Fiat-Shamir for ICICLE

Karthik Inbasekar,[a]

E-mail: karthik@ingonyama.com

Abstract: Fiat-Shamir Transformation for ICICLE

1 Purpose

The Fiat–Shamir transformation [1] uses a one way (hash) function to convert a public-coin interactive protocol between a prover and a verifier into a non-interactive protocol. The core principle of the transformation is that the verifier's random messages can be replaced with suitable outputs of a secure hash function. This enables practical real world application of Zero Knowledge Proofs in non interactive mode.

This work mainly follows the DSFS (Duplex Sponge Fiat-Shamir) prescription for Fiat-Shamir Transformation as laid out for Sponge hash functions in [2] and the corresponding implementation [3]. We also consult the hash API definitions used in the SAFE prescription (for Duplex sponges) laid out in [4] and [5]. In the past (For sumcheck and FRI) we have used the IOP specific (Interactive Oracle Proof) pattern described by the Merlin library [6] and STROBE [7].

The goal is to move towards a generic Fiat-Shamir IOP pattern which allows gluing of arbitrary ICICLE primitives to create ZKP protocols. The implementation [3] is written in rust and will undergo formal verification. The main features of spongefish are as follows.

- Main principles are templated in terms of a sponge hash function, without depending on details of the specific hash implementation.
- Sponge (size $n = r + c$,where r: rate, c: capacity) operates in duplex (absorb/squeeze methods) function, in overwrite mode. In over write mode, the hash replaces within the first r (rate) bits or field elements with the input from the user, while the capacity part is inaccessible. This follows the design aspects emphasized in sponge API design [5].
- The squeeze challenge mode only reads from the r part of the hash, and the hash chaining is maintained by the capacity part of the sponge (which is inaccessible). It follows that we do not need to hash chain previous challenges (like in sumcheck or FRI), which makes sponges more efficient compared to non-sponge hash modes for Fiat-Shamir.
- Clear separation of meta data and data by enforcing formatted Domain separator strings. The meta data such as labels are treated as schema and are part of the

Initialization vector only, while during run time, the sponge hash **only absorbs actual data** generated by the prover which makes it more efficient.

- The prover can in addition (optionally) maintain a protocol bound private sponge (unrelated to Fiat Shamir) for producing randomness related to zero knowledge. **Note:** sponge based randomness based on duplex sponge design can be used in ICICLE as a standalone CSPRNG (Cryptographically Secure Pseudo Random) generator.

Some Guidelines are:

- The theory description follows [2] and the reference implementation is here [3].
- The original implementations are in rust, and we need to adapt it to $c++$ and ensure compatibility with different backends.
- The rust wrappers should closely match [3] interface so existing rust users of spongefish could easily make the switch.
- Future works need to make this lattice compatible. It mostly affects the Pseudorandom generator in §4.3 for which we need rejection sampling, I have not researched it, and I believe given our current direction in lattices/pqc, this could be important. (TO DO)

We organize the article as follows

- Classic Fiat-Shamir §2: We review Fiat-Shamir and its Hash chain version. This is educattional, for understanding the background and how Fiat-Shamir arguments are structured in general. Mostly theory and can be skipped if one is familiar.
- Duplex Sponge 3: We review definitions and properties of Duplex sponge. This needs to be a design class for all hash functions that support it in ICICLE. The Fiat-Shamir depends on this definition of sponge operation. **Must read** and check how all ICICLE hashes can be used in this mode.
- Duplex Sponge Fiat-Shamir §4. Once the sponge is defined, DSFS is simply defining generic classes that can organize functionalities.
- Example: Sumcheck §5,
- Security summary §6 for best practices.
- Flow diagrams are available here (please use drawio app). Provides a complete visualization of the whole process, for sponge operation, and prover verifer interaction with the sponge.

2 Classic Fiat-Shamir

We use the same notation of [2] as much as possible. A Fiat-Shamir transformation maps an Interactive Protocol $\mathbf{IP} = (\mathbf{P}, \mathbf{V})$ for a relation $\mathcal{R}$ to a non interactive argument (proof) $\texttt{NARG}= (\mathcal{P}, \mathcal{V})$ for the same relation $\mathcal{R}$ (within a given random oracle model). The $\mathcal{P}$ is referred to as the argument prover (non-interactive) and $\mathcal{V}$ as the argument verifier (non-interactive). While P and V are referred to as **IP** prover and **IP** verifier respectively (interactive).

- **Oracle:** In theory: One specifies a distribution such as uniform rand, and a list of oracles f_i is sampled. In practice: f_i is a hash function that has good pseudo random properties and security features such as collision resistance and pre-image resistance.
- **Argument Prover:** $\mathcal{P}(x, w)$ takes input instance x and witness w and is able to query oracles f_i and outputs a proof string π that consists of prover messages α_i. The Fiat-Shamir protocol simulates the interaction between **IP** prover $P(x, w)$ and **IP** verifier V interaction by causally deriving verifier messages ρ_i from prover messages α_i and queries to the oracle f_i
- **Argument Verifier:** $\mathcal{V}(x, \pi)$ takes input instance x, proof string π, is able to query oracles f_i and outputs a decision bit 1/0 (valid/invalid proof). The Fiat-Shamir protocol simulates the interaction between **IP** verifier $V(x, \pi)$ and **IP** verifier V interaction by causally deriving verifier messages ρ_i from prover messages α_i and queries to the oracle f_i.

There are two common variants of the classic Fiat-Shamir that are implemented:

- The basic $\mathbf{FS}[IP]$ where the **IP** verifier message ρ_i in the i'th round in a $n > i$ round protocol, is derived from a random oracle f_i

$$\rho_i = f_i(x, \alpha_1, \alpha_2, \ldots \alpha_i) \tag{2.1}$$

  This is described in fig 1.

- The hash chain version $\mathbf{HCFS}[IP]$ where **IP** verifier message ρ_i in the i'th round is derived by hashing the **IP** prover message α_i with the verifier message ρ_{i-1} in the previous round

$$\rho_i := \begin{cases} f_i(x, \alpha_1), & i = 1 \\ f_i(\rho_{i-1}, \alpha_i), & i > 1 \end{cases} \tag{2.2}$$

  In fig 1 the hashing pattern is replaced with (2.2).

The main disadvantage with the theoretical models in this section is that it leaves lot of dependence on the specific IP used, in particular the domain and range of the hash functions is unfixed. In practice we use a fixed oracle (a single hash function) and initiate multiple hash calls per IP (one for each round and one for initializations). In sumcheck and FRI we have used the HCFS model with a single fixed hash (that is fixed on initiation of the IP) which forced us to

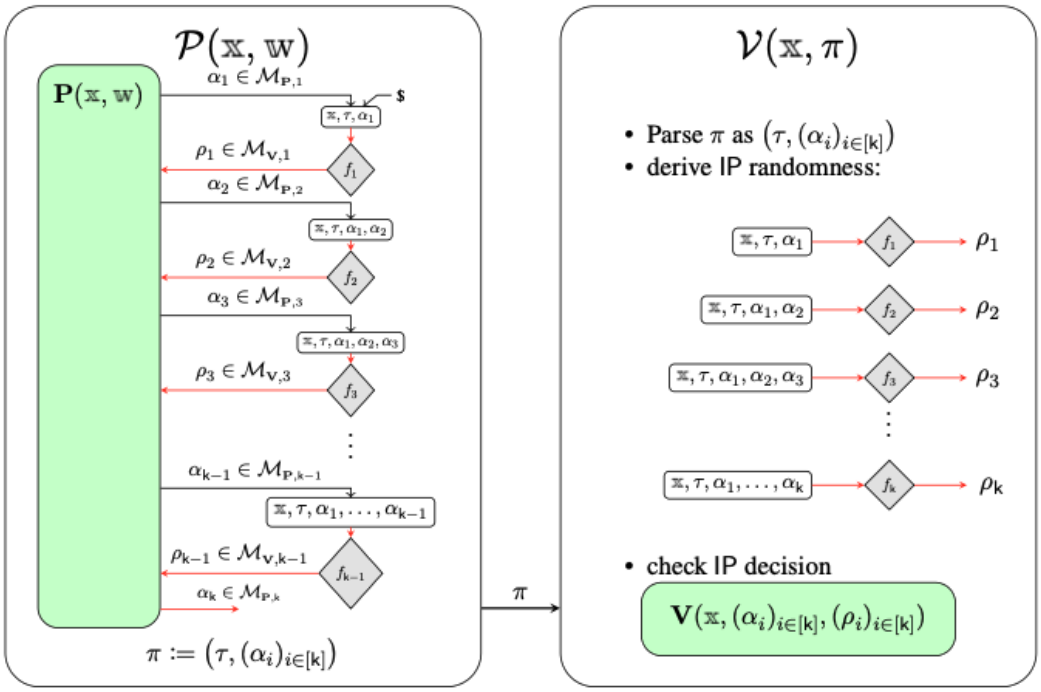

Figure 1. The standard FS[IP] model, in the above x is the instance, w the witness, $\tau \in \mathbb{N}$ is a random salt (seed rng) , $\mathcal{P}(x, w)$ is the argument prover, f_i are the oracles (hash), α_i are the prover messages, and ρ_i are the challenges. Fig taken from [2]

- Include meta data in every hashing in run time.
- include previous round challenges (manual chaining)
- We structure the input as $[metadata||data]$ and we had to skip over the metdata every time.

Most of these steps are quite redundant, and what is truly important is that the sequence of operations send message, get challenge etc is maintained. Spongefish solves a lot of these problems by introducing a sponge with an overwrite buffer, and keeping the hash chain internal by allowing read/write only from part of the buffer. Which means in run time, we dont care about anything other than the actual pure data produced by the prover. We describe this in the next section.

3 Duplex Sponges in Overwrite mode

We provide the description of the duplex sponge as in the sponge fish paper [3] and the code for references wherever possible. We will deal with the specifics of Spongefish Fiat-Shamir in the next section §4. First we define the state of a duplex sponge as a $n = r + c$ byte non empty alphabet (string) Σ^{r+c} that is divided into two parts

- **Rate** r: These are Σ^r bytes (convention in spongefish is first r bytes) that are used for absorbing user data or squeezing out data for the user.
- **Capacity** c: These are the remaining $c = n - r$ bytes that are never exposed to the user except during initialization. This makes the sponge hashes, stateful objects. These bytes maintain context and binding of the sequence of hash states.
- **Permute:** Every sponge has a mathematical algorithm called a permutation P (keccak, poseidon2, blake etc) that acts on Σ^{r+c} with a given unit U i.e $(u8, u16 \dots)$ etc).

Note that the ordering of "rate" bytes first and "capacity" bytes last is considered to be a design optimization to avoid skipping over "c" bytes every time when one wants to overwrite "r" bytes. The reason it is called duplex is because the sponge supports both reading and writing from the buffer.

The following operations are part of the duplex interface. The permute function is wrapped by the Duplex Sponge operations defined below. We will define all of these steps in detail further.

- **State initialization** §3.1 : The state is initialized to (see algorithm 2) a default value.
- **Absorb** §3.2 : The basic algorithm 2 where user messages are absorbed into the hash state in a specified prescription.
- **Squeeze** §3.3. In the squeeze operation bytes are squeezed **only from the rate part** of the state. Due to overwrite property, any number of bytes can be squeezed out.
- **Ratchet:** §3.4 Ratchet function clears the state following prescribed methods.

The entire sequence of operations is summarized in figure 2. The methods used in the duplex sponge interface are

```rust
pub trait DuplexSpongeInterface<U = u8>: Default + Clone + zeroize::Zeroize
where
    U: Unit, //Fields or bytes
{
    /// Initializes a new sponge, setting up the state.
    fn new(iv: [u8; 32]) -> Self;
    /// Absorbs new elements in the sponge.
    fn absorb_unchecked(&mut self, input: &[U]) -> &mut Self;
    /// Squeezes out new elements.
    fn squeeze_unchecked(&mut self, output: &mut [U]) -> &mut Self;
    /// Ratcheting to reset state
    /// - permuting the state and zeroize rate elements
    fn ratchet_unchecked(&mut self) -> &mut Self
}
```

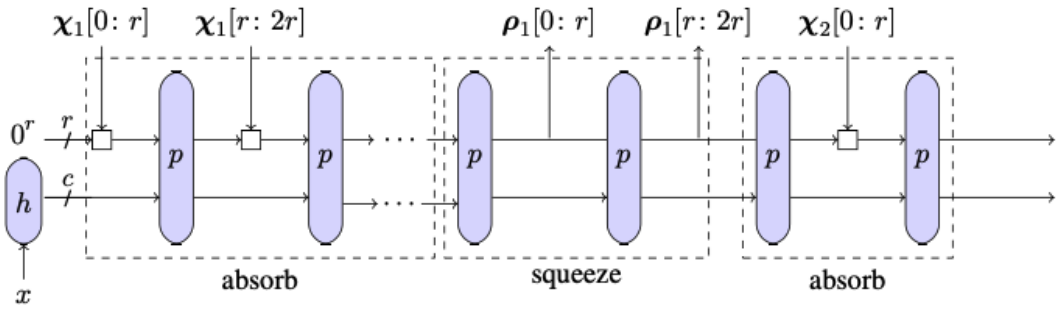

Figure 2. Duplex sponge in overwrite mode. Fig taken from [2]

The ratcheting operation is used to clear state from previous absorption values.

3.1 Initialize state

In algorithm 1 we summarize the prescription to initialize the state. The duplex sponge has a default starting value $\Sigma = [0^n]$ and we initialize the sponge defining a initialization vector, based on domain separator or some other string provided by the user. The prescription for Fiat-Shamir Domain separator will be covered later in §4.1. In this section, we keep it general.

- Bitwise hashes:

$$[0^r||IV_{32}||0^{n-r-32}] \tag{3.1}$$

  where IV_{32} is the initialization vector of 32 bytes. If user inputs $IV.len() > 32$ bytes, we absorb it with the hash initialized by zeros, and squeeze a 32byte output. Then we reset the initial state of the hash to be as above, and only use IV of length 32 bytes.

- For Finite Field hashes, the random initialization byte is mapped to a single field element $IV_{\mathbb{F}} = IV \mod p$ using (A.1)

$$[0^r||IV_{\mathbb{F}}||0^{n-r-1}] \tag{3.2}$$

  where IV_p is obtained from bytes and reduced to mod order. This ensures that we always get a **single unique field element** for the given bytes irrespective of their original byte length. We describe in appendix a method to get this consistently all the time with minimal bias §A.

This is a universal definition for the initialization for any duplex sponge hash. Note: Take into account that sponging may be used for Fiat-Shamir or sometimes just to produce short digests.

1. For basic sponge operations (digesting independent of Fiat-Shamir protocol), use a standard string (user specified) as a domain separator. There are no restrictions for

Algorithm 1 DS.Start(x) $\to st_0$, rate: r, capacity c, state size $n = r + c \in \mathbb{N}$, message: $x \in \{0,1\}^n$, Domain separator $h : \{0,1\}^{\leq n} \in \Sigma^c$, unit: U, initial state $st_0 : (\Sigma, i_A, i_S)$

$\Sigma \leftarrow [0U, n]$
if $U \in \{0,1\}$ **then** ▷ bitwise hashes
  $IV_{32} \leftarrow [0u8; 32]$
  $DS.Absorb(\Sigma, x.bytes())$
  $IV_{32} \leftarrow DS.Squeeze(\Sigma, 32bytes)$ ▷ This should be 32 bytes now
  $\Sigma = [0^r || IV_{32} || 0^{n-r-32}]$ ▷ Initial state for protocol sponge
else if $U \in \mathbb{F}$ **then** ▷ Finite Field hashes
  $IV_{\mathbb{F}} \leftarrow 0_{\mathbb{F}}$
  $x_{\mathbb{F}} \leftarrow \psi(x.bytes()) \in \mathbb{F}$ ▷ See (A.1) , byte to bigint mod reduce
  $IV_{\mathbb{F}} \leftarrow DS.Absorb(\Sigma, x_{\mathbb{F}})$
  $\Sigma = [0^r || IV_{\mathbb{F}} || 0^{n-r-1}]$ ▷ Initial state for protocol sponge in $\mathbb{F}$
end if
Set $i_A = 0$ ▷ Set Absorbing index =0
Set $i_S = r$ ▷ Set Initial Squeeze index at r
return st_0 ▷ Sponge state class stores the state, and index values

formatting the string other than the IV length is $32bytes$ or a single field element converted from these bytes.

2. For Fiat-Shamir specific, the domain separator is restricted and we define in detail how to restrict it in §4.1.

In the next few subsections, we describe the "unchecked" algorithms where the hasher does not do data type checks. The "checked" algorithms are implemented as wrapped checks around the unchecked algorithms an implement standard error messages.

3.2 Absorb

Given the initial sponge state Σ^{r+c}, the input block χ is sequenced in $[\chi_0[0:r]||\chi_1[r:2r]||\dots||\chi_k[r:t]]$ with $0 < t \leq r$ where the sponge will try to fill messages in sequence of size r. If χ_k is less than size r it doesn't zero pad, just writes whatever it has into the state and leaves the rest of the rate part as it was in the existing state. This is a deliberate choice for efficiency and does not affect security.

This is different from way sponges are traditionally applied and is known as overwrite mode. In overwrite mode, the m_i are directly written **only on the rate part** (no XORing just overwrite), and permute is applied everytime the absorb index $i_A = r$ reaches rate size. i.e if we have $[\chi_1||\chi_2||\chi_3]$ where we would apply

$$\begin{aligned}
\Sigma.start &= [0^r||IV||0^{n-r-32}] \\
\Sigma.absorb &= [\chi[0:r]||IV||0^{n-r-32}]\ ,\ \Sigma' = \Sigma.P() \\
&= [\chi[r:2r]||\Sigma'[n-r:r]\ ,\ \Sigma'' = \Sigma'.P() \\
&= [\chi[2r:t]||\Sigma''[n-t:r] = \Sigma'''
\end{aligned} \tag{3.3}$$

note that in each sequence it completely overwrote the "rate" part, and since absorb index reaches $i_A = r$ it also applies permute. In the last message, the size $t < r$ (say), then the absorb index is at $i_A = t < r$ and hence no permute is applied.

This process continues on un-till it reaches the end of the message. The Absorb algorithm is summarized in algorithm (2). The checked algorithm can be found here.

Algorithm 2 DS.Absorb$^P(st, \chi) \rightarrow st$, State: $st = (\Sigma, i_A, i_S)$, message χ, permutation : P, the resulting st is the output state

while $\chi.len() != 0$ **do**
  if $st.i_A = r$ **then**
    $st.\Sigma.P()$ ▷ Permute happens only when the index reaches the rate size
    $st.i_A = 0$
  else if $st.i_A < r$ **then**
    $l = min(\chi.len(), r - st.i_A)$ ▷ Chunk length
    $(chunk, rest) = \chi.splitat(l)$
    $st.\Sigma[i_A : i_A + l] \leftarrow chunk$ ▷ No Xoring or field operation, just overwrite
    $i_A += l$
    $\chi = rest$
  end if
end while
$st.i_S = r$
return st

3.3 Squeeze

Squeeze is the operation, where the prover can read challenges from the rate part of the sponge. If the number of squeeze bytes required is $t < r$

$$\Sigma^{r+c}.squeeze = c[0:t] \ , \ \Sigma^{r+c} \rightarrow \Sigma^{r+c} \tag{3.4}$$

the next squeeze operation if any continues from the index $i_S = t$ until it reaches r. (see §3. When the squeeze bytes index reaches $i_S = r$, a permute operation is applied. For example, if the number of bytes required is $t = 2r + 4$

$$\begin{aligned} \Sigma^{r+c}.squeeze &= \Sigma[0:r] = c_1 \ , \ \Sigma' = \Sigma.P() \\ &= \Sigma'[0:r] = c_2 \ , \Sigma'' = \Sigma'.P() \\ &= \Sigma''[0:4] = c_3 \ , \Sigma''^{r+c} \end{aligned} \tag{3.5}$$

The final user bytes are $[c_1||c_2||c_3]$ of size $2r + 4$ and the state remains as it is at Σ'' with the squeeze index at $i_S = 4$. If there is a subsequent squeeze request it continues from 4. If there is a subsequent absorb operation, the state is **overwritten on the rate part only** as described in absorb mode. The pseudo code for the Squeeze algorithm is given by algorithm 3. The checked wrapper can be found here.

Note: The hash state is not explicitly reset between sequential squeeze operations. If required by the prover, they needs to call a ratchet function §3.4.

Algorithm 3 DS.Squeeze$^P(st, out : U) \rightarrow (st, out)$, $st = (\Sigma, i_A, i_S)$ U unit, out is an output buffer in U units, eg bytes or field elements

if out=Null **then** ▷ termination condition: When all empty buffers are filled
  return st, out
end if
if $st.i_S = r$ **then**
  $st.\Sigma.P()$ ▷ Whenever squeeze index reaches rate size, permute state
  $st.i_S = 0$, $st.i_A = 0$
end if
assert$(st.i_S < r)$
$l = min(out.len(), r - st.i_S)$ ▷ bytes that need to be squeezed out (all or remaining)
$(out, rest) = out.splitat(l)$
$out \leftarrow st.\Sigma[i_S : i_S + l]$ ▷ Copy bytes from state to output
$i_S += l$
DS.Squeeze$^P(st, rest : U)$ ▷ Recurse till the requested number of bytes are met

3.4 Ratchet

The Ratchet is a state clearing algorithm, generally we dont use it in a regular sponge, but in protocols that require forward secrecy or state erasure it may be called to clear the state. It essentially,

- Permutes the state
- Zeros out **ONLY** the "rate" elements - The rate elements are the part of the state that is directly exposed during Absorb/Squeeze.

The unchecked ratchet algorithm is given by algorithm 4. The checked wrapping can be found here. In particular, ratcheting can be invoked on an incomplete block. By incomplete block, what we mean is that entries in the state are not filled by new values. Note that the state size never changes, so what ratcheting accomplishes is state memory erasure while keeping context.

Algorithm 4 DS.ratchet(st) State $st : (\Sigma, i_A, i_S)$

$st.\Sigma.P()$
$st.\Sigma[0 : r] \leftarrow 0^r$ ▷ Zeroize rate
$st.i_S = r$ ▷ Reset squeeze index
return: st

3.5 Summary

A sample of the sponge operation for different scenarios is summarized in a diagram 3. If figure is not clear, it can also be accessed from here (please use drawio app)

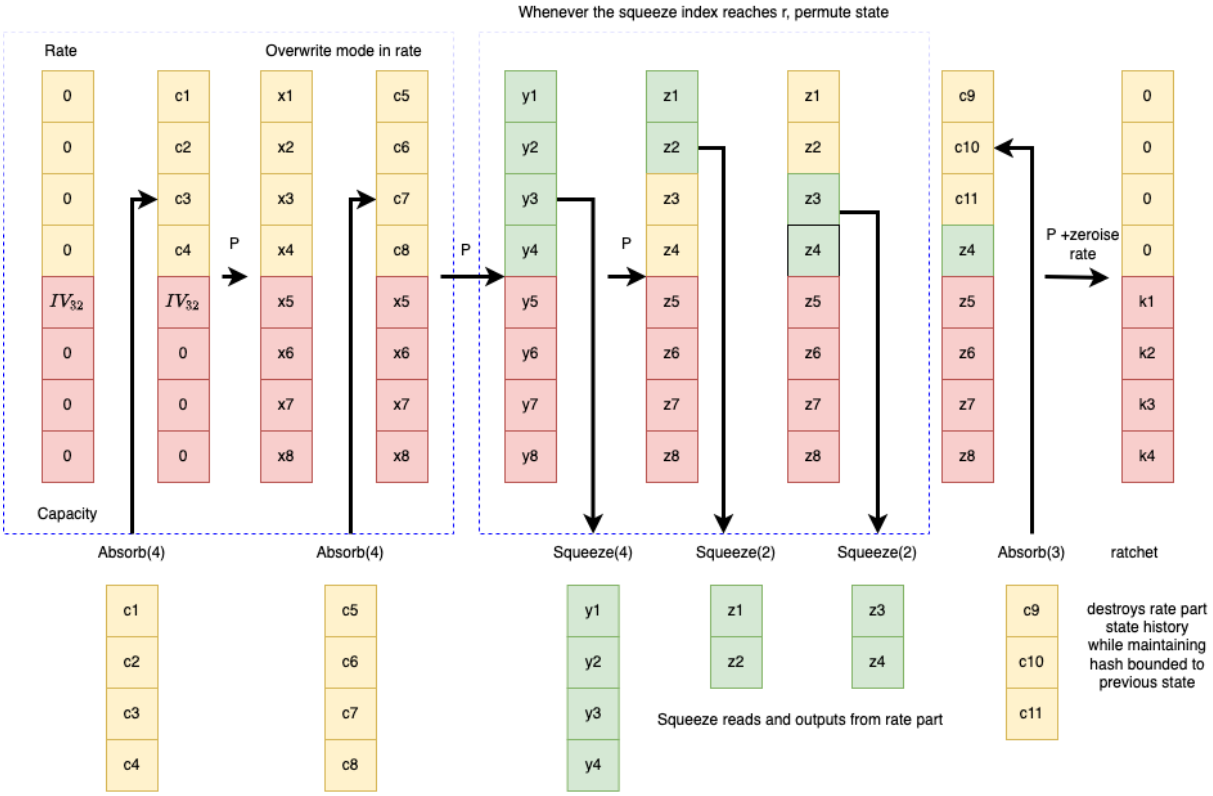

Figure 3. Sponge operation: The hash chaining, happens by making the capacity part inaccessible after initialization. In this sense, the sponge is a stateful object. The P is the permutation algorithm of the relevant hash. The absorb, writes only on the rate part, and overwrites pre existing data. If the absorb index reaches the maximum (rate size), it triggers a permute. Squeeze reads only from the rate part, and if the squeeze index reaches maximum (rate size) , it triggers a permute. Ratchet can be used to destroy rate history while keeping hash chain intact.

4 DSFS

The Duplex sponge Fiat-Shamir has the general format as described in fig 4. The legend below defines all the symbols used in the diagram.

- x: instance
- $\tau \in \mathbb{N}$: salt/seed rng
- $\mathcal{P}(x, w)$: non interactive argument (NARG) prover
- $\mathcal{V}(x, w)$: non interactive argument (NARG) verifier
- α_i: NARG Prover message,
- ψ_i: encoding map user data type to le-bytes, (serialization)
- $\hat{\alpha}_i = \psi_i(\alpha_i)$: encoded prover message in le-bytes that will be absorbed by the sponge,

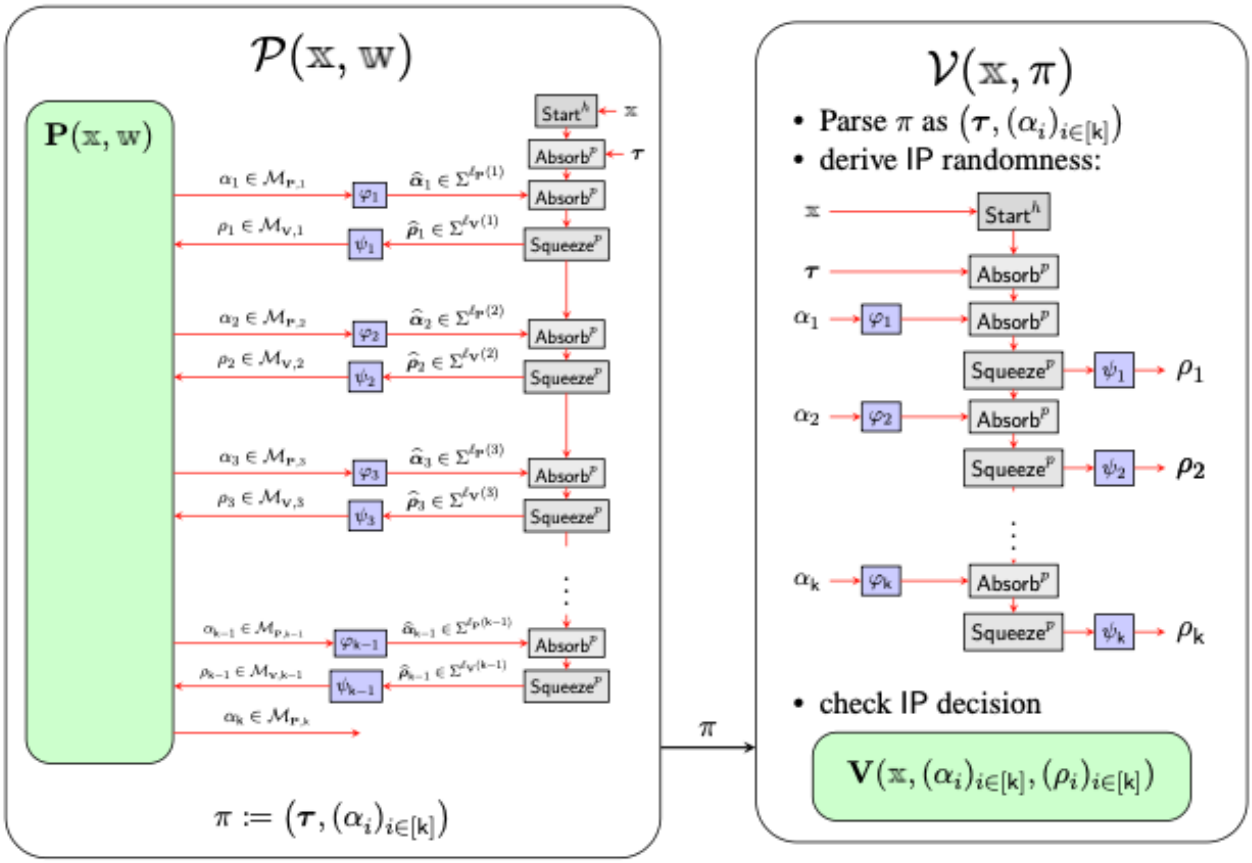

Figure 4. DSFS protocol: The prover and verifier states represent the corresponding operations

- $\hat{\rho}_i$: squeeze le-bytes from sponge or equivalently verifier challenge
- $\hat{\psi}_i$: decoding map from le-bytes to user data type (de-serialization)
- $\rho_i = \hat{\psi}_i(\hat{\rho}_i)$: decoded verifier message into user data type.
- $\pi = (\tau, \alpha_i)$ proof tuple, with seed rng, and prover messages.

There are three main definitions that need to be followed.

1. Define a Domain separator §4.1 data common to prover and verifier. Allows uniqueness per protocol and prevents cross protocol attacks. The domain separator is a formatted string, that the prover and verifier must agree apriori.

2. Define a Stateful-hash construct §4.2, that processes the domain separator string to identify runtime hash instructions. The stateful hash construct runs the sponge §3 privately and responds to requests. The interaction with the sponge via absorb/squeeze/ratchet do not access the capacity part by construction.

3. Define a prover state §4.3(definition constructors for protocols, seed rng, methods that custom provers can use, maintains proof string, interact with stateful hash construct to absorb/squeeze/ratchet from hash).

4. define a verifier state §4.4 (der). (definition constructors for protocols, accepts proof string, methods that custom verifiers can use, read methods for proof string, interact with stateful hash construct to absorb/squeeze/ratchet from hash)

4.1 Domain Separator

The domain separator structure is used to create and manage domain separator strings in protocols. As we saw earlier in §3 there are three runtime algorithms Absorb 2, Squeeze 3 and Ratchet 4. The domain separator string is formatted in a specific way that the entire set of operations in Fiat-Shamir for the specific protocol is known at the time of initiation of the protocol.

The set of information needed for encoding of this string is summarized below. The prover and verifier need to construct the same encoded string for the exact sequence of Fiat-Shamir sponge hashing for completeness.

- The data type U (e.g., bytes, base field, extension field, group element), which is a type parameter in the implementation.
- Protocol unique identifier string (domain seperator label),
- For each prover message: its label and size (number of elements to absorb).
- For each verifier message (challenge): its label and size (number of elements to squeeze).
- The sequence and number of operations (Absorb, Squeeze, Ratchet) as required by the protocol.

The prover and verifier are expected to have a "contract" where these details are agreed upon and is encoded in a formatted string.

```
pub struct DomainSeparator<H = crate::DefaultHash, U = u8>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
{
    /// Encoded domain separator string representing the sequence of sponge
    ↪ operations.
    io: String,
    /// Marker for the sponge hash function and unit type used.
    _hash: PhantomData<(H, U)>,
}
```

The struct creates a valid domain separator string x, whose lengths are consistent with data types described in the protocol. No information about the data type is explicitly stored in the io string of a domain separator, although it is enforced at an API level.

The format to encode a domain seperator string x is explained below.

```
x = inital_string|| O_1||O_1elementslen||O_1label||
↪  O_2||O_2elementslen||O_2label|| ...
```

- O_i are the operations Absorb (A), Squeeze (S) , Ratchet (R) and they appear in the formatted string (or tag) as

  ```
  pub enum Op {
      /// In a tag, absorb is indicated with 'A'.
      Absorb(usize),
      /// In a tag, squeeze is indicated with 'S'.
      Squeeze(usize),
      /// In a tag, ratchet is indicated with 'R'
      Ratchet,
  }
  ```

  The sequence of O_i specify the binding sequence of sponge operations performed in the protocol both for prover and for verifier (i.e they need to initialize with the same formatted string).

- $O_i elementslen$ following a O_i are the number of elements for that operation, **the format of the size is the number of elements in base 10**.

- $O_i label$ is the label for that operation O_i. **NOTE: The label cannot start with a digit or contain the NULL byte**. This is because it is used in spongefish as a formatting operation to indicate " ".

  ```
      const SEP_BYTE: &str = "\0";
  ```

- The formatted Domain separator is a schema or protocol contract, for both prover and verifier states. If either the prover or verifier change the sequence of operations, the contract breaks and an error is produced.

- As we saw in algorithm 2 The domain separator string x is processed by the given sponge to create the IV of the hash by algorithn 1 $DS.Start(x)$. So all the metadata and operator sequence in the formatted string is processed to set the IV in the initial state of the hash. This frees up any need for keeping track of meta data in run time.

4.1.1 example 1

An example for a domain separator for keccak is

```
let tag = "Domain-separator\0A32generator\0A32publickey\0R\0A32commitment\0S32
↪  challenge\0A32response";
// This generates the domain seperator string
Domain-seperator A32generator A32publickey R A32commitment S32challenge
↪  A32response
```

here 32 stands for the message length which is 32 bytes. The string is parsed, and the sequence of instructions for the sponge apart from initialization are determined using the stateful hash object interface §4.2

```
ops=vec![Op::Absorb(32), Op::Absorb(32), Op::Ratchet, Op::Absorb(32),
↪  Op::Squeeze(32),Op::Absorb(32)]
```

These are the actual instructions in the exact sequence that the sponge hash is allowed to do during protocol run time. **No other sequence can be performed by the prover or the verifier, if so, it should result in an error.**

Another way to achieve the encoding is to use explicit methods (see here) (which appears a bit error free but can become very cumbersome.)

```
let domain_separator =
↪  DomainSeparator::<hash>::new("Domain-separator").absorb(32, "generator").a
↪  bsorb(32,"publickey").absorb(32,"commitment").ratchet().squeeze(32,
↪  "challenge").absorb(32,"response");
```

For vectors of field elements, say we want to add 3 babybear field elements (base field), we put in the full size

```
// BabyBear has 31-bit modulus: bytes_modp(31) = 4
// 3 scalars * 4 bytes = 12
```

our domain seperator string for absorbing 3 babybear field elements with message "commitment" in a protocol with initialization string "Domain-separator" would look like

```
let domain_seperator = "Domain-separator\0A12commitment"
```

4.1.2 example 2

The formatted domain separator x, ensures that for a given protocol, this is deterministic and unique sequence. Note that the op sequence is label unaware, and will be the same

independent of the labels for a given x. For example,if we lool at two domain separator strings $ds1, ds2$

```
    fn test_domain_separator_absorb_and_squeeze2() {
        let ds1 = DomainSeparator::<H>::new("proto")
            .absorb(2, "input")
            .squeeze(1, "challenge");
        let ops1 = ds1.finalize();
        assert_eq!(ops1, vec![Op::Absorb(2), Op::Squeeze(1)]);

        let ds2 = DomainSeparator::<H>::new("proto2")
        .absorb(2, "input2")
        .squeeze(1, "challenge2");
    let ops2 = ds2.finalize();
    assert_eq!(ops2, vec![Op::Absorb(2), Op::Squeeze(1)]);
    assert_eq!(ops1,ops2);

    ///BUT the byte representation of the DS string is different! and will
    ↪  result in different values of the IV
    assert_eq!(ds1.bytes,ds2.bytes)
    }
```

However the byte representation of the DS strings will be different

```
assertion left == right failed
  left: [112, 114, 111, 116, 111, 0, 65, 50, 105, 110, 112, 117, 116, 0, 83,
  ↪  49, 99, 104, 97, 108, 108, 101, 110, 103, 101]
 right: [112, 114, 111, 116, 111, 50, 0, 65, 50, 105, 110, 112, 117, 116, 50,
 ↪  0, 83, 49, 99, 104, 97, 108, 108, 101, 110, 103, 101, 50]
```

This is also the catch 69 of this spongefish approach, i.e the user provided string is hashed to either a 32 byte IV or a singe Field element IV and set as the initial state of the duplex sponge. So now once a prover or a verifier agreed to a set of operations and labels, changing the labels or the sequence of operation can be identified easily, since their IV would be different.

The prover state §4.3 and verifier state §4.4 instances can also be derived from the fixed Domain separator using the into functions. This may be easier to do for application developers who can ensure that both prover and verifier for a given protocol will always start with the same inital sponge hash state (in frameworks, like gnark, jolt or for eg our sumcheck/fri prover/verifier). But in most cases, it is prover/verifier responsibility.

4.2 Stateful hash object

The process of converting the formatted string into a sequence of sponge operations is achieved by the stateful hash object interface. This helper class is used by both prover §4.3 and verifier §4.4 to interface with duplex sponge §3. The Stateful hash only handles the

sponge hash operations at run time of protocol. The intialization is done always by the domain separator, if there are any absorb/squeeze operations at definition due to domain separator length $>$ 32bytes, that is not handled/checked by stateful hash.

```rust
/// A stateful hash object that interfaces with duplex interfaces.
#[derive(Clone)]
pub struct HashStateWithInstructions<H, U = u8>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
{
    /// The internal duplex sponge used for absorbing and squeezing
    ↪ data.Already initalized in domain separator. it comes with either Sigma
    ↪ =[0^r,IV_{32},0^{n-r-32}] or [0^r,IV_F, 0^{n-r-1}] depending on
    ↪ bytes/fields.
    ds: H,
    /// A stack of expected sponge operations
    stack: VecDeque<Op>,
    /// Marker to associate the unit type `U` without storing a value.
    _unit: PhantomData<U>,
}
```

The stack is processed by popping the instructions and identifying the specific sponge operation needed. **Note:** The stack enforces the protocol transcript structure, and any deviation (e.g., out-of-order operations) should result in an error.

1. **new:** Initializes from domain separator's finalized stack
2. **absorb:** Pops/checks/updates stack for absorb operations
3. **squeeze:** Pops/checks/updates stack for squeeze operations
4. **ratchet:** Pops/checks stack for ratchet operation
5. **Drop:** Warns if stack is not empty at destruction
6. **Debug:** Prints stack for debugging
7. **Tests:** Asserts stack state after operations

The Hash state with instructions code, implements the above by reading a stack vector of operations processed from the Domain Separator string and calling the relevant sponge operation. Below we show only ratchet for example.

```rust
impl<U: Unit, H: DuplexSpongeInterface<U>> HashStateWithInstructions<H, U> {
/// Initializes from domain separator's finalized stack |
    pub fn new(domain_separator: &DomainSeparator<H, U>) -> Self {
    }
```

```
/// Pops/checks stack for ratchet operations. (example)
    pub fn ratchet(&mut self) -> Result<(), DomainSeparatorMismatch> {
        match self.stack.pop_front() {
            Some(Op::Ratchet) => {
                self.ds.ratchet_unchecked();
                Ok(())
            }
            Some(op) => Err(format!("Expected Ratchet, got {op:?}").into()),
            None => Err("Expected Ratchet, but stack is empty".into()),
        }
    }
/// Pops/checks stack for absorb operations.
    pub fn absorb(&mut self, input: &[U]) ->Result<(), DomainSeparatorMismatch>
/// Pops/checks stack for squeeze operations.
    pub fn squeeze(&mut self, output: &mut [U]) -> Result<(),
    ↪  DomainSeparatorMismatch>
```

Which are processed by the Duplex Sponge hash interface §3. Feel free to design stateful hash or squash everything in domain separator as per icicle needs. I found this a useful control separation from a design pov.

4.2.1 example

Let us revisit the example we saw in the previous section §4.1

```
let domain_separator =
↪  DomainSeparator::<hash>::new("Domain-separator").absorb(32, "generator").a
↪  bsorb(32,"publickey").absorb(32,"commitment").ratchet().squeeze(32,
↪  "challenge").absorb(32,"response");
```

The formatted string

```
"Domain-separator\0A32generator\0A32publickey\0R\0A32commitment\0S32challenge\
↪  0A32response";
```

is parsed, and the sequence of operations is determined using the stateful hash object interface §4.2

```
ops=vec![Op::Absorb(32), Op::Absorb(32), Op::Ratchet, Op::Absorb(32),
↪  Op::Squeeze(32),Op::Absorb(32)]
```

The actual set of operations for the sponge is (see algorithms §1 §2, §3, and §4

```
///Assume Sigma is initialized with DS.start(x)
```

```
/// Prover can call these from the prover state with
↪  addscalar/addpoints/generate challenge/ratchet methods.
DS.Absorb(Sigma, input1)
DS.Absorb(Sigma, input2)
DS.Ratchet(Sigma)
DS.Absorb(Sigma,input3)
DS.Squeeze(Sigma, output: 32bytes)
DS.Absorb(Sigma,input4)
//At end of protocol destroy state
DS.ratchet()
Sigma.clear()
```

Note:

- the stateful hash constructor maintains that for this given protocol, the only allowed instructions are the above.
- If any prover/verifier tries to add more data or remove more data than in the sequence we should throw an error.
- Note that the sponge doesnt care about the metadata such as the label, since these are taken into account at design.
- What is relevant for cryptographic security is the actual message and the sequence of the messages, hash chaining is done internally due to sponge design, so we dont need to worry about it.

4.3 Prover state

ProverState is a core struct in the spongefish library, representing the state of the prover in an interactive proof (IP) system. A very clear data flow diagram for this is given in here.

In spongefish it is defined by

```
pub struct ProverState<H = DefaultHash, U = u8, R = DefaultRng>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
    R: RngCore + CryptoRng,
{
    /// The randomness state of the prover.
    pub(crate) rng: ProverPrivateRng<R>,
    /// The public coins for the protocol
    pub(crate) hash_state: HashStateWithInstructions<H, U>,
    /// The encoded data.
    pub(crate) narg_string: Vec<u8>,
}
```

- H : Hasher used for the protocol
- U : Unit for hashing (field/bytes) etc.
  - Byte-based sponge (u8): $DomainSeparator ::< Keccak, u8 >:: new(...)$
  - Field-based sponge: Field element size.
- R : specify a cryptographically secure random generator (csrng) eg $rand :: CryptoRng, RngCore$
- **ProverPrivateRng:** This is strictly optional. Can be easily substituted with any commerically available CSPRNG. It contains the prover's private randomness for Zero Knowledge. This hash is different from the Fiat-Shamir hash, in fact it can be any hash that can produce a safe PRNG like SHA256/Keccak or whatever we think is best for ICICLE. In protocols like Groth16, and plonk sometimes randomness not derived from transcript is used, these randomness are generated by provers private RNG. In particular this hash does not need a formatted Domain separator string, since its purpose is to produce pure randomness. If we use a sponge for this, all the methods absorb/squeeze/ratchet are applicable with *csrng* as the IV value.

```
pub struct ProverPrivateRng<R: RngCore + CryptoRng> {
    /// The duplex sponge that is used to generate the random coins.
    pub(crate) ds: Keccak,
    /// The cryptographic random number generator that seeds the sponge.
    pub(crate) csrng: R,
}
```

This has methods that fill a buffer with random bytes, and can be used to generate seed randomness unique to prover state. **Note:** We recommend using this precise prescription for prng generation especially for fill bytes since it erases its state by ratcheting after squeezing. So it is impossible to reverse engineer the state from the random value. Note that proverprivateRng can be used as a standalone sponge based CSPRNG (Cryptographically Secure Pseudo Random Generator).

```
impl<R: RngCore + CryptoRng> RngCore for ProverPrivateRng<R> {
    fn next_u32(&mut self) -> u32 {
        let mut buf = [0u8; 4];
        self.fill_bytes(buf.as_mut());
        u32::from_le_bytes(buf)
    }

    fn next_u64(&mut self) -> u64 {
        let mut buf = [0u8; 8];
        self.fill_bytes(buf.as_mut());
        u64::from_le_bytes(buf)
    }
```

```
    fn fill_bytes(&mut self, dest: &mut [u8]) {
        // Seed (at most) 32 bytes of randomness from the CSRNG
        let len = usize::min(dest.len(), 32);
        self.csrng.fill_bytes(&mut dest[..len]);
        self.ds.absorb_unchecked(&dest[..len]);
        // fill `dest` with the output of the sponge
        self.ds.squeeze_unchecked(dest);
        // erase the state from the sponge so that it can't be reverted
        self.ds.ratchet_unchecked();
    }

    fn try_fill_bytes(&mut self, dest: &mut [u8]) -> Result<(),
    ↪  rand::Error> {
        self.ds.squeeze_unchecked(dest);
        Ok(())
    }
}
```

- **Hash State with instructions**: §4.2 Contains a hash function state (sponge construction) that is synchronized with the verifier's state for Fiat-Shamir transforms. i.e When verifier uses the hashwithinstructions for the relevant protocol, they will arrive at the same state. Can use all methods of the stateful hash: absorb/squeeze/ratchet.
- **NARG string**: This does the transcript Management by maintaining a protocol transcript (the NARG string), which records the sequence of messages (e.g., group elements, scalars, bytes) sent by the prover. The protocol transcript does not have any information (metadata) about the length or the type of the messages being read. This is because the information is pre-shared by specifying the formatted string for the DomainSeparator. The NARG string does not store in addition the verifier challenges, since these are supposed to be deterministically generated. In short, any protocol using Fiat-Shamir, will read off the relevant proof elements from the NARG string and serialize it in anyway they want.
- **Security**: Does not implement Clone or Copy to prevent accidental leakage of secret state.

For a given protocol, A prover state is built with a given domain separator, this needs a constructor §4.1, which takes as input a hasher H, a data type U and a rng generator R.

```
impl<H, U, R> ProverState<H, U, R>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
```

```
    R: RngCore + CryptoRng,
{
    pub fn new(domain_separator: &DomainSeparator<H, U>, csrng: R) -> Self {
        let hash_state = HashStateWithInstructions::new(domain_separator);

        let mut duplex_sponge = Keccak::default();
        duplex_sponge.absorb_unchecked(domain_separator.as_bytes());
        let rng = ProverPrivateRng {
            ds: duplex_sponge,
            csrng,
        };

        Self {
            rng, ///the rng that appears in new is seed rng
            hash_state,
            narg_string: Vec::new(),
        }
    }
    ///seed rng is prover specific and corresponding verifier can access it
    ///or can be added to proof.
    pub fn rng(&mut self) -> &mut (impl CryptoRng + RngCore) {
        &mut self.rng
    }
    ///NARG string can be read by prover at any time
    pub fn narg_string(&self) -> &[u8] {
        self.narg_string.as_slice()
    }
```

or use from methods to create directly from domain separator. For eg

```
impl<U, H> From<&DomainSeparator<H, U>> for ProverState<H, U, DefaultRng>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
{
    fn from(domain_separator: &DomainSeparator<H, U>) -> Self {
        Self::new(domain_separator, DefaultRng::default())
    }
}
```

Constructor methods for adding different types of data need to be defined by user using methods defined in prover state. In Spongefish the used method is to have a generic functionality and implement codecs for different libraries. In our case, we recommend to implement explicit add units for different data types, in the prover state definition.

```
    impl<H, U, R> ProverState<H, U, R>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
    R: RngCore + CryptoRng,
pub fn add_units(&mut self, input: &[U]) -> Result<(),
↪  DomainSeparatorMismatch> {
        let old_len = self.narg_string.len();
        ///update hash state with prover input first.
        self.hash_state.absorb(input)?;
        ///update the proof string (NARG) with prover input next.
        U::write(input, &mut self.narg_string).unwrap();
        //This next step is needed for protocols that need ZK (pure unbounded
        ↪  PRNG)
        ///for protocols that need ZK, the rng generating sponge is synced with
        ↪  the prover messages. But for every random squeeze from rng sponge,
        ↪  the state history is erased. This is critical for ZK security.
        self.rng.ds.absorb_unchecked(&self.narg_string[old_len..]);

        Ok(())
}
```

Note: The hash state always absorbs the prover messages, and the verifier would reconstruct the same hash state. The rng sponge is used only for protocols that need ZK, and in general it stays bound to the prover messages. We can also use the rng sponge once in initialization to generate a secure seed rng or leave it to user to do it themselves.

Public inputs can be added in the same way using the above constructor, these are common to prover and verifier. These are absorbed in the hash state but we do not need to update the NARG string. In particular, we can include it as part of domain separator construction to set IV. Spongefish implements a different trait called unittranscript for this purpose. But instead for simplicity one could implement these methods directly in the prover state and also in the verifier state. I think it is mostly an attempt to enforce common API interface.

```
impl<H, U, R> UnitTranscript<U> for ProverState<H, U, R>
where
    U: Unit,
    H: DuplexSpongeInterface<U>,
    R: RngCore + CryptoRng,
{
    /// Add public messages to the protocol transcript.
    /// Messages input to this function are not added to the protocol
    ↪  transcript.
    /// They are however absorbed into the sponge for Fiat-Shamir, and used to
    ↪  re-seed the prover state.
```

```
    fn public_units(&mut self, input: &[U]) -> Result<(),
    ↪  DomainSeparatorMismatch> {
        let len = self.narg_string.len();
        self.add_units(input)?;
        /// this is because they defined add_units for prover state to do both
        ↪  update hash state and update NARG string
        self.narg_string.truncate(len);
        ///public data is not added to proof string
        Ok(())
    }

    /// Fill a slice with uniformly-distributed challenges from the verifier.
    /// this is the actual challenge generator
    fn fill_challenge_units(&mut self, output: &mut [U]) -> Result<(),
    ↪  DomainSeparatorMismatch> {
        self.hash_state.squeeze(output)
    }
```

For specific types of messages, we need helper constructors with serialization built in for Vectors of field elements base or any extension, vectors of group elements G1/G2.

- **Modifying the transcript** (NARG string) directly is not allowed; only methods defined in the class or implementations of the class (like addunits, addbytes, addpoints, addscalars etc.) which call the sponge operations absorb, can append to it. Note that the instructions also predefine exactly what the prover/verifier can do.
- The prover sends the NARG string to the verfier at the end of the protocol so it can be read by verifier and processed in a deterministic way.

In summary we should define a prover state class that does all the following methods for data addition/interacting with the Sponge/prng. Note that this is the only way any protocol can access or make requests to the DSFS hash which is not exposed.

```
//Action: Prover sends polynomial coefficients or vector of Field elements
 add_scalars(&[F])
///Function: Absorb a slice of field elements (scalars) into the transcript.

//Action: Prover sends commitments of group elements.
add_points(&[G])
///Function: Absorb a slice of group elements (points)

// Action: Prover sends arbitrary data
add_bytes(&[u8])
/// Function: Absorb raw bytes into transcript.
```

```
//Action: Prover requests challenge for next round (unit is specified in
↪  definition of state)
challenge_scalars()
///Function: Squeeze one or more field elements.

//Action:  Prover requests random bytes
challenge_bytes<N>()
///Function: Squeeze N bytes as a challenge from the transcript.

//Action: Prover requests to advance the transcript to a new phase (prevents
↪  replay/collision between protocol steps)
ratchet()
///Function: perform ratchet.

//Action: Prover requests proof string
narg_string()
///Function: Get the current transcript as a byte string (the proof).

//Action: get challenge vector
challenge_vector()
///Function: Get a vector of all challenges produced, for debugging, and for
↪  ease of use in some protocols eg: MLE commit/open for sumcheck.

//Action: Prover needs private coins (randomness) for ZK.
rng()
///Function: Get a CSPRNG seeded by the transcript for private randomness.
```

4.4 Verifier state

The verifier state is a functionality that enables construction methods that allow to process the "NARG string" (Non Interactive Argument i.e proof) using the formatted string approach in domain separator §4.1, the hash state with instructions defines the Duplex sponge interface common to prover and verifier §4.2

```
/// Internally, it simply contains a stateful hash.
/// Given as input an [`DomainSeparator`] and a NARG string, it allows to
/// de-serialize elements from the NARG string and make them available to the
↪  zero-knowledge verifier.
pub struct VerifierState<'a, H = DefaultHash, U = u8>
where
    H: DuplexSpongeInterface<U>,
    U: Unit,
{
    pub(crate) hash_state: HashStateWithInstructions<H, U>,
    pub(crate) narg_string: &'a [u8],
}
```

For a given protocol, A verifier state is built with a given domain separator §4.1

```
impl<'a, U: Unit, H: DuplexSpongeInterface<U>> VerifierState<'a, H, U> {
    pub fn new(domain_separator: &DomainSeparator<H, U>, narg_string: &'a
    ↪  [u8]) -> Self {
        let hash_state = HashStateWithInstructions::new(domain_separator);
        Self {
            hash_state,
            narg_string,
        }
    }
    /// Read `input.len()` elements from the NARG string (read proof elements)
    #[inline]
    pub fn fill_next_units(&mut self, input: &mut [U]) -> Result<(),
    ↪  DomainSeparatorMismatch> {
        U::read(&mut self.narg_string, input)?;
        self.hash_state.absorb(input)?;
        Ok(())
    }
    /// Signals the end of the statement.
    #[inline]
    pub fn ratchet(&mut self) -> Result<(), DomainSeparatorMismatch> {
        self.hash_state.ratchet()
    }
}
```

and it can call the DS hash interface §3 to request the absorb and squeeze calls for simulating interaction. For public data/challenge, this interface is the same as used by the prover state, so one can define a common "unit transcript" functionality for prover/verifier or we recommend just adding methods directly to the verifier state class.

```
impl<H: DuplexSpongeInterface<U>, U: Unit> UnitTranscript<U> for
↪  VerifierState<'_, H, U> {
    /// Add native elements to the sponge without writing them to the NARG
    ↪  string.
    #[inline]
    fn public_units(&mut self, input: &[U]) -> Result<(),
    ↪  DomainSeparatorMismatch> {
        self.hash_state.absorb(input)
    }

    /// Fill `input` with units sampled uniformly at random.
    #[inline]
    fn fill_challenge_units(&mut self, input: &mut [U]) -> Result<(),
    ↪  DomainSeparatorMismatch> {
        self.hash_state.squeeze(input)
```

```
    }
}
```

This definition allows external verifiers to be constructed using our Fiat-Shamir prescription. As we saw before, the unit transcript trait can be skipped and all the relevant methods directly added to the verifier state. In summary we should define a verifier state class that does all the following methods for reading from the NARG and data addition/interacting with the Sponge/prng.

```
//Action: Verifier reads polynomial coefficients or vector of field elements
↪  from transcript
 fill_next_scalars(n)
 ///Function: Read the next n field elements from the transcript and absorbs
 ↪  into sponge.

//Action: Verifier reads commitments from transcript
fill_next_points(n)
///Function: Read the next n group elements from the transcript and absorbs
↪  into sponge.

//Action: Verifier reads arbitrary data from transcript
fill_next_bytes(n)
///Function: Read the next n bytes from the transcript and absorbs into sponge.

//Action: verifier request challenges
fill_challenge_scalars()
///Function: Squeeze one or more field elements as Fiat-Shamir challenges from
↪  the transcript.

//Action: Verifer request challenge bytes
fill_challenge_bytes<N>()
///Function: Squeeze N bytes as a challenge from the transcript.

//Action: Advance the transcript to a new phase (prevents replay/collision
↪  between protocol steps)
ratchet()
///Function: Ratchet
```

Note: The completeness of the protocol is ensured by constructing the verifier state from the same Domain separator string. Error messages must be used for malformed NARG strings or if the expected sequence in the formatted domain separator string is not met.

5 Example: Sumcheck

First define a sumcheck domain constructor. We want users to use our domain constructor as a class to define the encoded domain separator string for the sumcheck protocol. This is

just a constructor that helps a user to define the DSFS operations for the sumcheck protocol as a formatted string.

5.1 Step 1: Define domain separator

```rust
/// Extend the domain separator with the Sumcheck protocol.
trait SumcheckDomainSeparator<F: Field> {
    /// Create a new sumcheck domain separator with public input.
    fn new_sumcheck(domsep: &str, num_vars: usize, claimed_sum: F, degree:
    ↪  usize) -> Self;

    /// Add the sumcheck statement (public input).
    fn add_sumcheck_statement(self, num_vars: usize, claimed_sum: F) -> Self;

    /// Add the sumcheck protocol rounds.
    fn add_sumcheck_protocol(self, num_vars: usize, degree: usize) -> Self;
}

impl<F, H> SumcheckDomainSeparator<F> for DomainSeparator<H>
where
    F::Field,
    H: DuplexSpongeInterface,
{
    fn new_sumcheck(domsep: &str, num_vars: usize, claimed_sum: F) -> Self {
        Self::new(domsep)
            .add_sumcheck_statement(num_vars, claimed_sum)
            .add_sumcheck_protocol(num_vars)
    }

    fn add_sumcheck_statement(self, num_vars: usize, claimed_sum: F) -> Self {
        self.add_scalars(1, "num_vars")
            .add_scalars(1, "claimed_sum")
            .add_scalars(1, "degree")
    }

    fn add_sumcheck_protocol(self, num_vars: usize, degree:usize) -> Self {
        let mut sep = self;
        for i in 0..num_vars {
            sep = sep
                .add_scalars(degree+1, &format!("round-{}-poly", i))
                .challenge_scalars(1, &format!("round-{}-challenge", i));/// if
                ↪  in extension field need to be taken into account by user
        }
        sep.add_scalars(1, "final-check")
    }
}
```

Both prover and verifier can access this and must define it and can define their formatted string using the above.

The utf encoded string should look like

```
let io: DomainSeparator<H> =
        SumcheckDomainSeparator::<F>::new_sumcheck("sumcheck-protocol",
        ↪  num_vars, claimed_sum, degree);

io= "sumcheck-protocol\0num_vars\0claimed_sum\0degree\0A{d+1}round-0-poly\0S1r
↪  ound-0-challenge\0A{d+1}round-1-poly\0S1round-1-challenge...\0A1final-poly"
```

The *addsumcheckstatement*() initializes the hash and sets to its Initial state.

5.2 Step 2: Decode hash instructions

The statefulhash §4.2 should decode the sumcheck io string, and maintain a set of sponge instructions. (Assume the no of sumcheck vars is n)

```
ops =vec![Op::Absorb((d+1)*U),Op::Squeeze(U),Op::Absorb((d+1)*U),Op::Squeeze(U
↪  ),...,n-times]
```

- Stateful hash will listen for prover/verifier of their requests in sequence, like add scalars/challenge scalars etc,
- It checks if it aligns with the prescribed order and execute the Duplex sponge operations of the instruction by using pop operations
- Both prover and verifier can talk only to the statefulhash interface after defining protocols. Which keeps its interaction with the Duplex sponge private.
- For misc requests like get challenge vector, prover state can be redefined to store challenges for future use.

5.3 Step 3: Prover and verifier state

5.3.1 Prover state

Define methods for prover state §4.3. Note that it has 3 objects (proverprivaterng,hashstate,nargstring).

For classic sumcheck, we dont have perfect zero knowledge yet, so we are going to ignore the proverprivaterng.

- Every add action from the prover triggers an absorb function on hash state. An update to the NARG string.
- Every challenge action from prover triggers a squeeze function on hash state and outputs a challenge of unit U.

```rust
pub struct SumcheckProverState<H: DuplexSpongeInterface, F: Field> {
    proverstate: ProverState<H>,
    _field: std::marker::PhantomData<F>,
}

impl<H: DuplexSpongeInterface, F: Field> SumcheckProverState<H, F>
where
    ProverState<H>: FieldToUnitSerialize<F> + UnitToField<F>,
{
    pub fn new(inner: ProverState<H>) -> Self {
        Self {proverstate }
    }

///add same parameters so this gets the correct IV.
    pub fn add_statement(&mut self, num_vars: usize, claimed_sum: F) ->
    ↪  ProofResult<()> {
        self.proverstate.add_scalars(num_vars);
        self.proverstate.add_scalars(claimed_sim);
        self.proverstate.add_scalars(degree);
        self.ratchet();
        Ok(())
    }

///allows prover to talk to stateful hash for adding proof item. (note this
↪  also adds item to the narg string by design)
    pub fn add_round(&mut self, coeffs: &[F]) -> ProofResult<F> {
        self.proverstate.add_scalars(coeffs)?;
    }

///allows prover to talk to stateful hash for getting challenge, for sumcheck
↪  one can define to store the challenges for future use.
    pub fn get_round_challeng(&mut self, &[F])-> ProofResult<F> {
          let [challenge]: [F; 1] = self.proverstate.challenge_scalars()?;
    Ok(challenge)
    }

///prover can see the narg string any time, also useful for debug,
    pub fn proof(&self) -> &[u8] {
        self.inner.narg_string()
    }

    //for MLE commitment etc
    pub fn get_round_challenges_vector(&mut self, &[F])
}
```

5.3.2 Verifier state

Similar to prover state, one can define methods for the verifier state

```
//initiate protocol with the same Domain separator string as prover, accept
↪  NARG string from prover.

impl<'a, H: DuplexSpongeInterface, F: Field> SumcheckVerifierState<'a, H, F>
where
    VerifierState<'a, H>: F
{
/
    pub fn new(verifierstate: VerifierState<'a, H>) -> Self {
        Self { verifierstate, _field: std::marker::PhantomData }
    }

///add same parameters as prover so this gets the correct IV. Initialize sponge
↪  by statefulhash
    pub fn add_statement(&mut self, num_vars: usize, claimed_sum: F) ->
    ↪  ProofResult<()> {
        self.proverstate.add_scalars(num_vars);
        self.proverstate.add_scalars(claimed_sim);
        self.proverstate.add_scalars(degree);
        self.ratchet();
        Ok(())
    }

//reads from NARG string, proof elements, (makes request to stateful hash for
↪  absorb and squeeze
    pub fn read_round(&mut self, degree: usize) -> ProofResult<(Vec<F>, F)> {
        let coeffs = self.verifierstate.round_polys(degree)?;
        let [challenge]: [F; 1] = self.verifier_state.challenge_scalars()?;
        Ok((coeffs, challenge))
    }

}
```

6 Security summary

1. Hash Function/Sponge Security

    - Collision Resistance: The underlying sponge/hash function (e.g., Keccak, Poseidon2) must be collision-resistant to prevent two different transcripts from producing the same challenge.
    - Preimage Resistance: It must be hard to find an input that maps to a given output.

- Pseudorandomness: The output of the sponge must be indistinguishable from a random value
- Overwrite Mode: The sponge must operate securely in overwrite mode, where new data overwrites the rate part of the state, and only the capacity part maintains the hash chain.

2. Domain Separation

   - Unique Domain Separator: Each protocol must use a unique, well-formatted domain separator string as defined in §4.1 to prevent cross-protocol attacks.
   - Deterministic Protocol Description: The domain separator must encode the exact sequence of absorb/squeeze/ratchet operations, so that both prover and verifier agree on the protocol flow.
   - No NULL Bytes in Labels: Labels in the domain separator must not contain NULL bytes or start with digits, to avoid parsing ambiguities.
   - Ensure public data is included at the protocol definition, included in the Domain separator string.

3. Transcript Consistency

   - Operation Order Enforcement: The sequence of sponge operations (absorb, squeeze, ratchet) must strictly follow the protocol as encoded in the domain separator. Any deviation must result in an error. Maintain some kind of check that the exact sequence is followed as in §4.2. This is internal and not exposed to user.
   - Transcript Binding: The transcript (NARG string) must include all prover messages (and seed rng if any) in the correct order, and the verifier must reconstruct the same state from the transcript and domain separator.

4. Zero-Knowledge and Randomness

   - Private Prover RNG: The prover must use a private, sponge-based CSPRNG for zero-knowledge protocols, seeded independently from the public transcript.
   - State Erasure: After generating randomness, the prover's private sponge state must be ratcheted (erased) to prevent state recovery and ensure forward secrecy.
   - No Leakage: The prover state must not be cloneable or copyable to prevent accidental leakage of private randomness.
   - Consider using this as ICICLE CSPRNG generator. (We can work on this to make it a standard)

5. Serialization and Deserialization

   - Canonical Serialization: Field and group elements must be serialized in a canonical, unambiguous way (little-endian, compressed for groups - please take into account the square root complexity for decompression).

- Validation: Deserialization must check that elements are valid (e.g., field elements are in range, group elements are on the curve). Consistent Encoding: The same element must always serialize to the same byte string.

6. Error Handling

   - Strict Error Propagation: Any mismatch in domain separator, transcript length, or operation order must result in a clear error and abort protocol execution.
   - No Silent Failures: All protocol violations must be reported with error messages.

7. Protocol Completeness and Soundness

   - Deterministic Challenges: For a given transcript and domain separator, the challenges generated must be deterministic and reproducible by both prover and verifier.
   - No Malleability: The transcript and domain separator must prevent an adversary from modifying the proof or protocol flow without detection.

8. Implementation Notes

   - No Big-Endian Support: The code/implementation explicitly does not support big-endian targets, to avoid subtle bugs in serialization. Neither does ICICLE, so it should be stated without ambiguity.
   - No Type Enforcement in Domain Separator: The domain separator enforces only data lengths, not data types, so protocol designers must ensure type consistency at a higher level. I am not sure exactly how to do this.

A Appendix

A.1 Minimal Bias random Bytes to fields

For a bitwise hasher, the output is binary/hex/bytes, in many examples the verifier messages are elements in a finite field $\mathbb{F}_p^d$ where p is a prime modulus, and $d = 1, 2, ..$ covers Fields and their extensions. So quite often one need to decode a field element from a random binary string $x \in \{0,1\}^m$ for $m > \log_2 p$. The common method to decode bytes to field is to compute

$$\psi(x) = \left(\sum_{i=1}^{m} x_i 2^{i-1} \right) \mod p \tag{A.1}$$

This map ensures that, when x is uniformly random or the output of a pseudo random hash function, the map $\psi(x)$ is uniformly random (pseudo random) up to negligible bias. In other words for any field element in $\mathbb{F}_p$ there are many preimages for $\psi(x)$ that are uniformly distributed when $2^m > p$. For proof we refer to the statistical distance calculation of $\psi(x)$ in appendix B of [3].

A.2 Serialization and Deserialization

The serialization format is a convention for how to represent field elements as bytes, independent of their internal representation (montgomery or not). The serialize compressed method commonly used has the features

1. Takes the raw bytes of the data type (field/extension field/group)
2. Serializes in little-endian format
3. Uses exactly compressed size bytes. eg: babybear 31 bit field is stored as a 4 byte element.
4. The output bytes are always the same for an element of the same data type.

The deserialize function should implement

1. Read the raw bytes in little-endian
2. interpret the bytes as a data type
3. check that the result is a valid data type.
4. The output bytes are always the same for the same field element

The basic interfaces commonly used in arkworks are

```
// Main serialization trait
pub trait CanonicalSerialize {
    fn serialize<W: Write>(&self, writer: W) -> Result<(), SerializationError>;
    fn serialized_size(&self) -> usize;
}
// Main deserialization trait
pub trait CanonicalDeserialize {
    fn deserialize<R: Read>(reader: R) -> Result<Self, SerializationError>;
}
// Compressed serialization (eg: remove leading zeros in field elements, store
↪  only x coordinate in points etc)
fn serialize_compressed<W: Write>(&self, writer: W) -> Result<(),
↪  SerializationError>;
fn deserialize_compressed<R: Read>(reader: R) -> Result<Self,
↪  SerializationError>;

// Unchecked compressed serialization (no validataion checks, user
↪  responsiblity)
fn serialize_compressed_unchecked<W: Write>(&self, writer: W) -> Result<(),
↪  SerializationError>;
fn deserialize_compressed_unchecked<R: Read>(reader: R) -> Result<Self,
↪  SerializationError>;
```

```rust
// Uncompressed serialization (to raw bytes)
fn serialize_uncompressed<W: Write>(&self, writer: W) -> Result<(),
↪  SerializationError>;
fn deserialize_uncompressed<R: Read>(reader: R) -> Result<Self,
↪  SerializationError>;

// Unchecked uncompressed serialization
fn serialize_uncompressed_unchecked<W: Write>(&self, writer: W) -> Result<(),
↪  SerializationError>;
fn deserialize_uncompressed_unchecked<R: Read>(reader: R) -> Result<Self,
↪  SerializationError>;

// Get size of compressed serialization
fn compressed_size(&self) -> usize;

// Get size of uncompressed serialization
fn uncompressed_size(&self) -> usize;

// For validating deserialized data (eg if point is a valid point or if element
↪  is a field element mod p
pub trait Valid {
    fn check(&self) -> Result<(), SerializationError>;
}
```

- For field data types, the compressed/uncompressed are the same. The check is essentially applying mod rules to ensure the data is a valid field element.
- For group element, the compressed/uncompressed are different, and are implemented with flags, to decode correctly. In principle, this saves memory.
  - Raw bytes: direct, full representation (not space-optimized).
  - Compressed bytes = minimal, space-efficient encoding (especially for group elements).

For a group element, we can have flags that represent the compression type. The compression just stores the x coordinate with flags, that indicate the nature of the y coordinate, which can be reconstructed by reading the flags in the serialized x. This adds the complexity of computing y coordinate which may be expensive. So we can decide if we want this or not.

```rust
pub enum SWFlags {
    /// Represents a point with positive y-coordinate by setting all bits to 0.
    YIsPositive = 0,
    /// Represents the point at infinity by setting the setting the
    ↪  last-but-one bit to 1.
    PointAtInfinity = 1 << 6,
```

```
    /// Represents a point with negative y-coordinate by setting the MSB to 1.
    YIsNegative = 1 << 7,
}
```

Obviously in this case the checks should include, whether the point is a valid point, which we already have in ICICLE.

B CPP: Class definition

This section is a suggestion, and please feel free to do whatever that suits ICICLE design.Please keep §6 in mind.

Note: The suggested C++ is based on cursor AI construction of C++ classes matching the document and the code description. It closely follows the actual rust code. BUT use with CARE!

- A class for Duplex sponge interface, these will relate to the actual mathematical operation *permute*() as defined by the particular hash

```
template<typename Unit>
class DuplexSpongeInterface {
public:
    virtual ~DuplexSpongeInterface() = default;
    virtual void initialize(const std::vector<uint8_t>& iv) = 0;
    virtual void absorb(const std::vector<Unit>& input) = 0;
    virtual void squeeze(std::vector<Unit>& output) = 0;
    virtual void ratchet() = 0;
};
```

- A class for Domain separator construction

```
class DomainSeparator {
public:
    std::string io; // Encoded domain separator string
    // ... other members as needed

    // Constructor
    DomainSeparator(const std::string& tag, const
    ↪  std::vector<std::tuple<char, size_t, std::string>>& ops);
    // Helper to encode/decode the separator string
};
```

- A class for parsing domain separator string into hash operations

```
template<typename Sponge>
class HashStateWithInstructions {
    Sponge sponge;
    std::deque<OpType> stack;
public:
    HashStateWithInstructions(const DomainSeparator& ds, const
    ↪  std::vector<uint8_t>& iv);
    void absorb(const std::vector<uint8_t>& input);
    void squeeze(std::vector<uint8_t>& output);
    void ratchet();
    // ... error handling for mismatched operations
};
```

- Class prover state and a Class for Proverprivaterng (ZK use case). Note that proverprivateRng can be used as a standalone sponge based CSPRNG.

```
template<typename Sponge, typename RNG>
class ProverState {
    ProverPrivateRng<Sponge, CSPRNG> rng;
    HashStateWithInstructions<Sponge> hash_state;
    std::vector<uint8_t> narg_string; // Transcript

public:
    // Constructor: initializes hash state and rng
 ProverState(const DomainSeparator& ds, CSPRNG csprng){}

 void add_scalars(const std::vector<FieldElement>& scalars) {
    }
//example
 void add_points(const std::vector<GroupElement>& points) {
        std::vector<uint8_t> bytes = serialize_points(points);
        hash_state.absorb(bytes);//update hash first
        narg_string.insert(narg_string.end(), bytes.begin(),
        ↪  bytes.end());//update NARG
        rng.sponge.absorb(bytes);//Sync rng sponge for ZK use cases
    }
const std::vector<uint8_t>& get_transcript() const;
    // ... other methods
};

///for the ZK use cases to use sponge to generate csprng's
template<typename Sponge, typename CSPRNG>
class ProverPrivateRng {
    CSPRNG csprng;
    Sponge sponge;
public:
```

```
    ProverPrivateRng(CSPRNG csprng_init, Sponge sponge_init)
        : csprng(std::move(csprng_init)), sponge(std::move(sponge_init))
        ↪  {}

    void fill_bytes(std::vector<uint8_t>& dest) {
        std::vector<uint8_t> seed(std::min(dest.size(), size_t(32)));
        csprng.generate(seed.data(), seed.size());
        sponge.absorb(seed);
        sponge.squeeze(dest);
        sponge.ratchet(); // this is important if we use the spo
    }

    uint32_t next_u32() {
       //implement
    }
    uint64_t next_u64() {
     //implememt
    }
};
```

- Class verifier state:

```
template<typename Sponge>
class VerifierState {
    HashStateWithInstructions<Sponge> hash_state;
    const std::vector<uint8_t>& narg_string;
    size_t transcript_pos = 0; // Track position in transcript

public:
    VerifierState(const DomainSeparator& ds, const std::vector<uint8_t>&
    ↪  transcript)
        : hash_state(ds, /*iv=*/std::vector<uint8_t>(32, 0)),
        ↪  narg_string(transcript) {}

    // Read and absorb the next units from the transcript
    void fill_next_units(std::vector<uint8_t>& output) {
        if (transcript_pos + output.size() > narg_string.size())
            throw std::runtime_error("Transcript too short");
        std::copy(narg_string.begin() + transcript_pos,
                  narg_string.begin() + transcript_pos + output.size(),
                  output.begin());
        transcript_pos += output.size();
        hash_state.absorb(output);
    }
```

```
    // Absorb public units (not from transcript)
    void public_units(const std::vector<uint8_t>& input) {
        hash_state.absorb(input);
    }

    // Squeeze challenge units
    void fill_challenge_units(std::vector<uint8_t>& output) {
        hash_state.squeeze(output);
    }

    // Ratchet the hash state
    void ratchet() {
        hash_state.ratchet();
    }
};};
```

C Definitions

Acknowledgments

We stand on the shoulders of giants.

References

[1] A. Fiat and A. Shamir, "How to prove yourself: practicalsolutions to identification and signature problems"."

[2] A. Chiesa and M. Orrù, "A fiat-shamir transformation from duplex sponges." Cryptology ePrint Archive, Paper 2025/536, 2025.

[3] "spongefish: a duplex sponge fiat-shamir library."

[4] J. Aumasson, D. Khovratovich, B. Mennink, and P. Quine, "SAFE: Sponge API for field elements." Cryptology ePrint Archive, Paper 2023/522, 2023.

[5] J. Aumasson, D. Khovratovich, B. Mennink, and P. Quine, *SAFE: Sponge API for field elements (blog)*, 2023.

[6] "Merlin library for fiat shamir."

[7] M. Hamburg, "The STROBE protocol framework." Cryptology ePrint Archive, Paper 2017/003, 2017.