Hardware implementation of $Keyak_2$

Jos Wetzels & Wouter Bokslag

Radboud University

Content

- Authenticated Encryption
- CAESAR Competition
- Keyak_2
- Scope
- GMU API
- Design Decisions
- Test Benches
- Performance Figures

Authenticated Encryption

- Encryption provides confidentiality but not (per se) *integrity* or *authenticity*
- Problem when attacker can modify ciphertext without detection (eg. padding oracle attacks, bitflipping, etc.)
- We want authentication of our data:
  - AE: Authenticated Encryption
  - AEAD: Authenticated Encryption with Associated Data (also include plaintext metadata in check)
- Traditional solution: Message Authentication Codes (MACs)
- Implementation often goes wrong (see POODLE attack on TLS)
- Desire for more dedicated AEAD schemes outside of AES in GCM/CCM/OCB mode

CAESAR Competition

- Competition for Authenticated Encryption, selects portfolio of schemes

- Primary candidate goals:
  - Offer advantages over AES-GCM
  - Suitable for widespread adoption

- Candidates specify *family* of authenticated ciphers

- Five inputs, one output:
  - Input Data (eg. plaintext or ciphertext)
  - Associated Data (AD), also called metadata (eg. packet headers)
  - Key
  - (*optional*) Public Message Number
  - (*optional*) Secret Message Number

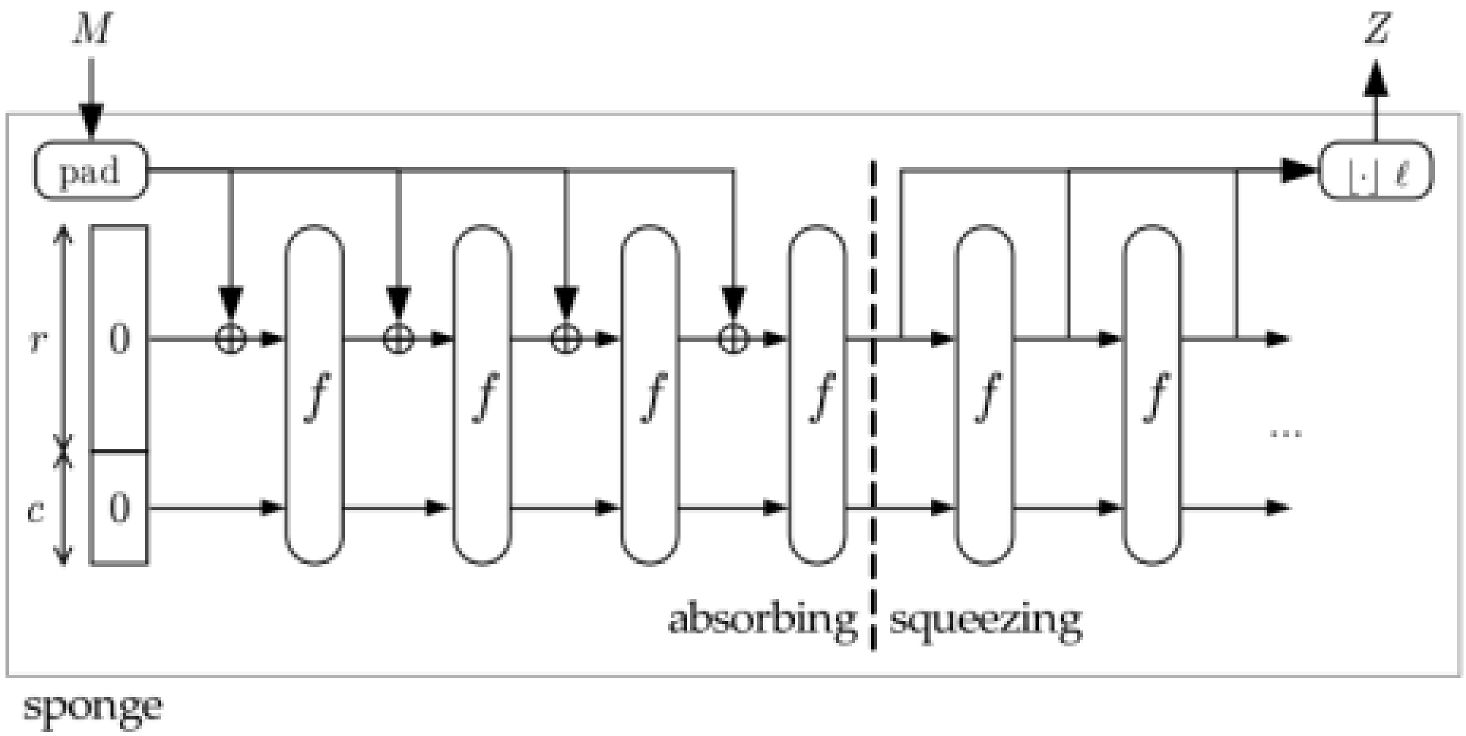
  - Output Data (eg. ciphertext or decrypted plaintext)

Keyak$_2$ - Overview

- Parameterized permutation-based authenticated encryption scheme
- 2nd round CAESAR candidate
- 5 named instances aimed at spectrum of platforms
- Offers support for *session-based* authentication (full sequence of messages is authenticated)
- In-place encryption & decryption
- Stream-compatible (does not require prior knowledge of input length)

Keyak$_2$ - Sponges

- Mode of operation is 'sponge-based' construction



- Data is first 'aborbed' into sponge state, later 'squeezed' out

- Primitive suitable to make hashes, MACs, stream ciphers, etc.
  - eg. 'squeeze' keystream from it, absorb data in it for later authentication

Keyak$_2$ – Motorist Mode

- New mode of operation ("*Motorist*") compared to Keyak$_1$

- Consists of 3 components:
  - Piston: Has internal state, performs 'sponge' operations on it
    - *Crypt*: Encryption/Decryption, squeeze 'keystream' from state, absorb input into state
    - *Inject*: Absorb metadata into state
    - *Spark*: Perform permutation (core of scheme security)
    - *GetTag*: Extract authentication tag from state

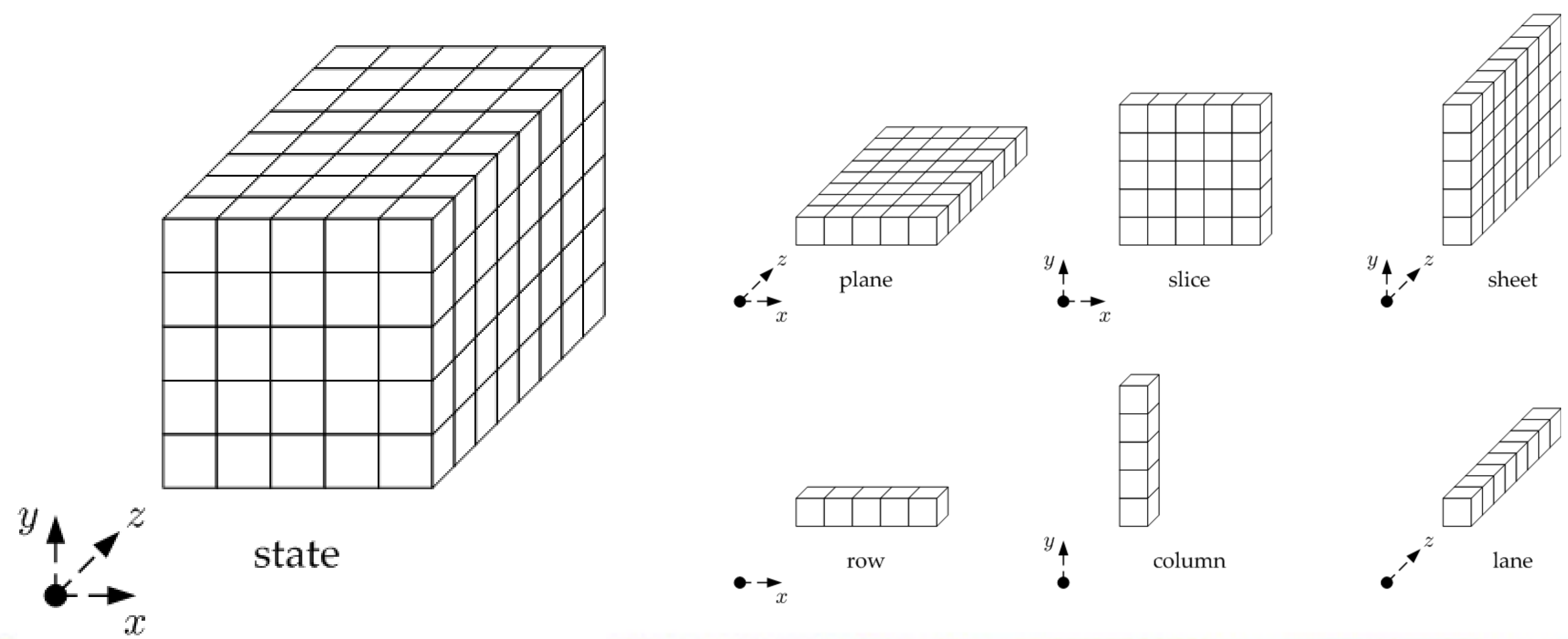
  - Engine: Controls array of π parallel pistons, effectively wrapper around piston functionality

  - Motorist: Top-level component, performs initialization and 'wrapping'
    - *StartEngine*: Initializes state using SUV (Secret and Unique Value) which is derived from key + nonce
    - *Wrap*: Performs AEAD call (encrypt or decrypt + authenticate data + metadata)

Keyak$_2$ - Permutation

- Uses Keccak$_P$ (derivative of Keccak$_f$) as underlying permutation
- Keccak$_f$ also underlies SHA-3 winner Keccak
- Difference Keccak$_P$ and Keccak$_f$: starting round index
- State represented as 5x5 array of n-bit '*lanes*'

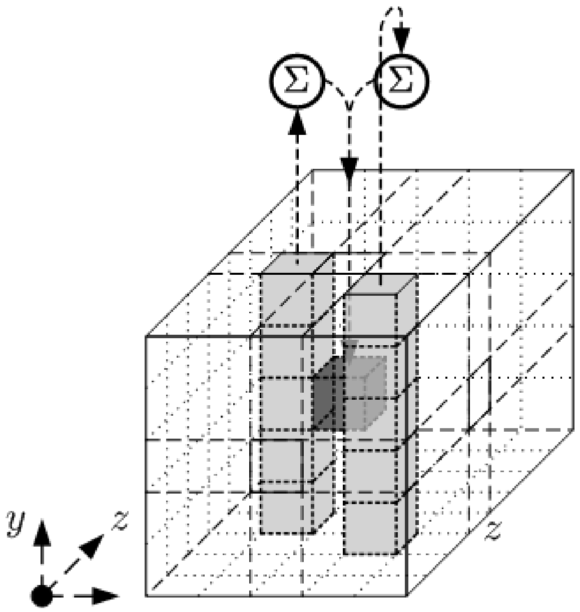
Keccak-f[b](S):
  for $i \in \{0, .., (n_r - 1)\}$:
    S = **Round**[b](S, RC_i)
  Return S



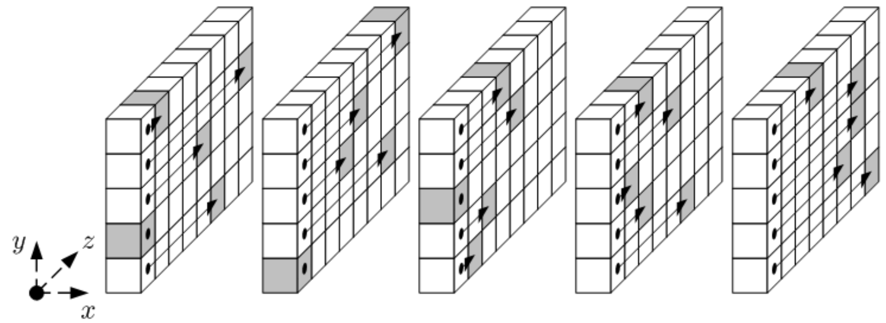
Keyak$_2$ - Permutation

- Round function 'shuffles' state according to several criteria

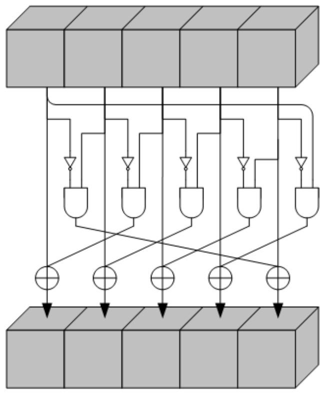
```
Round[b](A, RC):
 //θ-step for bit diffusion
 for x ∈ {0,..,4}:
   C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4]

 for x ∈ {0,..,4}:
   D[x] = C[x-1] xor rot(C[x+1], 1)

 for (x, y) ∈ {{0,..,4}x{0,..,4}}:
   A[x,y] = A[x,y] xor D[x]

 //ρ-step for inter-slice diffusion and
 //π-step for disturbing x,y alignment through lane-transposition
 for (x, y) ∈ {{0,..,4}x{0,..,4}}:
   B[y, 2x + 3y] = rot(A[x,y], r[x,y])

 //χ-step for non-linear mapping
 for (x, y) ∈ {{0,..,4}x{0,..,4}}:
   A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y])

 //ι-step to break symmetry
 A[0,0] = A[0,0] xor RC

 Return A
```



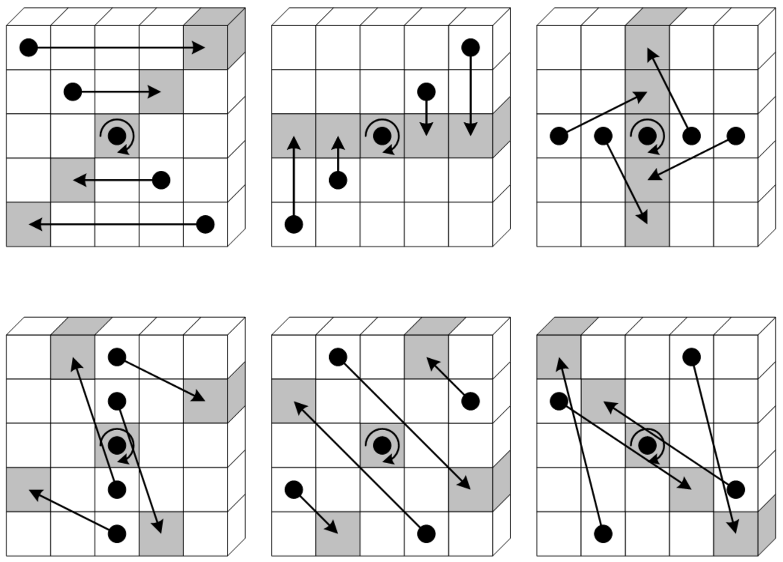
θ step



ρ step



ι step



π step

Keyak$_2$ – Named Instances

- 5 Named Instances
- Differ in state-width *b* and piston-parallelism *π*
- Suitable for different platforms

River Keyak $b = 800, \Pi = 1$

Lake Keyak $b = 1600, \Pi = 1$ (primary recommendation)

Sea Keyak $b = 1600, \Pi = 2$

Ocean Keyak $b = 1600, \Pi = 4$

Lunar Keyak $b = 1600, \Pi = 8$

Keyak$_2$ – Pure Python Implementation

```
def _KeccakFonLanes(self, lanes):
        R = 1
        for roundIndex in range(self.aStartRoundIndex, (self.aStartRoundIndex + self.aNrRounds)):
                # θ
                C = [lanes[x][0] ^ lanes[x][1] ^ lanes[x][2] ^ lanes[x][3] ^ lanes[x][4] for x in range(5)]
                D = [C[(x+4)%5] ^ self._ROL(C[(x+1)%5], 1) for x in range(5)]
                lanes = [[lanes[x][y]^D[x] for y in range(5)] for x in range(5)]

                # ρ and π
                (x, y) = (1, 0)
                current = lanes[x][y]
                for t in range(24):
                        (x, y) = (y, (2*x+3*y)%5)
                        (current, lanes[x][y]) = (lanes[x][y], self._ROL(current, (t+1)*(t+2)//2))

                # χ
                for y in range(5):
                        T = [lanes[x][y] for x in range(5)]
                        for x in range(5):
                                lanes[x][y] = T[x] ^((~T[(x+1)%5]) & T[(x+2)%5])

                # ι
                lanes[0][0] ^= self.RC[roundIndex]
        return lanes
```

```
def StartEngine(self, SUV, tagFlag, T, unwrapFlag, forgetFlag):
    if (self.phase != MotoristPhase.ready):
        raise Exception("The phase must be ready to call Motorist.StartEngine().")

    self.engine.InjectCollective(SUV, True)

    if (forgetFlag):
        self._MakeKnot()

    res = self._HandleTag(tagFlag, T, unwrapFlag)

    if (res):
        self.phase = MotoristPhase.riding
    return res

def Wrap(self, I, O, A, T, unwrapFlag, forgetFlag):
    if (self.phase != MotoristPhase.riding):
        raise Exception("The phase must be riding to call Motorist.Wrap().")

    if(not(hasMore(I)) and not(hasMore(A))):
        self.engine.Inject(A)

    while (hasMore(I)):
        self.engine.Crypt(I, O, unwrapFlag)
        self.engine.Inject(A)

    while (hasMore(A)):
        self.engine.Inject(A)

    if ((self.Pi > 1) or (forgetFlag)):
        self._MakeKnot()

    res = self._HandleTag(True, T, unwrapFlag)

    if not(res):
        O.erase()
    return res
```

Internship Scope

- CAESAR 3rd round candidate process requires (reference) hardware implementation

- River & Lake Keyak most suitable (no piston-parallelism)

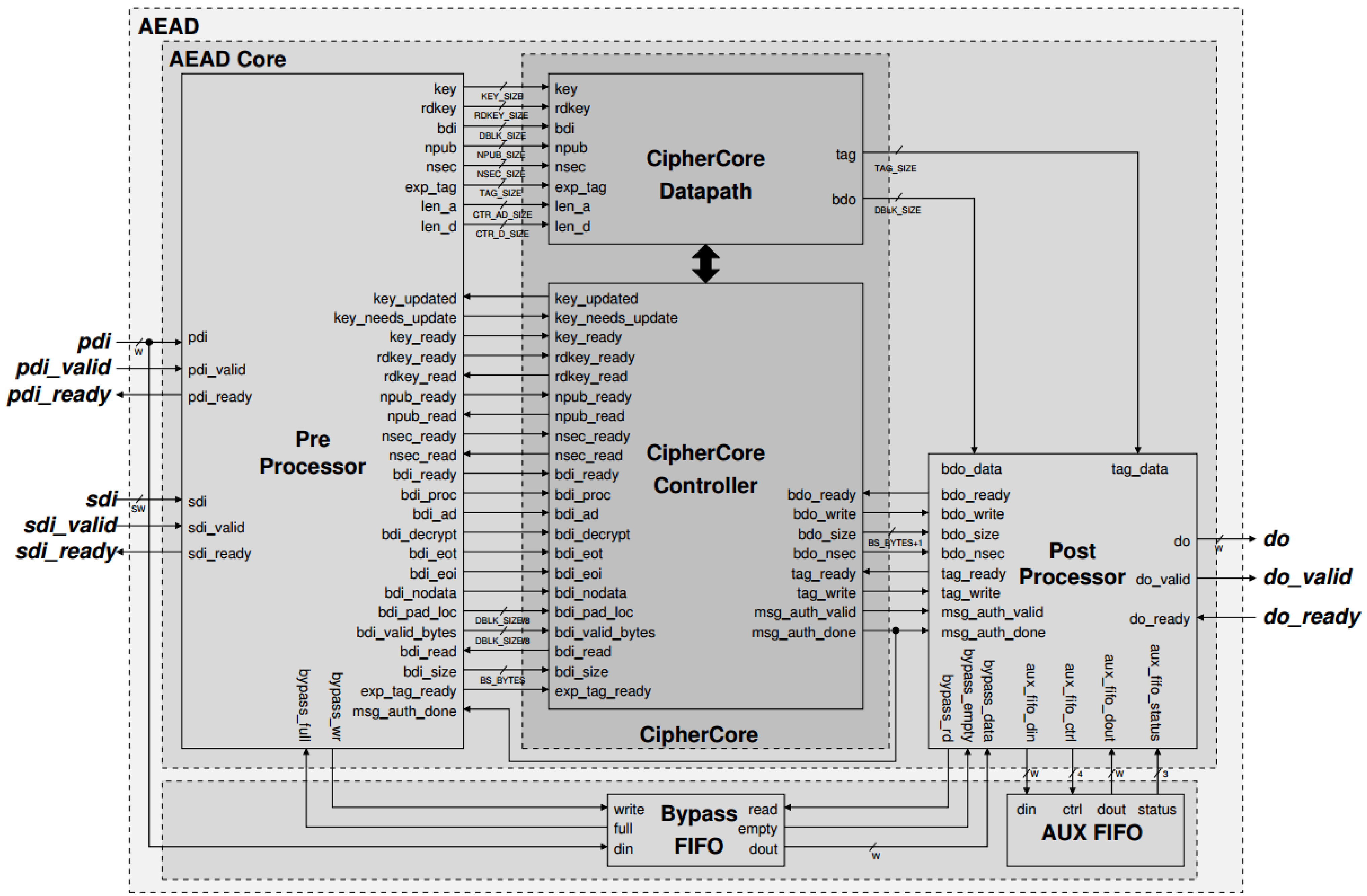
- River Keyak
  - Lane size: 32 bits
  - State width: 800 bits (5x5x32)
  - Squeeze rate: 68 bytes
  - Absorb rate: 96 bytes

- Lake Keyak
  - Lane size: 64 bits
  - State width: 1600 bits (5x5x64)
  - Squeeze rate: 168 bytes
  - Aborb rate: 192 bytes

GMU API

- CAESAR call for submissions only defined a software API
- 3rd round candidates require hardware implementation
- George Mason University (GMU) proposed hardware API
- Motivation:
  - Hardware API influences area & throughput/area ratio of implementation
  - Uniform hardware API facilitates candidate comparison
- Features:
  - Inputs of arbitrary number of bytes
  - Streaming-support: no need to know size of plain/cipher/AD in advance
  - Independent data & key inputs
  - Encryption/Decryption in same core, toggled with flag
  - External communication with AXI4 or FIFOs

GMU API



AEAD
AEAD Core
Pre Processor
CipherCore Datapath
CipherCore Controller
CipherCore
Post Processor
Bypass FIFO
AUX FIFO
pdi
pdi_valid
pdi_ready
sdi
sdi_valid
sdi_ready
do
do_valid
do_ready
key
rdkey
bdi
npub
nsec
exp_tag
len_a
len_d
tag
bdo
KEY_SIZE
RDKEY_SIZE
DBLK_SIZE
NPUB_SIZE
NSEC_SIZE
TAG_SIZE
CTR_AD_SIZE
CTR_D_SIZE
BS_BYTES
BS_BYTES+1
key_updated
key_needs_update
key_ready
rdkey_ready
rdkey_read
npub_ready
npub_read
nsec_ready
nsec_read
bdi_ready
bdi_proc
bdi_ad
bdi_decrypt
bdi_eot
bdi_eoi
bdi_nodata
bdi_pad_loc
bdi_valid_bytes
bdi_read
bdi_size
exp_tag_ready
msg_auth_done
bypass_full
bypass_wr
bdo_ready
bdo_write
bdo_size
bdo_nsec
tag_ready
tag_write
msg_auth_valid
bdo_data
tag_data
bypass_rd
bypass_empty
bypass_data
aux_fifo_din
aux_fifo_ctrl
aux_fifo_dout
aux_fifo_status
write
full
din
read
empty
dout
ctrl
status

GMU API

- Input processing handled by PreProcessor
  - Key loading/activation
  - Data block loading, padding
  - Keeping track of data to process
  - Etc.

- Output processing handled by PostProcessor
  - Clearing output blocks not belonging to ciphertext or plaintext
  - Conversion of output blocks to data words
  - Holding decrypted plaintext in AUX FIFO buffer until full authentication
  - Generating errors upon authentication failure
  - Etc.

- Actual encryption/decryption handled by CipherCore

- Implementers only have to design CipherCore internals

Design Decisions

- Implementations concern named instance rather than family
- Hence can identify many simplifications, reductions, etc. in general algorithm (see design doc for exhaustive list and details)
- Several variables can be assume constant:
  - CAESAR requires static nonce, key & tag lengths. Use recommended values from Keyak specs:
    - Nonce: 464 or 1200 bits (River vs Lake)
    - Key: 128 bits
    - Tag: 128 bits
  - Sponge-related variables (squeeze & aborb rates, capacity, chaining value length)
  - Named instances have fixed state width & permutation round count

Design Decisions

- Input and Metadata limitations
  - GMU API specifies decrypted plaintext held in AUX FIFO until auth.

  - Hence can ciphertext per *Wrap* call limited to size of AUX FIFO

  - Decided to impose general inputdata limit:
    - Initially squeeze rate Rs
    - Later (Rs-τ) since otherwise extra permutation call would be needed for τ bytes

  - Two versions of each named instance:
    - Version A
      - Also limit on metadata (Ra-Rs), results in reduced area, only 1 permutation call per *Wrap* call (since total data consumed is ((Rs-τ)+(Ra-Rs)) < Ra)

    - Version B
      - No limit on metadata, only limit on inputdata

Design Decisions

- River & Lake Keyak have π = 1, hence no parallelism
  - Require logic for only 1 Piston

  - Eliminate related loops, certain functions (eg. *InjectCollective*)

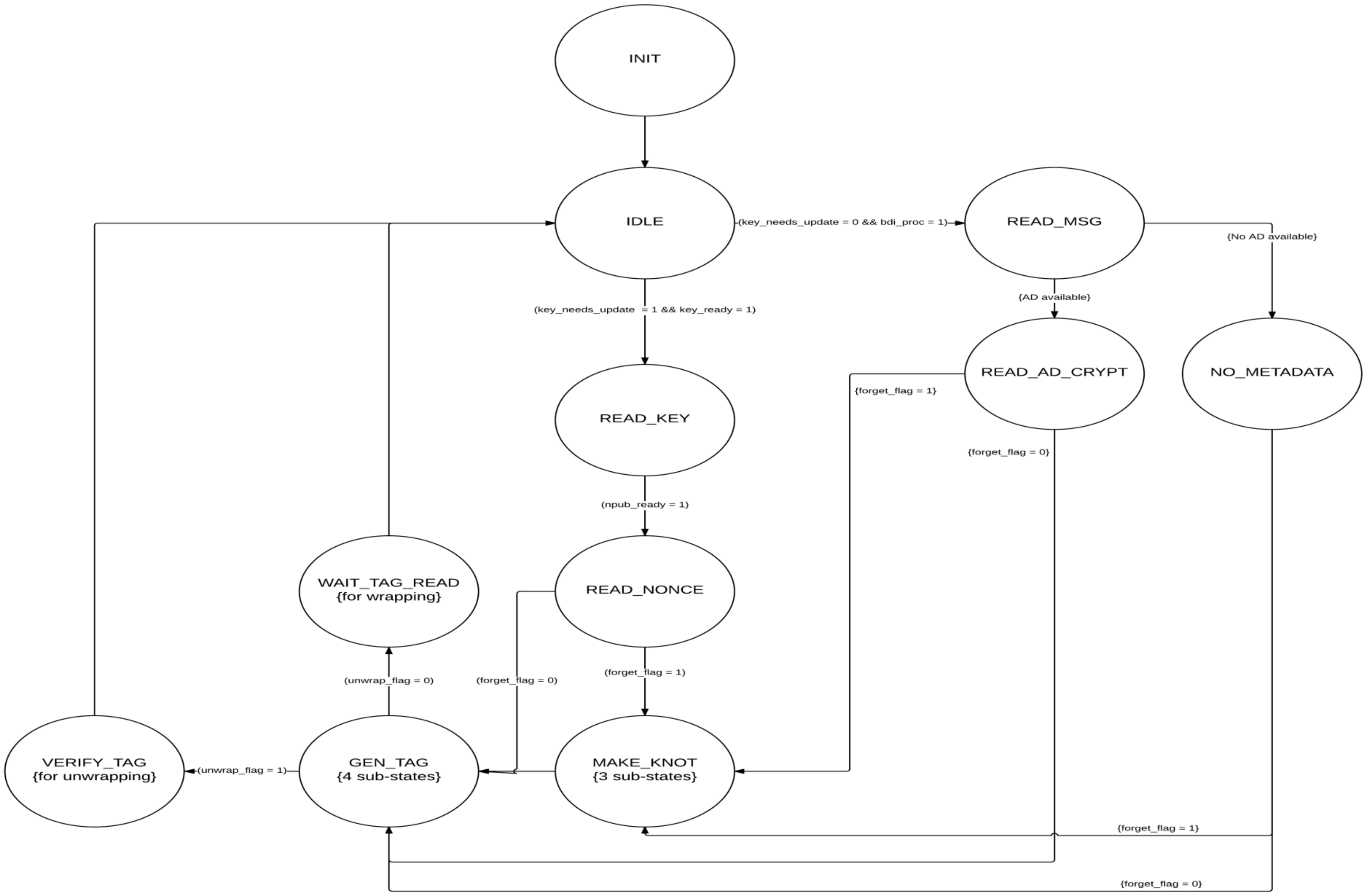
  - Related arrays (eg. Et remembering how much output was used as tag or chaining value) reduced to single values

  - Static SUV diversification bytes (0x01 0x00)

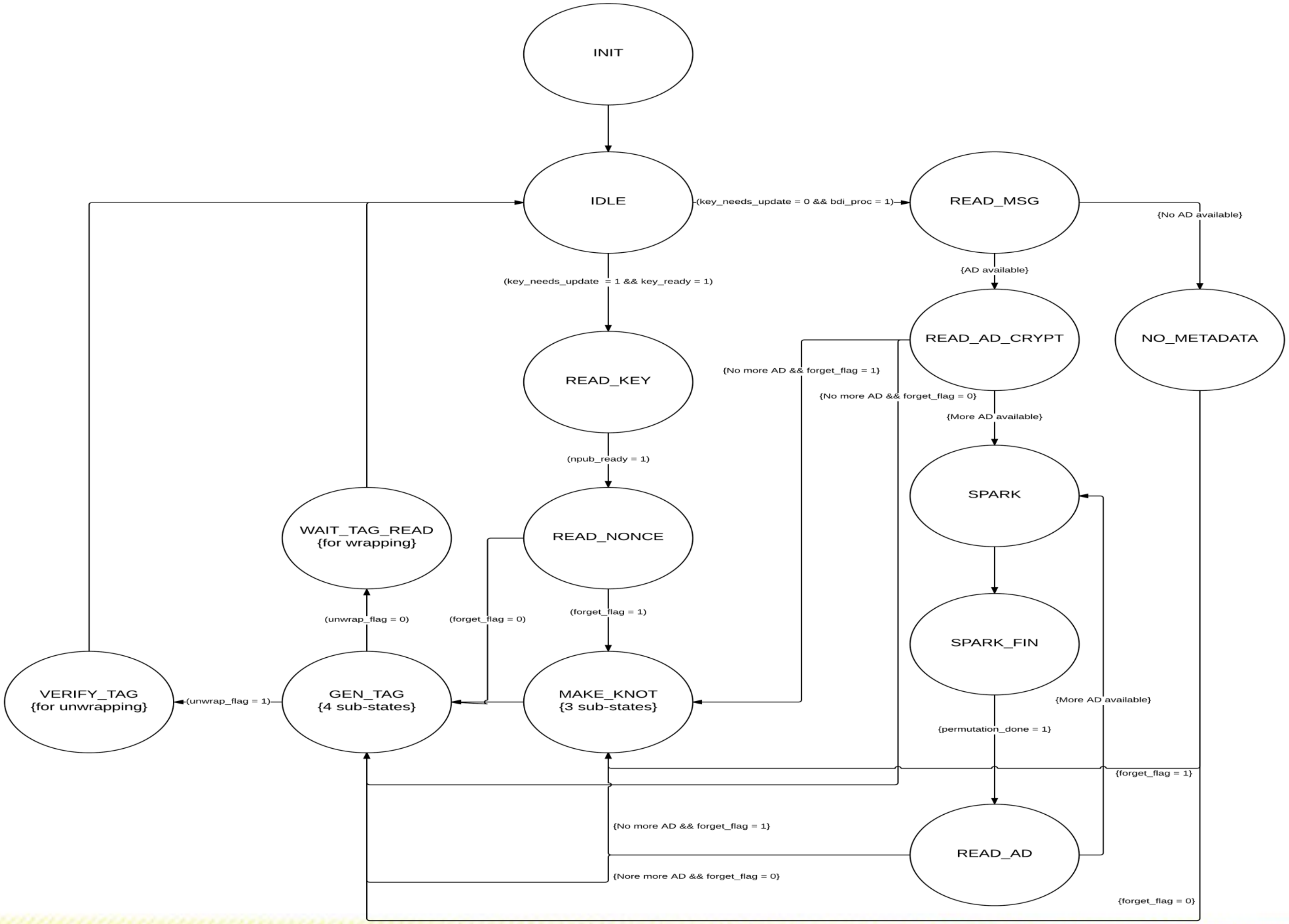
  - Eliminate related conditional execution (eg. (π > 1))

- Collapse sub-components into singular CipherCore
  - Piston, Engine, Motorist: Spread out over VHDL processes in CipherCore
  - Keccak_P round function as sub-component

Design Decisions – FSM (vA)



Design Decisions – FSM (vB)



Design Decisions

- State / Piston functionality in single VHDL process
  - Supports all required piston functionality (crypting, injection, etc.)
  - State is indexed in row/col/lane fashion, transformed to 8-bit byte-wise state indexing variable ω when necessary
- Permutation is done by dedicated VHDL process
  - Drives $Keccak_P$ round sub-component (operates on state register)
  - Iterates for #rounds
  - Signals for 'activate' and 'done'

Design Decisions

- Version B problem: metadata internal buffer
  - When we consume more than (Ra-Rs) metadata we don't consume full blocks per permutation call
  - Need to store remnant data internally for usage during next call
  - Use Ra-sized internal buffer, indexed using exhaustive muxer
  - Alternative: shift register (not explored)
- Diversions from GMU API
  - Separate ports for input and metadata for ease of processing
  - Support for custom flags (eg. forget_flag)
  - Support alternate data processing order (first inputdata, then metadata)

Test Benches

- Test Benches test 1 base case and 3 edge cases, wrapping + unwrapping for each

  - Base Case
    - Data generated using *KeccakTools* test vector generation code
    - Maximum input & (Ra-Rs) metadata

  - Edge Case 1 (vB only)
    - Maximum input & long metadata

  - Edge Case 2
    - Single input & single metadata byte

  - Edge Case 3
    - Some input & no metadata

Performance Figures

- Implementation was *reference* implementation
- Emphasis on legibility / understandability to implementers rather than particular area/throughput optimizations
- Synthesis for FPGA and ASIC *very* intensive when writing non-optimized VHDL code
- Area figures obtained through separate synthesis of round function and preliminary synthesis of 1 instance
- Throughput figures expressed in bits per clock cycle

Performance Figures - Area

Architecture	Area (#slice LUTs)
Keccak$_P$ Round Function (River Keyak)	1,899
Keccak$_P$ Round Function (Lake Keyak)	2,956

Lake Keyak (version A)	Count
RAM (20x64-bit)	1
Multipliers (20x6-bit)	1
Adders/Subtractors	75 (of various bit-sizes)
Comparators	1443 (mostly 32-bit)
Multiplexers	828884 (mostly 1-bit 2-to-1)
XORs	7761
Registers (1216 bit)	1
Registers (1200-bit)	1
Registers (128-bit)	4
Registers (64-bit)	25
Registers (32-bit)	2
Registers (8-bit)	1
Registers (1 bit)	15
Counters	1
FSMs	2

Performance Figures - Throughput

Architecture	Initialization	Wrap
Keyak (version A)	$4 + i * (3 + \#rounds) + j * 2$	$3 + i * (3 + \#rounds) + j * 2 + m$
Keyak (version B)	$4 + i * (3 + \#rounds) + j * 2$	$3 + i * (3 + \#rounds) + j * 2 + m * l$

Architecture	Throughput	Throughput rate (Mbit/s)
$Keccak_P$ Round Function (River Keyak)	$\frac{statesize}{T_{clk}}$	6004.563*
$Keccak_P$ Round Function (Lake Keyak)	$\frac{statesize}{T_{clk}}$	11967.359*

Architecture	Inputdata, Metadata	Throughput rate (bits/clockcycle)
River Keyak (version A)	Inputdata: 52 bytes Metadata: 28 bytes	37.647
River Keyak (version B)	Inpudata: 52 bytes Metadata: 192	56.163
Lake Keyak (version A)	Inputdata: 152 bytes Metadata: 24 bytes	74.105
Lake Keyak (version B)	Inpudata: 152 bytes Metadata: 384	87.510