Software implementation of the NSA's ARADI block cipher on Arm Cortex–M4

José Eduardo Santos Rabelo[1] **, Julio López**[1]

[1]Institute of Computing – University of Campinas (Unicamp)
Campinas, SP – Brazil

j260551@dac.unicamp.br, jlopez@ic.unicamp.br

Abstract. *ARADI is a block cipher proposed by the NSA for inline memory encryption and paired with LLAMA, an authenticated-encryption mode for low-latency memory protection. This work evaluates the software viability of the NSA's ARADI block cipher on a 32-bit Arm Cortex–M4 microcontroller. We derive an optimized linear map and a shuffle-based two-block formulation that leverages packed halfword parallelism. Our optimized implementation achieves a* $2.36\times$ *performance improvement over the two-block NSA baseline. Finally, we provide comparative benchmarks against ChaCha–Poly1305 and AES-256.*

1. Introduction

Off-chip DRAM is exposed to cold-boot and remanence attacks [Halderman et al. 2008]; DMA can bypass IOMMU protection [Markettos et al. 2019]. We assume an attacker can observe or tamper with external DRAM contents or traffic, while the CPU core and secret-key storage remain trusted. Inline authenticated encryption reduces this risk, but every added cycle stalls the memory path. This has motivated low-latency designs such as PRINCE and SPEEDY [Borghoff et al. 2012, Leander et al. 2021]. ARADI/LLAMA follows this line with a block cipher and a memory-oriented AE mode for verify-then-decrypt reads [Greene et al. 2024]. Since its release, third-party work has reported reduced-round distinguishers and related-IV failures [Bellini et al. 2024, Avanzi et al. 2024]. Later results include 12-round key-recovery attacks [Mandal et al. 2025, Dunkelman and Ghosh 2025], a 14-round differential MITM attack [Ghosh et al. 2025], and full-round algebraic cryptanalysis at high cost [La Scala and Tiwari 2026].

Microcontroller deployments often rely on software-only stacks. On Armv7-M cores, rotations, subword permutations, and linear layers dominate more than in hardware pipelines; timing-disciplined code should also avoid lookup tables and secret-dependent memory accesses [Almeida et al. 2016]. Inspired by bitslicing and fixslicing on 32-bit cores [Adomnicai and Peyrin 2020], we ask whether ARADI/LLAMA can serve inline DRAM protection on MCU-class cores and which linear-layer changes narrow the gap to ChaCha/Poly1305. Our harness uses ARMed-SPHINCS ChaCha assembly [Hülsing et al. 2016a, Hülsing et al. 2016b] and portable Poly1305 following RFC 8439 [Nir and Langley 2018, Moon 2014].

Contributions. We derive two software-oriented implementations of ARADI's linear map: the single-block `lm_new` and the two-block shuffle map `lm_two_new`. Both preserve ARADI's round count, S-box, key schedule, and algebraic linear-layer behavior [Greene et al. 2024]. We benchmark them on Arm Cortex–M4 against ChaCha, ChaCha–Poly1305, and fixsliced AES-256.

2. Preliminaries

2.1. Specification of ARADI

ARADI is a 16-round substitution–permutation network over a 128-bit state arranged as four 32-bit words (w, x, y, z), under a 256-bit master key [Greene et al. 2024]. Greene, Motley, and Weeks designed it for extremely low-latency memory encryption, including inline CPU–DRAM deployment. Each round applies a key XOR τ^{k_i}, a bitsliced substitution layer π, and a round-dependent linear layer Λ_i; encryption repeats this sequence for $i = 0, \ldots, 15$ and finishes with the whitening key k_{16}.

Substitution Layer (π). ARADI uses a 4×4 bitsliced S-box realized as a cascade of four Toffoli gates [Greene et al. 2024]. For a fixed bit-slice, let x, y, z, w denote the four Boolean word registers holding the inputs on the S-box wires; each bit lane across these registers corresponds to one parallel block. The S-box is computed *in place* by the following ordered updates:

$$x \leftarrow x \oplus (w \wedge y), \qquad z \leftarrow z \oplus (x \wedge y), \qquad y \leftarrow y \oplus (w \wedge z), \qquad w \leftarrow w \oplus (x \wedge z). \quad (1)$$

Each assignment in (1) implements one Toffoli-style operation, with the right-hand side as controls and the left-hand side as target. Because the updates are evaluated in order, no temporaries are needed; at the end, the same registers x, y, z, w contain the S-box outputs for that bit-slice, for a cost of four ANDs and four XORs per round.

Linear Layer (Λ). At round i, the linear layer applies the same 32-bit involution L_j to each state word, with $j = i \bmod 4$:

$$\Lambda_i(x_0, x_1, x_2, x_3) = \big(L_j(x_0), L_j(x_1), L_j(x_2), L_j(x_3)\big).$$

For a word written as two 16-bit halves $u \,\|\, \ell$, this map is

$$L_j(u \,\|\, \ell) = \big(u \oplus S_{16}^{a_j}(u) \oplus S_{16}^{c_j}(\ell)\big) \;\|\; \big(\ell \oplus S_{16}^{a_j}(\ell) \oplus S_{16}^{b_j}(u)\big), \quad (2)$$

where $S_{16}^{b}(x)$ denotes the circular left rotation of the 16-bit word x by b bits. The parameter triples (a_j, b_j, c_j), for $j = 0, 1, 2, 3$, are $(11, 8, 14)$, $(10, 9, 11)$, $(9, 4, 14)$, and $(8, 9, 7)$. They satisfy $2a_j \equiv b_j + c_j \pmod{16}$, the identity exploited by our software optimization in Section 3.

The ARADI Key Schedule ARADI expands the 256-bit master key into seventeen 128-bit round keys. The schedule keeps eight 32-bit words, initialized directly from the master key. At each step, four adjacent word pairs are updated by one of two invertible linear mixers: M_0 uses rotations by 1 and 3, while M_1 uses rotations by 9 and 28. A small word permutation follows ($P_0 = (12)(56)$ or $P_1 = (14)(36)$), and a round counter is XORed into the last word. Subkey extraction alternates between the two halves of the register: even rounds use $K_0^i \,\|\, \cdots \,\|\, K_3^i$, odd rounds use $K_4^i \,\|\, \cdots \,\|\, K_7^i$, and one extra key is emitted for final whitening [Greene et al. 2024]. Our implementations precompute these keys; for `lm_two_new`, they are stored in the matching shuffled representation.

Implementation notes and two-block optimization. For 32-bit CPUs we use a bitsliced representation that processes two blocks in parallel. Using the *perfect shuffle*/unshuffle

permutation on 32-bit words, we interleave the upper/lower 16-bit halves so that a pair of 16-bit rotations can be effected by a single 32-bit rotation; this lets us rewrite (2) as an XOR of three rotated 32-bit quantities per word, significantly cutting the cost of Λ_i. This follows the same spirit as fixslicing/representation tricks used in AES-like ciphers [Adomnicai and Peyrin 2020].

2.2. LLAMA context

LLAMA pairs ARADI-CTR confidentiality with a PMAC-like LMAC authenticator for memory protection [Black and Rogaway 2002, Greene et al. 2024]. Fig. 1 shows the CTR and LMAC paths; both reuse the same optimized ARADI engine.

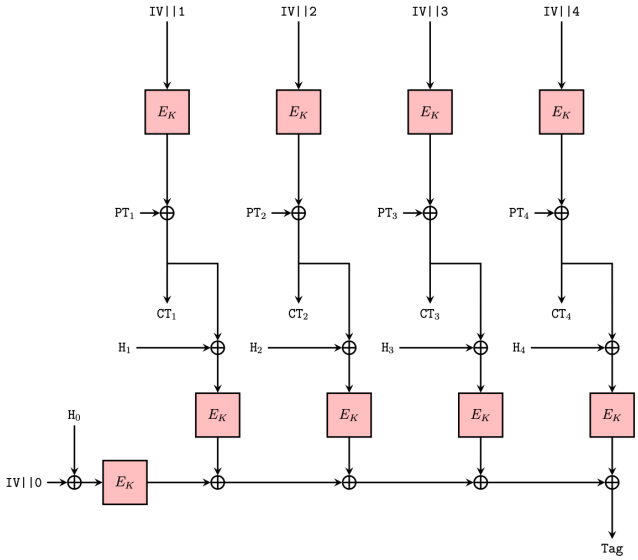


Figure 1. LLAMA authenticated encryption: ARADI provides CTR confidentiality and LMAC integrity, enabling parallel verify-then-decrypt reads [Greene et al. 2024].

The once-per-key-period H_i setup is measured separately.

3. Optimized Linear Maps

The linear layer dominates ARADI's Cortex–M4 cost: a 16-round block applies L_j 64 times, and the direct map repeatedly extracts, rotates, and merges 16-bit halves. We refactor this layer while preserving ARADI's public round function [Greene et al. 2024].

3.1. Algebraic single-block optimization (`lm_new`)

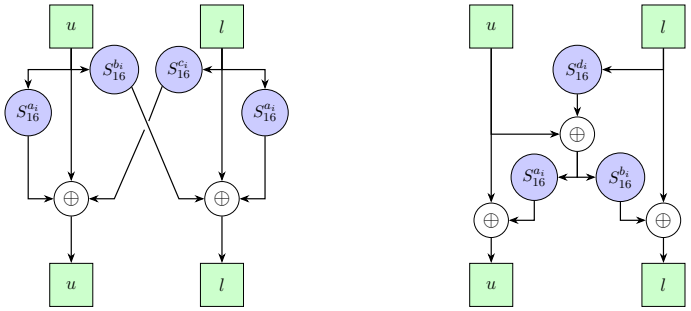
The baseline map, illustrated in Fig. 2(a), uses four 16-bit rotations per 32-bit word. The identities $d_j \equiv c_j - a_j \pmod{16}$ and $2a_j \equiv b_j + c_j \pmod{16}$ allow the computation to share the temporary

$$t \;=\; S_{16}^{d_j}(\ell) \;\oplus\; u.$$

The outputs then use only two subsequent rotations:

$$u' \;=\; u \;\oplus\; S_{16}^{a_j}(t), \qquad \ell' \;=\; \ell \;\oplus\; S_{16}^{b_j}(t).$$

This `lm_new` reformulation reduces the per-word rotation count from four to three. By the identities above and the original definition [Greene et al. 2024], it is algebraically equivalent to L_j and remains an involution.



(a) Original map. (b) `lm_new` map.

Figure 2. Comparison between the original ARADI linear map (left) and an optimized software-oriented variant (right). The new formulation, known as `lm_new`, consolidates the shift operations into a shared intermediate variable, significantly reducing the number of rotations.

3.2. Shuffle-based optimizations

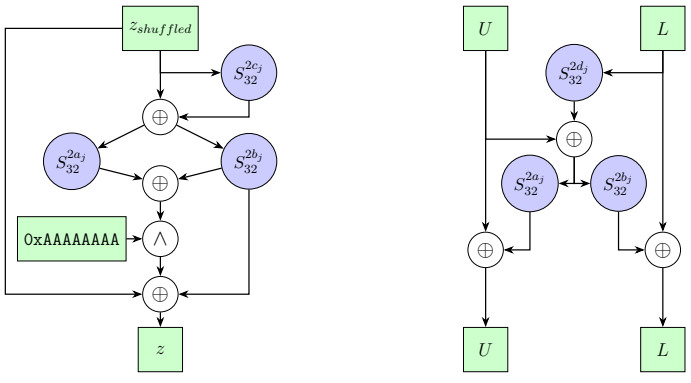
The perfect shuffle [Warren 2013] interleaves two 16-bit lanes inside one 32-bit register. For $z = u \parallel v$, with $u, v \in \{0,1\}^{16}$, it produces

$$\text{shuffle}(z) = u_{15}v_{15}u_{14}v_{14} \cdots u_0 v_0.$$

This bijection has a fixed unshuffle inverse and turns two equal-distance half-word rotations into one 32-bit barrel rotation. For two ARADI blocks, we pack corresponding halves as $U = \text{shuffle}(u_0 \parallel u_1)$ and $L = \text{shuffle}(\ell_0 \parallel \ell_1)$, then apply the shuffled `lm_new` equations, with $d_j \equiv c_j - a_j \pmod{16}$:

$$T = U \oplus S_{32}^{2d_j}(L), \quad U' = U \oplus S_{32}^{2a_j}(T), \quad L' = L \oplus S_{32}^{2b_j}(T).$$

The factor of two keeps the interleaved lanes aligned: a 32-bit rotation by $2r$ implements a 16-bit rotation by r in each lane. The resulting `lm_two_new` map is shown in Fig. 3(b); shuffle and unshuffle are paid only at the representation boundary.



(a) Single-block shuffle. (b) `lm_two_new`.

Figure 3. Shuffle-based linear-map layouts optimized for 32-bit software: a single-block shuffle layout (left) and the evaluated two-block map `lm_two_new` (right). Rotation labels show the actual doubled 32-bit distances; a 32-bit rotate by $2r$ implements two 16-bit rotates by r.

The inner layer then uses three rotations and three XORs per pair of words, amortizing the work across two ARADI blocks while preserving equivalence to L_j.

Table 1. Inner linear-map cost per state-word position. Boundary shuffle and unshuffle costs are excluded from the inner-loop counts.

Map	Blocks	16-bit rot.	32-bit rot.	Software effect
NSA baseline	1	4	0	half-word extract/merge path
`lm_new`	1	3	0	one fewer half-word rotation
NSA baseline	2	8	0	duplicated state, high register pressure
`lm_two_new`	2	0	3	shared word rotations after shuffle

4. Benchmarking details

All experiments run bare-metal on an STM32F407VG Arm Cortex–M4 at 168 MHz, without data cache. The firmware uses C11 GCC 14.2.1 with `-O3` and loop unrolling; cycle counts come from the Armv7-M DWT counter zeroed before each measurement window.

We evaluate the NSA single-block map, `lm_new`, the NSA map applied to two independent blocks, and `lm_two_new`. All keep ARADI's round count, S-box, and key schedule unchanged [Greene et al. 2024]; `lm_two_new` stores round keys in the shuffled domain. External baselines are fixsliced AES-256 [Adomnicai and Peyrin 2020] and ARMed-SPHINCS ChaCha12/20 assembly cores [Hülsing et al. 2016a, Hülsing et al. 2016b].

Each cipher runs $N = 1000$ in-place iterations on fixed pseudo-random vectors. We report total cycles and cpb; key precomputation is excluded from the microbenchmark window. The cacheless bare-metal target is deterministic; cycle counts were reproducible across repeated runs, with variation below 0.5%. The portable C benchmark harness is released as an open-source artifact.

5. Results

Table 2 reports cycles and cpb on Cortex–M4. ARADI uses portable C/GCC; ChaCha12/20 uses ARMed-SPHINCS assembly [Hülsing et al. 2016a, Hülsing et al. 2016b]; AES-256 uses a two-block fixsliced implementation [Adomnicai and Peyrin 2020]. The comparison reflects practical software stacks, not equal implementation effort.

Table 2. Cycle counts and cpb on Cortex–M4 for $N = 1000$ in-place encrypt iterations.

Cipher	Variant	Key (bits)	Block (bits)	Total cycles	cpb
ChaCha-12	ARMed-SPHINCS	256	512	843 451	13.18
ChaCha-20	ARMed-SPHINCS	256	512	1 175 878	18.37
ARADI 2-block	`lm_two_new` (ours)	256	2×128	1 263 900	39.49
ARADI 2-block	NSA baseline (2-block)	256	2×128	2 983 511	93.23
ARADI	`lm_new` (ours)	256	128	2 491 511	155.72
ARADI	NSA baseline (1-block)	256	128	2 795 511	174.72
AES-256	Fixsliced	256	2×128	3 733 623	116.67

The one-block and two-block rows should be read separately because they amortize work over different granularities. `lm_new` reduces the one-block NSA baseline from 174.72 to 155.72 cpb, about 11%. Against the two-block NSA baseline, `lm_two_new`

drops from 93.23 to 39.49 cpb, a $\approx 2.36\times$ speedup. ChaCha12 remains the fastest software baseline, but optimized ARADI narrows the gap and outperforms our fixsliced AES-256 baseline by nearly $3\times$. Thus `lm_new` is the single-block option, whereas `lm_two_new` applies when two adjacent 128-bit lines can be processed together.

6. LLAMA on real workloads (encrypt-then-MAC)

We compare one- and two-block LLAMA to software ChaCha12/20–Poly1305 on Cortex–M4 with $N = 1000$, $L = 16$ blocks, 96-bit IV/nonce, deterministic inputs, and DWT cycle counts. LLAMA includes CTR encryption and LMAC over ciphertext, but excludes once-per-session H precomputation. ChaCha–Poly1305 includes RFC 8439 one-time-key derivation, encryption, Poly1305, and verify/decrypt; ChaCha uses ARMed-SPHINCS assembly [Hülsing et al. 2016a, Hülsing et al. 2016b, Nir and Langley 2018], while Poly1305 uses portable poly1305-donna [Moon 2014].

Table 3. LLAMA vs. ChaCha–Poly1305 on Cortex–M4 for encrypt-and-tag plus verify-and-decrypt iterations ($N=1000$, $L=16$ blocks).

Cipher	Variant	Key (bits)	Block (bits)	Total cycles	cpb
ChaCha–Poly1305	12 rounds	256	512/stream	13 449 762	26.26
ChaCha–Poly1305	20 rounds	256	512/stream	16 306 761	31.84
LLAMA	two-block (ours)	256	2×128	45 893 007	89.63
LLAMA	one-block baseline	256	128	95 518 508	186.55

Two-block LLAMA costs about $2.82\times$ more cpb than ChaCha20–Poly1305 and $3.41\times$ more than ChaCha12–Poly1305, while cutting LLAMA's own one-block cost by roughly $2\times$. The comparison already favors LLAMA because H setup is outside the window; including it would widen the gap for non-amortized sessions. Within LLAMA, the two-block engine is the relevant configuration for verify-then-decrypt RAM-encryption flows with non-repeating nonces and ciphertext-authenticating tags [Greene et al. 2024].

7. Conclusion and limitations

This work presents an in-depth Cortex–M4 software evaluation of ARADI and shows that its linear layer can be made substantially more efficient without changing the cipher interface. `lm_new` reduces single-block cost, while `lm_two_new` amortizes 32-bit rotations across adjacent blocks and gives a $2.36\times$ speedup over the two-block NSA baseline. These changes preserve the round count, S-box, key schedule, and algebraic linear-layer behavior. ChaCha and ChaCha–Poly1305 remain strong software baselines, but optimized ARADI narrows the throughput gap and outperforms the fixsliced AES-256 baseline, establishing a reference point for unified hardware/software memory-encryption pipelines.

The released portable C artifact gives future work a concrete baseline for Thumb-2 tuning, Cortex-M33 and Cortex-A cores, SIMD targets, RV64 implementations, and side-channel evaluation. Energy and code-size footprint remain open for future work. Recent cryptanalysis reports reduced-round distinguishers, 12- and 14-round key recovery, related-IV issues in LLAMA, and full-round algebraic attacks [Bellini et al. 2024, Mandal et al. 2025, Dunkelman and Ghosh 2025, Ghosh et al. 2025, Avanzi et al. 2024, La Scala and Tiwari 2026].

Acknowledgements. The authors gratefully acknowledge the Cryptographic Research Centre at the Technology Innovation Institute for its partial financial support during the development of this work. They also thank the anonymous reviewers for their helpful suggestions and constructive feedback.

References

Adomnicai, A. and Peyrin, T. (2020). Fixslicing AES-like ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425.

Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016). Verifying constant-time implementations. In *25th USENIX Security Symposium*, pages 53–70.

Avanzi, R., Dunkelman, O., and Ghosh, S. (2024). A note on ARADI and LLAMA. Cryptology ePrint Archive, Paper 2024/1328.

Bellini, E., Formenti, M., Gérault, D., Grados, J., Hambitzer, A., Huang, Y. J., Huynh, P., Rachidi, M., Rohit, R., and Tiwari, S. K. (2024). CLAASPing ARADI: Automated analysis of the ARADI block cipher. In *Progress in Cryptology – INDOCRYPT 2024*, volume 15496 of *Lecture Notes in Computer Science*, pages 90–113. Springer, Cham.

Black, J. and Rogaway, P. (2002). A suggestion for handling arbitrary-length messages with a single-key MAC. In *Fast Software Encryption – FSE 2002*, volume 2365 of *LNCS*, pages 81–105. Springer.

Borghoff, J. et al. (2012). PRINCE – a low-latency block cipher for pervasive computing applications. In *ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer.

Dunkelman, O. and Ghosh, S. (2025). Improved Key-Recovery Attacks on ARADI up to 12 Rounds Using ZeroSum and the Fast Hadamard Transform. Cryptology ePrint Archive, Paper 2025/1227.

Ghosh, S., Michel, B., and Naya-Plasencia, M. (2025). Differential-MITM attack on 14-round ARADI. Cryptology ePrint Archive, Paper 2025/1918.

Greene, P., Motley, M., and Weeks, B. (2024). ARADI and LLAMA: Low-Latency Cryptography for Memory Encryption. Cryptology ePrint Archive, Paper 2024/1240.

Halderman, J. A. et al. (2008). Lest we remember: Cold boot attacks on encryption keys. In *17th USENIX Security Symposium*, pages 45–60.

Hülsing, A., Rijneveld, J., and Schwabe, P. (2016a). ARMed SPHINCS – Computing a 41KB Signature in 16KB of RAM. In Persiano, G. and Yang, B.-Y., editors, *Public Key Cryptography – PKC 2016*, volume 9614 of *Lecture Notes in Computer Science*, pages 446–470. Springer-Verlag Berlin Heidelberg.

Hülsing, A., Rijneveld, J., and Schwabe, P. (2016b). ARMed SPHINCS code package. `https://joostrijneveld.nl/papers/armedsphincs/`. ChaCha12/20 Arm code package.

La Scala, R. and Tiwari, S. K. (2026). Oracle-based multistep strategy for solving polynomial systems over finite fields and algebraic cryptanalysis of the ARADI cipher. *Advances in Mathematics of Communications*, 23:255–273.

Leander, G., Moos, T., Moradi, A., and Rasoolzadeh, S. (2021). The SPEEDY family of block ciphers: Engineering an ultra low-latency cipher from gate level for secure processor architectures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):510–545.

Mandal, S., Mondal, S. K., Rohit, R., and Sarkar, S. (2025). Improved differential cryptanalysis of ARADI. Cryptology ePrint Archive, Paper 2025/1976.

Markettos, A. T. et al. (2019). Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals. In *NDSS Symposium*.

Moon, A. (2014). poly1305-donna. `https://github.com/floodyberry/poly1305-donna`.

Nir, Y. and Langley, A. (2018). ChaCha20 and Poly1305 for IETF Protocols. RFC 8439.

Warren, Jr., H. S. (2013). *Hacker's Delight*. Addison-Wesley, 2nd edition.