Cryptography Experiments In Lean 4: SHA-3 Implementation

Gérald Doussot*

November, 2024

Abstract

In this paper we explain how we implemented the Secure Hash Algorithm-3 (SHA-3) family of functions in Lean 4, a functional programming language and theorem prover. We describe how we used several Lean facilities including type classes, dependent types, macros, and formal verification, and then refined the design to provide a simple one-shot and streaming API for hashing, and Extendable-output functions (XOFs), to reduce potential for misuse by users, and formally prove properties about the implementation.

1 Introduction

Lean 4[1][2] is a functional programming language, and theorem prover. It has many features including first-class functions, dependent types, metaprogramming, verification and extensible syntax to name a few, making it interesting and suitable for a wide range of problems. Mathlib[2][3], the Lean mathematical library, is the most significant and impactful project written and formalized in Lean. Lean's dual nature makes it compelling for a number of reasons, one of which is being able to formally prove properties about a program written in Lean itself. Implementation, and usage of cryptography libraries is notoriously difficult, and error-prone. At the least, this makes Lean 4 a good candidate language for prototyping executable cryptographic primitive and protocol implementations, and proving properties about them.

Cryptographic hash functions are arguably simpler to implement than other primitives such as those found in public key cryptography. Yet, their implementations are not immune to memory corruption in memory unsafe languages, and their design and implementation may lead to misuse and incorrect results. There has recently been renewed interest in the Secure Hash Algorithm-3 family of functions on data, and in SHAKE128/SHAKE256 in particular, due to their adoption in post-quantum cryptography schemes. SHA-3 also has an interesting design with implications for implementers, and consequences for users; it is a

*gerald.doussot@nccgroup.com

[1]Lean 4: https://lean-lang.org/

[2]Mathlib: https://leanprover-community.github.io/mathlib-overview.html

sponge construction[3], and care must be taken to enforce the correct sequencing of operations, and management of internal data structures.

We present our experience in implementing SHA-3 in Lean 4 in this paper. We first provide a brief introduction of SHA-3, focusing on interesting aspects for our Lean implementation. We then describe how we implemented parts of SHA-3 including state management, hash function parameters, permutation, the sponge construction, user-facing API, and formal verification. We show how we proved that all access to the input data, the SHA-3 round constants table, the internal buffer, and state are in bounds, and how we enforced a prohibition on undue absorb and squeeze alternation. We follow with a discussion on how we tested our implementation for correctness, and performance. We present potential issues to consider when implementing Lean code in high-assurance systems, before providing concluding remarks.

Our SHA-3 implementation in Lean 4 is available on GitHub[4].

2 SHA-3

The "FIPS 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions"[5] 2015 standard specifies the Secure Hash Algorithm-3 family of functions on data. It is based on an instance of the KECCAK algorithm that NIST selected as the winner of the "SHA-3 Cryptographic Hash Algorithm Competition". The SHA-3 family consists of four cryptographic hash functions SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two extendable-output functions (XOFs), SHAKE128 and SHAKE256. The latter are now mandated in post-quantum algorithms, as part of the Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM) FIPS 203[6], and of the Module-Lattice-Based Digital Signature Standard (ML-DSA) FIPS 204[7] 2024 standards. SHA-3 is a sponge construction[8] for building a function with variable-length input and output, with a permutation on a fixed length block, the state. It has two distinct phases, absorption and squeezing, and one cannot alternate these phases in SHA-3. The fixed length block consists of two parts: the rate and the capacity. All but two SHA-3 functions have different capacity parameters; with the block size being set to 200 bytes, this drives the size of the matching rate part of the block.

At a high level, in the absorption phase the input message is chunked, and each chunk is XOR'ed with the rate part of the fixed block. The KECCAK-f permutation is called on the overall block after each chunk has been processed. In the squeezing phase, the rate part of the block is returned, interleaved with calls to the hash function on the whole block. Switching from the absorbing phase to the squeezing phase will result in padding of the input message. Depending of the implementation, this process of working on different parts of the block, with block parts of different sizes depending on the hash functions, and for different input and output sizes, can be error-prone. In fact, researchers

[3]Sponge construction: https://en.wikipedia.org/wiki/Sponge_function
[4]SHA-3 Lean 4 implementation: https://github.com/gdncc/Cryptography
[5]FIPS 202: https://csrc.nist.gov/pubs/fips/202/final
[6]FIPS 203: https://csrc.nist.gov/pubs/fips/203/final
[7]FIPS 203: https://csrc.nist.gov/pubs/fips/204/final
[8]Cryptographic sponge functions: https://keccak.team/files/CSF-0.1.pdf

found a buffer overflow vulnerability in the implementation of the Secure Hash Algorithm 3 (SHA-3) that had been released by its designers[1]. Proving that accesses to data structures are in bounds is therefore of interest to avoid memory corruption and other adverse impacts.

3 Implementing The SHA-3 Family Of Functions

3.1 SHA-3 State

The SHA-3 core functions revolve around managing a state, which is 200 bytes in length. This is large, and we need to find a way to represent this state in an efficient manner. Lean allows to implement code closer to the machine, using native types. It has support for efficient structures including array (`Array`), and byte array (`ByteArray`), which is an array of unsigned byte elements. Note that while `Array` is a native array, the elements of the array are Lean objects, whereas `ByteArray` is a completely native byte array, and would probably be more performant[9] where applicable. Another potentially suitable structure is the vector[10](`Vector`) type in the Lean community `Batteries` library. This library is officially maintained and relatively small, but for this project, we wanted to limit our experiments to core Lean 4. `Array` is appropriate to represent the underlying SHA-3 state, and we chose this type. However, `Array` can have an arbitrary size (and can change size), therefore it would be best to put restrictions on its size to better convey our implementation intent, and reduce possibilities of bugs. In our implementation, we created a Lean subtype (a dependent type) to achieve that, as shown in listing 1.

```
-- An array of 25 UInt64 values
private abbrev Arr25 := { val : Array UInt64 // val.size = 25}
private abbrev State := Arr25
```

Listing 1: State subtype

We first defined `Arr25`, a subtype with a value of type `Array UInt64`, which should work well on modern architectures, and the property that the array has 25 elements, in the form of a logical statement. We then defined another name, `State` for this data type. We used `abbrev` instead of `def` to allow Lean to automatically find[11] type class instance values based on the unfolded definition.

We then defined a number of utility functions, and tools as shown in listing 2.

The first function instantiates an empty state, by creating an array of 25 zero values, and providing a proof that the array has indeed 25 elements, as

[9] `Array` and `ByteArray` representation: https://leanprover.zulipchat.com/#narrow/channel/113489-new-members/topic/Code.20review.3A.20proof.20about.20Array.20map/near/454906535

[10] Batteries library vector type: https://github.com/leanprover-community/batteries/blob/31a10a332858d6981dbcf55d54ee51680dd75f18/Batteries/Data/Vector/Basic.lean#L23

[11] def vs abbrev: https://leanprover-community.github.io/archive/stream/270676-lean4/topic/def.20vs.20abbrev.html

```
private def mkState : State :=
  ⟨ mkArray 25 0, by decide ⟩

instance : GetElem Arr25 Nat UInt64 (λ _ i ↦ i < 25) where
  getElem state idx _ :=  state.val[idx]

-- proof that array size does not change upon modification
private def subtypeModify
        {α  : Type u}
        {n : Nat}
        (xs : { val : Array α  // val.size = n })
        (i : Nat)
        (elem: α  )
        : { a : Array α  // a.size = n } :=
  let val := xs.val.modify i (λ _ => elem)
  ⟨val, (Array.size_modify xs.val i (λ _ => elem )) ▸ xs.property⟩
```

Listing 2: State utility functions and tools

required by our subtype definition, using the `decide`[12] tactic.

In the second definition, we overload the indexing notation for the `State` value, which is a collection, by implementing an instance of the `GetElem` type class. In effect, this permits using a nicer notation to access a value of a given `State A`, with `A[i]` instead of `A.val.get i`, where `i` is an index that ranges from 0 to 24. Note that we will need to provide a proof that `i` is lower than 25, when we use either notation.

The third definition is more complex. Because modifying a Lean array may change the number of its elements, we must provide a proof that the `State` inner array `val` size did not change, if we want to return a modified `State` that conforms to its type definition. Function `subtypeModify()` returns this proof along with the modified state value, based on lemmas included in Lean.

3.2 Modeling Sponge Direction and Hash Parameters

One of the goals of this project was to attempt to make it more difficult for users of the SHA-3 library to inadvertently misuse it. For instance, one should not be able to compile code that absorbs data, after squeezing the sponge, as this is not a valid state transition. The latter case was enforced by the use of different sponge subtypes for absorbing and squeezing (see listing 3), and functions that would accept an allowed state, and return the next valid state. Callers passing a sponge to callees, and callees returning a sponge, will be responsible for providing proofs that the sponges are in the expected states.

```
def AbsorbingKeccakC : Type  :=
    {keccak : KeccakC // keccak.state = SpongeState.absorbing }

def SqueezingKeccakC : Type :=
   {keccak : KeccakC // keccak.state = SpongeState.squeezing }
```

Listing 3: Absorbing and squeezing sponge types

In an effort to catch other potential mistakes one could make in using the

[12] decide tactic: https://github.com/leanprover/lean4/blob/79428827b80255884d46fc422ed709e2f5427e57/src/Lean/Parser/Tactic.lean#L71

library, we tried to force various call chains that should be rejected. In one instance, our library implementation happily absorbed data in one primitive state e.g. SHA3-256, and squeezed data using another primitive e.g. SHA3-512, on this SHA3-256 state. We chose to use dependent types to remediate this. Recall that the SHA-3 family of functions differ in their handling of state by three parameters:

- capacity
- padding delimiter
- (default) output length.

For instance, SHA3-224, and SHA3-256 have the following capacity, padding delimiter, and output length values, respectively: 56, 6, 28, and 64, 6, 32.

We implemented the `HashFunction α β γ` type, which is parameterized with three values of arbitrary types for now, representing capacity, padding delimiter, and default output length as shown in listing 4. `HashFunction 56 6 28` is actually a type in the same way `UInt64` is, but it is distinct from `HashFunction 1 2 3`, another type instantiated from different capacity, padding delimiter, and output length values. We then augmented our sponge state types, namely `AbsorbingKeccakC`, `SqueezingKeccakC`, and their underlying type `KeccakC`, to depend on this newly created hash function type. We can now have type `AbsorbingKeccakC (HashFunction 56 6 28)` to represent a sponge for hash function SHA3-224, in absorbing state.

```
-- Sponge function, defined by its capacity, padding,
-- and output bit length
private inductive HashFunction
  {α : Type u}
  {β : Type v}
  {γ : Type w}
  : α → β → γ → Type (max u v w) where
    | f (capacity : α) (paddingDelimiter : β) (outputBitsLen : γ)
      (property :
        (capacity = c) ∧
        (paddingDelimiter = p) ∧  (outputBitsLen = o))
  : HashFunction c p o

def AbsorbingKeccakC (α : Type) : Type :=
  {keccak : KeccakC α // keccak.state = SpongeState.absorbing}

def SqueezingKeccakC (α : Type) : Type :=
  {keccak : KeccakC α // keccak.state = SpongeState.squeezing}
```

Listing 4: HashFunction type

After the initial implementation was publicly released, the Lean community suggested and implemented a simplification of the unneeded complexity of this design in a GitHub pull request[13], which was merged into the main branch of the project. It replaces the `HashFunction α β γ` dependent type with a simple `HashFunction` structure, and makes `KeccakC`, and other definitions

[13] GitHub pull request 1: https://github.com/gdncc/Cryptography/commit/d50aa61efab7d9ea9a6469f5a2a9c7616d5d0d24

dependent on the value of this new `HashFunction` structure only. See listing 5 for a sample extract of these changes.

```
private structure HashFunction where
  capacity : Capacity
  paddingDelimiter : Nat
  outputBitsLen : Nat

/-- The base cryptographic sponge context -/
private structure KeccakC (hf : HashFunction)  where
  A  : State
  state : SpongeState := SpongeState.absorbing
  rate : RateValue hf.capacity := ⟨ 0, by omega ⟩
  buffer  : FixedBuffer
  bufPos : RateIndex hf.capacity :=
         ⟨ 0, by simp [KeccakPPermutationSize]; omega ⟩
  outputBytesLen := 0
```

Listing 5: New definition of `HashFunction` and simplified definition of `KeccakC`

3.3 The Block Permutation

The SHA-3 permutation consists of the repetition of several steps θ (theta), ρ (rho), π (pi), χ (chi), and ι (iota). Lean conveniently allows defining function names using Greek symbols, at the expense of making it difficult to identify them in performance data collected by tools such as Linux `perf`[14]. Our first implementation was more or less a textbook algorithmic implementation of these steps, apart from the usage of native types (`UInt64` instead of arbitrary large `Nat` for instance). See listing 6, for the first algorithmic implementation of θ. Notice that for a pure functional programming language, Lean 4 permits to model mutations in a simple manner.

```
private def theta (A : State) : State := Id.run do
  let mut C : Array UInt64 := mkArray 5 0
  let mut D : Array UInt64 := mkArray 5 0
  let mut A := A

  for x in [:5] do
    C := C.set! x ((A.get! x) ^^^ (A.get! (x + 5)) ^^^
                   (A.get! (x + 10)) ^^^ (A.get! (x + 15)) ^^^
                   (A.get! (x + 20)))

  for x in [:5] do
    D := D.set! x ((C.get! ((x + 4) % 5))
                ^^^ ((((C.get! ((x + 1) % 5)) <<< 1) |||
                        ((C.get! ((x + 1) % 5)) >>> 63))))
    for y in [:5] do
      A := A.set! (x + 5 * y) ((A.get! (x + 5 * y) ^^^ (D.get! x)))

  A
```

Listing 6: Algorithmic implementation of θ step

We profiled the code to establish a performance baseline, and then attempted

[14]perf: https://en.wikipedia.org/wiki/Perf_(Linux)

to unroll the loops across all steps (see listing 7). This resulted in a significant performance increase, at the cost of longer source code.

```
private def theta (A : State) : State := Id.run do
  let C0 := (A.get! 0) ^^^ (A.get! 5) ^^^ (A.get! 10) ^^^
            (A.get! 15) ^^^ (A.get! 20)
  let C1 := (A.get! 1) ^^^ (A.get! 6) ^^^ (A.get! 11) ^^^
            (A.get! 16) ^^^ (A.get! 21)
  let C2 := (A.get! 2) ^^^ (A.get! 7) ^^^ (A.get! 12) ^^^
            (A.get! 17) ^^^ (A.get! 22)
  let C3 := (A.get! 3) ^^^ (A.get! 8) ^^^ (A.get! 13) ^^^
            (A.get! 18) ^^^ (A.get! 23)
  let C4 := (A.get! 4) ^^^ (A.get! 9) ^^^ (A.get! 14) ^^^
            (A.get! 19) ^^^ (A.get! 24)

  let mut D0 := C4 ^^^ ((C1 <<< 1) ||| (C1 >>> 63))

-- (Snip ...)

  A := A.set! 24 (A.get! 24 ^^^ D0)

  A
```

Listing 7: Unrolled loop implementation of θ step (abbreviated)

In the profiled code, we noticed a number of calls checking that the indices used to access elements of data structures were within bounds. Indeed, the `get!()` function accesses an element from a byte array, or panics if the index is out of bounds[15], so the Lean 4 runtime repeatedly checks whether this is the case or not.

Providing formal proofs that indices are within bounds permits eliding this runtime check in the compiled code. We decided to move to the next step, which was to provide these proofs. Our first attempt was to write a proof that accesses to the SHA-3 round constants table was within bounds. Recall that SHA-3 executes 24 rounds of a sequence of steps. In each round, the ι step performs a bitwise XOR operation between a per round constant stored in the round constants table, and the first element of the state array. The initial implementation of the ι step, along with how it is called by the Keccak-p function `keccalP()` is shown in listing 8. The Lean runtime checks that the table assignment `A.set!`, and read `A.get!` operations are in bounds.

We rewrote the ι step to accept a proof `h` where the round index `ir` is valid. We use the `roundConstants[ir]'h` notation to pass this proof to the body of ι. Therefore, the responsibility falls upon the caller to provide the proof. In function `keccakP`, the `for` loop provides a proof tuple `h` (using the Lean `ForIn'` type class) that `round` is an element of collection $[0, \mathit{roundConstants.size}\ )$, and specifically the second part of the tuple h_2 is a proof that `ir` is lower than the size of `roundConstants`. The code, updated to include the in bounds access proof, is presented in listing 9.

[15] The runtime behavior for an out of bounds access is a panic. These kinds of runtime-fallible methods are accounted for in Lean and do not cause soundness issues; they are defined as returning a prescribed default value on failure. To illustrate, kernel-reducing a bad array access via `Array.get!()` returns `Nat.zero`, the default element: `#reduce (#[] : Array Nat).get! (1 : Nat)`. Trying to evaluate generated code for a bad array access yields a runtime error `#eval (#[] : Array Nat).get! (1 : Nat)`.

```
private def iota
  (A : State)
  (ir : Nat)
  : State := Id.run do
  A.set! 0 ((A.get! 0) ^^^ (roundConstants.get! ir))

 /--
 The KECCAK-p[b, nr] permutation consists of nr iterations of:
 Rnd(A, ir) = iota(chi(π(ρ(theta(A)))), ir).
 -/

 private def keccakP (k : KeccakC α) : KeccakC α := Id.run do
   let mut A := k.A
  for round in [:k.numberRounds] do
    A :=  A |> theta |> rhopi |> chi |> (iota · round)
   {k with A := A}
```

Listing 8: ι and KECCAK-p implementation without proofs

```
-- iota
private def ι
  (A : State)
  (ir : Nat)
  (h : ir < roundConstants.size)
  : State := Id.run do
   A.set! 0 ((A.get! 0) ^^^ (roundConstants[ir]'h))

private def keccakP (k : KeccakC α) : KeccakC α := Id.run do
  let mut A := k.A
  -- KECCAK[c] number round nr := 24
  for h :  round in [:roundConstants.size] do
  -- h1 : col.start <= round, h2 := round < 25
    let ⟨_h₁, h₂⟩ := h
    A :=  A |> θ |> ρπ |> χ |> (ι · round h₂ )
  {k with A := A}
```

Listing 9: Access to round constants table with formal proof

Based on this success, we then attempted to provide in bounds access proofs for all accesses to the state array, and any temporary arrays in all steps of the KECCAK-p permutation. Unfortunately, we seemingly experienced a performance issue in Lean's elaborator, a component in charge of turning the user-facing syntax into a representation suitable for the rest of the Lean compiler. We filed an issue[16] on GitHub. Faced with this issue, we dropped our exploration on improving the performance of the implementation, and decided to focus on Lean 4 other strengths, including the usage of formal proofs to verify correctness of aspects of the implementation. We returned to the implementation of the step functions with loops, and provided the in bounds access proofs, as shown in listing 10.

For accessing values, note that we now use the same notation as the one we employed to access the round constants table e.g. `A[index]'(h)`, where `A` is the state, with collection indexing notation `A[]` permitted by the implementation of the `GetElem` type class, and `h` a proof. The additional difficulty in step θ

[16]Non-Linear Growth In Elaboration Time With A Number Of Local Vars Declared With Let: https://github.com/leanprover/lean4/issues/5324

```
theorem StateIndexWithinBounds521
  (index : Nat)
  (offset : Nat)
  (hCol : index ∈ [:5])
  (hOffset : offset < 21)
  : index + offset < 25 := by
  let ⟨ _h₁, h₂ ⟩ := hCol
  simp at h₂
  omega

theorem StateIndexWithinBounds55
  (index : Nat)
  (offset : Nat)
  (hCol : index ∈ [:5])
  (hOffset : offset ∈ [:5])
  : index + 5 * offset < 25 := by
  let ⟨ _hc₁, hc₂ ⟩ := hCol
  let ⟨ _ho₁, ho₂ ⟩ := hOffset
  simp at hc₂ ho₂
  omega

-- theta
private def θ (A : State) : State := Id.run do
  let mut C : Arr5 := ⟨ mkArray 5 0, by decide ⟩
  let mut D : Arr5 := ⟨ mkArray 5 0, by decide ⟩
  let mut A := A

  for hx : x in [:5] do
    C := subtypeModify C x
          ( A[x     ]'(StateIndexWithinBounds521 x 0 hx  (by trivial)) ^^^
            A[x + 5 ]'(StateIndexWithinBounds521 x 5 hx  (by trivial)) ^^^
            A[x + 10]'(StateIndexWithinBounds521 x 10 hx (by trivial)) ^^^
            A[x + 15]'(StateIndexWithinBounds521 x 15 hx (by trivial)) ^^^
            A[x + 20]'(StateIndexWithinBounds521 x 20 hx (by trivial)))
  for hx : x in [:5] do
    D := subtypeModify D x
          ( C[(x + 4) % 5] ^^^
            ((((C[(x + 1) % 5]) <<< 1) |||
            -- Lean's `%` is remainder, not modulo
            ((C[(x + 1) % 5]) >>> 63))))
    for hy : y in [:5] do
      A := subtypeModify A (x + 5 * y)
            ((A[(x + 5 * y)]'(StateIndexWithinBounds55 x y hx hy)  ^^^
            (D[x])))

  A
```

Listing 10: θ step implementation with access proofs

compared to the round constants table case is that the index varies differently e.g. `x`, `x + 5`, ..., `x + 20` in one instance, instead of just incrementing `x` by one. We moved some of the proof logic to several Lean 4 theorems to reduce repetition, and for clarity. Notice that in all θ step loops, as for the round constants table, we get a proof that both indexing variables `x` and `y` are within the interval $[0, 5)$. This proof is accessible in the scope delimited by `` `() ``. Lean permits inspecting the proof state, including what must be proven. Listing 11 shows the initial proof state for case `x + 5`, and the outstanding proof $x + 5 < 25$. Notice that proof `hx` was obtained from the `for` loop.

For this proof, we wrote theorem `StateIndexWithinBounds521`, which

```
A† : State
C† : Arr5 := ⟨mkArray 5 0, …⟩
D : Arr5 := ⟨mkArray 5 0, …⟩
A : State := A†
col† : Std.Range := { start := 0, stop := 5, step := 1 }
x : Nat
hx : x ∈ col†
r† : Arr5
C : Arr5 := r†
⊢ x + 5 < 25
```

Listing 11: θ step initial proof state for case `x + 5`

accepts a variable `x` (a natural number $\mathbb{N}$, an `offset` from `x` e.g. 5, another natural number, a proof `hCol` that `x` is in the interval $[0, 5)$, and a (trivial) proof `hOffset` that the `offset` is less than 21, which the Lean `trivial` tactic provides for us. In the lemma, we then decomposed the `hCol` proof to obtain h_2, which is a proof that `index` is less than 4. and specifically:

`index < { start := 0, stop := 5, step :=1 }.stop`.

We use the Lean `simp` tactic to rewrite h_2 as `h₂ : index < 5`. At this stage, our proof state is a system of simple inequalities, as shown in listing 12.

```
index offset : Nat
hCol : index ∈ { start := 0, stop := 5, step := 1 }
hOffset : offset < 21
_h₁ : { start := 0, stop := 5, step := 1 }.start ≤ index
h₂ : index < 5
⊢ index + offset < 25
```

Listing 12: θ step second-last proof state for case `x + 5`

The Lean `omega` tactic uses the two inequalities `hOffset : offset < 21`, and `h₂ : index < 5`, to solve the final goal `h₂ : index < 5`. All other in bounds read access proofs in the θ and other steps follow a similar pattern.

We continued using function `subtypeModify()`, presented earlier, to prove that mutation of the state underlying data structure values did not affect the size of this data structure (and that therefore further read and write accesses at the given index are still valid).

3.4 Sponge Construct Handling

We described how we implemented most of the SHA-3 state, sponge direction, hash parameters, and the core SHA-3 permutation function. We also formally proved all index-based accesses to the state data, and round constant table are in bounds, thus eliding runtime checks that accesses are in bounds. Then, we modeled getting data in, and out of our state in our sponge construct.

Our implementation of the construct went through several iterations. Early in the project, we abstracted common behavior for all hash, and extendable-output functions, and for one-shot and streaming APIs, using Lean type classes, and this design persisted. `Absorb` defines a generic absorption operation that takes a state, an input, and returns an updated state. `Squeeze` defines another generic squeeze operation that takes a state and an output length, and returns a product type consisting of the updated state, and the desired output (see listing 13). These generic operations should give leeway to specialize what kind of input and output data structures we can work with in future implementations, without API changes.

```
private class Absorb (α : Type) (β : Type) where
  absorb : α  → β →  α

private class Squeeze (α : Type) (β : Type) (γ : outParam Type) where
  squeeze : α  → β → γ
```

Listing 13: `Absorb` and `Squeeze` type classes

The original type classes instance implementations of these operations were again algorithmic in nature, and worked as expected. However, we encountered difficulties in providing in bounds access proofs for handling the internal temporary buffer, where input to be hashed is copied to in a round-robin basis in the absorption phase, and when copying data from the state to the output buffer in the squeeze phase. The reason for this is that we did not have enough information conveyed by the types we initially chose to use to write a suitable proof. It was akin trying to prove that for every natural numbers a and b, which can be seen as index and offset to a data structure of size 42, $a + b < 42$, which is false. Moreover in this inequality, the ceiling may vary depending on the hash parameters including the capacity. We therefore needed to carefully choose our types to restrict what potential values are permissible, and render the problem solvable.

Recall that the capacity is a parameter that differs depending of the hash function, and that the rate is derived from the width `b` of a Keccak-p permutation (200 bytes for all SHA-3 functions) as follows `rate = b - capacity`. We summarized these values in table 1.

Table 1: Capacity and rate of SHA-3 hash functions

Function	Capacity	Rate
SHA3-224	56	144
SHA3-256	64	136
SHA3-384	96	104
SHA3-512	128	72
SHAKE128	32	168
SHAKE256	64	136

Notice that the capacity does not exceed 128 bytes in all cases. We used the `Fin 129` type for the capacity, where `Fin n` is a natural number `i` with the constraint that `0 ≤ i < n`.

We declared a `FixedBuffer` subtype for the internal buffer, a byte array with a proof obligation that it has the size of the Keccak-p permutation. Recall that this buffer is made of two parts of different lengths, the capacity, followed by the rate section. We implemented dependent types `RateValue n`, `RateIndex n` which are based on the capacity and the Keccak-p permutation size. `RateIndex n` is an index into the rate component of the internal buffer `FixedBuffer` rate section. We also implemented a number of theorems to prove simple properties about these structures. Listing 14 shows the definitions of these newly defined types, and one sample theorem.

```
def KeccakPPermutationSize := 200

private abbrev RateValue (capacity : Capacity) :=
  Fin (KeccakPPermutationSize - capacity + 1 )

private abbrev RateIndex (capacity : Capacity) :=
  Fin (KeccakPPermutationSize - capacity )

private abbrev FixedBuffer :=
  {val : ByteArray // val.size =  KeccakPPermutationSize }

@[simp] theorem FixedBufferSize (fb : FixedBuffer)
  :  fb.val.size = KeccakPPermutationSize :=
  by exact fb.2
```

Listing 14: `RateValue n`, `RateIndex n`, and `FixedBuffer` dependent types

Introduction of these definitions required refactoring of several other structures, and functions. More importantly, it allowed conveying more fine-grained information in functions accessing the internal buffer. For example in function `absorb()` (listing 15), notice the following:

- The `AbsorbingKeccakC α n` structure depends on `n` of type `Capacity`, instead of `Nat` (ℕ), and thus allows deriving the domain of values for `bufPos` of type `RateIndex n`, for all the possible hash function capacity values.
- We implemented and used `fixedBufferModify()`, which like `subtypeModify()` for the state, proves that the buffer size does not change upon modification of the buffer. it depends on variable `bufPos`, for which we maintain a proof that it is always in bounds, even after increasing.
- In the secondary loop with index `j` from 0 to 24, we read 8 bytes from the buffer at an increasing `start` position, and provide a proof that the end positions (and therefore the start) are always in bounds, and specifically that 7 + `start` < `buffer.val.size`. Note that `start` is a `Nat`, and we are just proving that `start` + 7 will always be less than 200, the permutation width.
- As a side remark, `hi` contains the proof that index `i` is within bounds of `inputBytes`, allowing to use the notation `inputBytes[i]`, and therefore removing in bounds access checks on the input buffer as well.

Another function, `DomainDelimitAndPad101()` (listing 16) was also formally proven to have all access to the buffer in bounds. As specified in the FIPS PUB 202 standard, the total number of bytes denoted by `q` that are appended to a message in the internal buffer is determined by `m` and the rate `r`: `q` = (`r` / 8) − (`m` mod (`r` / 8)), which translates in our byte aligned implementation to `rate` - `bufPos`. Our function has slightly more complex inequalities to prove, including `rate` - `bufPos` < `KeccakPPermutationSize` - `n` + 1, in the case only one padding byte is required. We were able to write the proofs with the

```
private def absorb
  {α : Type}
  {n : Capacity}
  (k : AbsorbingKeccakC α n)
  (inputBytes : ByteArray)
  : AbsorbingKeccakC α n := Id.run do
  let mut k := k
  let mut buffer := k.val.buffer
  let mut bufPos := k.val.bufPos

  for hi : i in [:inputBytes.size] do
    if bufPos.val == k.val.rate.val - 1 then
      buffer := fixedBufferModify buffer ⟨ bufPos, by omega⟩  inputBytes[i]
      let mut A := k.val.A
      for hj : j in [:25] do
        let start := j <<< 3 -- lane size = 8
        A := subtypeModify A j ((A[j]) ^^^
               (FixedBuffer.toUInt64LE buffer start (by
                 let ⟨ _hj₁, hj₂  ⟩ := hj ;
                 simp at hj₂  ;
                 simp [KeccakPPermutationSize]; omega)))
      k := {k with val := keccakP
              {k.val with A := A, buffer := buffer, bufPos :=
                ⟨ 0, by simp [KeccakPPermutationSize]; omega⟩}}

    buffer := fixedBufferModify buffer ⟨ bufPos, by omega ⟩  inputBytes[i]
    bufPos := bufPos + ⟨ 1, by simp [KeccakPPermutationSize]; omega⟩

  {k with val := {k.val with buffer := buffer, bufPos := bufPos }}
```

Listing 15: Function `absorb()` with internal buffer in bounds access proofs

availability of richer type information, and with the assistance of the `omega` tactic.

Coming back to our usage of type classes, we implemented a single instance of the `Absorb` type class, which absorbs a byte array input for the `AbsorbingKeccakC α n` sponge, where `α`, and `n` are respectively, a specific SHA-3 function, and a capacity. We implemented two instances of the `Squeeze` type class, `squeezeAbsorbedInput()`, and `squeezeNotFullyAbsorbedInput()` to facilitate streaming and non-streaming APIs, and to prevent invalid states such that a squeezing sponge cannot be used to absorb again, using types. Specifically, `squeezeAbsorbedInput()` only accepts and returns `SqueezingKeccakC α n` sponges, whereas `squeezeNotFullyAbsorbedInput()` only accepts `AbsorbingKeccakC α n` sponges, and returns `SqueezingKeccakC α n` sponges.

3.5 Implementing the API

We implemented commonly expected APIs for traditional hash functions including:

- One-shot:
  - initialize
  - hash
- Streaming:

```
private def DomainDelimitAndPad101
  { n : Capacity }
  (buffer : FixedBuffer )
  (bufPos : RateIndex n)
  (rate : RateValue n)
  (paddingDelimiter : Nat)
  : FixedBuffer := Id.run do
  let mut buffer := buffer
  -- padding bytes required
  let q : RateValue n :=  ⟨rate - bufPos, by omega ⟩
  if hq : q == 1 then
    buffer := fixedBufferModify buffer ⟨ bufPos, by omega ⟩
                                (paddingDelimiter + 0x80).toUInt8
  else
    buffer := fixedBufferModify buffer ⟨ bufPos, by omega ⟩
                                paddingDelimiter.toUInt8
    for hi : i in [bufPos + 1 : rate - 1 ] do
      buffer := fixedBufferModify buffer
                                  ⟨ i, by
                                    let ⟨ _hi₁, hi₂⟩ := hi;
                                    simp at hi₂ ;
                                    omega ⟩  0
    buffer := fixedBufferModify buffer
                                ⟨ rate - 1, by simp [KeccakPPermutationSize];
                                  omega  ⟩
                                 (0x80).toUInt8
  buffer
```

Listing 16: Function `DomainDelimitAndPad101()` in bounds access proofs

- initialize
- update
- finalize

We only implemented the streaming API for the SHA-3 extendable-output functions, and used API function names such as `absorb()`, and `squeeze()`. We also found that we needed to implement another public function for the XOF streaming API, `toSqueezing()`, to facilitate entering mutating loop of the SHA-3 context, as exemplified in listing 17.

```
let mut ctx1 := SHAKE128.mk |> (SHAKE128.absorb · a) |> (SHAKE128.toSqueezing ·)
let mut s1 := ByteArray.mk $ mkArray 10 0
for _ in [0:3] do
  (ctx1, s1) := SHAKE128.squeeze ctx1 10
  -- (do something with s1 SHAKE output)
```

Listing 17: `toSqueezing()` sample usage

We used two Lean 4 macros to implement, and expose the public API in a consistent manner for XOF and hash functions (see listing 18 for the latter). This allowed to implement a hash function such as SHA3-224 with a simple statement, as shown in listing 19. In this example, `SHA3_224` is a name space, that exposes our functions, with the correct hash function type (we chose to represent the padding byte in hexadecimal notation, but this is not required).

The Lean 4 community later submitted a GitHub pull request[17] to com-

[17] GitHub pull request 2: https://github.com/gdncc/Cryptography/pull/2

```
macro "defhash" id:ident ":=" e:term  : command => `(
    def kf := $e
    def kfType := match kf with | .f c p o _h => HashFunction c p o
    def c : Capacity := match kf with | .f c _p _o _h => c

    instance : Absorb
               (AbsorbingKeccakC kfType c)
               ByteArray where
      absorb := absorb
    instance : HashFunctionParameters
               kfType
               (Capacity × Nat × Nat) where
      params := params
    instance : Squeeze
               (SqueezingKeccakC kfType c)
                Nat
               (Id (SqueezingKeccakC kfType c × ByteArray)) where
      squeeze  := squeezeAbsorbedInput
    instance : Squeeze
               (AbsorbingKeccakC kfType c)
               Nat
               (Id (SqueezingKeccakC kfType c × ByteArray)) where
      squeeze  := squeezeNotFullyAbsorbedInput

  namespace $id
    def $(mkIdent `final)
      (s : AbsorbingKeccakC kfType c)
      : ByteArray  :=
      (Squeeze.squeeze s s.val.outputBytesLen).2

    def $(mkIdent `update)
      (s : AbsorbingKeccakC kfType c)
      (bs : ByteArray)  :=
        Absorb.absorb s bs

    def $(mkIdent `mk) := mkKeccakC kf c

    def $(mkIdent `hashData)
        (bs : ByteArray)
        : ByteArray :=
        let k : AbsorbingKeccakC kfType c := mkKeccakC kf c
          (Squeeze.squeeze (Absorb.absorb k bs) k.val.outputBytesLen).2
  end $id
)
```

Listing 18: `defhash` macro

```
defhash SHA3_224 := mkHashFunction 56 0x06 28
```

Listing 19: Implementing the SHA3-224 hash function using the `defhash` macro

pletely remove the use of macros to implement the hash, XOF APIs and functions (see listing 20). The main idea is that we don't need to define constructors, and other API functions for each hash/XOF SHA-3 function; we can just define one `HashFunction` type and constructor, namespaces for the respective APIs, and Lean resolves the method notation for us. This designs manages to make the code shorter than the equivalent macro code, and is much easier to debug.

the pull request was merged to the main branch of the project.

```
-- Implement the hash and xof function APIs.

namespace HashFunction

def final
  {hf : HashFunction} (s : AbsorbingKeccakC hf) : ByteArray  :=
  (Squeeze.squeeze s s.val.outputBytesLen).2

def update {hf : HashFunction}
  (s : AbsorbingKeccakC hf) (bs : ByteArray)  :=
  Absorb.absorb s bs

def hashData
  {hf : HashFunction} (bs : ByteArray) : ByteArray :=
  let k : AbsorbingKeccakC hf := hf.mk
  (Squeeze.squeeze (Absorb.absorb k bs) k.val.outputBytesLen).2

end HashFunction

def XOF := HashFunction

namespace XOF

def toSqueezing
  {xof : XOF} (k : AbsorbingKeccakC xof) : SqueezingKeccakC xof :=
  (Squeeze.squeeze k 0).1

nonrec def absorb
  {xof : XOF} (s : AbsorbingKeccakC xof) (bs : ByteArray)
  : AbsorbingKeccakC xof :=
  absorb s bs

def squeeze
{xof : XOF} {α : Type} [Squeeze α Nat ((SqueezingKeccakC xof) × ByteArray)]
    (k : α) (l : Nat) : ((SqueezingKeccakC xof) × ByteArray)  :=
  Squeeze.squeeze k l

end XOF

-- Implement our hash, and xof functions
def SHA3_224 : HashFunction := HashFunction.ofParams 56 0x06 28
def SHA3_256 : HashFunction := HashFunction.ofParams 64 0x06 32
def SHA3_384 : HashFunction := HashFunction.ofParams 96 0x06 48
def SHA3_512 : HashFunction := HashFunction.ofParams 128 0x06 64
def SHAKE128 : XOF := HashFunction.ofParams 32 0x1f 32
def SHAKE256 : XOF := HashFunction.ofParams 64 0x1f 64
```

Listing 20: Implementation of the hash, XOF APIs, and functions without macros

Usage of the APIs is demonstrated[18] in the `example` build target of the implementation.

[18] Example API usage: https://github.com/gdncc/Cryptography/blob/main/Cryptography/Hashes/SHA3/example.lean

4 Testing

4.1 Correctness

We detailed how we proved that all access to the input data, the SHA-3 round constants table, the internal buffer, and state are in bounds, and how we enforced a prohibition on undue absorb and squeeze alternation. Lean 4 also did not report being unable to determine whether any implemented function is terminating. Therefore, we did not have to explicitly provide any termination proof, and, more importantly, Lean was able to prove that all our functions are terminating.

We implemented the Secure Hash Algorithm-3 Validation System (SHA3VS)[19] test vectors including the short, long message, and pseudorandomly generated messages (Monte Carlo) tests for all implemented hash and XOF functions, and in addition, the variable-length output tests for all XOFs. We did not decide on a suitable test framework, so the tests only output whether they pass or fail.

Of interest, we made use of the built-in Lean 4 parser combinator library `Parsec` to extract all test vectors and pass them to the SHA-3 library.

4.2 Performance

To measure the performance impact of changes to the code, we initially wrote a foreign function interface (FFI) wrapper around the Read Time-Stamp Counter (RDTSC) instruction on x86 processors using Lean FFI facilities. In the process we erroneously omitted to use an `IO` monad, resulting in incorrect timestamps being returned. Lean is a pure functional programming language, and assumes that functions return the same result when called with the same input so such result can be re-used, and/or the functions may be called in arbitrary order. Returning an `IO` monad instructs Lean to carry the effectful operation, and to use the result of this operation. We later discovered that Lean provides the `IO.monoNanosNow` function, which returns monotonically increasing time since an unspecified past point in nanoseconds. We removed our RDTSC wrapper, and used this function instead.

We implemented one performance test, SHAKE128 absorption speed of 32B, 1KB, and 1MB chunks. Our implementation is 2 orders of magnitude slower than the results reported by the `xoflib`[20] benchmark, which wraps the native `sha3` Rust crate[21]. Table 2 summarizes the absorption performance results on a 2.3GHz quad-core 10th-generation Intel Core i7 processor.

Table 2: Absorption performance of 32B, 1KB, and 1MB chunks

Implementation	32B	1KB	1MB
xoflib	110 MB/s	344 MB/s	381 MB/s
Lean	1.29 MB/s	2.93 MB/s	2.97 MB/s

Several Lean users have had difficulties instrumenting performance of Lean

[19] SHA3VS: https://csrc.nist.gov/csrc/media/projects/cryptographic-algorithm-validation-program/documents/sha3/sha3vs.pdf

[20] xoflib: https://github.com/GiacomoPope/xoflib/tree/main

[21] Crate `sha3`: https://docs.rs/sha3/latest/sha3/

programs in the past. For reference, we were able to use to capture performance data using the Linux perf `perf` tool on our target binary:

```
perf record -g --call-graph=dwarf Cryptography-Hashes-SHA3-perftest
```

Listing 21: Calling `perf` to gather performance data

Adding the following options to our performance test binary target, and the cryptography library in our project `lakefile.lean`, which contains the configuration that `lake` needs to build the application, provided more fine-grained information:

```
moreLeancArgs := #["-UNDEBUG", "-O3", "-ggdb",
                   "-g3", "-fno-omit-frame-pointer"]
```

Listing 22: `lakefile.lean` configuration data for performance profiling

Hotspot[22] can display the resulting data for analysis. We did not spend much effort analyzing the data in this project.

We measured the impact of inlining functions on x86 platform, and kept inlining declarations that showed consistent performance increase using declaration modifiers `always_inline`, and `inline`. The latter declaration appears[23] to be conditional on some other information, but the logic is currently not implemented in the Lean compiler. For now, using both modifiers is redundant, but they are not mutually exclusive.

5 Potential Issues When Implementing Cryptography

Readers may have noticed a surprising behavior in function `absorb()` (previous listing 15) with variable `bufPos` of type `RateIndex n`, which is in effect a finite (`Fin`) type; the variable is never explicitly set to zero, and silently wraps around zero. Lean currently implements the addition operation modulo n, as illustrated in listing 23.

This does not appear to be a problem in our implementation but this is likely going to be a source of bugs in systems using the `Fin` type, where data operations may not lead to the desired values. Non-overflowing operations could be added to the `Fin` type, or an alternative data structure that provides such operations could be developed. We modified our `absorb()` function implementation to explicitly set `bufPos` to zero when needed, and provided a proof that the value will not wrap around, as shown in listing 24.

Another potential issue to be mindful of, and which is not specific to Lean 4, is that any strong abstractions such as type classes permit to create APIs that are generic. We explained in a previous section that we could absorb data in one cipher primitive state, and squeeze data using another primitive in an early implementation. Lean 4 makes it easy to work with generic abstractions, and one should be careful that it does not permit unwanted interactions.

[22]Hotspot: https://github.com/KDAB/hotspot

[23]Function inlining: https://leanprover.zulipchat.com/#narrow/channel/113488-general/topic/Function.20inlining/near/480299272

```
/--
Returns `a` modulo `n + 1` as a `Fin n.succ`.
-/
protected def ofNat {n : Nat} (a : Nat) : Fin (n + 1) :=
  ⟨a % (n+1), Nat.mod_lt _ (Nat.zero_lt_succ _)⟩

/--
Returns `a` modulo `n` as a `Fin n`.

The assumption `NeZero n` ensures that `Fin n` is nonempty.
-/
protected def ofNat' (n : Nat) [NeZero n] (a : Nat) : Fin n :=
  ⟨a % n, Nat.mod_lt _ (pos_of_neZero n)⟩

-- We intend to deprecate `Fin.ofNat` in favor of `Fin.ofNat'` (and later rename).
-- This is waiting on https://github.com/leanprover/lean4/pull/5323
-- attribute [deprecated Fin.ofNat' (since := "2024-09-16")] Fin.ofNat

private theorem mlt {b : Nat} : {a : Nat} → a < n → b % n < n
  | 0,   h => Nat.mod_lt _ h
  | _+1, h =>
    have : n > 0 := Nat.lt_trans (Nat.zero_lt_succ _) h;
    Nat.mod_lt _ this

/-- Addition modulo `n` -/
protected def add : Fin n → Fin n → Fin n
  | ⟨a, h⟩, ⟨b, _⟩ => ⟨(a + b) % n, mlt h⟩
```

Listing 23: Lean `Fin` type implementation

Lean operations on convenient data types such as `Fin`, which is backed by `Nat` are not constant-time, and this can have disastrous consequences in cryptography. For instance, the `Nat` type is unboxed up to 63 bits, then Lean uses the `GMP`[24] library beyond that[25]. The switch from unboxed to boxed values and vice-versa can reveal information about the size of the operands and result. The `GMP` library itself is not constant-time, and may also reveal information about these.

6 Concluding Notes And Future Work

We implemented the SHA-3 family of functions in Lean 4, and described our experience. In this process, we gained a better understanding some salient features of Lean including but not limited to dependent types, and formal verification, and how to apply them. We also learnt more about the SHA-3 algorithm and the sponge construction. For the latter, actually providing proofs for some aspect of the implementation permitted us to gain additional insights that we initially missed.

During the course of our project, we encountered a performance issue with Lean's elaborator, preventing us from proving properties of better performing code. We then chose to focus on program verification, instead of improving performance.

[24] GMP: https://gmplib.org/

[25] `Nat` implementation: https://leanprover.zulipchat.com/#narrow/channel/270676-lean4/topic/.E2.9C.94.20Type.20erasure.2Fcompilation.20questions/near/473374900

```
abbrev Capacity := Fin 129

def KeccakPPermutationSize := 200

abbrev RateIndex (capacity : Capacity) :=
  Fin (KeccakPPermutationSize - capacity )

theorem RateIndexLTBlockMinCap
  {n : Capacity}
  (ri : RateIndex n)
  : ri < KeccakPPermutationSize - n := by
  omega

theorem RateIndexLTBlockMinCapMinOne
  {n : Capacity}
  (ri : RateIndex n)
  (h1 : ¬(ri == KeccakPPermutationSize - n - 1) = true)
  : ri + 1 < KeccakPPermutationSize - n := by
  simp at h1
  have h2 : ri < KeccakPPermutationSize - n := (by exact RateIndexLTBlockMinCap ri)
  refine Nat.add_lt_of_lt_sub ?h
  omega

private def absorb
  {α : Type}
  { n : Capacity}
  (k : AbsorbingKeccakC α n)
  (inputBytes : ByteArray)
  : AbsorbingKeccakC α n := Id.run do
  let mut k := k
  let mut buffer := k.val.buffer
  let mut bufPos := k.val.bufPos
  for hi : i in [:inputBytes.size] do
    if hif : bufPos.val == KeccakPPermutationSize - n - 1 then
      buffer := fixedBufferModify buffer ⟨ bufPos, by omega⟩  inputBytes[i]
      let mut A := k.val.A
      for hj : j in [:25] do
        let start := j <<< 3 -- lane size = 8
        A := subtypeModify A j ((A[j]) ^^^
                    (FixedBuffer.toUInt64LE buffer start (by
                        let ⟨ _hj₁, hj₂  ⟩ := hj;
                        simp at hj₂;
                        simp [KeccakPPermutationSize];
                        omega)))
      k := {k with val := keccakP
                  {k.val with A := A, buffer := buffer, bufPos :=
                    ⟨ 0, by simp [KeccakPPermutationSize]; omega⟩}}
      buffer := fixedBufferModify buffer ⟨ bufPos, by omega ⟩  inputBytes[i]
      bufPos := ⟨ 0, by simp [KeccakPPermutationSize]; omega⟩
    else
      buffer := fixedBufferModify buffer ⟨ bufPos, by omega ⟩  inputBytes[i]
      bufPos := RateIndex.add bufPos
                     ⟨ 1, by simp [KeccakPPermutationSize]; omega⟩
                     (by exact RateIndexLTBlockMinCapMinOne bufPos hif)

  {k with val := {k.val with buffer := buffer, bufPos := bufPos }}
```

Listing 24: `absorb()` function updated to prevent silent wrapping of `bufPos` value

We have not explored other avenues of potential interest:

- Lean 4 code is ultimately transformed into machine language and we lose formally proven properties in the process. It would be extremely useful to be able to parse the generated assembly code, and to prove properties about it.
- We proved that access to several data structures including the state was in bounds. We did not prove, nor did we test that input bytes are injected at the correct place in the state when using the streaming API. The Rust FN-DSA crate tries to catch issues in its SHAKE implementation test suite[26] by running each test vector twice, once as a single chunk, and then again by injecting it one byte at a time. Proving that input bytes are injected at the correct location would prevent the rare bugs that this Rust implementation test is looking for in the first place.
- In the current API implementation, one could inadvertently duplicate a state, and use the wrong state instance later to absorb and squeeze data. Implementing the typestate pattern or similar, to ensure that a previous state cannot be reused, could be of interest.
- Implementing the duplex construction in addition to the sponge construction would permit to access and experiment with a wider variety of cryptographic functions.

7 Acknowledgments

We thank Thomas Pornin, Eli Sohl, and the anonymous reviewers for their useful comments and suggestions.

References

[1] Nicky Mouha and Christopher Celi. "A Vulnerability in Implementations of SHA-3, SHAKE, EdDSA, and Other NIST-Approved Algorithms". en. In: 13871. CT-RSA 2023: Cryptographers' Track at the RSA Conference, San Francisco, CA, US, 2023. DOI: https://doi.org/10.1007/978-3-031-30872-7_1. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=936243.

[2] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28*. Ed. by André Platzer and Geoff Sutcliffe. Cham: Springer International Publishing, 2021, pp. 625–635. ISBN: 978-3-030-79876-5.

[26]: FN-DSA: https://github.com/pornin/rust-fn-dsa/blob/main/fn-dsa-comm/src/shake.rs#L862-L891

[3] The mathlib Community. “The Lean Mathematical Library”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. POPL ’20: 47th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. New Orleans LA USA: ACM, Jan. 20, 2020, pp. 367–381. ISBN: 978-1-4503-7097-4. DOI: 10.1145/3372885.3373824. URL: https://dl.acm.org/doi/10.1145/3372885.3373824 (visited on 11/13/2024).