsliced implementation can be obtained through equational reasoning. To support such reasoning, we crucially rely on the algebraic structure set forth in Section 2.

We believe that there is also a broader take-away for an audience of French functional programmers. The present work surfs on the wake of the Squiggol school of programming [13]. Because our government-approved programming language does not yet support ad-hoc polymorphism [14], the French community may be missing out on some interesting programming patterns. The present article can thus be read as a case-study in "functor-oriented programming" [15], where the notion of functor comes from category theory (recalled in Section 2 and unrelated to the notion used in ML module systems). In effect, we forbid ourselves from programming over a concrete data structure: instead, we rely on (categorical) functors, peppered with some more structure, so as to 1. unlock algebraic manipulation during compilation and 2. fully exploit parametricity for reasoning.

The present work is but the beginning of our research program. We shall focus *exclusively* on the semantics aspects, only briefly touching upon implementation aspects (which remain the raison d'être of the project!). Our experience designing the Usuba [6] programming language and implementing its compiler gives us some confidence that a syntax can indeed be tailored to dress up this semantics.

We nonetheless could not resist the temptation of developing the semantics in the Coq theorem prover, as executable programs. After all, we are interested in combinational circuits: if there has ever been a time where we do not have to worry about termination, this is it! So we have implemented cryptographic primitive in pure Gallina, following the lead of some of our colleagues who went even further, down to deeply-embedded assembly code [16]. Besides the ability to prove the correctness and, even more usefully, test our code, this also enables us to easily prototype a fixslicing compiler, using Coq's `autorewrite` tactics to emulate a transformation-based simplification engine [17] (as part of a fictional optimizing compiler).

This article grew out of the Bachelor's research project of the first author and the Master thesis of the second author. The former worked on the specification, bitslicing and fixslicing of the SKINNY cipher [18, 5]. The latter worked on the specification, bitslicing and fixslicing of the GIFT cipher [19, 4]. Both ciphers are provided in the accompanying source code[1]. For clarity, we shall focus exclusively on SKINNY as our running example here.

Note that none of the programs below were easy to write in the first place: it took weeks of careful study to weed out the essential complexity from the accidental mismanagement of array indices. If the code seem somewhat trivial, one should bear in mind that this simplicity was hard-won. All the more reason to, some other day, design a programming language, so as to automatically inhabit this semantics, and write its compiler, so as to automatically optimize code following our theorems.

2 Functional specification

In this section, we intend to give a specification for the SKINNY cipher, with an eye towards implementation. We shall therefore aim for a rather operational description, to give a sense of the computational cost of the cipher, without premature concern for implementation performance just yet.

[1] `https://github.com/pedagand/bitfix`

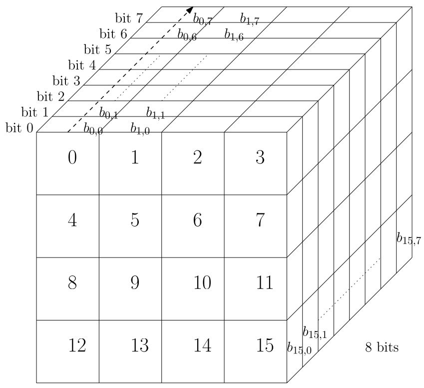


Figure 1. SKINNY state matrix

As often in functional programming, it helps tremendously to first lay out the types our program will have to deal with. In our case, the focal point is the *state* of the cipher, which is specified as a 3-dimensional matrix (Figure 1). We model each dimension in turn through a dedicated type constructor:

- `Rows.T` : `Type` → `Type` is a data container representing 4 rows of data (horizontal dimension),
- `Cols.T` : `Type` → `Type` is a data container representing 4 columns of data (vertical dimension), and
- `Slice8.T` : `Type` → `Type` is a data container representing 8 slices of data (depth dimension).

The cube drawn in Figure 1 could just be modeled as `Rows.T (Cols.T (Slice8.T bool))`, `Slice8.T (Rows.T (Cols.T bool))`, or any other composition of these 3 type constructors. They are all isomorphic to a sequence of 128 bits. We shall wait until Section 2.4 to settle on the most natural representation for our specification effort. We revisit this choice in Section 3 when we are concerned with producing a memory-efficient representation.

For all intents and purposes, these type constructors shall remain abstract throughout this article: we interact with them solely through their algebraic interface. The first of which is the categorical notion of functor:

```
Class Functor F :=
  { map: ∀ A B, (A → B) → F A → F B }.
```

To the disciples of Reynolds and Girard, this signature has come to mean "parametric data container": how would you go about asserting that the type constructor `Rows.T` is not doing something non-trivial with the type it is provided as an argument? In other words, how to be sure that a `Rows.T bool` behaves "similarly" as a `Rows.T nat`? One could try and argue that Coq does not allow pattern matching on types but it is rather unsavory to involve the design of the whole programming language into this argument. Instead, we require `Rows.T` to offer a `map` operation: this constructively witnesses the fact that a `Rows.T A` is (functionally) related to a `Rows.T B`, as long as we can (functionally) relate `A` and `B`. Put otherwise, we ask that `Rows.T`, and similarly `Cols.T` and `Slice8.T`, come equipped with their free theorems [20, 21].

Having modeled the core data types, we now turn to modeling the cipher as a purely functional program. To stay clear from the temptation of array mismanagement, we disallow index arithmetic altogether: no indexing an array-like structure by `i+1` or, even worse, `j−i` in this paper!

2.1 Key mixing

Much like λ in functional programming, the exclusive-or $\oplus$ features prominently in the cryptographic liturgy. In particular, it is instrumental to mix the derived keys during ciphertext computation. For genericity, we model this layer as an operation defined over any Boolean algebra [22]:

```
Variable B: Type.
Context `(Boolean B).

Definition add_round_key_ (constkey: B)(s: B): B :=
  xor s constkey.
```

The type `bool` is an obvious instance of a Boolean algebra. However, any commutative applicative [23] functor `F` applied to a Boolean algebra also yields a Boolean algebra (dispatching the Boolean operations pointwise to the underlying elements). For the sake of completeness, we recall that an applicative functor offers the following operations:

```
Class Applicative {F} `(Functor F) :=
  { pure: ∀ A, A → F A
  ; app: ∀ A B, F (A → B) → F A → F B}.
```

2.2 Diffusion

The role of the diffusion layer is to divert individual bits across the whole structure. Inevitably, this calls for a notion of indexing. Naperian functors [24] identify a type `Ix` of indices as the logarithm of an exponential type:

```
Class Naperian {F} `(Functor F) Ix :=
  { lookup: ∀ A, F A → Ix → A
  ; init: ∀ A, (Ix → A) → F A }.
```

through the fact that `lookup` and `init` form a bijection. This is also known as a representable functor in categorical circles [25].

Following our earlier discussion, we ask for `Rows.T` (respectively, `Cols.T`) to be a Naperian functor indexed by a set of 4 elements. Similarly, `Slice8.T` must be a Naperian functor indexed by 8 elements. We define

```
Inductive Ix := R0 | R1 | R2 | R3.
```

as the logarithm of `Rows.T`,

```
Inductive Ix := C0 | C1 | C2 | C3.
```

as the logarithm of `Cols.T`, and

```
Inductive Ix :=
  | S0 | S1 | S2 | S3
  | S4 | S5 | S6 | S7.
```

as the logarithm of `Slice8.T`.

Note that we have a notion of indexing but no arithmetic: we remain in line with our objectives. Note also that being Naperian implies being a commutative applicative functor. As a corollary, we get that any combination of `Rows.T`, `Cols.T` and `Slice8.t` applied to `bool` yields a Boolean algebra, over which we can therefore write combinational circuits.

Applying `init` to the identity function, we generically compute the container of indices:

```
Variable Ix: Type.
Variable F : Type → Type.
Context `{Naperian F Ix}.

Definition indices: F Ix := init (fun ix ⇒ ix).
```

We can witness the fact that `Rows.T` contains at least as many elements as `Cols.T` through the following construction:

```
Definition reindex_R (i: Rows.Ix): Cols.Ix :=
  match i with
  | Rows.R0 ⇒ Cols.C0
  | Rows.R1 ⇒ Cols.C1
  | Rows.R2 ⇒ Cols.C2
  | Rows.R3 ⇒ Cols.C3
  end.

Definition indices_C: Rows.T Cols.Ix :=
  map reindex_R (indices Rows.Ix).
```

Note that, since they actually have the same number of elements, we could also go the other way around, defining an inhabitant of `Cols.T Rows.Ix`. We do not need this construction for Skinny but it is necessary for Gift, which proceeds by transposition of a 4×4 matrix.

The first step of the diffusion process consists in applying a right rotation over each individual column. However, we do not have enough structure to identify a "right" or "left" direction over our containers. To do so, we introduce the (regretfully ad-hoc[2]) notion of circulant functor

```
Class Circulant {F} `(Functor F) :=
 { circulant: ∀ A, F A → F (F A)
 ; anticirculant: ∀ A, F A → F (F A) }.
```

taking an `F`-vector to an `F` × `F` circulant matrix (performing a right-rotation of `F A` at each step) and `F` × `F` anticirculant matrix (performing a left-rotation of `F A` at each step).

More usefully for our purposes, we can derive generic left and right rotation operators for any circulant functor `T` Naperian over a type `Ix`:

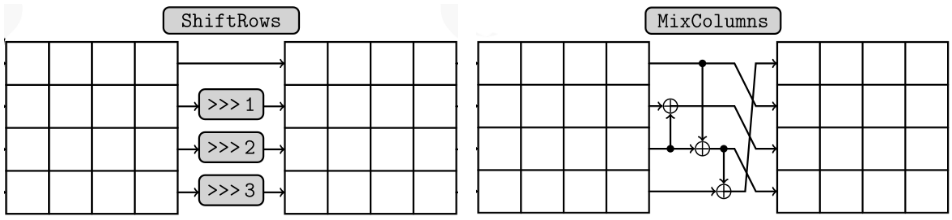
```
Definition ror {A} (ix: Ix)(xs: F A): F A :=
  lookup (circulant xs) ix.
Definition rol {A} (ix: Ix)(xs: F A): F A :=
  lookup (anticirculant xs) ix.
```

After suitable generalization, this leads us to the following model for the `shiftrows_` operation, depicted in Figure 2a:

```
Variable A: Type.

Definition shiftrows_: Rows.T (Cols.T A → Cols.T A) :=
  map ror indices_C.
```

[2] We were hoping to be able to piggy-back on the notion of foldable [26] and Naperian functor to identify directions. Intuitively, `fold` denotes a unique left-to-right traversal order. However, we could only turn this into a rotation if we asked for a decidable equality over the logarithm of the functor. We are not sure yet whether we want to make such a commitment.



(a) Shiftrows (from Beierle et al. [18]) **(b)** Mixcolumns (from Beierle et al. [18])

Figure 2. SKINNY diffusion layer

which represents the first half of the diffusion layer. The second half corresponds to a combinational circuit `psi_`

```
Variable B: Type.
Context `(Boolean B).

Definition psi_ (s: Rows.T B): Rows.T B :=
  let r0 := lookup s Rows.R0 in
  let r1 := lookup s Rows.R1 in
  let r2 := lookup s Rows.R2 in
  let r3 := lookup s Rows.R3 in
  let r0' := r0 in
  let r1' := xor r2 r1 in
  let r2' := xor r0 r2 in
  let r3' := xor r2' r3 in
  init (fun r => match r with
               | Rows.R0 => r0
               | Rows.R1 => r1'
               | Rows.R2 => r2'
               | Rows.R3 => r3'
               end).
```

which exercises the Naperian structure of `Rows.T` together with the underlying Boolean algebra. This is followed by a right rotation of rows. Altogether, this defines the `mixcolumns_` operation, depicted in Figure 2b:

```
Definition mixcolumns_ (rs: Rows.T B): Rows.T B :=
  ror Rows.R1 (psi_ rs).
```

2.3 Confusion

The confusion layer builds upon two permutations over `Slice8.T`. Once again, this is merely exploiting the Naperian structure of `Slice8.T`: for any Naperian functor `F` indexed by `Ix`, there exists a generic permutation operator `perm` : ∀ `A`, (`Ix` → `Ix`) → `F A` → `F A`. This leads to the following definitions:

```
Variable B: Type.

Definition bperm (xs: Slice8.T B): Slice8.T B :=
  perm (fun s => match s with
               | Slice8.S7 => Slice8.S2 | Slice8.S6 => Slice8.S1
               | Slice8.S5 => Slice8.S7 | Slice8.S4 => Slice8.S6
```

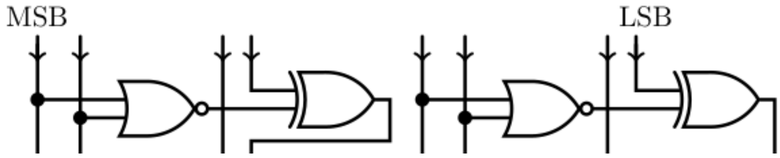


Figure 3. Substitution box S8 (from Beierle et al. [18])

```
                | Slice8.S3 => Slice8.S4 | Slice8.S2 => Slice8.S0
                | Slice8.S1 => Slice8.S3 | Slice8.S0 => Slice8.S5
                end) xs.

Definition bperm_out (xs: Slice8.T B): Slice8.T B  :=
  perm (fun s => match s with
                | Slice8.S0 => Slice8.S0 | Slice8.S1 => Slice8.S2
                | Slice8.S2 => Slice8.S1 | Slice8.S3 => Slice8.S3
                | Slice8.S4 => Slice8.S4 | Slice8.S5 => Slice8.S5
                | Slice8.S6 => Slice8.S6 | Slice8.S7 => Slice8.S7
                end) xs.
```

At the heart of this layer stands the substitution box `s8` (Figure 3), which is just another combinational circuit:

```
Context `(Boolean B).

Definition gate x y z := xor x (not (or y z)).

Definition s8 (xs: Slice8.T B): Slice8.T B :=
  let b0 := lookup xs Slice8.S0 in
  let b2 := lookup xs Slice8.S2 in
  let b4 := lookup xs Slice8.S4 in
  let b6 := lookup xs Slice8.S6 in
  let b7 := lookup xs Slice8.S7 in

  let b0' := gate b0 b2 b2 in
  let b4' := gate b4 b6 b7 in
  init (fun i => match i with
                | Slice8.S0 => b0'
                | Slice8.S4 => b4'
                | i => lookup xs i
                end).
```

These building blocks are chained together to form the confusion layer, `subcells_`:

```
Definition subcells_ (xs: Slice8.T B): Slice8.T B :=
  let xs := bperm (s8 xs) in
  let xs := bperm (s8 xs) in
  let xs := bperm (s8 xs) in
  bperm_out (s8 xs).
```

2.4 Making the rounds

A round of Skinny is obtained by composing the following operations in this particular order:

1. `subcells_ : ∀ {B : Type}, Boolean B → Slice8.T B → Slice8.T B`
2. `add_round_key_ : ∀ {B : Type}, Boolean B → B → B → B`
3. `shiftrows_ : ∀ {A : Type}, Rows.T (Cols.T A → Cols.T A)`
4. `mixcolumns_ : ∀ {B : Type}, Boolean B → Rows.T B → Rows.T B`

Each of these operations will have to be lifted up to proceed over the entire state of the cipher. We shall therefore settle on the most convenient composition order for the type constructors `Rows.T`, `Cols.T` and `Slice8.T`.

The solution can be read off from the type constraints induced by individual elements and their composition. First, the type of `shiftrows_` dictates that we compose `Rows.T` followed by `Cols.T`. Then, we observe that `subcells_` can be grounded to `bool` without disturbance. This leads to following composition order:

```
Definition cube A := Rows.T (Cols.T (Slice8.T A)).
Definition state := cube bool.
```

Consequently, our hands are tied when it comes to lifting up individual components:

```
Definition round (s: state)(constkey: state): state :=
 let s := map subcells_ s in
 let s := add_round_key_ constkey s in
 let s := app shiftrows_ s in
 mixcolumns_ s.
```

In particular, we have that `subcells_` is iterated pointwise across rows and columns, `add_round_key_` handles the entire cube as the support for a Boolean algebra, `shiftrows_` applicatively applies its column transformation over individual rows and `mixcolumns` handles horizontal faces of the cube as Boolean algebras.

The overall primitive is nothing but the iteration of a single round over the list of round subkeys, produced by an offline key schedule procedure:

```
Definition skinny (constkeys: list state)(s: state): state :=
 fold_left round constkeys s.
```

3 Bitslicing

If we were to adopt `state` as our actual representation, the memory usage of the resulting program would be quite poor. It would take at least 128 bytes to store that many bits (assuming that booleans are stored in the smallest addressable memory size, a byte). This would be a terrible waste of memory bandwidth and register usage.

We must therefore look toward adopting a *packed* representation, grouping individual booleans into a single machine word. Doing so would improve memory density. However, we have to be very careful that we can still efficiently compute over this packed representation. As it turns out, for Skinny and Gift, we will not achieve this just now: we will have to wait until Section 4, to have our cake and eat it too (using the *same* data representation).

Bitslicing is the cryptographer's jargon for such a representation change. In the case of Skinny, we observe that only `shiftrows_` imposes a strict composition order of `Rows.T` followed by `Cols.T`. Aside from that, we can commute `Slice8.T` out of the composition. We would thus have the type `Rows.T (Cols.T bool)` representing 16 bits and fitting snugly in a 32 bits machine word. The overall composition `Slice8.T (Rows.T (Cols.T bool))` would take 8 registers in total, imposing a very reasonable amount of register pressure on the CPU. In effect, to return back to the jargon of compiler designers, we propose to turn a structure-of-arrays (SoA) into an array-of-structures (AoS), only we are dealing with bit-level quantities here.

Storing only 16 bits in a 32 bits word (on ARM Cortex M) remains sub-optimal. We can further increase register usage and instantly double throughput by processing two blocks at the same time. To this end, we introduce a new Naperian functor, `Double.T`, indexed by the two elements set `Fst` and `Snd`. This leads to

```
Definition reg32 A := Rows.T (Cols.T (Double.T A)).
Definition cube2 A := Slice8.T (reg32 A).
Definition state := cube2 bool.
```

which denote our intent to manipulate objects of type `reg32` as if they were stored in a single machine word. In the present work, this observation will remain at the state of wishful thinking. One can think of the code in this section (as well as the fixsliced code, in the next section) as playing, themselves, the role of specifications to lower-level refinements. Our previous work on Usuba suggests that we will be able to deliver on this promise in the future. Further, remark that the `state` now corresponds to two intermingled blocks but as we shall process them in lock-step throughout the cipher, this does not make much difference.

Given two Naperian functors `F` and `G`, there exists a generic notion of reindexing [27]:

```
 Variables A FIx GIx: Type.
 Variable F G : Type → Type.
 Context `{Naperian F FIx} `{Naperian G GIx}.

 Definition reindex (fgs : F (G A)): G (F A) :=
   init (fun j ⇒
   init (fun i ⇒
   lookup (lookup fgs i) j)).

```

Operationally, reindexing effects a change of the iteration order of the composed data container: we go from supporting iteration over `G` within an external iteration over `F` to an iteration over `F` within an external iteration over `G`.

Using such reindexing, we can specify the transposition process relating two input blocks (following the specification) and their bitsliced counterpart:

```
Definition to_bitslice {A} (s0: Spec.cube A)(s1: Spec.cube A): Bitslice.cube2 A :=
  reindex (F := Double.T)
    (init (fun i ⇒
           match i with
           | Double.Fst ⇒ reindex (G := Slice8.T) s0
           | Double.Snd ⇒ reindex (G := Slice8.T) s1
           end)).
```

The correctness of a candidate bitsliced implementation, dubbed `Bitslice.skinny`, amounts to the following:

Theorem 1 (Correctness of bitslicing)**.** *For any list of pairs of round subkeys (of type `list (Spec.state * Spec.state)`) and for any pair of input blocks (of type `list Spec.state` individually), we have that a single execution of `Bitslice.skinny` produces the same output (after transposition) as two runs of the specification `Spec.skinny`:*

```
   let constkeys0 := List.map fst constkeys in
   let constkeys1 := List.map snd constkeys in
```

```
  to_bitslice
    (Spec.skinny constkeys0 s0)
    (Spec.skinny constkeys1 s1)
  = Bitslice.skinny
      (List.map (fun '(x, y) ⇒ to_bitslice x y) constkeys)
      (to_bitslice s0 s1).
```

In practice, it is useful to generalize this statement and, instead, state that `Spec.skinny` and `Bitslice.skinny` must preserve the following relation

```
Definition R {A} (s0: Spec.cube A)(s1: Spec.cube A)(s: Bitslice.cube2 A): Prop :=
  to_bitslice s0 s1 = s.
```

from their input to their outputs.

The relational invariant will then naturally and compositionally percolate through the various layers of the ciphers. We can then read off the bitsliced code from the bitsliced types and relational invariant. A single round of the cipher becomes:

```
Definition round (s: state)(constkey: state): state :=
  let s := subcells_ s in
  let s := add_round_key_ constkey s in
  let s := map (F := Slice8.T) (app shiftrows_) s in
  map mixcolumns_ s.
```

Considering (very carefully!) the instances of `subcells_`, `add_round_key_` and `psi_` (which is part of `mixcolumns_`), we check that these can be efficiently implemented over machine words: they only involve boolean operations, applied bit-wise over `reg32`.

Unfortunately, `shiftrows_` and `Rows.ror` would entail some very fiddly and computationally costly bit twiddling. These amount to performing bit-level permutations within a machine word. This is a frequent pain point when bitslicing the diffusion layer of ciphers: it is also true in the case of Gift and Aes, for instance. In the next section, we present this one weird trick to get permutations to nearly vanish.

4 Fixslicing

Studying the Gift cipher, Adomnicai et al. [4] noticed that the permutation layer can be decomposed into, first, a transformation that can be efficiently implemented over machine words and, second, a permutation that commutes with the other layers (confusion and key mixing). Similarly, the diffusion layer of Skinny can be decomposed as follows

Proposition 1. *∀ {B} ‘(Boolean B) (rs: Rows.T (Cols.T B)),*
mixcolumns_ (app shiftrows_ rs) = phi_ (sigma_ rs).

where `phi_` is a permutation with inverse `inv_phi_` that captures all the costly bit-twiddling operations

```
Definition phi_ (rs: Rows.T (Cols.T A)): Rows.T (Cols.T A) :=
  ror Rows.R1 (app shiftrows_ rs).


Definition inv_phi_ (rs: Rows.T (Cols.T A)): Rows.T (Cols.T A) :=
  let rs := rol Rows.R1 rs in
  app (map rol indices_C) rs.

```

while, in turn, `sigma_` admits an efficient implementation on machine words

```
Definition sigma_ (s: Rows.T (Cols.T B)): Rows.T (Cols.T B) :=
  let r0 := lookup s Rows.R0 in
  let r1 := lookup s Rows.R1 in
  let r2 := lookup s Rows.R2 in
  let r3 := lookup s Rows.R3 in
  let r0' := r0 in
  let r1' := xor (ror Cols.C1 r2) r1 in
  let r2' := xor (ror Cols.C2 r0) r2 in
  let r3' := xor (ror Cols.C3 r2') r3 in
  init (fun r => match r with
                 | Rows.R0 => r0'
                 | Rows.R1 => r1'
                 | Rows.R2 => r2'
                 | Rows.R3 => r3'
                 end).
```

Indeed, aside from the logical operations, the rotations are in fact almost free thanks to the ARM Cortex barrel shifter.[3]

Being able to postpone `phi_` would do us no good if we were nonetheless forced to run it anyway. The second, crucial observation is that, after only 4 steps, `phi_` is almost an identity.

Proposition 2. *For any state `rs: Rows.T (Cols.T A)`, we have:*

```
tau_ (iter 4 phi_ rs) = rs.
```

where `tau_` is defined as

```
Definition tau_ (rs: Rows.T (Cols.T A)): Rows.T (Cols.T A) :=
  map (ror Cols.C2) rs.
```

which can be implemented reasonably efficiently.

In typical mode of operation, SKINNY will perform between 32 and 56 rounds. Being able to accumulate `phi_` across chunks of 4 iterations, we can effectively make them disappear, leaving only a `tau_` step every four iterations.

We verify that `phi_` commutes with `subcells_`:

Proposition 3. *`∀ {B} `{Boolean B} (s: Slice8.T (Rows.T (Cols.T B))),`*
`subcells_ (map phi_ s) = map phi_ (subcells_ s).`

This is intuitively obvious since `subcells_` proceeds pointwise over rows and columns, hence changing their respective position does not influence the outcome.

Similarly and for the same reason, `phi_` commutes with `add_round_key`:

Proposition 4. *`∀ B `(Boolean B) (s constkey: Rows.T (Cols.T B)),`*
`add_round_key_ (phi_ s) constkey`
`= phi_ (add_round_key_ s (inv_phi_ constkey)).`

[3] By this remark, we hope to have established our credentials as budding cryptographers.

Note that we have to transform the input key before hand, so as to keep both blocks aligned. In particular, this means that we do not have to correct the first round subkey, we will have to apply `inv_phi` to the second round subkey, `iter 2 inv_phi` to the third round subkey and, finally, `iter 3 inv_phi` to the fourth round subkey. The fifth round subkey starts back in sync, *etc.*

The crux of the matter is the interaction between `phi_` and the diffusion layer. We observe that putting a state `phi_ s` into the diffusion layer first yield a transformation relativized to a machine word followed by the emission of two `phi_` steps out of the layer:

Corollary 1. *∀ B '(Boolean B) (s: Rows.T (Cols.T B)),*
mixcolumns_ (app shiftrows_ (phi_ s))
= phi_ (phi_ (inv_phi_ (sigma_ (phi_ s)))).

This means that, in turn, the subsequent diffusion layer will have to absorb two `phi_` steps. We thus generalize our statement for any number of `phi_` steps, the previous statement corresponding to the case $n = 1$:

```
Definition mixcolumns_mod i (s: Rows.T (Cols.T B)): Rows.T (Cols.T B) :=
  iter i inv_phi_ (sigma_ (iter i phi_ s)).
```

Ultimately, there are only 4 relativized diffusion layers, namely `mixcolumns_mod` 0, `mixcolumns_mod` 1, `mixcolumns_mod` 2 and `mixcolumns_mod` 3 and they correspond to the diffusion layers absorbing that many `phi_` steps:

Proposition 5. *∀ B '(Boolean B) n (s: Rows.T (Cols.T B)),*
mixcolumns_ (app shiftrows_ (iter n phi_ s))
= iter (1 + n) phi_ (mixcolumns_mod n s).

Given 4 (suitably generated) round subkeys, we merely have to chain 4 rounds with the corresponding diffusion layer, ending with `tau_` to re-synchronize back to identity

```
Definition round_mod (s: state) constkeys :=
 let '(constkey0, constkey1, constkey2, constkey3) :=
  constkeys in
 let s := round (map (mixcolumns_mod 0)) s constkey0 in
 let s := round (map (mixcolumns_mod 1)) s constkey1 in
 let s := round (map (mixcolumns_mod 2)) s constkey2 in
 let s := round (map (mixcolumns_mod 3)) s constkey3 in
 map tau_ s.
```

The correctness statement is unsurprising, modulo some light bureaucracy to ensure that round subkeys are in the right format

Theorem 2. *∀ f_constkeys s,*
let constkeys := flatten_constkeys f_constkeys in
Bitslice.skinny constkeys s
= Fixslice.skinny f_constkeys s.

The meat of the proof consists in, as for bitslicing, generalizing this statement to a relational one. With fixslicing, there are in fact 4 relations, depending on the number of `phi_` steps accumulated. We move from one to the next every time we go through a diffusion layer, resetting from the last to the first every 4 steps.

The definitions of `mixcolumn_mod` is somewhat disappointing: we would be hard-pressed to make any precise claim about the potentiality of an efficient implementation. The current form is symbolically useful, as it allows us to easily prove the correctness of fixslicing. However, it remains a mathematical specification, not a piece of software. In effect, we are now looking for a closed expression in the language of Boolean algebras (`xor`, *etc.*), Naperian operations (`init`, `lookup`) and circulant operations (more precisely, `ror` and `rol`).

One solution[4], which consists in using Coq as a term rewriting engine, is to postulate the existence of such an explicit form, let us call it `mixcolumns_mod_explicit`. We then claim that this definition ought to be equivalent to its specification, `mixcolumns_mod` for $n \in \{0, 1, 2, 3\}$. Making sure that the operations of Boolean algebra, Naperian and circulant functors are opaque, β reduction yields a term in our target language. However, it suffers from significant redundancy: by working over the term algebra, we have not quotiented out the equational theory. To do so, we orient the equational theory in a rewrite database and apply the `autosubst` tactics to obtain simplified forms. Similarly, we retrieve convenient `let` forms thanks to the `set` (`ident` := `term`) tactics. We are then left with reading off the definition of `mixcolumns_mod_explicit` from the proof goal.

5 Conclusion

We have thus completed our journey toward a rationalized treatment of bitslicing and fixslicing in their quintessential form, taking the SKINNY cipher as our running example. We have similarly treated the GIFT cipher, whose design is in fact at the origin of the fixslicing technique [4]. GIFT works over the type `Rows.T (Cols.T (Slice4.T bool))` where `Slice4.T` denotes a depth of 4 slices. In bitsliced form, we once again extract out the `Slice4.T` functor and `Double.T` the amount of data processed per run. Fixslicing is conceptually easier to understand as the diffusion layer is defined as the combination of an in-register transformation followed by a 90-degree rotation of the matrix of rows and columns. It simply cancels out after 4 steps.

Of note, the propositions that lead to the bitslicing and fixslicing correctness theorems hide a somewhat surprising secret: all the proofs were obtained by `vm_compute; congruence`. Having defined `Rows.T`, `Cols.T`, `Slice4.T` and `Slice8.T` as records with primitive projections, all the identities boil down to the specification, bitsliced forms and fixsliced forms having the same β-normal η-long form through the equivalence relation!

As part of future work, we intend to extend our formalization to bridge the gap to machine word. Once again, we expect parametricity to kick in: we currently have an implementation defined over `reg32` (and its supporting operations) which ought to be equivalent to an implementation defined over the type of 32 bits machine words (and its supporting operations). We also wish to pursue the automated generation of fixsliced code, eventually implementing a proper simplification engine. In the process, we ought to develop methods to check for commutativity of the diffusion layer. A litmus test to this project will be our ability to deliver a fixsliced implementation of AES: mark our machine words!

[4]Needless to say, in reality, we first went looking for 4 direct, simplified implementations, inspired by Adomnicai et al. [4]. Then we showed that these implementations were equivalent to their respective specification. Only later did we use Coq to understand how these solutions *could* have been derived from first principles.

References

[1] Daniel J Bernstein. The death of optimizing compilers, 2015. URL `https://cr.yp.to/talks/2015.04.16/slides-djb-20150416-a4.pdf`.

[2] A. P. Shivarpatna Venkatesh, A. Bhat Handadi, and M. Mory. Security implications of compiler optimizations on cryptography - A review. *CoRR*, abs/1907.02530, 2019. URL `http://arxiv.org/abs/1907.02530`.

[3] Eli Biham. A fast new DES implementation in software. In *FSE*, 1997. doi: 10.1007/BFb0052352.

[4] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A new GIFT representation. Cryptology ePrint Archive, Paper 2020/412, 2020. URL `https://eprint.iacr.org/2020/412`. `https://eprint.iacr.org/2020/412`.

[5] Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers new bitsliced AES speed records on arm-cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021. doi: 10.46586/TCHES.V2021.I1.402-425. URL `https://doi.org/10.46586/tches.v2021.i1.402-425`.

[6] Darius Mercadier. *Usuba, Optimizing Bitslicing Compiler. (Usuba, Compilateur Bitslicing Optimisant)*. PhD thesis, Sorbonne University, France, 2020. URL `https://tel.archives-ouvertes.fr/tel-03133456`.

[7] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. URL `http://cr.yp.to/papers.html#cachetiming`.

[8] T.R. Johnson and United States. National Security Agency. *American Cryptology During the Cold War, 1945-1989: The struggle for centralization, 1945-1960*. Series VI the NSA period 1952 - present. Center for Cryptologic History, National Security Agency, 1995.

[9] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: a white paper for the black hat. Cryptology ePrint Archive, Paper 2014/571, 2014. URL `https://eprint.iacr.org/2014/571`.

[10] William J Buchanan and Leandros Maglaras. Review of the nist light-weight cryptography finalists. In *19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 469–474, 2023. doi: 10.1109/DCOSS-IoT58021.2023.00079.

[11] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK lightweight block ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015. doi: 10.1145/2744769.2747946. URL `https://doi.org/10.1145/2744769.2747946`.

[12] Daniel J. Bernstein. Chacha, a variant of salsa20. In *State of the Art of Stream Ciphers Workshop, SASC 2008, Lausanne, Switzerland*, January 2008. URL `https://cr.yp.to/papers.html#chacha`.

[13] Richard S. Bird, Jeremy Gibbons, Ralf Hinze, Peter Höfner, Johan Jeuring, Lambert G. L. T. Meertens, Bernhard Möller, Carroll Morgan, Tom Schrijvers, Wouter Swierstra, and Nicolas Wu. Algorithmics. In Michael Goedicke, Erich J. Neuhold, and Kai Rannenberg, editors, *Advancing Research in Information and Communication Technology - IFIP's Exciting First 60+ Years, Views from the Technical Committees*

and Working Groups, volume 600 of *IFIP Advances in Information and Communication Technology*, pages 59–98. Springer, 2021. doi: 10.1007/978-3-030-81701-5_3. URL `https://doi.org/10.1007/978-3-030-81701-5_3`.

[14] Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*, volume 198 of *EPTCS*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL `https://doi.org/10.4204/EPTCS.198.2`.

[15] Russell O'Connor. Functor-oriented programming. `http://r6.ca/blog/20171010T001746Z.html`.

[16] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1202–1219, 2019. doi: 10.1109/SP.2019.00005.

[17] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for haskell. *Sci. Comput. Program.*, 32(1-3):3–47, 1998. doi: 10.1016/S0167-6423(97)00029-4. URL `https://doi.org/10.1016/S0167-6423(97)00029-4`.

[18] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II 36*, pages 123–153. Springer, 2016.

[19] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: A small present. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 321–345, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66787-4.

[20] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[21] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404. URL `https://doi.org/10.1145/99370.99404`.

[22] Steven Givant and Paul R. Halmos. *Introduction to Boolean Algebras*. Undergraduate Texts in Mathematics. Springer, 2006. ISBN 978-0-387-40293-2.

[23] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi: 10.1017/S0956796807006326.

[24] Peter Hancock. Napier's combinators and Böhm's logarithms. `http://docs.hancock.fastmail.fm.user.fm/arithmetic.pdf`.

[25] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[26] Brent Yorgey. Typeclassopedia. *The Monad.Reader*, (13):17–68, March 2009.

[27] Jeremy Gibbons. Aplicative programming with naperian functors. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 556–583. Springer, 2017. doi: 10.1007/978-3-662-54434-1_21. URL `https://doi.org/10.1007/978-3-662-54434-1_21`.