Anatomy of a STARK

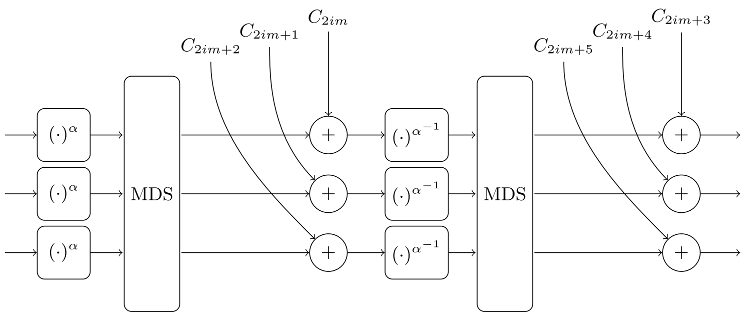
Anatomy of a STARK, Part 5: A Rescue-Prime STARK

This part of the tutorial puts the tools developed in the previous parts together to build a concretely useful STARK proof system. This application produces a STARK proof of correct evaluation of the Rescue-Prime hash function on a secret input with a known output. It is concretely useful because the resulting non-interactive proof doubles as a post-quantum signature scheme.

Rescue-Prime

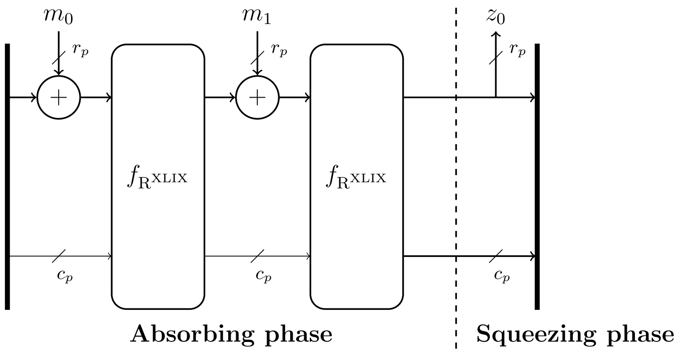
Rescue-Prime is an arithmetization-oriented hash function, meaning that it has a compact description in terms of AIR. It is a sponge function constructed from the Rescue-XLIX permutation $f_{\mathrm{R}^{\mathrm{XLIX}}} : \mathbb{F}^m \rightarrow \mathbb{F}^m$, consisting of several almost-identical rounds. Every round consists of six steps:

1. Forward S-box. Every element of the state is raised to the power α, where α is the smallest invertible power.
2. MDS. The vector of state elements is multiplied by a matrix with special properties.
3. Round constants. Pre-defined constants are added to every element of the state.
4. Backward S-box. Every element of the state is raised to the power α^{-1}, which is the integer whose power map is the inverse of $x \mapsto x^\alpha$.
5. MDS. The vector of state elements is multiplied by a matrix with special properties.
6. Round constants. Pre-defined constants are added to every element of the state.



The rounds are *almost* identical but not quite because constants are different in every rounds. While the Backward S-box step seems like a high degree operation, as we shall see, all six steps of the Rescue-XLIX round function can be captured by non-deterministic transition constraints of degree α.

Once the Rescue-XLIX permutation is defined, one obtains Rescue-Prime by instantiating a sponge function with it. In this construction, input field elements are absorbed into the top r elements of the state in between permutations. After one final permutation, the top r elements are read out. The Rescue-Prime hash digest consists of these r elements.



For the present STARK proof the following parameters are used:

- prime field of $p = 407 \cdot 2^{119} + 1$ elements
- $\alpha = 3$ and $\alpha^{-1} = 180331931428153586757283157844700080811$
- $m = 2$
- $r = 1$

Furthermore, the input to the hash computation will be a single field element. So in particular, there will be only one round of absorbing and one application of the permutation.

Implementation

The Rescue-Prime paper provides a nearly complete reference implementation. However, the code here is tailored to this one application.

```
class RescuePrime:
    def __init__( self ):
        self.p = 407 * (1 << 119) + 1
        self.field = Field(self.p)
        self.m = 2
        self.rate = 1
        self.capacity = 1
        self.N = 27
        self.alpha = 3
        self.alphainv = 180331931428153586757283157844700080811
        self.MDS = [[FieldElement(v, self.field) for v in [270497897142
                    [FieldElement(v, self.field) for v in [270497897142
        self.MDSinv = [[FieldElement(v, self.field) for v in [210387253
                       [FieldElement(v, self.field) for v in [901659657
        self.round_constants = [FieldElement(v, self.field) for v in [17
                                        10979758935699315327977538331866
                                        22820955900114355144222324832454
                                        26806570341117507762848324759622
                                        25014578629479310330371287650973
                                        15407792598648894396046384275381
                                        20435111991682398903226296606340
                                        57645879694647124999765652767459
                                        10259511070209448059707229051734
                                        85474390402060953238965247602744
                                        50572190394727023982626065566525
                                        87212354645973284136664042673979
                                        64194686442324278631544434661927
                                        23568247650578792137833165499572
                                        26400738596223484923791696610642
                                        22735830035453464339116453978421
                                        17970823399297229278827091448671
                                        10254493506276773963860368427274
                                        65916940568893052493361867756647
                                        14464015980752806066454380054852
                                        58854991566939066418297427463486
                                        14403053317130920196971556932351
                                        26450872243290657206637321658326
                                        22822825100935314666408731317941
                                        33847779135505989201180138242500
                                        14601928459310067359003664020862
                                        51518045467620803302456472369449
                                        73980612169525564135758195254813
                                        31385101081646507577789564023348
                                        27044002175874948259965791469559
                                        18523087799284533234417223423409
                                        21058192526199530348370033183384
                                        23320623552000086538251046002993
                                        17826406047821564310583255646639
                                        69838834175855952450551936238929
                                        75130152423898813192534713014890
                                        59548275327570508231574439445023
                                        43940979610564284967906719248029
                                        95698099945510403318638730212513
                                        77477281413246683919638580088082
                                        20678230433749740727375338748354
                                        14135467467888546341062992692979
                                        19199940390616847185791261689448
                                        17761361801981722293183261130717
                                        26790775110400509581136115681006
                                        33296937002574626161968730356414
                                        63869971087730263431297345514089
                                        20048128236185863835621187479372
                                        69328322389827264175963301685224
                                        23970159143769923596250553611388
                                        17960711445525398132996203513667
                                        21947563597282592084930017902696
                                        23003861106193195090131641372834
                                        14944681490699419681440381176738
```

```
25535582028106779796087284957910
93289417880348777872263904150910
47794802862111969844512383842308
20876224164132836934759800949450
34228805619823025763071411313049
15826163946006067936812298460724
65048656051037025727800046057154
13408288547776619894729309556570
23967684755547703714152865513907
85099105046897588972183075364233
23230501809141464311531960812337
17007238945443068217768778926177
62135161769871915508973643543011
15206455074148527786017895403501
20178926662621174884406053934450
17918479834729103356590263393280
96154153056489728639907128079436
95833504353120759807903032286346
18197598166282579162743995853119
26759026754839231133734899008522
49899900194200760923895805362651
89154519171560176870922732825690
26564972829058756198883514505969
14058385065911128084221211598104
26661390827474629787573402671814
23664512061479664542420999593491
26599406539009169295119874296277
59082836245981276360468435361137
26520064393601763202002257967586
10878169287684594077512357551815
13865803494798046491243642009217
45127926643030464660360100330441
21064870723840560652431859710752
42375307814689058540930810881506
23765338383691295304308235023237
23663877147548256281048410604892
16836667729797994334886606944152
19530126222676103611729005345454
21238196048554356213950107201025
96986567016099155020743003059932
24805732445613858920110710030276
19855022740661843292098944484417
17781267625420146897635247199202
21137413617037619862821357708402
10578571244551877573283063426067
12217936817579393468778075306367
12684821636117316049784444421486
22264167580742653700039698161547
23427590865863485892991884292379
18940981129458969702879685602315
75017033107075630953974011872571
14494534486035107558657512948957
26199115261693345516943712125431
18450316039330448878816627264054
```

```
    def hash( self, input_element ):
        # absorb
        state = [input_element] + [self.field.zero()] * (self.m - 1)

        # permutation
        for r in range(self.N):

            # forward half-round
            # S-box
            for i in range(self.m):
                state[i] = state[i]^self.alpha
            # matrix
            temp = [self.field.zero() for i in range(self.m)]
            for i in range(self.m):
                for j in range(self.m):
                    temp[i] = temp[i] + self.MDS[i][j] * state[j]
            # constants
            state = [temp[i] + self.round_constants[2*r*self.m+i] for i

            # backward half-round
            # S-box
            for i in range(self.m):
                state[i] = state[i]^self.alphainv
            # matrix
            temp = [self.field.zero() for i in range(self.m)]
            for i in range(self.m):
                for j in range(self.m):
                    temp[i] = temp[i] + self.MDS[i][j] * state[j]
            # constants
            state = [temp[i] + self.round_constants[2*r*self.m+self.m+i

        # squeeze
        return state[0]
```

Rescue-Prime AIR

The transition constraints for a single round of the Rescue-XLIX permutation are obtained by expressing the state values in the middle of the round in terms of the state values at the beginning, and again in terms of the state values at the end, and then equating both expressions. Specifically, let $\boldsymbol{s}_i$ denote the state values at the beginning of round i, let $\boldsymbol{c}_{2i}$ and $\boldsymbol{c}_{2i+1}$ be round constants, let M be the MDS matrix, and let superscript denote element-wise powering. Then the transition of a single round is captured by the equation:

$$M(\boldsymbol{s}_i^\alpha) + \boldsymbol{c}_{2i} = \left(M^{-1}(\boldsymbol{s}_{i+1} - \boldsymbol{c}_{2i+1})\right)^\alpha$$

To be used in a STARK, transition constraints cannot depend on the round. In other words, what is needed is a single equation that describes all rounds, not just the ith round. Let $\boldsymbol{X}$ denote the vector of variables representing the current state (beginning of the round), and $\boldsymbol{Y}$ denote the vector of variables represnting the next state (at the end of the round). Furthermore, let $\mathbf{f}_{\boldsymbol{c}_{2i}}(W)$ denote the vector of m polynomials that take the value $\boldsymbol{c}_{2i}$ on o^i, and analogously for $\mathbf{f}_{\boldsymbol{c}_{2i+1}}(W)$. Suppose without loss of generality that the execution trace will be interpolated on the domain $\{o^i | 0 \leq i \leq T\}$ for some T. Then the above family of arithmetic transition constraints gives rise to the following equation capturing the same transition conditions:

$$M(\boldsymbol{X}^\alpha) + \mathbf{f}_{\boldsymbol{c}_{2i}}(W) = \left(M^{-1}(\boldsymbol{Y} - \mathbf{f}_{\boldsymbol{c}_{2i+1}}(W))\right)^\alpha$$

The transition constraint polynomial is obtained by moving all terms to the left-hand side and dropping the equation to zero. Note that there are $2m+1$ variables, corresponding to $m =$ w registers.

```
    def round_constants_polynomials( self, omicron ):
        first_step_constants = []
        for i in range(self.m):
            domain = [omicron^r for r in range(0, self.N)]
            values = [self.round_constants[2*r*self.m+i] for r in range(
            univariate = Polynomial.interpolate_domain(domain, values)
            multivariate = MPolynomial.lift(univariate, 0)
            first_step_constants += [multivariate]
        second_step_constants = []
        for i in range(self.m):
            domain = [omicron^r for r in range(0, self.N)]
            values = [self.field.zero()] * self.N
            #for r in range(self.N):
            #    print("len(round_constants):", len(self.round_constants
            #    values[r] = self.round_constants[2*r*self.m + self.m +
            values = [self.round_constants[2*r*self.m+self.m+i] for r in
            univariate = Polynomial.interpolate_domain(domain, values)
            multivariate = MPolynomial.lift(univariate, 0)
            second_step_constants += [multivariate]

        return first_step_constants, second_step_constants

    def transition_constraints( self, omicron ):
        # get polynomials that interpolate through the round constants
        first_step_constants, second_step_constants = self.round_constan

        # arithmetize one round of Rescue-Prime
        variables = MPolynomial.variables(1 + 2*self.m, self.field)
        cycle_index = variables[0]
        previous_state = variables[1:(1+self.m)]
        next_state = variables[(1+self.m):(1+2*self.m)]
        air = []
        for i in range(self.m):
            # compute left hand side symbolically
            # lhs = sum(MPolynomial.constant(self.MDS[i][k]) * (previous
            lhs = MPolynomial.constant(self.field.zero())
            for k in range(self.m):
                lhs = lhs + MPolynomial.constant(self.MDS[i][k]) * (prev
            lhs = lhs + first_step_constants[i]
            # compute right hand side symbolically
            # rhs = sum(MPolynomial.constant(self.MDSinv[i][k]) * (next_
            rhs = MPolynomial.constant(self.field.zero())
            for k in range(self.m):
                rhs = rhs + MPolynomial.constant(self.MDSinv[i][k]) * (r
            rhs = rhs^self.alpha

            # equate left and right hand sides
            air += [lhs-rhs]

        return air
```

The boundary constraints are a lot simpler. At the beginning, the first state element is the unknown secret and the second state element is zero because the sponge construction defines it as such. At the end (after all N rounds or T cycles), the first state element is the one element of known hash digest $[h]$, and the second state element is unconstrained. Note that this second state element must be kept secret to be secure – otherwise the attacker can invert the permutation. This description gives rise to the following set $\mathcal{B}$ of triples $(c, r, e) \in \{0, \ldots, T\} \times \{0, \ldots, \mathsf{w} - 1\} \times \mathbb{F}$:

- $(0, 1, 0)$
- $(T, 0, h)$.

```
    def boundary_constraints( self, output_element ):
        constraints = []

        # at start, capacity is zero
        constraints += [(0, 1, self.field.zero())]

        # at end, rate part is the given output element
        constraints += [(self.N, 0, output_element)]

        return constraints
```

The piece of the arithmetization is the witness, which for STARKs is the execution trace. In the case of this particular computation, the trace is the collection of states after every round, in addition to the state at the very beginning.

```
    def trace( self, input_element ):
        trace = []

        # absorb
        state = [input_element] + [self.field.zero()] * (self.m - 1)

        # explicit copy to record state into trace
        trace += [[s for s in state]]
        # permutation
        for r in range(self.N):

            # forward half-round
            # S-box
            for i in range(self.m):
                state[i] = state[i]^self.alpha
            # matrix
            temp = [self.field.zero() for i in range(self.m)]
            for i in range(self.m):
                for j in range(self.m):
                    temp[i] = temp[i] + self.MDS[i][j] * state[j]
            # constants
            state = [temp[i] + self.round_constants[2*r*self.m+i] for i

            # backward half-round
            # S-box
            for i in range(self.m):
                state[i] = state[i]^self.alphainv
            # matrix
            temp = [self.field.zero() for i in range(self.m)]
            for i in range(self.m):
                for j in range(self.m):
                    temp[i] = temp[i] + self.MDS[i][j] * state[j]
            # constants
            state = [temp[i] + self.round_constants[2*r*self.m+self.m+i]

            # record state at this point, with explicit copy
            trace += [[s for s in state]]

        return trace
```

STARK-based Signatures

A non-interactive zero-knowldge proof system can be transformed into a signature scheme. The catch is that it must be capable of proving knowledge of a solution to a cryptographically hard problem. STARKs can be used to prove arbitrarily complex computational statements. However, the whole point of Rescue-Prime is that it generates cryptographically hard problem instances in a STARK-friendly way – concretely, with a compact AIR. So let's transform a STARK for Rescue-Prime into a signature scheme.

Rescue-Prime STARK

Producing a prover and verifier for STARK for Rescue-Prime consists of little more than linking together existing code snippets.

```
class RPSSS:
    def __init__( self ):
        self.field = Field.main()
        expansion_factor = 4
        num_colinearity_checks = 64
        security_level = 2 * num_colinearity_checks

        self.rp = RescuePrime()
        num_cycles = self.rp.N+1
        state_width = self.rp.m

        self.stark = Stark(self.field, expansion_factor, num_colinearity

    def stark_prove( self, input_element, proof_stream ):
        output_element = self.rp.hash(input_element)

        trace = self.rp.trace(input_element)
        transition_constraints = self.rp.transition_constraints(self.sta
        boundary_constraints = self.rp.boundary_constraints(output_eleme
        proof = self.stark.prove(trace, transition_constraints, boundary

        return proof

    def stark_verify( self, output_element, stark_proof, proof_stream ):
        boundary_constraints = self.rp.boundary_constraints(output_eleme
        transition_constraints = self.rp.transition_constraints(self.sta
        return self.stark.verify(stark_proof, transition_constraints, bo
```

Note the explicit argument concerning the proof stream. This needs to be a special object that simulates a *message-dependent* Fiat-Shamir transform, as opposed to a regular one.

Message-Dependent Fiat-Shamir

In order to transform a zero-knowledge proof system into a signature scheme, a non-interactive proof must be tied to the document that is being signed. Traditionally, the way to do this via Fiat-Shamir is to define the verifier's pseudorandom response as the hash digest of the document concatenated with the entire protocol transcript up until the point where its output is required.

In terms of implementation, this requires a new proof stream object – one that is aware of the document for which the signature is to be generated or verified. The next class achieves just this.

```
class SignatureProofStream(ProofStream):
    def __init__( self, document ):
        ProofStream.__init__(self)
        self.document = document
        self.prefix = blake2s(bytes(document)).digest()

    def prover_fiat_shamir( self, num_bytes=32 ):
        return shake_256(self.prefix + self.serialize()).digest(num_byte
    def verifier_fiat_shamir( self, num_bytes=32 ):
        return shake_256(self.prefix + pickle.dumps(self.objects[:self.r

    def deserialize( self, bb ):
        sps = SignatureProofStream(self.document)
        sps.objects = pickle.loads(bb)
        return sps
```

Signature Scheme

At this point it is possible to define the key generation, signature generation, and signature verification functions that make up a signature scheme. Note that these functions are members of the Rescue-Prime STARK Signature Scheme (`RPSSS`) class whose definition started earlier.

```
# class RPSSS:
    def keygen( self ):
        sk = self.field.sample(os.urandom(17))
        pk = self.rp.hash(sk)
        return sk, pk

    def sign( self, sk, document ):
        sps = SignatureProofStream(document)
        signature = self.stark_prove(sk, sps)
        return signature

    def verify( self, pk, document, signature ):
        sps = SignatureProofStream(document)
        return self.stark_verify(pk, signature, sps)
```

This code defines a *provably secure*[1], *post-quantum* signature scheme that (almost) achieves a 128 bit security level. While this description sounds flattering, the scheme's performance metrics are much less so:

- secret key size: 16 bytes (yay!)
- public key size: 16 bytes (yay!)
- signature size: ~**133 kB**
- keygen time: 0.01 seconds (acceptable)
- signing time: **250 seconds**
- verification time: **444 seconds**

There might be a few optimizations available that can reduce the proof's size, such as merging common paths when opening a batch of Merkle leafs. However, these optimizations distract from the purpose of this tutorial, which is to highlight and explain the mathematics involved.

In terms of speed, a lot of the poor performance is due to using Python instead of a language that is closer to the hardware such as C or Rust. Python was chosen for the same reason – to highlight and explain the maths. But the biggest performance gain in terms of speed is going to come from switching to faster algorithms for key operations. This is the topic of the next and last part of the tutorial.

0 - 1 - 2 - 3 - 4 - **5** - 6

1. More specifically, in the random oracle model, a successful signature-forger gives rise to an adversary who breaks the one-wayness of Rescue-Prime with polynomially related running time and success probability. ↩