datto

Encryption at Rest in ZFS

Tom Caputi
tcaputi@datto.com

Overview of Encryption Implementation

What is Encryption?

- **Want to prevent someone (an attacker) from accessing private data**

- **Permissions aren’t good enough**
  - Root user can always access every file
  - Kernel bugs can result in privilege escalation
  - Disks can always be moved to another machine / OS and read

- **Solution: Encryption**
  - Data on disk should look pseudorandom (no detectable patterns)
  - User has a secret key that can be used to access the data
  - Mathematically, data is extremely hard to decrypt

Problems with Non-Native Encryption

- **File Level Encryption** (eg. ecryptfs)
  - Encryption before compression -> no compression
  - No dedup capabilities (within dataset)
  - Writes a metadata header, can disturb file alignment or waste space
- **Disk Level Encryption** (eg. dm-crypt)
  - Multiple copies are encrypted multiple times
  - Keys must always be loaded or pool is useless
    - No scrub, resilver, etc
    - No possibility of doing `zfs send` without keys loaded

- **Complex management**

How is this important to Datto?

- **Our primary backup solution for our partners**
  - A backup agent runs on our client's machines
  - Backups are sent to our device on the client's network
  - Backups are replicated to servers in the cloud (`zfs send`)

- **Advantages of Native Encryption**
  - Higher performance encryption, without losing compression
  - Much cleaner implementation than current stacked block devices
  - Ability to backup customer data without liability

What is Encrypted?

Encrypted

- File data and metadata
  - ACLs, names, permissions, attrs
- Directory listings
- All Zvol data
- FUID Mappings
- Master encryption keys
- All of the above in the L2ARC
- All of the above in the ZIL

Not Encrypted

- Dataset / snapshot names
- Dataset properties
- Pool layout
- ZFS Structure
- Dedup tables
- Everything in RAM

Keystore API

- **ZFS Encryption Commands**
  - `zfs create -o encryption=<enc> -o keysource=<ks>`
  - `zfs key -l <dataset>` : Loads a user's key into zfs for use
  - `zfs key -u <dataset>` : Unloads a user's key from the system
  - `zfs key -c <dataset>` : Changes a user's key
  - `zfs mount, zfs umount, zpool import, zpool export`

- When key is loaded datasets are mountable (fs) / openable (zvol)
- Child datasets inherit encryption algorithm and keysource by default
- **Key / key source changeable without re-encrypting dataset**

Encryption Administration

- **Algorithms**
  - AES-CCM, AES-GCM
  - 128 bit, 192 bit, 256 bit
  - `encryption=on` defaults to AES-CCM-256 bit
- **Key Sources**
  - File, prompt
  - Raw, hex, passphrase
  - Variable PBKDF2 iterations (more later)
- **Properties**
  - `encryption, keysource, keystatus, pbkdf2iters`

Caveats of Native ZFS Encryption

- Limited to `copies=2`
- Dedup tables are not encrypted
  - Dedup will leak data about equivalent data blocks
  - Dedup will only work within “clone families”
- Encryption + compression could allow for a CRIME attack
  - Not relevant to most applications
  - Can be prevented with `compression=off`

Data Encryption in ZFS

From the Ground Up

Encryption Scope

- File Level Encryption
  - Store encryption parameters as file metadata
  - How to encrypt large files without rewriting for every update?
  - What happens if the file metadata is corrupted / lost?

- **Block Level Encryption**
  - **Encrypt each block separately**
  - **Store the encryption parameters in `blkptr_t`**
  - **Limits the scope to a single block**
    - **Encryption, decryption, data loss**

Types of Encryption

- Asymmetric encryption
  - Public / private keypair
  - Slow
  - Good for verifying identity of communicating parties
  - Examples: SSH handshake, TLS handshake

- **Symmetric encryption**
  - **Single key for encryption / decryption**
  - **Fast (AES-NI instruction set on Intel x86_64, almost 1000x faster)**
  - **Examples: TLS (post handshake), dm-crypt, etc.**

Symmetric Encryption: Block Cipher

On-Disk
RAM Only
Procedure

- **Block Cipher**
  - Used to transform individual blocks of plaintext
  - AES is the current standard (built into Intel x86_64)
  - Works on a fixed block (AES is 128 bits)

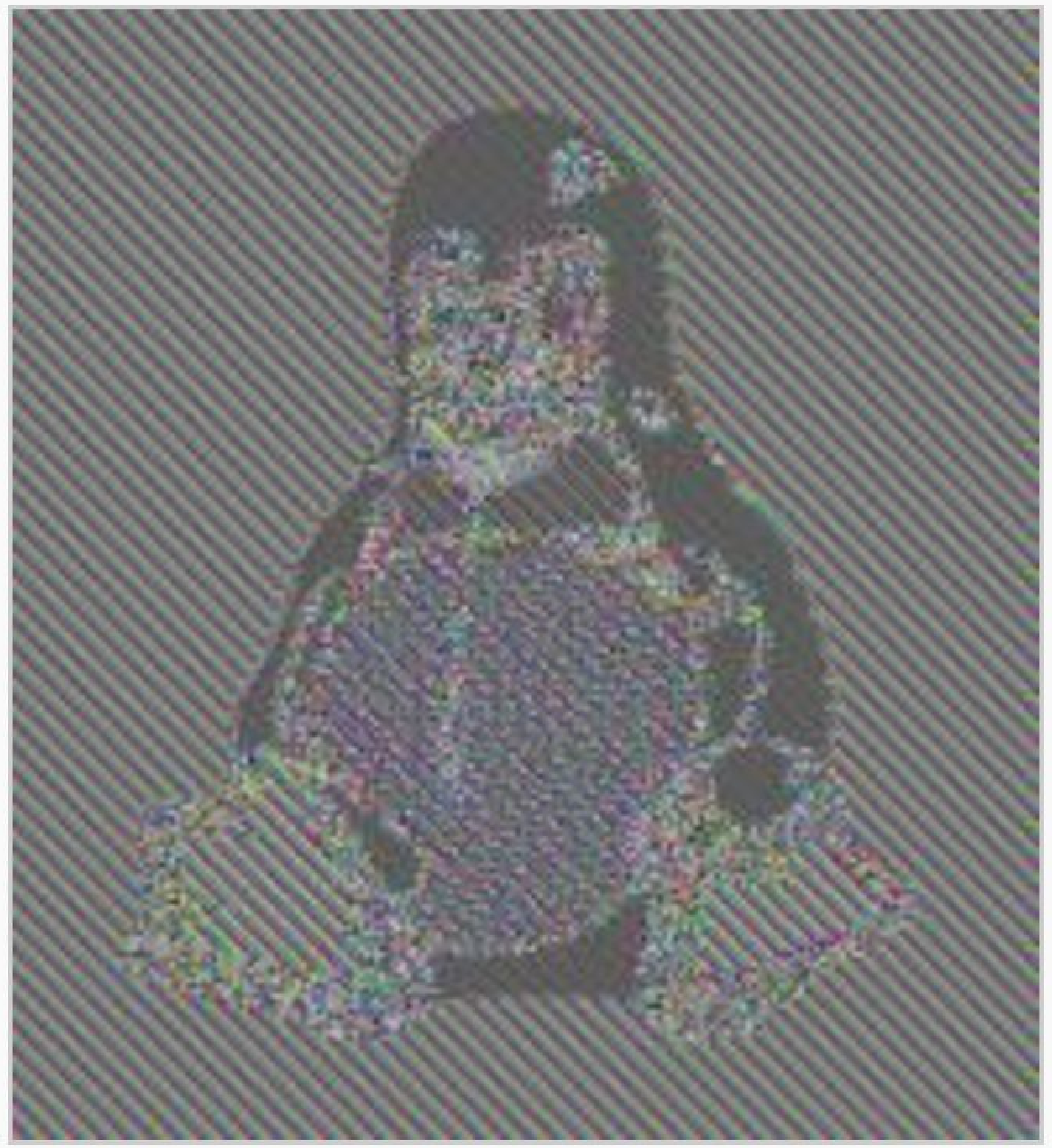
Symmetric Encryption: Stream Cipher

- On-Disk
- RAM Only
- Procedure

- **Block CIpher Mode of Operation**
  - Allows encryption of arbitrary lengths of plaindata
  - Successively applies AES to each block in the plaindata
  - Mode is called Electronic Cookbook (ECB)

ECB Encryption Problem

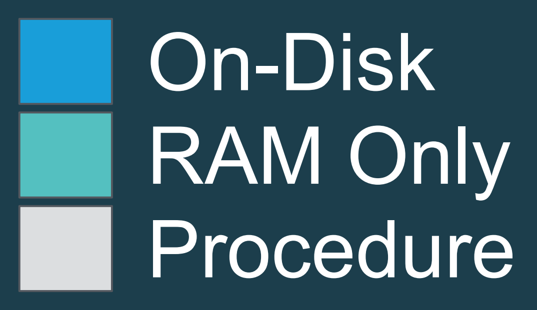
Original image

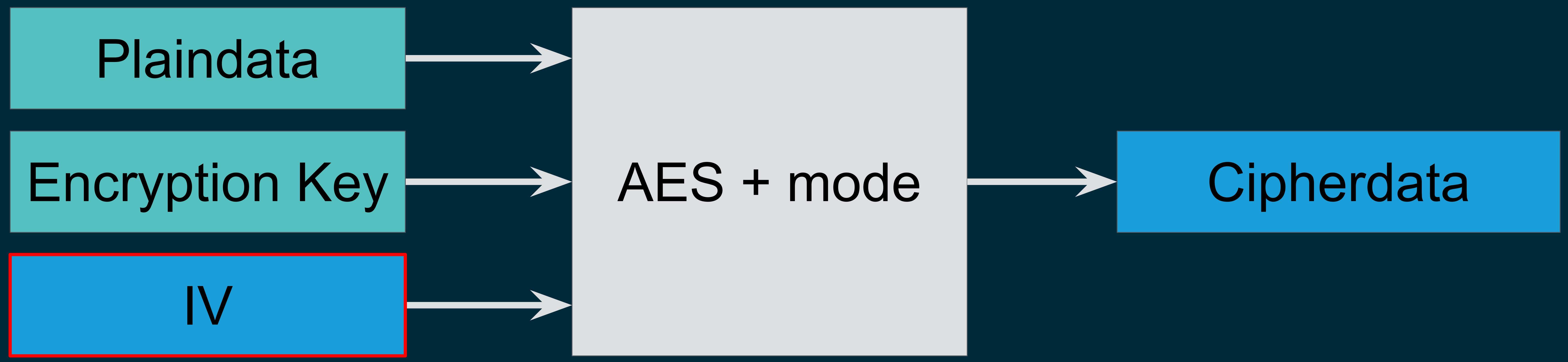


Encrypted using ECB mode

Modes other than ECB result in pseudo-randomness

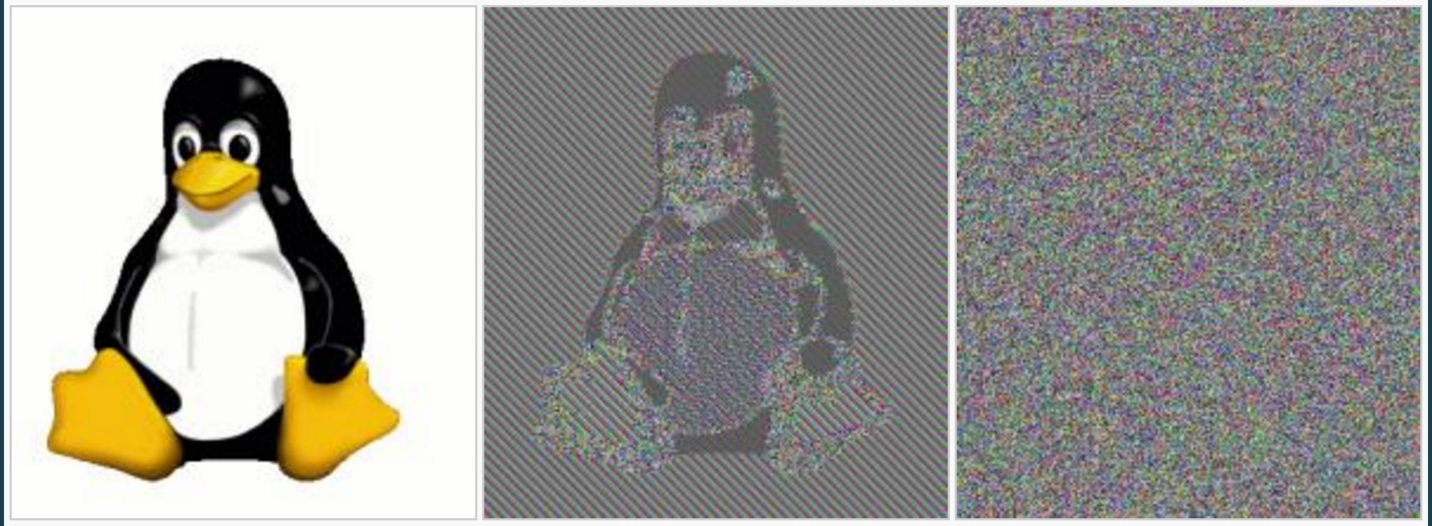
Confidential Stream Cipher





- **Confidential Block Cipher Modes**
  - Initialization Vector (IV) acts as salt for the first block
  - Blocks after the first are used to “salt” the next block

Initialization Vectors (IV)



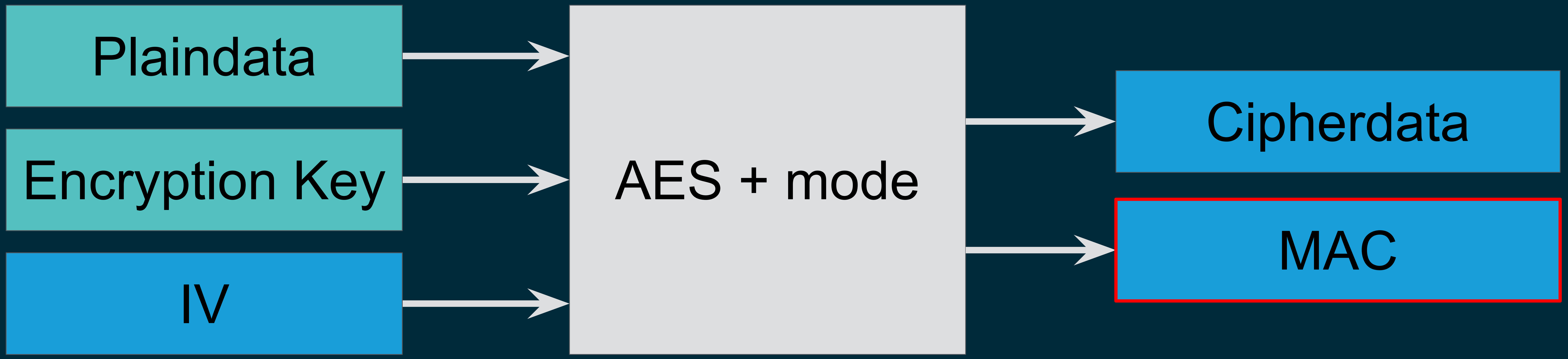
Original image | Encrypted using ECB mode | Modes other than ECB result in pseudo-randomness

- Used as a salt for the encryption algorithm
- Prevents equivalent plaintext blocks -> equivalent ciphertext blocks
  - When used with a proper mode

- Different modes have different IV requirements
  - GCM and CCM require:
    - Up to 104 bits (13 bytes), 96 bits recommended by NIST
    - Reusing an IV + key results in **CATASTROPHIC FAILURE**

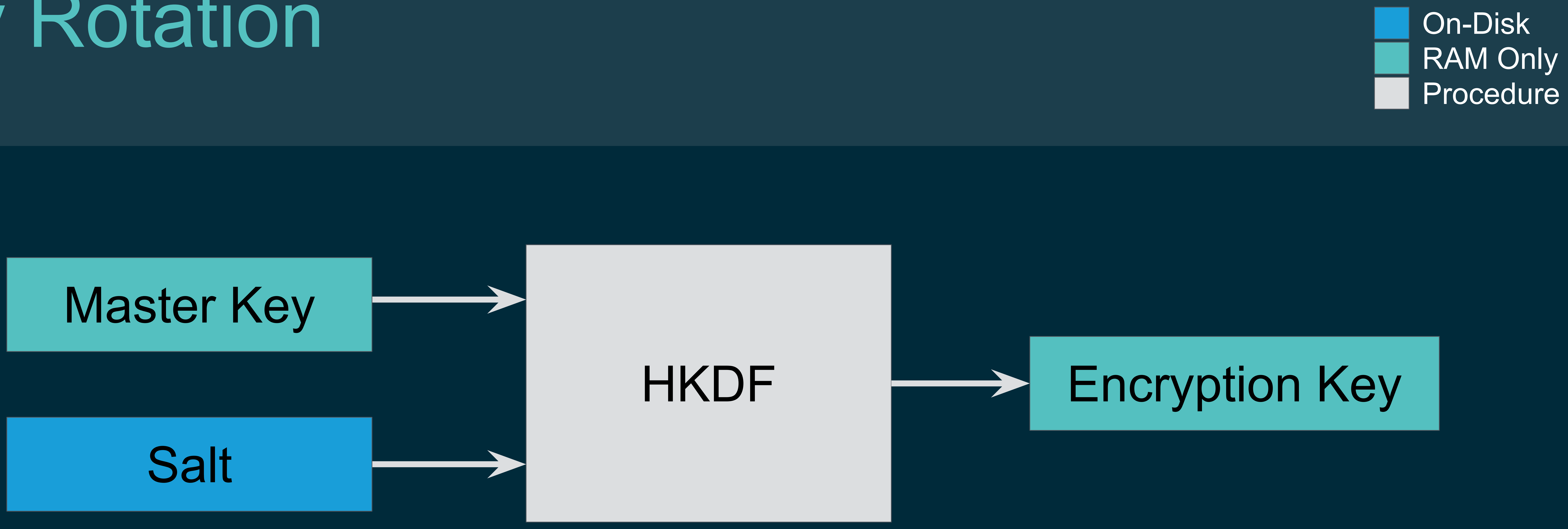
Authenticated Encryption

On-Disk
RAM Only
Procedure



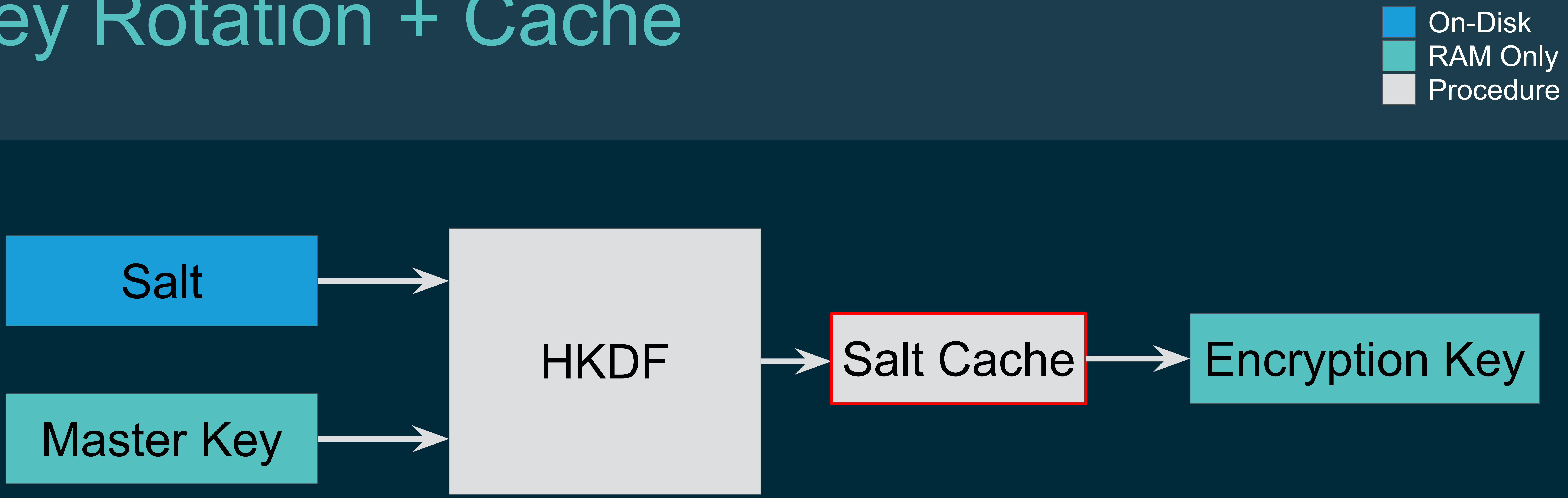
- **Authenticated Encryption (AE or AEAD)**
  - Encryption also produces a Message Authentication Code (MAC)
  - MAC is a checksum that requires a secret key to produce
  - Prevents an attacker from filling the ciphertext with garbage undetected

Key Rotation



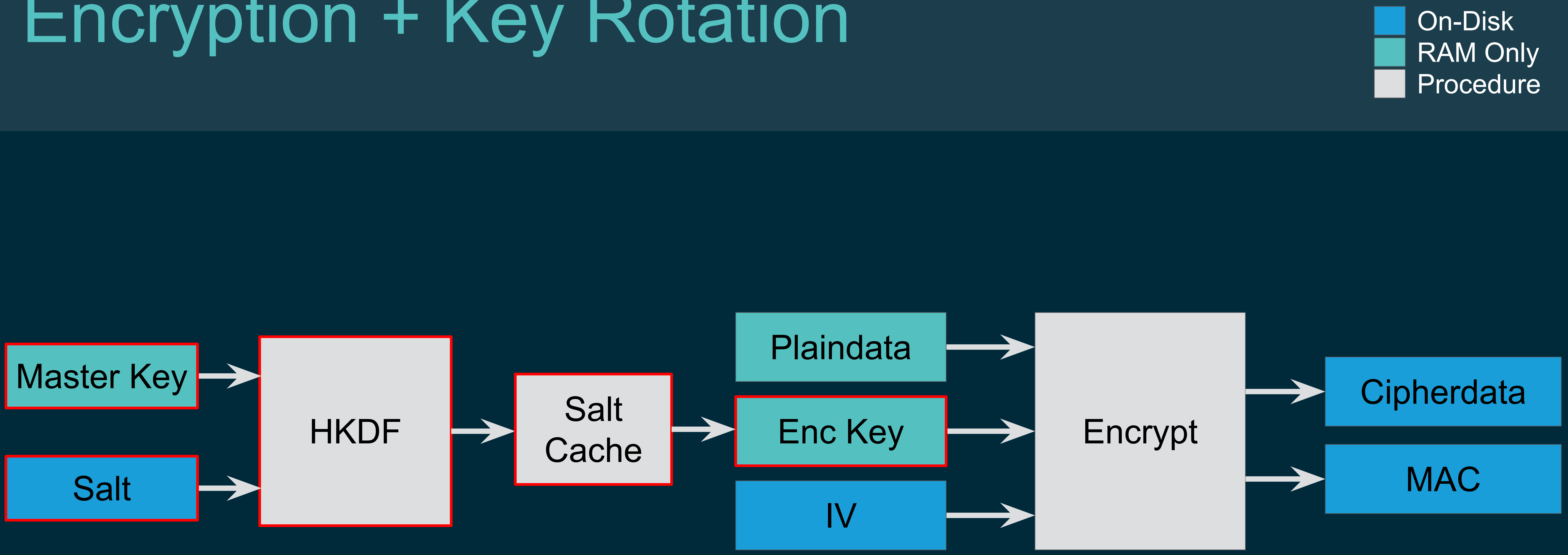
- **Hash-Based Key Derivation Function (HKDF)**
  - Generates an encryption key from a master key + salt
  - Relatively inexpensive to calculate
  - Prevents Master key from getting stale due to IV collisions, algorithm limits

Key Rotation + Cache



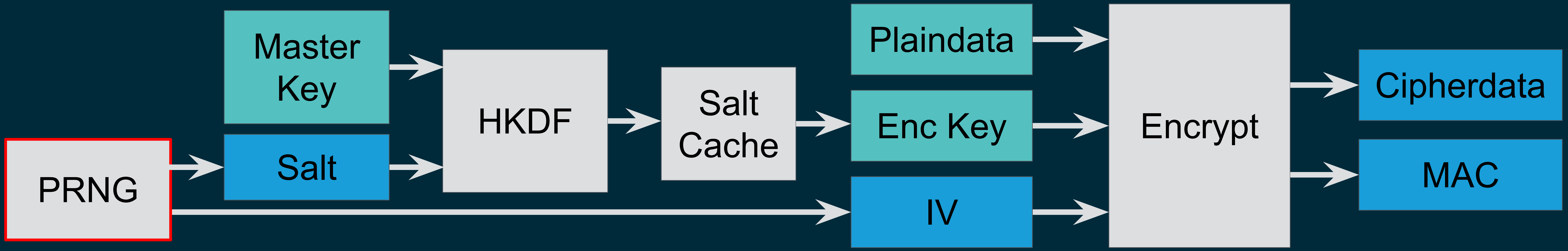
- **Salt + Encryption Key Cache**
  - Current key doesn’t go stale for a while
  - Cache the current one for faster encryption
  - Doesn’t help decrypting older data

Encryption + Key Rotation



Generating the IV and Salt

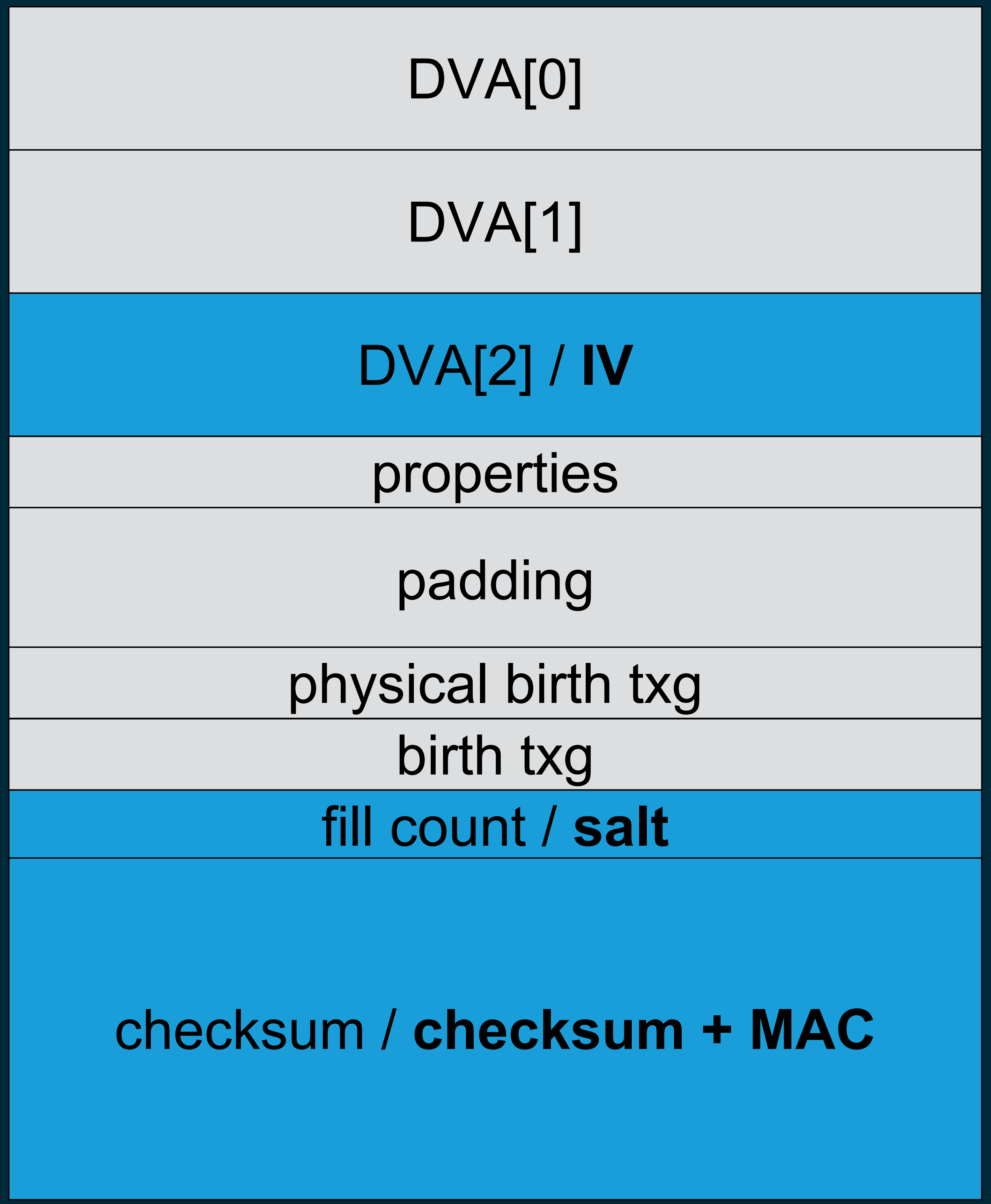
On-Disk
RAM Only
Procedure



- **Pseudo Random Number Generator (PRNG)**
  - 96 bit IV + 64 bit salt = 160 bits of entropy
  - 1 / 1 billion chance of collision after 5.406e+19 blocks
  - 41141552 years at 1 million blocks per second

Encryption Parameters: `blkptr_t`

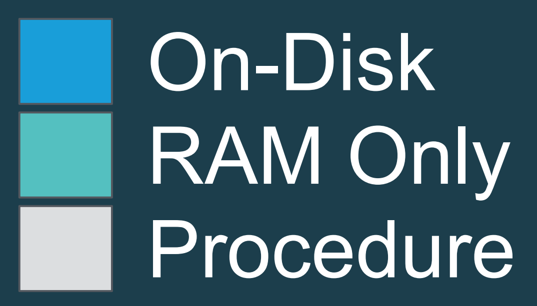
- **Salt** (64 bits)
- **MAC** (128 bits)
  - Occupies ½ of checksum
  - Serves similar purpose to checksum
  - Normal checksum allows for scrubbing
- **IV** (96 bits)
  - Would use too much of padding
  - Disadvantages to deriving from other fields
    - `zbookmark_phys_t`
    - DVA[0] + birth txg + salt
  - Limits copies=2

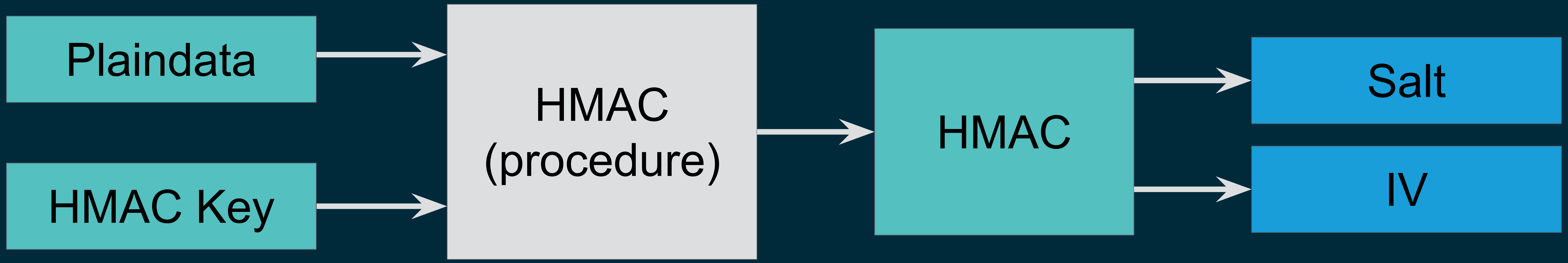


Dedup Encryption Parameters: Concept

- In order for dedup to work, MAC + checksum must match
  - IV + salt must match for equivalent data
  - Normally, reusing the IV + key results in **CATASTROPHIC FAILURE**
  - We will only use the same IV + key when data is equivalent as well
    - In this case we have simply duplicated what we had before
    - Leaks the info that the blocks are the same
    - Dedup leaks this info anyway

Dedup Encryption Parameters: HMAC

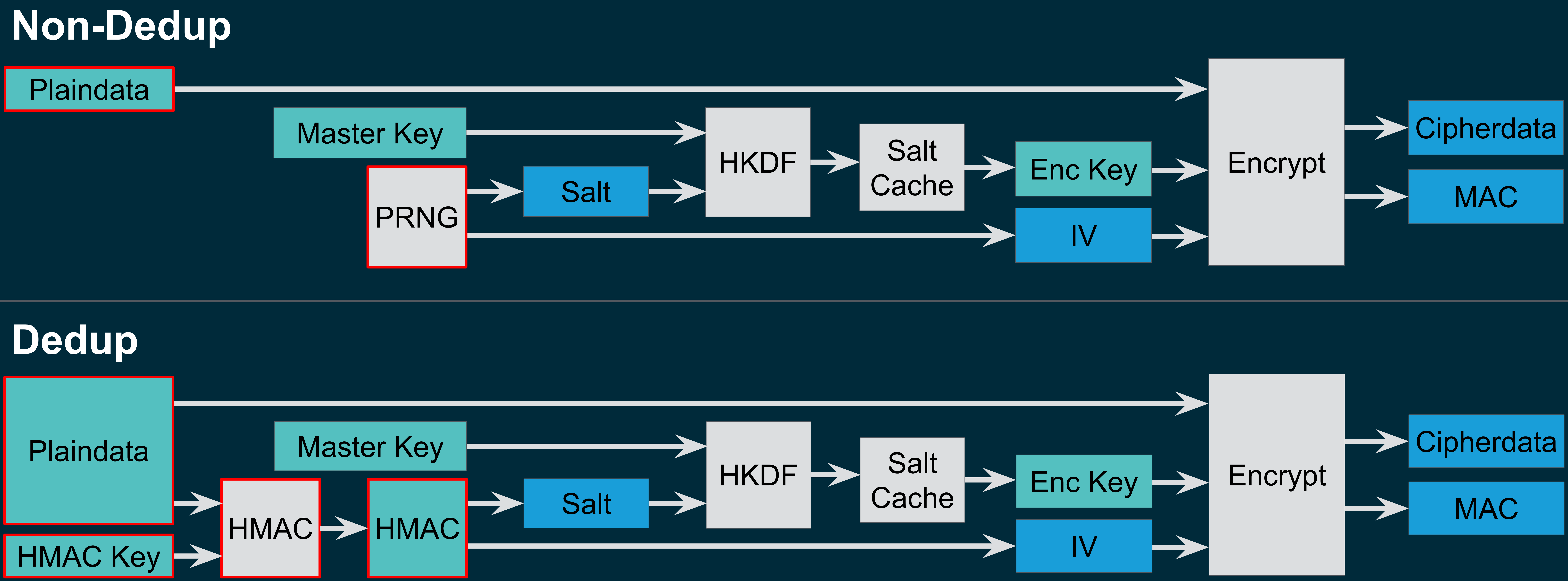




- **Hash-Based Message Authentication Code**
  - Similar to MAC, generated without producing ciphertext
  - HMAC key stored alongside the master key
  - 64 bits to salt, 96 bits to IV

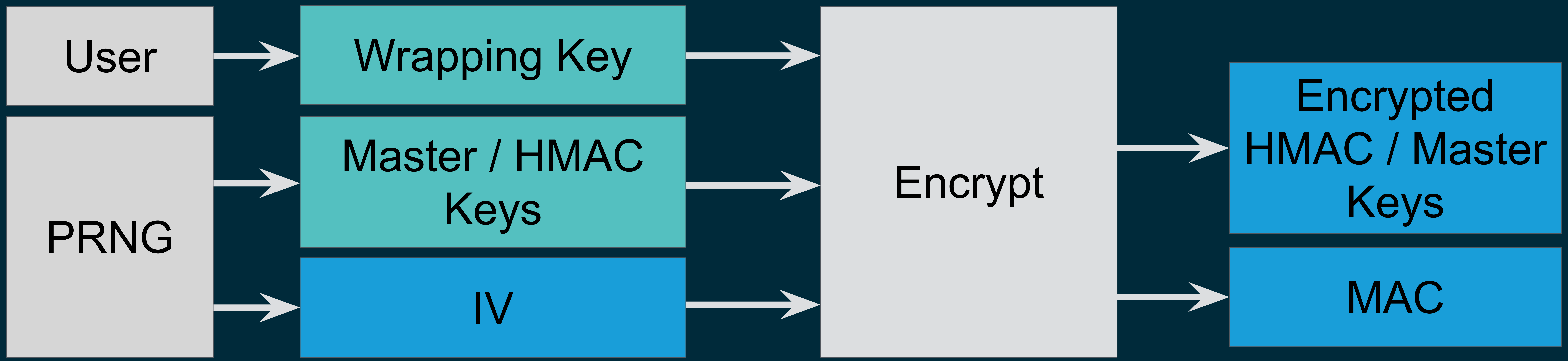
Dedup vs Non-Dedup Encryption

- On-Disk
- RAM Only
- Procedure



Allowing the User to Change the Key

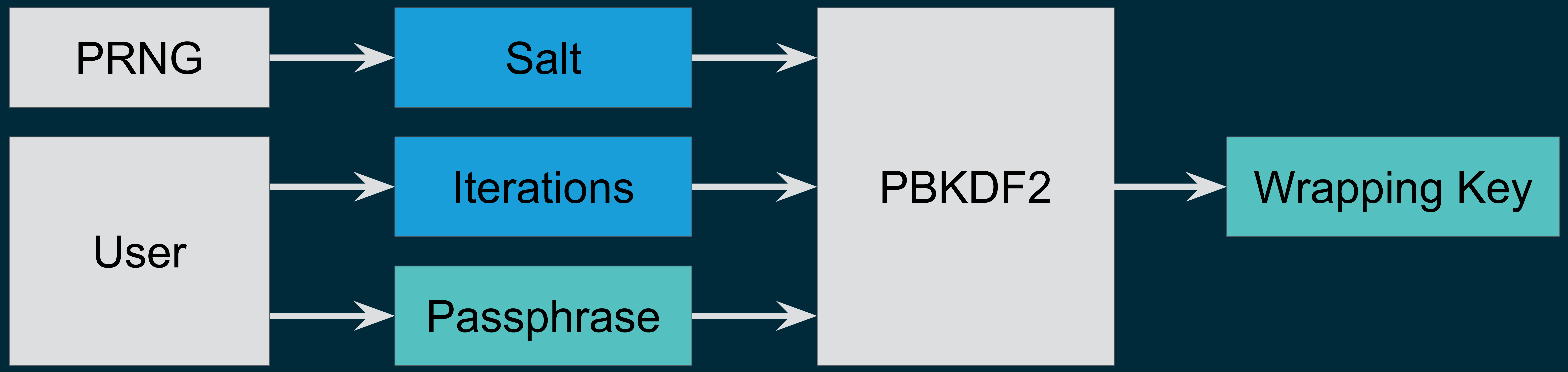
On-Disk
RAM Only
Procedure



- **Wrapping Key**
  - Provided by the user
  - Used to encrypt the randomly generated master key
  - Master key never exposed to the user

Passphrase Based Keys

On-Disk
RAM Only
Procedure



- **Passphrase Based Key Derivation Function (PBKDF2)**
  - Passphrases are variable length, low entropy
  - Turns passphrase into a high entropy key
  - CPU Intensive to calculate to prevent brute force attacks

Additional Topics

Additional Topics: ZIL Encryption

- **ZIL blocks are preallocated**
  - Must pre-assign salt / IV
  - Must store MAC in ZIL header (since bp will not be rewritten)

- **ZIL blocks need to be claimable without loaded keys**
  - Leave ZIL structure metadata unencrypted
    - `zil_chain_t`, `lr_common_t`, `blkptr_t` from `TX_WRITE`
    - Data blocks from `TX_WRITE` can be encrypted normally

- **ZIL blocks are rewritten for every log record**
  - Real IV = generated IV + `zc_nused` from `zil_chain_t`

Additional Topics: L2ARC Encryption

- **Goals / Challenges**
  - No extra data stored in L2ARC header
  - Data encrypted in the L2ARC, decrypted but compressed in L1ARC
  - L2ARC read code verifies against `blkptr_t`'s checksum

- **Implementation**
  - Store data on disk as it exists in the pool
  - New L1ARC header (normal L1 header + encryption params)
  - Encryption parameters move with the header until buffer is written out
  - On read, decryption params provided by caller's `blkptr_t`

Additional Topics: Raw Sends

- Ability to replicate a dataset without having the keys loaded
- Just send the data as it exists on disk
  - Also need to send the IV / MAC
  - Very similar concept to recently merged compressed send feature
- ZFS can be a true platform for end-to-end encryption
  - Backups to untrusted servers is possible
  - Admin can always replicate data
- Coming soon….

Current Status

- Fully implemented (except for raw sends)
- Ready for review
- Pull requests are out for Linux, OSX, Illumos
  - Primary PR is on Linux

- **Special Thanks**
  - Jorgen Lundman for maintaining the ports to OSX and Illumos
  - Matt Ahrens and Brian Behlendorf for all the help answering my questions
  - George Wilson and Dan Kimmel for helping me through the ARC changes

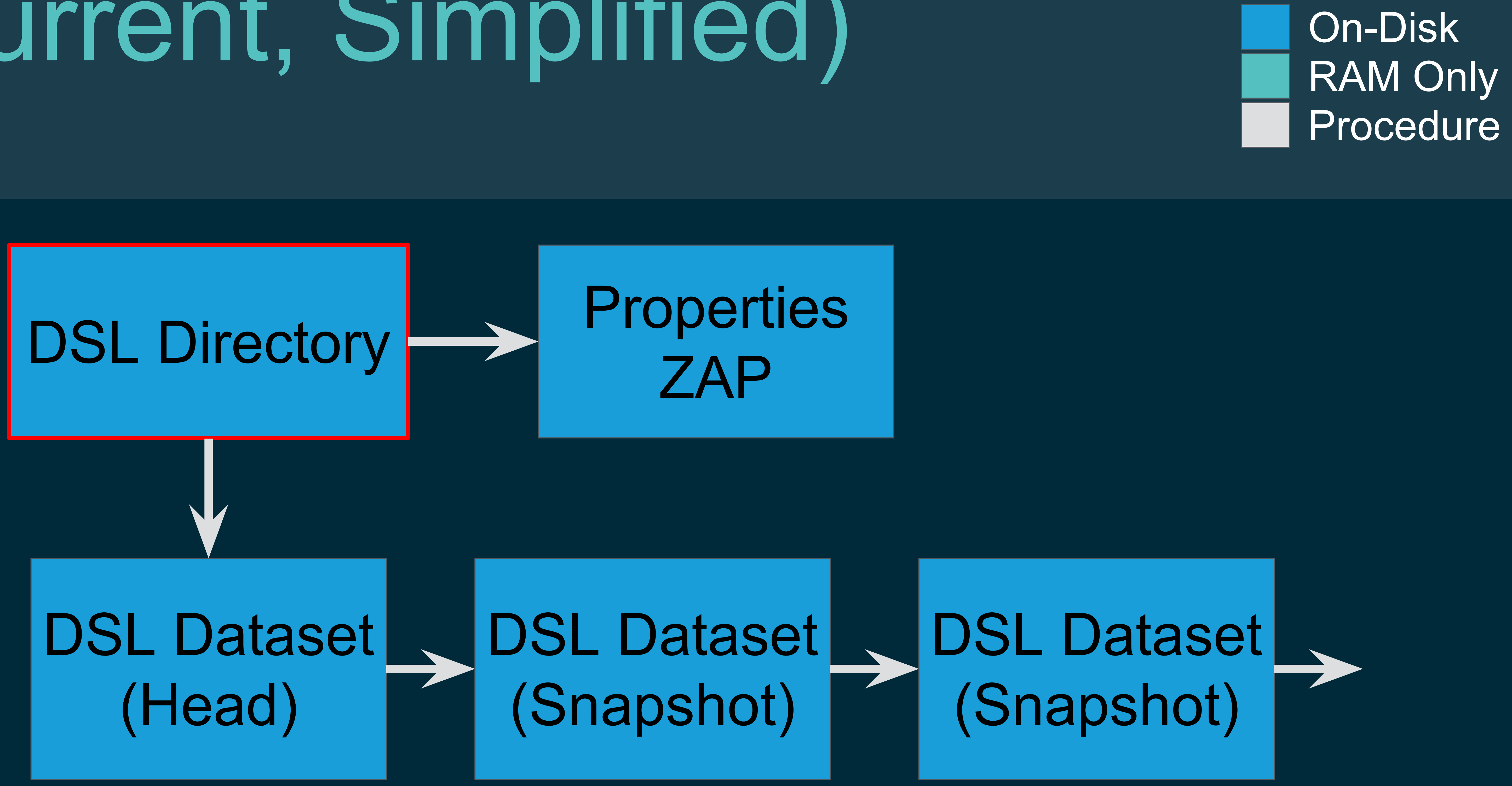
datto

Questions?

Tom Caputi
tcaputi@datto.com
https://github.com/zfsonlinux/zfs/pull/4329

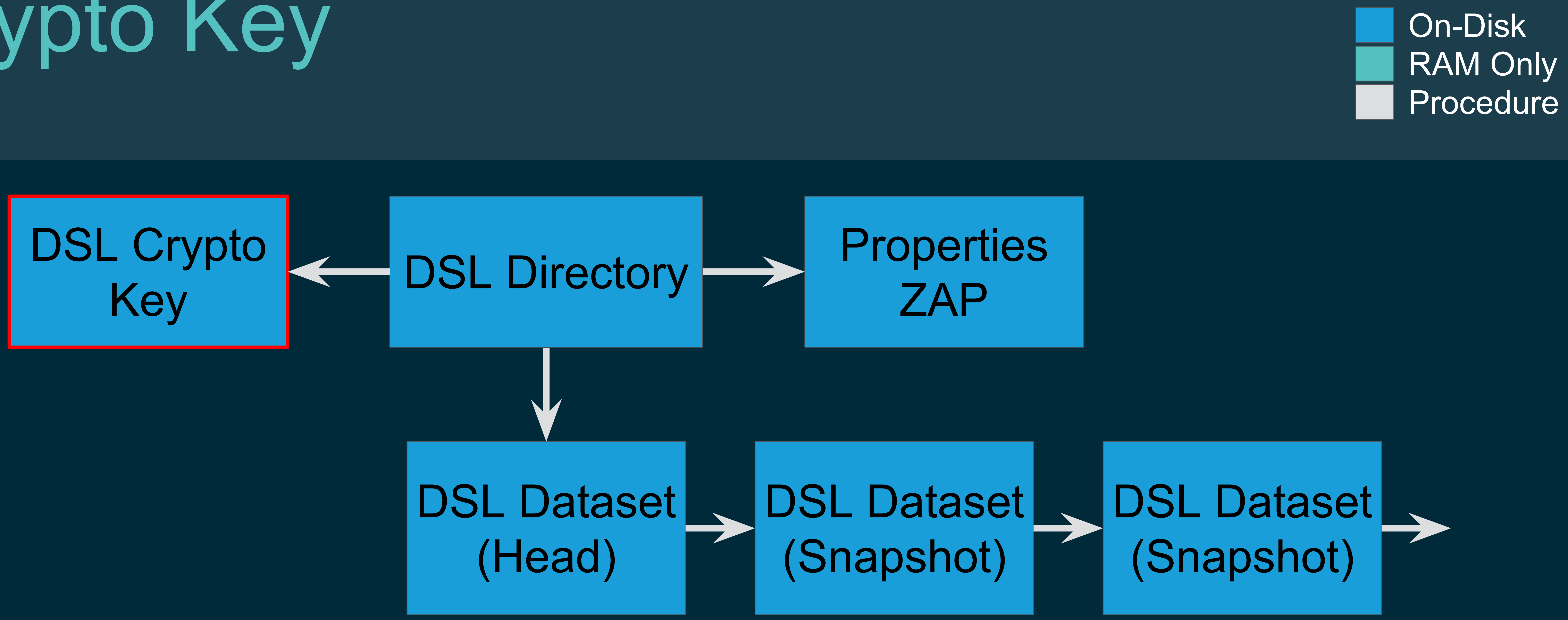
Appendix: Keystore

DSL Directory (Current, Simplified)



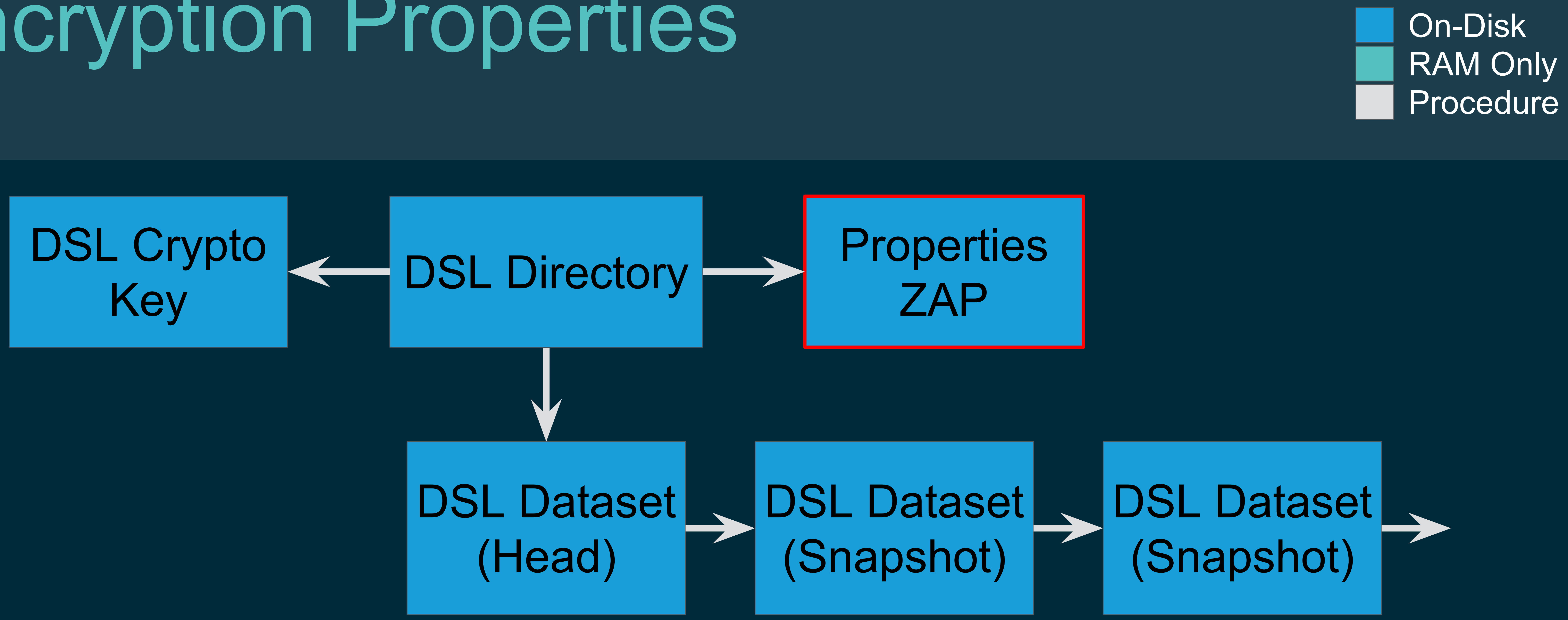
- **DSL Directory**
  - A dataset and all snapshots
  - Pointers to properties object, linked list of snapshots, child map

DSL Crypto Key



- **DSL Crypto Key**
  - ZAP
  - One per DSL Directory (snapshots share)
  - Holds Encrypted Master / HMAC Keys, wrapping IV + MAC

New Encryption Properties



- **New Encryption Properties**
  - Encryption algorithm
  - Key source
  - PBKDF2 params: salt, iterations

In-Core Keystore

On-Disk
RAM Only
Procedure

- **SPA Keystore**
  - Wrapping Key will work for DSL Directory and all children
  - All snapshots within a DSL Directory will share a DSL Crypto Key
  - All three structs maintained in AVL trees added to the SPA

In-Core Keystore: Wrapping Keys

On-Disk
RAM Only
Procedure

- **Wrapping Keys**
  - Provided by the user
  - Managed with `zfs key` command
  - Keys are unloadable when refcount is zero

datto

In-Core Keystore: DSL Crypto Keys

On-Disk
RAM Only
Procedure

- **DSL Crypto Keys**
  - Holds Master / HMAC keys, salt cache
  - Immediately evicted when refcount is zero

In-Core Keystore: Key Mappings

On-Disk
RAM Only
Procedure

- **Key Mappings**
  - Created when dataset is owned (with a few exceptions)
  - Loads the DSL Crypto Key from disk on creation (if it isn't already)
  - Simply allows ZIO layer to lookup DSL Crypto Keys via the Dataset ID