Implementations of PQC on Constrained Devices

A low-stack implementation of HAETAE[1]

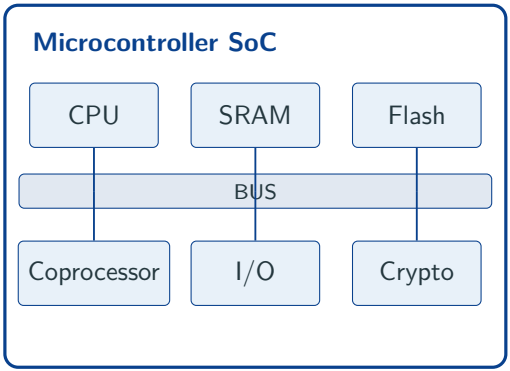
Gustavo Banegas

LIX, CNRS, Inria, École Polytechnique



[1] Joint work with: YoungBeom Kim, Christine van Vredendaal, and Seog Chung Seo

PQC belongs on the device, too



Typical System-on-Chip (SoC)

- **Secure boot:** verify the firmware before running it.
- **Updates:** authenticate new firmware in the field.
- **Attestation:** sign or verify device identity and state.

What does “constrained” mean here?

Devices in the evaluation

- 8-bit AVR: **8 KiB SRAM**
- Cortex-M0+: **12 KiB SRAM**
- Also: MSP430, Cortex-M4, and RISC-V

The CPU, memory size, and software environment vary, but the same cryptographic operation must still run.

The practical budget

- **SRAM:** stack, signature, keys, and application state
- **Flash:** implementation and firmware
- **Time / energy:** cycles for signing and verification
- **Communication:** public keys and signatures

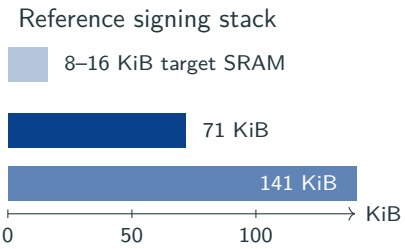
HAETAE: compact signatures, expensive temporary memory

Why consider HAETAE?

A module-lattice signature scheme with shorter signatures and public keys than ML-DSA at comparable security levels.

The obstacle

The reference signer uses **71–141 KiB of stack**, depending on the security level.



HAETAE in one slide

- Module-lattice signature over $R_q = \mathbb{Z}_q[X]/(X^{256}+1)$, $q = 64{,}513$.
- Fiat–Shamir with aborts: commitment w, challenge c, response z.
- Hyperball sampling produces compact signatures, but complicates memory use.

	H-2	H-3	H-5
(k, ℓ)	$(2, 4)$	$(3, 6)$	$(4, 7)$
Public key (B)	992	1,472	2,080
Secret key (B)	1,408	2,112	2,752
Signature (B)	**1,474**	**2,349**	**2,948**
ML-DSA sig. (B)	2,420	3,309	4,627

ML-DSA and HAETAE: same structure, different costs

	ML-DSA	HAETAE
Ephemeral y	Uniform coefficients in a hypercube	Gaussian vector rescaled to a hyperball
Rejection	Coefficientwise ℓ_∞ bounds	Euclidean bounds on the full vector
Signature	Fixed-length response and hint encoding	rANS-encoded high bits and hint; low bits and challenge

Shared: module lattices, matrix–vector products, Fiat–Shamir with aborts, and NTT arithmetic.

[2] https://eprint.iacr.org/2022/323.pdf

ML-DSA and HAETAE: same structure, different costs

	ML-DSA	HAETAE
Ephemeral y	Uniform coefficients in a hypercube	Gaussian vector rescaled to a hyperball
Rejection	Coefficientwise ℓ_∞ bounds	Euclidean bounds on the full vector
Signature	Fixed-length response and hint encoding	rANS-encoded high bits and hint; low bits and challenge

Shared: module lattices, matrix–vector products, Fiat–Shamir with aborts, and NTT arithmetic.

Let's try the tecniques from Dilithium for Memory Constrained Devices (Bos, Rennes and Sprenkels)[2]

[2] https://eprint.iacr.org/2022/323.pdf

Some information

1. **Why HAETAE?** Compact keys and signatures, but a large reference stack.
   - HAETAE sig 1,474 vs ML-DSA sig 2,420

Some information

1. **Why HAETAE?** Compact keys and signatures, but a large reference stack.
   - HAETAE sig 1,474 vs ML-DSA sig 2,420
2. **Where does memory go?** Live polynomial vectors, sampling, and encoding.

Some information

1. **Why HAETAE?** Compact keys and signatures, but a large reference stack.
   - HAETAE sig 1,474 vs ML-DSA sig 2,420
2. **Where does memory go?** Live polynomial vectors, sampling, and encoding.
3. **How do we reduce it?** Stream, recompute, and separate lifetimes.

Key generation: where does the memory go?

$$b = a + As_1 + s_2 \pmod{q}$$

- The reference keeps the matrix A, secret vector s_1, and output b live together.
- The secret-key norm check also needs a 2,048 B FFT workspace.

	HAETAE-2	HAETAE-3	HAETAE-5
Dominant $A + s_1 + b$ objects	14 KiB	27 KiB	39 KiB
Measured reference stack	23,844 B	41,236 B	52,500 B

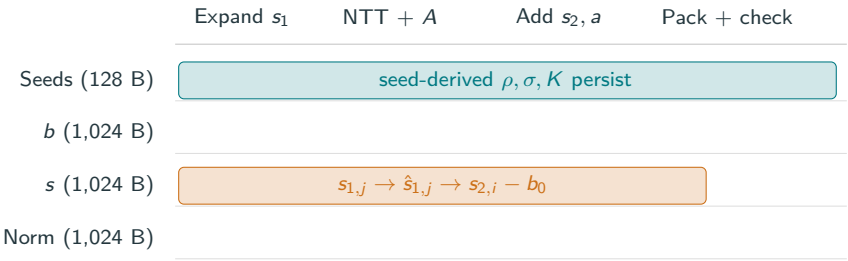
HAETAE-2/3 KeyGen: stream the matrix product

1. Expand one secret polynomial $s_{1,j}$; write it to the secret key.
2. Transform it in place; regenerate each needed $A_{i,j}$ from the seed.
3. Accumulate one output row b_i; add $s_{2,i}$ and a_i.
4. Update the singular-value norm accumulator; reject or pack the keys.

Two polynomial scratch slots plus a norm accumulator replace the full matrix and secret vectors.

HAETAE-2/3 key generation: reuse four small slots

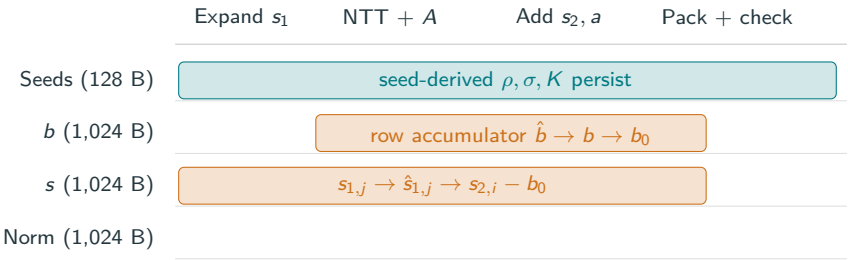
Live memory over the computation



The matrix is regenerated entry by entry; secrets are written to *sk* as soon as they are produced.

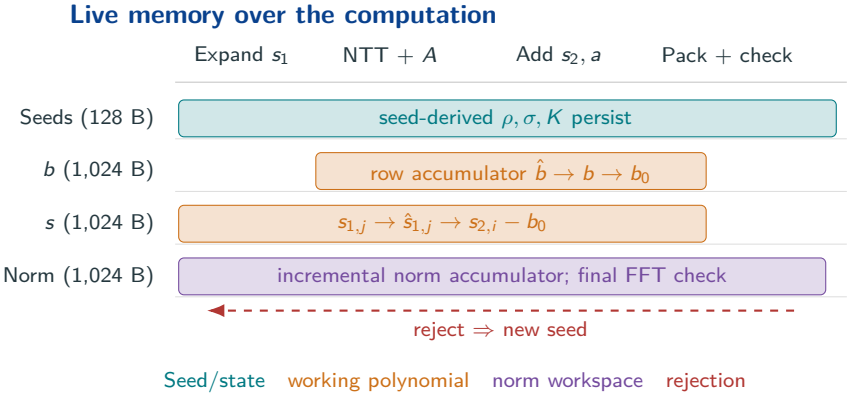
HAETAE-2/3 key generation: reuse four small slots

Live memory over the computation



The matrix is regenerated entry by entry; secrets are written to *sk* as soon as they are produced.

HAETAE-2/3 key generation: reuse four small slots



The matrix is regenerated entry by entry; secrets are written to *sk* as soon as they are produced.

Key generation differs at level 5

HAETAE-2/3: $d > 0$

- Public key stores rounded $\mathrm{HB}(b)$.
- Norm check is interleaved with the matrix product.
- Rejected candidates can pay for matrix expansion.

HAETAE-5: $d = 0$

- Check the norm first; expand A only after acceptance.
- Public key stores $\mathrm{NTT}(-2b)$.
- Two caller-level unions share FFT/sampling and norm/row storage.

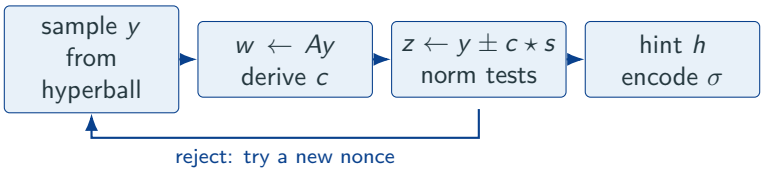
HAETAE-5 dominant caller temporaries: $2{,}048 + 1{,}024 = 3{,}072$ **B.**

KeyGen result: SRAM saved, cycles traded

	Stack (bytes)		Cycles (millions)	
	Ref. C	Ours C	Ref. C	Ours C
HAETAE-2	23,844	**5,816**	9.25	11.27
HAETAE-3	41,236	**5,816**	12.01	22.44
HAETAE-5	52,500	**4,776**	16.49	10.80

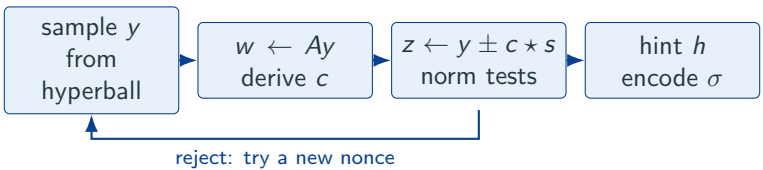
- Stack falls by **76%, 86%, and 91%** across levels 2, 3, and 5.
- Level 2/3 repeats streamed matrix work on rejected keys; level 5 checks the norm first, so matrix expansion happens only after acceptance.

The signing algorithm creates overlapping live data



- y depends on the norm of a full Gaussian vector.
- The reference keeps multiple y, w, z, and hint vectors live at once.
- A single polynomial is **1,024 B**; HAETAE-5 uses $\ell = 7$, $k = 4$.

The signing algorithm creates overlapping live data



- y depends on the norm of a full Gaussian vector.
- The reference keeps multiple y, w, z, and hint vectors live at once.
- A single polynomial is **1,024 B**; HAETAE-5 uses $\ell = 7$, $k = 4$.

The problem is lifetime overlap, not simply signature size.

A memory model before optimization

HAETAE-5 dominant objects

$$|\mathrm{poly}| = 256 \times 4 = 1{,}024 \text{ B}$$
$$|\mathrm{polyvecl}| = 7{,}168 \text{ B}$$
$$|\mathrm{polyveck}| = 4{,}096 \text{ B}$$

Minimum-state estimates

Key generation	4,096 B
Signing	8,960 B
Verification	6,144 B

The implementation strategy

Regenerate

Matrix entries, secrets, and ephemeral samples come from seeds.

Stream

Process one polynomial or row; absorb or encode it immediately.

Reuse

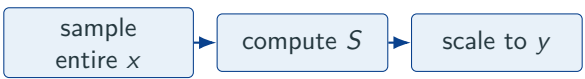
Overlay buffers whose live ranges cannot overlap.

More computation $\longrightarrow$ less peak SRAM

Matrix streaming and some buffer reuse follow the ML-DSA work of Bos–Renes–Sprenkels; HAETAE's sampler, hint, and rANS format need additional treatment.

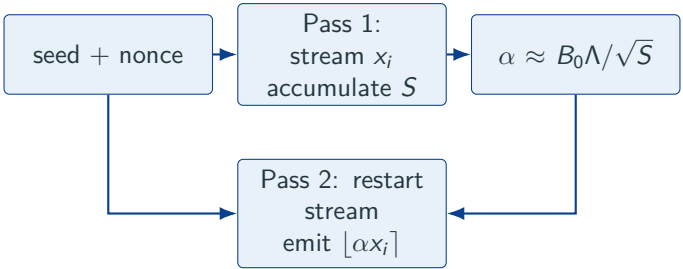
The hyperball sampler: why one pass is expensive

- Draw $(\ell + k)N$ Gaussian coefficients x_i.
- Compute $S = \sum_i x_i^2$ and a scaling factor $\alpha \approx B_0 \Lambda / \sqrt{S}$.
- Output $y_i = \lfloor \alpha x_i \rceil$, then test its norm.



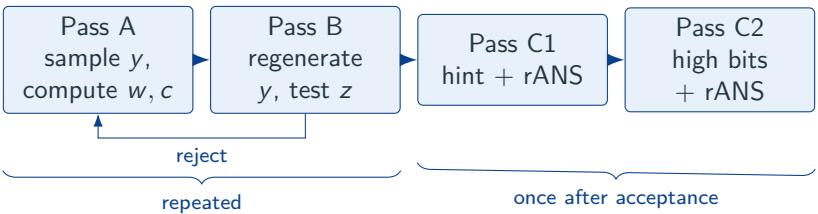
The scaling factor is known only after the last sample; buffering x costs $\Theta((\ell + k)N)$ words.

Two passes: same samples, constant extra space



- Restarting the same pseudorandom stream reproduces the same x_i.
- A streaming Gaussian backend removes the reference sampler's static arrays; scheme `.bss` $= 0$.

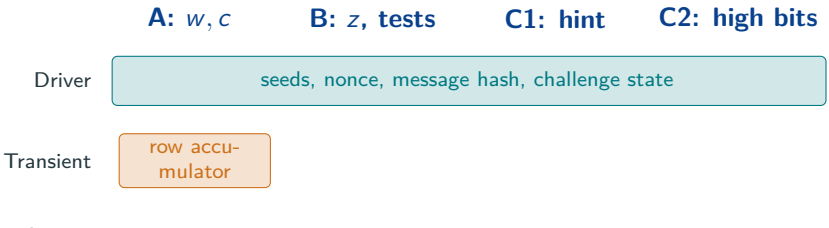
Signing: make the rejection loop small



$$S_{\text{sign}} = S_{\text{driver}} + \max(S_A, S_B, S_{C1}, S_{C2})$$

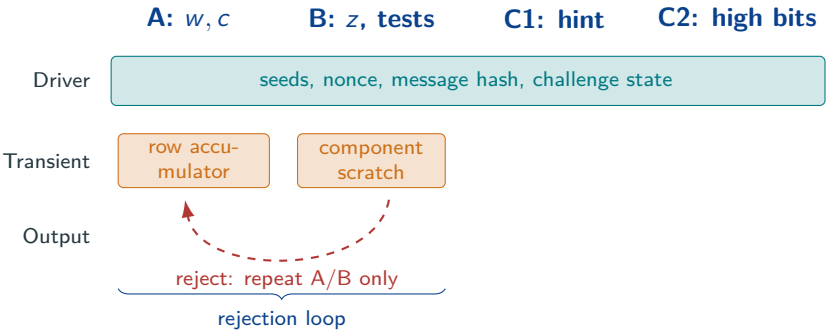
The passes live in a separate translation unit, preventing their large frames from being inlined into the driver.

Signing memory: one transient pass at a time



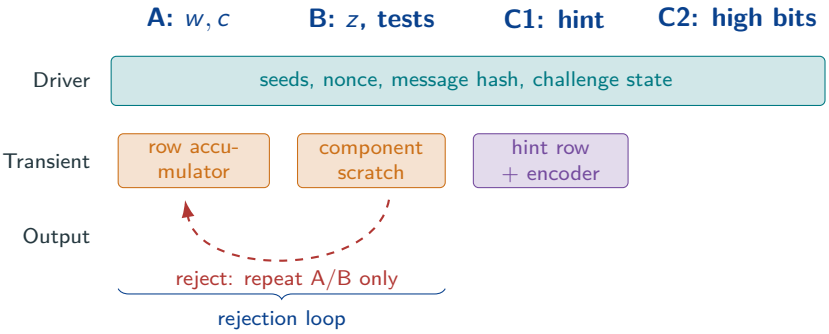
$S_{\text{sign}} = S_{\text{driver}} + \max(S_A, S_B, S_{C1}, S_{C2})$: transient buffers are reused, not added.

Signing memory: one transient pass at a time



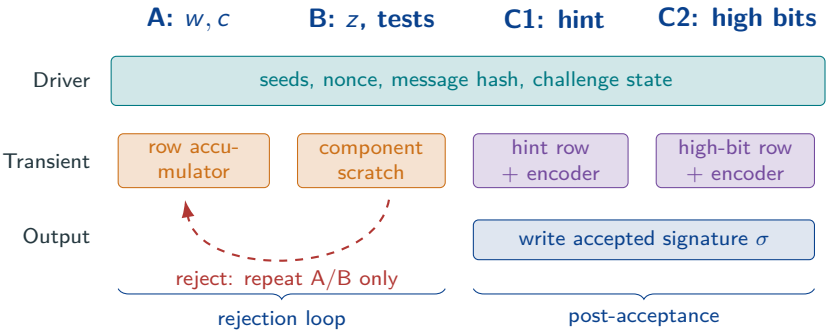
$S_{\text{sign}} = S_{\text{driver}} + \max(S_A, S_B, S_{C1}, S_{C2})$: transient buffers are reused, not added.

Signing memory: one transient pass at a time



$S_{\text{sign}} = S_{\text{driver}} + \max(S_A, S_B, S_{C1}, S_{C2})$: transient buffers are reused, not added.

Signing memory: one transient pass at a time



$S_{\text{sign}} = S_{\text{driver}} + \max(S_A, S_B, S_{C1}, S_{C2})$: transient buffers are reused, not added.

Pass A: build the commitment row by row

1. Derive the message digest and the seed for y.
2. Generate the hyperball sample; stream A and compute one w_i at a time.
3. Pack and absorb each row's high bits into SHAKE immediately.
4. Derive c from the transcript; reuse the row accumulator's storage for c.

A speed optimization within the same stack peak

Two matrix rows are batched, halving ephemeral-vector regenerations from $k\ell$ to $\lceil k/2 \rceil \ell$; the extra polynomial fits because Pass C sets the peak.

Pass B: reject as early as possible

1. Regenerate y from the accepted seed/nonce; compute $z_i = y_i \pm (c \star s)_i$ one component at a time.
2. First test $z_{1,0}$: it uses no secret-key material and can reject before NTT-based products.
3. Accumulate $\|z\|_2^2$ after each component; stop when the bound is exceeded.
4. Apply the remaining rejection predicate only after the needed components are available.

Most rejected attempts skip work; no full response vector is retained.

Pass C: compute the signature only after acceptance

C1: hint

Recompute w and z_2 row by row; derive each hint row and feed it directly to the rANS encoder.

C2: response high bits

Regenerate z_1 row by row; encode its high bits immediately and pack the low bits.

C1 and C2 do not need simultaneous hint and high-bit buffers.

The accepted seed, nonce, challenge, and scaling factors are enough to reproduce every value needed here.

Why reverse-order rANS enables streaming

Conventional

Materialize $X[0], \ldots, X[m-1]$, then encode

$$X[m-1], X[m-2], \ldots, X[0].$$

Reverse row traversal

For rows $i = R-1 \ldots 0$, compute coefficients $j = N-1 \ldots 0$ and encode $X[iN+j]$ immediately.

- Both paths present the **same symbol sequence** to a deterministic encoder.
- Full hint and high-bit staging arrays disappear; differential tests found byte-identical streams across all levels.

Two smaller signing details matter

Incremental transcript hashing

Pack one row of high bits and absorb it into SHAKE; reuse that row buffer. The full 608–1,184 B transcript buffer is unnecessary.

Fused challenge multiplication

For HAETAE-2/3, the challenge is sparse ($\tau = 58/80$): signed shifts accumulate directly into z. HAETAE-5 ($\tau = 128$) keeps NTT multiplication. Both avoid a separate product polynomial.

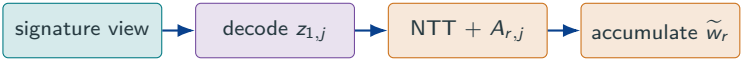
Working polynomials remain 32-bit: intermediates modulo $2q = 129{,}026$ exceed 16 bits.

Verification: one output row at a time

1. Parse the signature; keep low bits as a view into its buffer.
2. Pre-pass: decode high-bit columns, reconstruct z_1, test its norm, derive packed w'.
3. For each output row r, re-decode each $z_{1,\ell}$ column and accumulate $\widetilde{w}_r = \sum_\ell A_{r,\ell}\,\mathsf{NTT}(z_{1,\ell})$.
4. Decode one hint row; reconstruct the transcript row and $z_{2,r}$; absorb and test.
5. Recompute the challenge and compare with the signature.

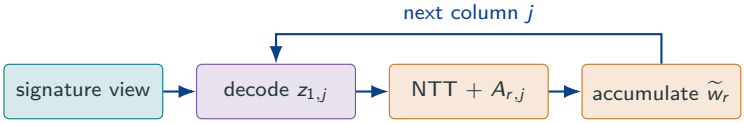
Re-decoding costs computation, but avoids a full k-polynomial accumulator.

Verification: re-decode columns, finish one row



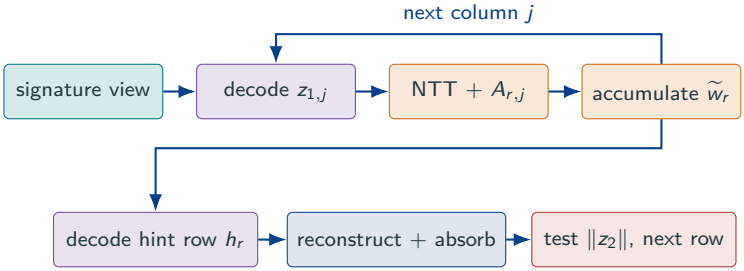
One output row replaces a full k-polynomial accumulator; the same 512 B decode slot alternates between columns and hint rows.

Verification: re-decode columns, finish one row



One output row replaces a full k-polynomial accumulator; the same 512 B decode slot alternates between columns and hint rows.

Verification: re-decode columns, finish one row



Working slots: 512 B decode union two 1,024 B polynomials 64 B persistent

One output row replaces a full k-polynomial accumulator; the same 512 B decode slot alternates between columns and hint rows.

Verification's compact working set

Persistent w' and challenge seed	64 B
High-bit column / hint-row union	512 B
Polynomial scratch slot 1	1,024 B
Polynomial scratch slot 2	1,024 B

- $\mathrm{LB}(z_1)$ stays in the signature buffer.
- A 512 B union alternates between decoded high bits and hints.
- Each row is hashed and discarded.

Measured verification stack: 4,824–4,848 B on Cortex-M4.

Peak stack on Cortex-M4, all three levels

	KeyGen	Sign	Verify
HAETAE-2	5,816 B	5,836 B	4,824 B
HAETAE-3	5,816 B	6,020 B	4,848 B
HAETAE-5	4,776 B	6,004 B	4,840 B

- For HAETAE-5, signing is **6,004 B** versus **8,092 B** in the prior low-stack implementation.
- The pure-C and assembly variants use the same high-level stack allocation.

Nucleo-L4R5ZI, −O3.

The cost of recomputation

Ours (C), million cycles	KeyGen	Sign	Verify
HAETAE-2	11.3	92.0	1.10
HAETAE-3	22.4	115.6	2.27
HAETAE-5	10.8	172.8	2.76

- Signing takes **4.62–5.70**× the cycles of the speed-optimized assembly reference.
- Against prior low-stack HAETAE-5, our C signing uses **0.57**× the cycles with a smaller stack.

Nucleo-L4R5ZI, -O3, medians of 100 runs

Physical 12 KiB Cortex-M0+: the full path fits

Total SRAM	KeyGen	Sign	Verify
HAETAE-2	8,512 B	9,064 B	7,560 B
HAETAE-3	9,696 B	10,832 B	8,928 B
HAETAE-5	9,912 B	**12,056 B**	10,128 B

- Sign total includes $|sk| + |\sigma|+$ signing stack.
- Tightest case: HAETAE-5 signing leaves **232 B** of 12,288 B SRAM.

STM32C031C6, Cycle/stack builds: `-Os`; Flash result: `-Os -flto`.

Thank you!



Scan for the paper

eprint.iacr.org/2026/635

Code

github.com/smallhaetae/haetae_lowstack