SAFE

Faster and simpler hashing for ZKPs

https://safe-hash.dev

Dmitry Khovratovich – Ethereum Foundation and Dusk Network
Jean-Philippe Aumasson - Taurus and Inference
Porçu Quine – Lurk Lab and Protocol Labs

zkSummit8, Berlin

Hashing and ZK proof systems

Cryptographic hashing is a **crucial ingredient of ZKP's**, as it is used..

- For commitments, Merkle trees, Fiat-Shamir transforms, etc.
- Via plain hashing, PRFs, DRBGs, XOFs, etc.
- Everywhere in recursive SNARKs and STARKs

Hashing and ZK proof systems

Cryptographic hashing is a **crucial ingredient of ZKP's**, as it is used..

- For commitments, Merkle trees, Fiat-Shamir transforms, etc.
- Via plain hashing, PRFs, DRBGs, XOFs, etc.
- Everywhere in recursive SNARKs and STARKs

The **efficiency metric** is not simply speed of a "vanilla" software implementation

It's mainly the **number of constraints** (R1CS or AIR) a.k.a. "algebraic complexity", in order to minimizes **proof generation and verification**

ZKP-friendly hash functions

To be efficient, these must work with similar structures as the constraint systems – usually, **finite fields**, where (for example) **XOR** becomes costly, on prime fields.

Fast BLAKE2 becomes slow and big – "**ZK-friendly**" **designs** are necessary

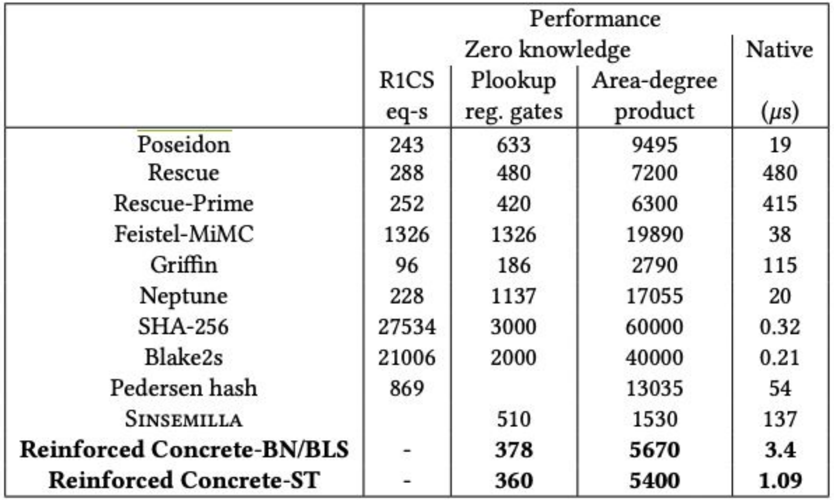
ZKP-friendly hash functions

To be efficient, these must work with similar structures as the constraint systems – usually, **finite fields**, where (for example) **XOR** becomes costly, on prime fields.

Fast BLAKE2 becomes slow and big – "**ZK-friendly**" **designs** are necessary

Poseidon family is the de facto standard
Used in Aleo, Anoma, Dusk, Filecoin, Penumbra, Polygon, zkSync, etc.

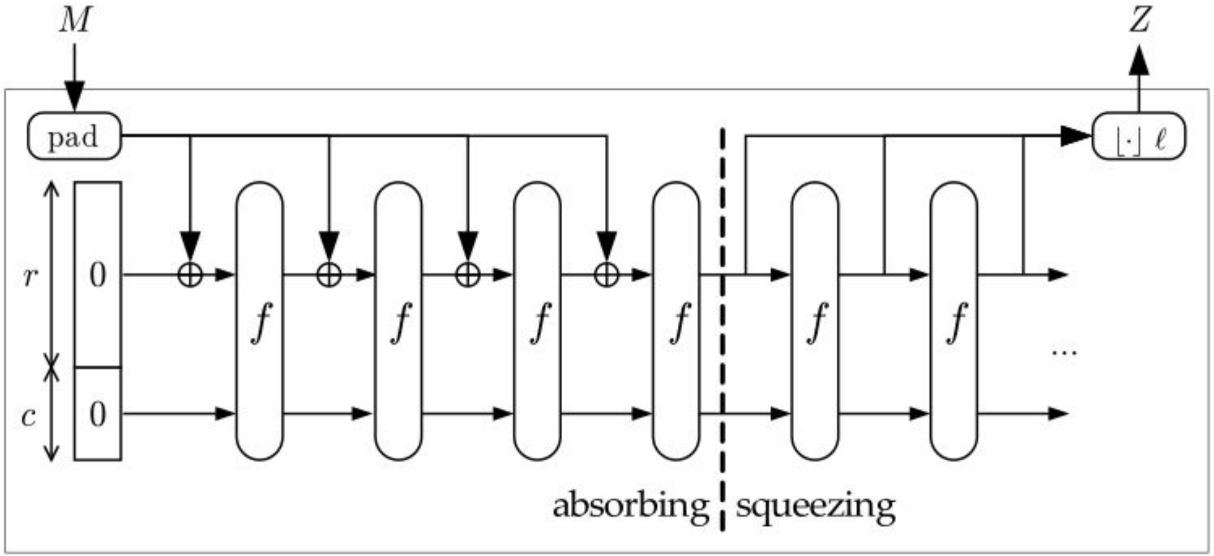
Other designs sometimes optimized for specific cases (field size, constraints type)



	Performance			
	Zero knowledge			Native
	R1CS eq-s	Plookup reg. gates	Area-degree product	(μs)
Poseidon	243	633	9495	19
Rescue	288	480	7200	480
Rescue-Prime	252	420	6300	415
Feistel-MiMC	1326	1326	19890	38
Griffin	96	186	2790	115
Neptune	228	1137	17055	20
SHA-256	27534	3000	60000	0.32
Blake2s	21006	2000	40000	0.21
Pedersen hash	869		13035	54
Sinsemilla		510	1530	137
Reinforced Concrete-BN/BLS	-	**378**	**5670**	**3.4**
Reinforced Concrete-ST	-	**360**	**5400**	**1.09**

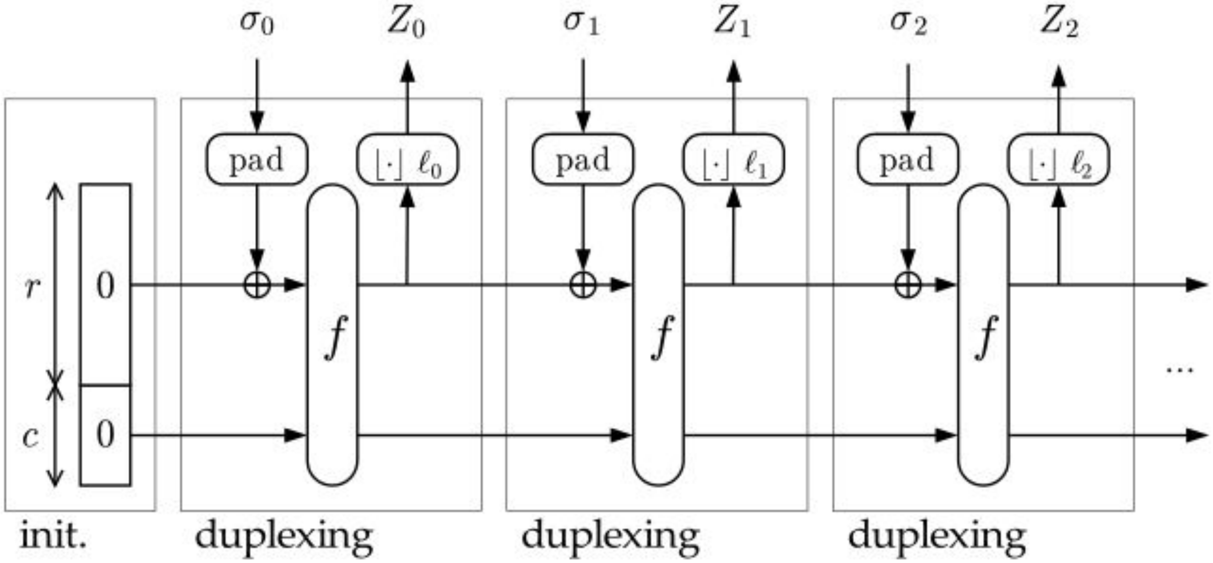
ZKP-friendly *sponge* functions

Simplest approach for ZK hashes, only requires a permutation



ZKP-friendly *duplex* functions

Generalization of sponges, to build hashes, PRFs, DRBGs, XOFs, etc.



Improving ZK hashing

Common "pain points" to address:

- **Security flaws** are common (modes design/choice, domain separation, etc.)
- ZK hashes expose an **inconsistent API** which is difficult to use securely
- **Padding schemes** reduces performance

Some "quick wins" in terms of simplicity and efficiency:

- Working with **field elements** rather than bits
- Assume **input length known** in advance

Hashing is hard

Did this small experiment while preparing the talk:

```
count = 1
while (true) {
    do("Pick a random Poseidon implementation on GitHub")
    do("Spend 5 minutes looking for bugs")
    if (bug found)
        return count
    count++
}
```

The "program" returned **2**

SAFE: making ZKP hashing easy and secure

Sponge ***A***PI for ***F***ield ***E***lements, a framework for protocol developers:

- Specification of a sponge state and **API**
- Eliminates padding, by introducing "**IO patterns**"
- **Implementation-ready** pseudo-code and models

SAFE aims to become the ***standard for ZK hashing***, bringing:

- **Interoperability** of libraries across protocols and proof systems
- A **common language** to specify hash-based protocols
- A basis for **hardware-accelerated** hashing

What SAFE is NOT

SAFE is NOT a new hash construction, but a **variant of the duplex mode** with interfaces defined in terms of *field elements rather than bits*

SAFE is NOT a new permutation, but can be instantiated with..

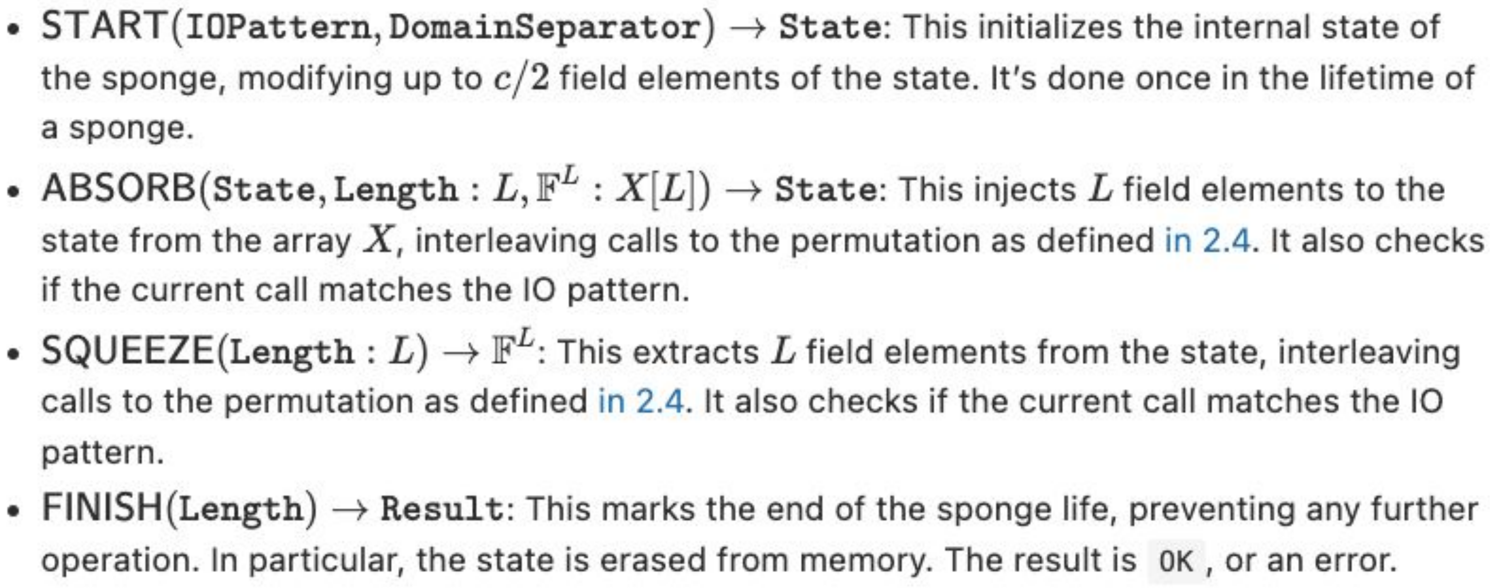
- Any existing permutation algorithm (such as Poseidon's)
- Any large enough finite field and field size

The SAFE API

Done once at init time

Series of calls of ABSORB and SQUEEZE in arbitrary order

Done once: verifies all calls were done and erase the state



- START($\mathtt{IOPattern}, \mathtt{DomainSeparator}$) $\rightarrow \mathtt{State}$: This initializes the internal state of the sponge, modifying up to $c/2$ field elements of the state. It's done once in the lifetime of a sponge.
- ABSORB($\mathtt{State}, \mathtt{Length} : L, \mathbb{F}^L : X[L]$) $\rightarrow \mathtt{State}$: This injects L field elements to the state from the array X, interleaving calls to the permutation as defined in 2.4. It also checks if the current call matches the IO pattern.
- SQUEEZE($\mathtt{Length} : L$) $\rightarrow \mathbb{F}^L$: This extracts L field elements from the state, interleaving calls to the permutation as defined in 2.4. It also checks if the current call matches the IO pattern.
- FINISH($\mathtt{Length}$) $\rightarrow \mathtt{Result}$: This marks the end of the sponge life, preventing any further operation. In particular, the state is erased from memory. The result is `OK`, or an error.

A SAFE state

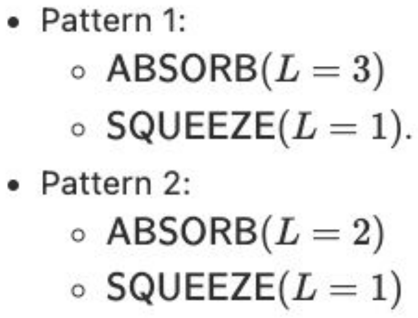
- A **permutation state** of ***n*** field elements (width ***n*** = *rate + capacity*)
- A permutation – operating on field elements
- Internal counters:
  - Absorb position
  - Squeeze position
- A *hasher* algorithm – a "vanilla" hash (only for *precomputation* of the IV)
- **A parameter tag *T*** – think "IV/initialisation value"

The parameter tag

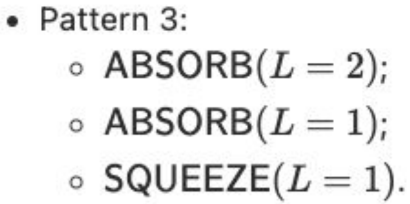
The initial value of the hash (precomputed), makes an **instance unique**

Derived from an **IO pattern** (sequence of ABSORB and SQUEEZE calls)

- Calls and their length parameters encoded to a byte string
- String hashed to a 128-bit value with the hasher (SHA3-256 by default)
- Optional *domain separator D* to distinguish identical IO patterns



- Pattern 1:
  - ABSORB($L = 3$)
  - SQUEEZE($L = 1$).
- Pattern 2:
  - ABSORB($L = 2$)
  - SQUEEZE($L = 1$)



- Pattern 3:
  - ABSORB($L = 2$);
  - ABSORB($L = 1$);
  - SQUEEZE($L = 1$).

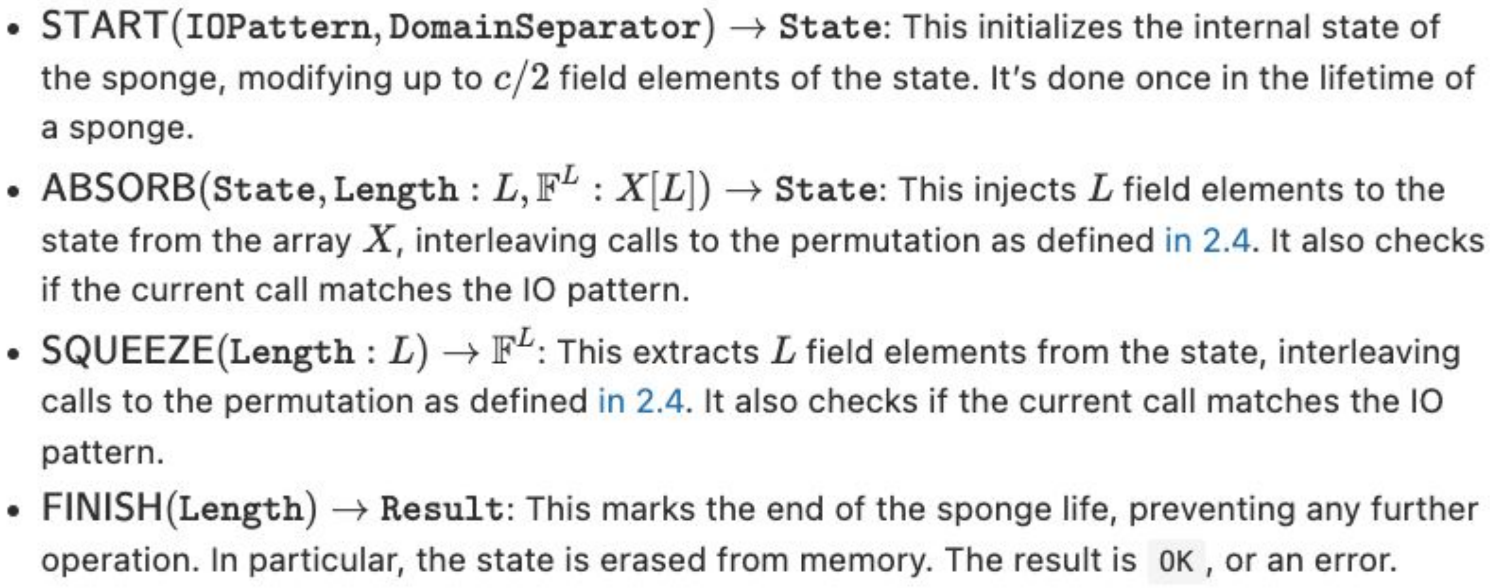
Which of these IO patterns correspond to equivalent instances? (and thus a same tag)

The SAFE API

Done once at init time: **commit to an IOpattern**

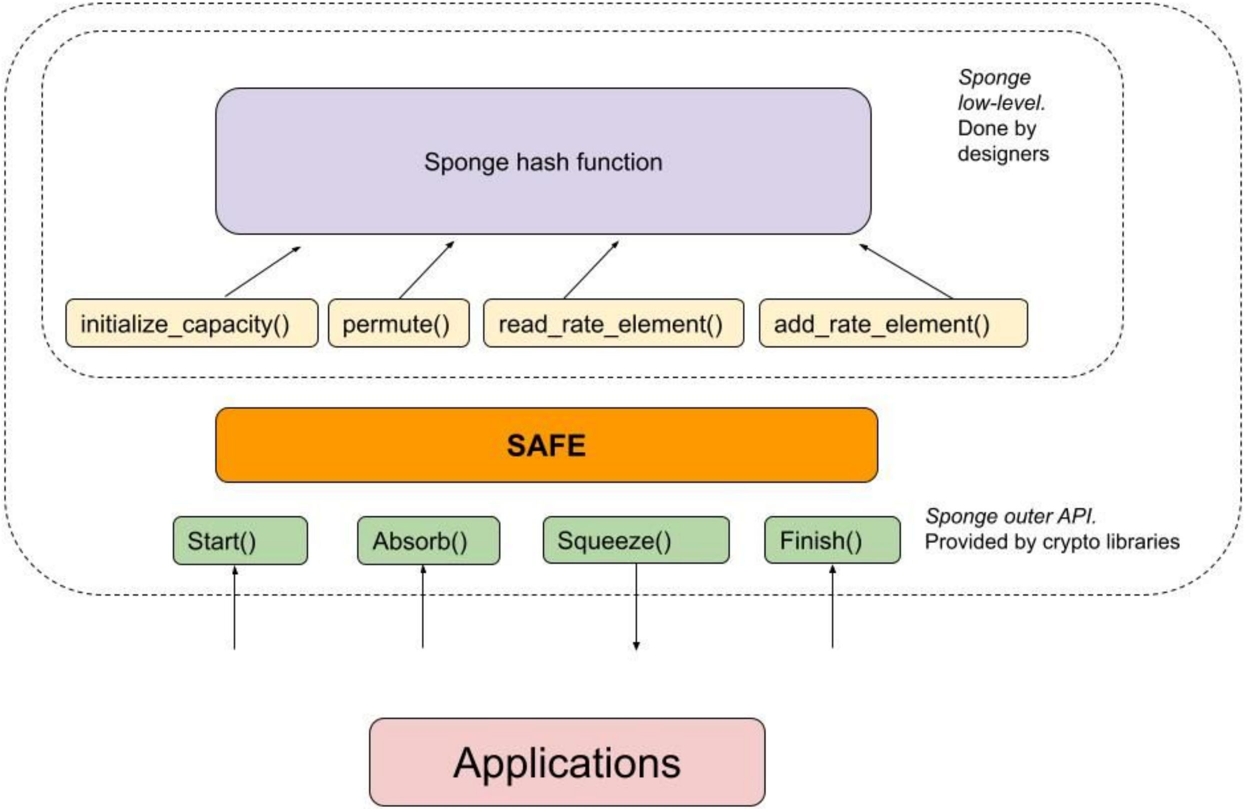
Series of calls of ABSORB and SQUEEZE in arbitrary order

Done once: verifies all calls were done and erase the state

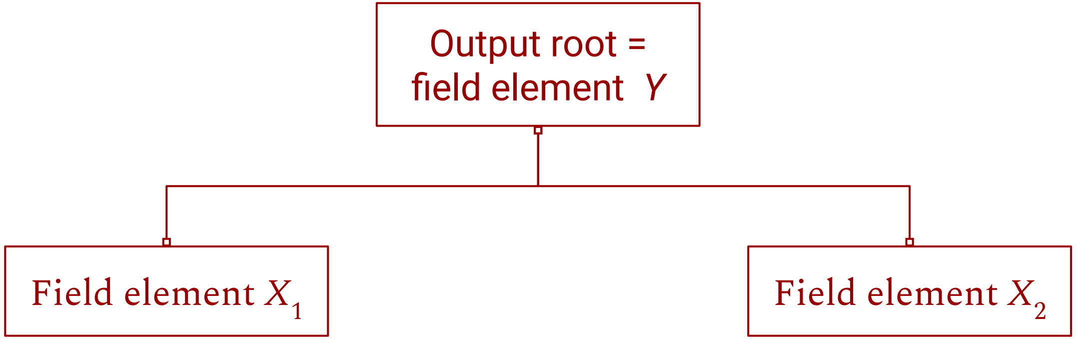


- $\mathsf{START}(\texttt{IOPattern}, \texttt{DomainSeparator}) \rightarrow \texttt{State}$: This initializes the internal state of the sponge, modifying up to $c/2$ field elements of the state. It's done once in the lifetime of a sponge.
- $\mathsf{ABSORB}(\texttt{State}, \texttt{Length} : L, \mathbb{F}^L : X[L]) \rightarrow \texttt{State}$: This injects L field elements to the state from the array X, interleaving calls to the permutation as defined in 2.4. It also checks if the current call matches the IO pattern.
- $\mathsf{SQUEEZE}(\texttt{Length} : L) \rightarrow \mathbb{F}^L$: This extracts L field elements from the state, interleaving calls to the permutation as defined in 2.4. It also checks if the current call matches the IO pattern.
- $\mathsf{FINISH}(\texttt{Length}) \rightarrow \texttt{Result}$: This marks the end of the sponge life, preventing any further operation. In particular, the state is erased from memory. The result is `OK`, or an error.

"Middleware" between applications and a permutation



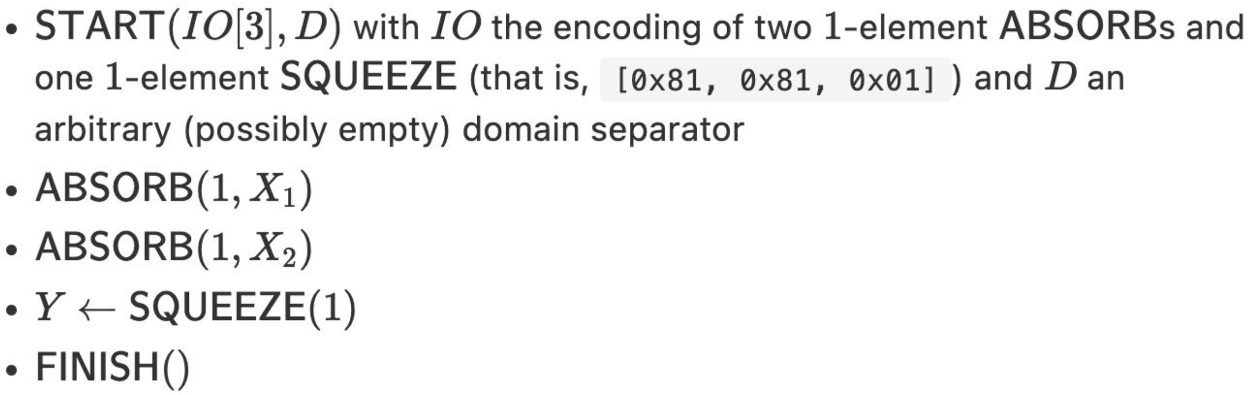
Example: Merkle tree



Output root = field element Y

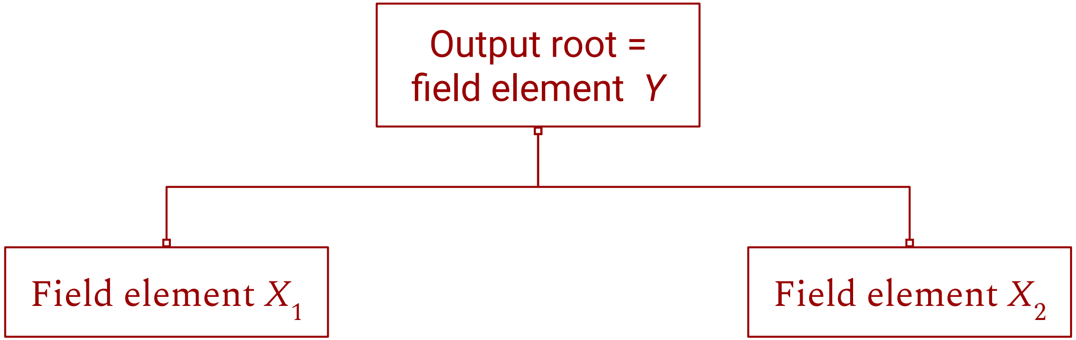
Field element X_1

Field element X_2



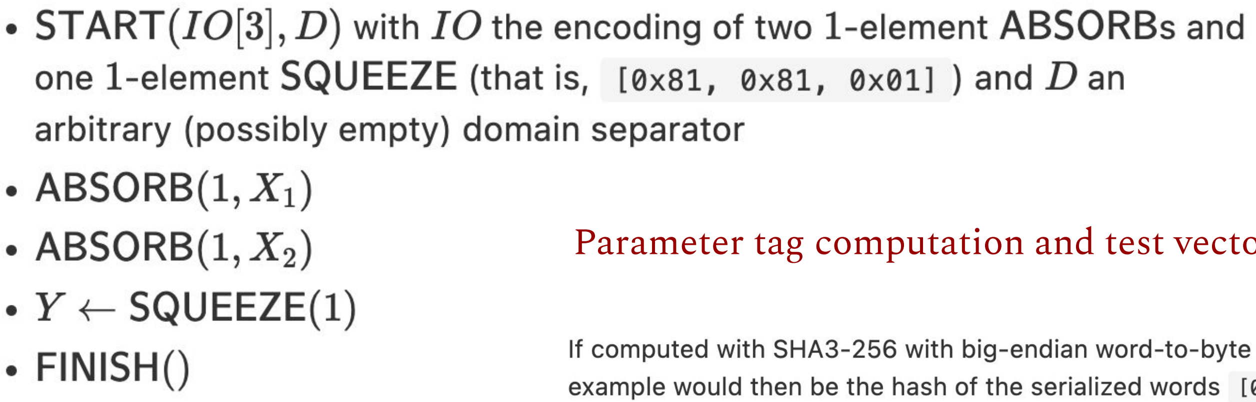
- START$(IO[3], D)$ with IO the encoding of two 1-element ABSORBs and one 1-element SQUEEZE (that is, `[0x81, 0x81, 0x01]`) and D an arbitrary (possibly empty) domain separator
- ABSORB$(1, X_1)$
- ABSORB$(1, X_2)$
- $Y \leftarrow$ SQUEEZE(1)
- FINISH$()$

Example: Merkle tree


Output root =
field element Y

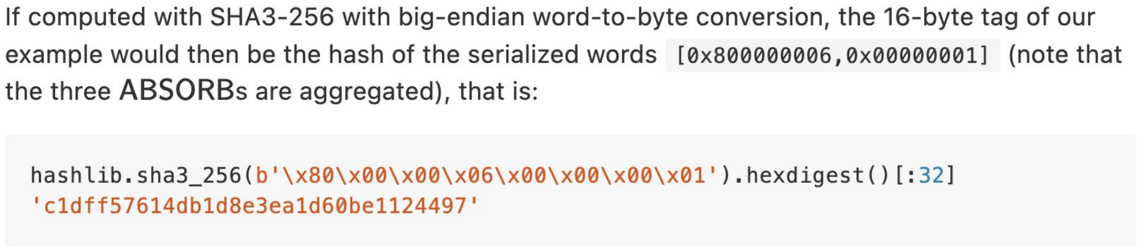
Field element X_1

Field element X_2



- $\mathsf{START}(IO[3], D)$ with IO the encoding of two 1-element ABSORBs and one 1-element $\mathsf{SQUEEZE}$ (that is, `[0x81, 0x81, 0x01]`) and D an arbitrary (possibly empty) domain separator
- $\mathsf{ABSORB}(1, X_1)$
- $\mathsf{ABSORB}(1, X_2)$
- $Y \leftarrow \mathsf{SQUEEZE}(1)$
- $\mathsf{FINISH}()$

Parameter tag computation and test vector:

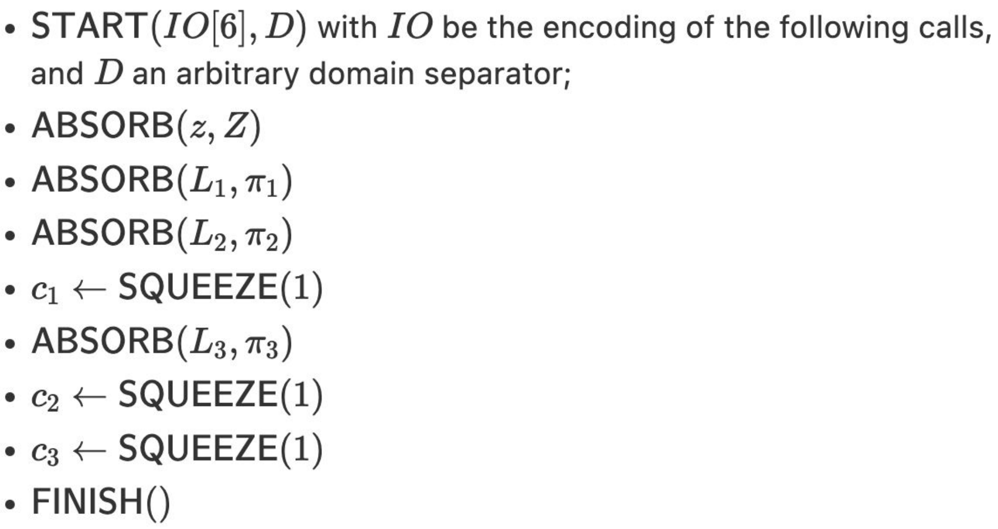


If computed with SHA3-256 with big-endian word-to-byte conversion, the 16-byte tag of our example would then be the hash of the serialized words `[0x800000006,0x00000001]` (note that the three ABSORBs are aggregated), that is:

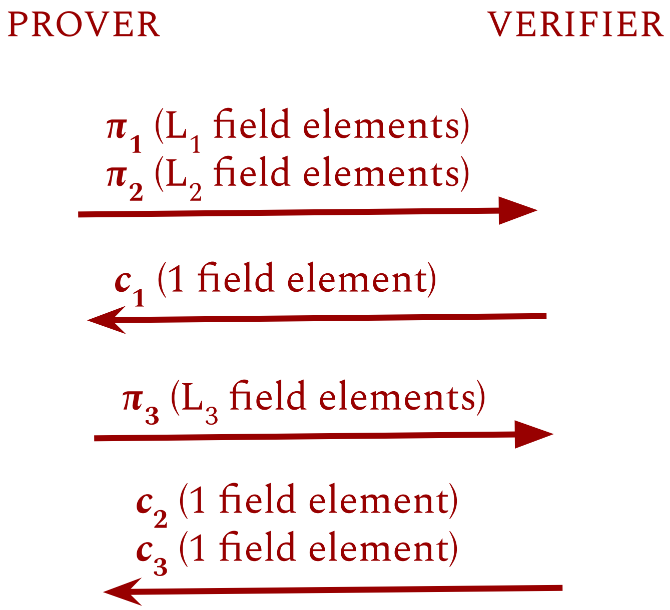
```
hashlib.sha3_256(b'\x80\x00\x00\x06\x00\x00\x00\x01').hexdigest()[:32]
'c1dff57614db1d8e3ea1d60be1124497'
```

Example: Interactive protocol

Challenges generation (simplified model):



- START($IO[6], D$) with IO be the encoding of the following calls, and D an arbitrary domain separator;
- ABSORB(z, Z)
- ABSORB(L_1, π_1)
- ABSORB(L_2, π_2)
- $c_1 \leftarrow$ SQUEEZE(1)
- ABSORB(L_3, π_3)
- $c_2 \leftarrow$ SQUEEZE(1)
- $c_3 \leftarrow$ SQUEEZE(1)
- FINISH()



PROVER

VERIFIER

π_1 (L_1 field elements)

π_2 (L_2 field elements)

c_1 (1 field element)

π_3 (L_3 field elements)

c_2 (1 field element)

c_3 (1 field element)

Limitations

- Length of data hashed must be **known in advance**:
  - Very few cases where it's a problem
  - This assumption makes the design simpler and more efficient
  - In the specs we describe an "infinite length" PRNG and an AEAD mode where the input is not known in advance

- Protocol-specific **input domain separation** is the responsibility of the protocol, not SAFE's (for ex, if different types are encoded to field elements)

- Need for a 128-bit **hash function**, to compute the initialization tag
  - Doesn't need to be circuitized, precomputed "offline", output can be hardcoded

- The duplex **security proof** must be adapted to fully apply to SAFE
  - Work in progress :)

How to adopt SAFE?

Follow the specs at https://safe-hash.dev

See Filecoin implemented it in https://github.com/filecoin-project/neptune

Hash function designers: Pick/design your permutation and parameters, don't worry about the mode

Protocol designers: Define the use of hashing in terms of SAFE calls, using SAFE API terminology – will make implementation straightforward

Implementers: Abstract out your software/hardware hash design as a SAFE instance, to be instantiated with the parameters received

Thank you!

https://safe-hash.dev

Get in touch on Telegram if you have questions or need help:

Dmitry /@khovratovich, **JP** /@veorq, **Porçu** /@porcuquine