**Goal:** SHA3-256 is too slow on Cortex-M0+ and similar targets. Propose either:

- Lightweight cryptographic hash alternatives (e.g. BLAKE2s, SHAKE)
- Partial truncation or pre-hashing strategies
- Hardware mapping or logic optimization

Output:

1. Comparative timing table (SHA3 vs. alternatives)
2. Suitability for deterministic constant-time usage
3. Impact on IND-CCA security under ROM
4. Implementation outline for M0+ or RISC-V

Use technical depth and embed constraints.

**Constraints:**

- Cortex-M0+: Limited clock speed, minimal instruction set, extremely low RAM.
- RISC-V M-class: Variable instruction sets (e.g., no multiplication), but generally more efficient than M0+.
- Constant-time: Crucial to prevent timing side-channel attacks.
- ROM/Flash: Limited space for code.

**1. Comparative Timing Table (SHA3 vs. Alternatives)**

It's difficult to give *exact* numbers without specific implementations and benchmarking, but here's a relative comparison:

| Algorithm | Relative Speed (vs. SHA3) | Notes |
| :---------- | :------------------------ | :---------------------------------------------------------------- |
| SHA3-256 | 1x | Baseline, slow |
| BLAKE2s | 2x - 4x | Faster, SIMD-friendly (less relevant on M0+), constant-time possible |
| SHAKE128 | 1.5x - 3x | Variable output, potentially faster for smaller key sizes |
| Trunc. SHA3 | 1.2x - 2x | Truncating output, speedup varies with truncation |
| Pre-hash + AES | 3x - 5x | If AES hw available |

**2. Suitability for Deterministic Constant-Time Usage**

- **SHA3-256:** Difficult to implement constant-time in software, especially on M0+. Any branching or table lookups based on input data must be carefully handled.
- **BLAKE2s:** Designed with constant-time implementations in mind. SIMD optimizations can be avoided for embedded, and the core logic is simpler than SHA3.
- **SHAKE128:** Similar to SHA3, but the variable output length adds complexity for constant-time. If output length is fixed, it's more feasible.
- **Trunc. SHA3:** Truncation itself is constant-time, but the underlying SHA3 must be constant-time.
- **Pre-hash + AES:** AES (if hardware-accelerated) is inherently constant-time. The pre-hash (e.g., BLAKE2s) must also be constant-time.

**3. Impact on IND-CCA Security under ROM**

- **SHA3-256 Replacement:**
  - Replacing SHA3-256 with a cryptographically sound alternative (BLAKE2s, SHAKE) *should* maintain IND-CCA security *if* the replacement is also modeled as a Random Oracle.
  - The core security argument (adversary's inability to distinguish ciphertexts) still holds if the hash function's output is unpredictable.
- **Truncation:**

  - Truncating the hash output *reduces* the security level. If we truncate SHA3-256 to 128 bits, the security is at most 128 bits.

  - This might be acceptable if the overall system security budget allows it.

- **Pre-hashing:**

  - Pre-hashing (e.g., `K = AES( prehash(v || r) )`) is acceptable *if* the pre-hash is collision-resistant and the AES key is derived securely.

  - This shifts the "randomness" requirement from SHA3 to the pre-hash.

**4. Implementation Outline for M0+ or RISC-V**

- **BLAKE2s (Recommended):**

  - M0+: Implement the BLAKE2s core logic carefully, avoiding table lookups where possible. Optimize the G-function. Unroll loops judiciously for speed, but be mindful of code size.

  - RISC-V: Leverage any available multiplication instructions. Explore register allocation to minimize memory accesses.

  - Both: Test *thoroughly* for constant-time behavior.

- **SHAKE128:**

  - Similar to SHA3, but focus on optimizing the KECCAK-p permutation.

  - If variable output length is needed, this adds complexity. If the output length is fixed, it simplifies the implementation.

- **Truncated SHA3:**

  - Implement SHA3-256 constant-time (very challenging on M0+).

  - Truncate the output (simple bitwise operation).

  - This is a trade-off: speed vs. security.

- **Pre-hash + AES:**

  - If *hardware AES* is available (some Cortex-M4F have it, less common on M0+ or RISC-V M), this is the fastest option.

  - Use BLAKE2s or a fast alternative for the pre-hash.

  - Derive the AES key carefully (e.g., from a master secret using HKDF).

**Example: BLAKE2s on Cortex-M0+**

```c
// Simplified example (not complete or fully optimized)

void blake2s_keyed_hash(uint8_t *out, const uint8_t *in, size_t inlen,
                        const uint8_t *key, size_t keylen) {
  uint32_t h[8] = { // IV
      0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A,
      0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19};

  // ... Key setup (if any) ...

  for (size_t i = 0; i < inlen; i += BLAKE2S_BLOCKBYTES) {
    uint8_t block[BLAKE2S_BLOCKBYTES];
    // ... Pad input block ...

    uint32_t m[16];
    // ... Load block into m ...

    // The G function (simplified)
    #define G(r, i, a, b, c, d)                                 \
      {                                                         \
        h[a] = h[a] + h[b] + m[blake2s_sigma[r][2 * i + 0]];    \
        h[d] = ROR32(h[d] ^ h[a], 16);                          \
        h[c] = h[c] + h[d];                                     \
        h[b] = ROR32(h[b] ^ h[c], 12);                          \
```

```
    h[a] = h[a] + h[b] + m[blake2s_sigma[r][2 * i + 1]];                \
    h[d] = ROR32(h[d] ^ h[a], 8);                                       \
    h[c] = h[c] + h[d];                                                 \
    h[b] = ROR32(h[b] ^ h[c], 7);                                       \
   }

  // ... Round loop (careful unrolling) ...
  for (int r = 0; r < 10; ++r) {
    G(r, 0, 0, 4, 8, 12);
    G(r, 1, 1, 5, 9, 13);
    G(r, 2, 2, 6, 10, 14);
    G(r, 3, 3, 7, 11, 15);
    G(r, 4, 0, 5, 10, 15);
    G(r, 5, 1, 6, 11, 12);
    G(r, 6, 2, 7, 8, 13);
    G(r, 7, 3, 4, 9, 14);
  }
 }

 // ... Finalization ...
 memcpy(out, h, BLAKE2S_OUTBYTES);
}
```