Phase 2: Challenge generation and Transcript protocol

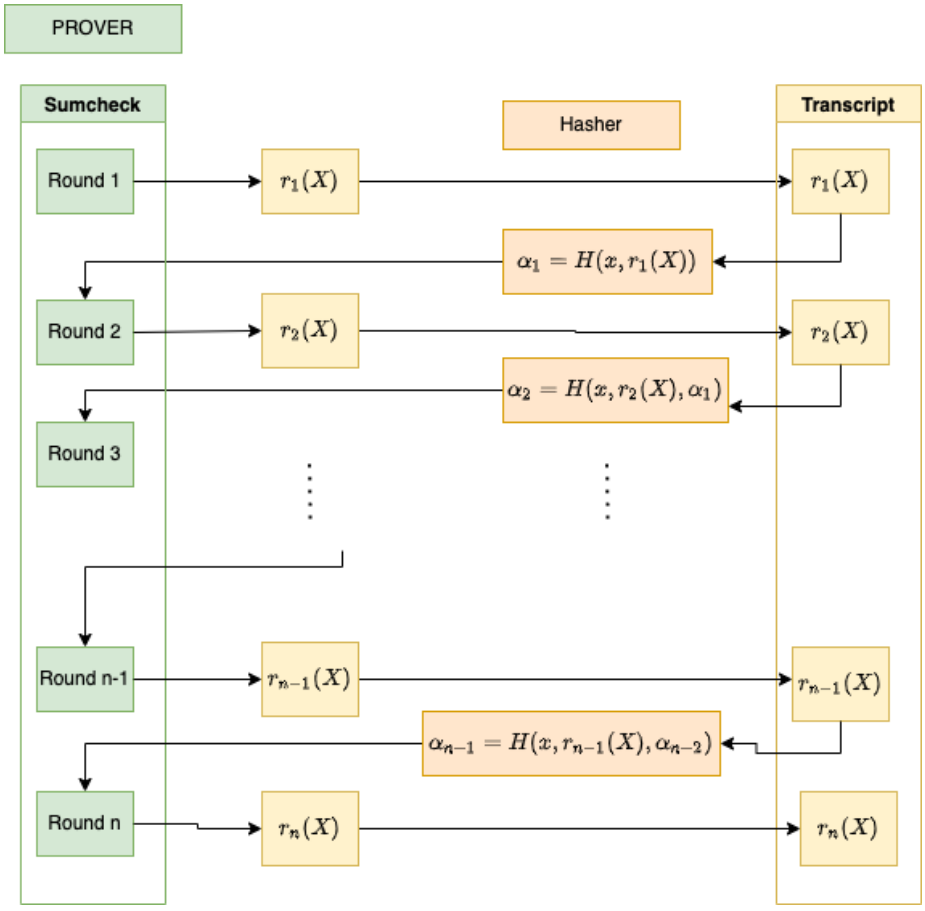

Figure 1: alt text

1. It records the prover output of the sumcheck protocol per round in a sequential manner.
   - x is a public input
   - r_i are $d+1$ Field elements each, if the max degree of the sumcheck is d. Thus in the case of single MLE, each round output is two field elements. This is called a prover state, and the prover records the prover state in the transcript at the end of each round.
2. Using a hasher, generates a random field element such that the output per round is hash chained to all the inputs in the previous round.
   - The hasher can be any hash, though in general Keccak is commonly used eg: Jolt
3. If the sumcheck protocol has n rounds, the prover records n states and the hasher activates $n-1$ times for the prover.

4. The verifier code must use the same transcript and hasher, and also in addition generate α_n to do the final check.

Generally speaking Fiat Shamir implemented in the following way using hash chaining is secure.

$$\alpha_i = Hash(x, i, \alpha_{i-1}, r_i)$$

It seems like there are two variations

- Strong FIat Shamir, where x is included in each hash.
- Weak Fiat Shamir, where x is only used in the first hash.

We only use the strong Fiat Shamir, though in practice both has been used in reakworld applications.

Example usage in supersumcheck

The sumcheck proof structure is

```
Proof struct
SumcheckProof<F: Field> {
   num_vars: usize,
   degree: usize,
   round_polynomials: Vec<Vec<F>>,
}
```

The main issue that this note addresses is what are the messages, how they are encoded, and what is the protocol. The key idea is based on strobe. A prover message or operation is seen as a tagged operation always.

`OP[label](prover_msg)`

where OP is some operation, for using a proof element for the hash first: encode the prover message as

`encoded_round_msg =[label||length(prover_msg)||round_number||prover_msg_in_round]`

Then hash the message with public data and encoded message

`challenge = hash(public||prev_challenge||round_challenge_label||encoded_round_msg)`

When prover and verifer adopt the same policy, it will generate the same challenges.

A transcript (Policy) for FIat Shamir

The sources used for compiling this information are

- Merlin library, as a basis for defining policy for Fiat Shamir.
  - Merlin c++ single file FS for templates
- jolt transcript

NOTE: It is user responsibility to ensure that prover and verifier implement the same policy. Generally it is enforced by the application, that hosts a prover and a verifier code.

Application **MUST SPECIFY** before calling sumcheck API a config

1. **hash function**: Merlin uses Keccak f/1600 at 128 bit security
   - Default use `keccak` ,
   - provide namespace options for icicle supported hashes like blake2s.
2. **Proof specific domain separator**: This identifies each proof statement and make it unique and prevents cross protocol attacks
   - ***Domain separator*** data type: `DS[u8]`
     - Default use `b"ICICLEV3Sumcheck"`
3. **Labels**
   - ***Label for round_poly*** data type:`label_round[u8]`
     - Default use`round_poly_label[u8] = b"ICICLE_round_poly"`
   - ***Label for challenge round*** data type `round_challenge[u8]`
     - Default use `round_challenge_label[u8] = b"ICICLE_round_challenge`
4. **encode policy: Data representations of messages**: Messages are data that the prover wants to append to proof, these data need to be encoded in a format before generating hashes from them.
   - Byte encoding Little Endian(LE)/Big Endian(BE).
   - Max length of byte strings `u32::max_value()`(internal)
   - Serialization/compression method for bigints. (internal?)
5. **seed rng**
   - Before generating the first challenge, the state needs to be initialized with some value. By default we use zero, but application can provide any random value.

We model this as a struct that encodes prover generated data to genrate randomness

```
This data should be FIXED by the application

Pub Struct Transcript {
    hash <namespace>,
    Domain seperator label<u8>,
    round_poly_label<u8>,
    round_challenge_label<u8>,
    encode_policy <LE/BE>,
    seed_rng <F::from(rng)>,
}

default values
Pub Struct Transcript {
    Hash = keccak,
    DS_label[u8] = b"ICICLEV3Sumcheck",
    round_poly_label[u8] = b"ICICLE_round_poly"
```

```
    round_challenge_label[u8] = b"ICICLE_round_challenge",
    encode_policy = Little Endian
    seed_rng = 0,
}
```

Example transcript 1 for $k = 0, 1, 2$ (i.e Deg 3): The principles from the FS and strobe can be encoded as how the data is encoded, on what form of data the hash operates, and generates challenges

```
Sumcheck prover messages: illustrating only phase2 for degree 3
Begin prove(){

//Initiate transcript - both prover and verifier should do this
DS = [DS_Label||Proof.num_vars.le()||Proof.degree.le()]
entry_0: [DS||public||sum_bytes]

// **Phase 2** Round m= 0

Proof.round_polynomials.append(r_0[x])
entry_1: [round_poly_label[u8] ||LE32(r_0[x]).len()||m=0||r_0[x].serialize.LE32()]
alpha_0 = Hash(entry_0||seed_rng||round_challenge_label[u8]||entry1).to_field()

// **Phase 2** Round m=1

Proof.round_polynomials.append(r_1[x])
entry_2: [round_poly_label[u8] ||LE32(r_1[x]).len()||m=1||r_1[x].serialize.LE32()]
alpha_1 = Hash(entry_0||alpha_0||round_challenge_label[u8]||entry2).to_field()

/ **Phase 2** Round m=2
Proof.round_polynomials.append(r_2[x])
entry_3: [round_poly_label[u8] ||LE32(r_2[x]).len()||m=2||r_2[x].serialize.LE32()]
alpha_2 = Hash(entry_0||alpha_1||round_challenge_label[u8]||entry3).to_field()

end prove
}
```

Verifier usage

```
read Transcript.DS_label
read Transcript.encode_policy
read Transcript.round_poly_label
read Transcript.round_challenge_label
read Transcript.seed_rng

read Proof.num_vars
read Proof.degree
```

```
read Proof.round_polynomials

read claimed_sum

//Initiate transcript
DS = [DS_Label||Proof.num_vars.le()||Proof.degree.le()]

Verifier Transcript encoding
entry_0: [DS||public||claimed_sum_bytes]
entry_1: [round_poly_label[u8] ||LE32(r_0[x]).len()||m=0||r_0[x].serialize.LE32()]
entry_2: [round_poly_label[u8] ||LE32(r_1[x]).len()||m=1||r_1[x].serialize.LE32()]
entry_3: [round_poly_label[u8] ||LE32(r_2[x]).len()||m=2||r_2[x].serialize.LE32()]

Hash generation by Verifier

alpha_0 = Hash(entry_0||seed_rng||round_challenge_label[u8]||entry1).to_field()
alpha_1 = Hash(entry_0||alpha_0||round_challenge_label[u8]||entry2).to_field()
alpha_2 = Hash(entry_0||alpha_1||round_challenge_label[u8]||entry3).to_field()
```

Here for example we list the definitions used in supersumcheck. This is a good guideline to implement it. In C++ this can be defined as a class/struct.

Basic functionality

```
pub trait Transcript<G: CurveGroup> {
    fn sumcheck_proof_domain_sep(&mut self, num_vars: u64, m: u64) {
        self.append_message(b"domain-separator", b"sumcheck v1");
        self.append_u64(b"n", num_vars);
        self.append_u64(b"m", m);
    }

    fn append_scalar(&mut self, label: &'static [u8], scalar: &G::ScalarField) {
        let mut buf = vec![];
        scalar.serialize_compressed(&mut buf).unwrap();
        self.append_message(label, &buf);
    }

    fn append_scalars(&mut self, label: &'static [u8], scalars: &[G::ScalarField]) {
        self.append_message(label, b"begin_append_vector");
        for item in scalars.iter() {
            <Self as TranscriptProtocol<G>>::append_scalar(self, label, item);
        }
        self.append_message(label, b"end_append_vector");
    }
    fn challenge_scalar(&mut self, label: &'static [u8]) -> G::ScalarField {
        let mut buf = [0u8; 64];
        self.challenge_bytes(label, &mut buf);
```

```
        G::ScalarField::from_le_bytes_mod_order(&buf)
    }
}
```