SARX: A 256-bit ARX Stream Cipher and a Password-Based AEAD Construction

Abstract

We present SARX, a 256-bit ARX stream cipher and password-based AEAD designed for high-throughput software vaults. The core cipher is a four-word (4×64-bit) add–rotate–xor map derived from an empirical search over candidate constructions, tuned for branch-free constant-time implementation and near-isometric diffusion: one-bit differences avalanche across the state in 2–3 rounds while avoiding structural short cycles. We compose SARX with Argon2id key derivation and BLAKE3 authentication in an Encrypt-then-MAC construction, and give a standard black-box reduction to IND-CCA and INT-CTXT in the derived-key model. A constant-time Rust implementation reaches up to 8 GB/s keystream and 2–3 GB/s end-to-end encryption on commodity CPUs, and passes NIST SP 800-22, Dieharder, and PractRand/BigCrush statistical test batteries. We discuss design trade-offs, resistance to generic attacks on ARX ciphers, and deployment for client-side vaults and backups.

1 Introduction

Stream ciphers and ARX (add–rotate–xor) constructions have become popular primitives for high-speed software encryption. Designs such as Salsa20 and ChaCha20 show that small ARX cores with simple counter-based keystream generation can achieve strong security and excellent performance on general-purpose CPUs. At the same time, password-based authenticated encryption remains a practical challenge: tools often combine a general-purpose AEAD scheme with an ad hoc KDF, without a dedicated design for the password-based setting or a clear treatment of offline dictionary attacks.

This paper proposes two related contributions:

- **SARX:** a 256-bit ARX stream cipher with a small, four-word state and an 8-round core, designed for high-speed keystream generation and a simple specification. The round structure and rotation constants are selected via an automated search over simple 64-bit ARX recurrences, using avalanche and distance distortion in the Hamming metric as evaluation criteria.
- **SARX-Vault:** a password-based authenticated-encryption (AEAD) construction using SARX, Argon2id, and keyed BLAKE3 in a conservative Encrypt-then-MAC composition, with an optional cost-amplifying hardening layer.

The design goals are:

- *Simplicity and auditability:* a small ARX core over four 64-bit words, a straightforward counter-mode keystream, and standard KDF and MAC primitives.

- *Standard security:* IND-CCA and ciphertext integrity (INT-CTXT) for the password-based scheme, under explicit PRF and MAC assumptions.
- *Performance:* software-friendly 64-bit ARX operations, parallel keystream generation, and good throughput on commodity CPUs.
- *Password resistance:* integration with Argon2id for memory-hard key derivation, plus an optional, deterministic hardening layer that raises the per-guess cost of offline brute-force attacks.

Our contributions

We summarise our contributions as follows.

- We specify the SARX stream cipher: a 256-bit key, 64-bit counter ARX construction with 32-byte blocks and an 8-round core. The specification is self-contained and directly matches the reference implementation. We also describe the automated ARX parameter search used to select the rotation constants and round structure, based on measured avalanche and near-isometry in the bit metric.
- We define SARX-Vault, a password-based AEAD scheme that uses: Argon2id to derive key material from a password and salt, BLAKE3 to derive a 256-bit keystream key for SARX, and keyed BLAKE3 as a MAC over the vault header and ciphertext. The vault format is compact and suitable for file encryption.
- In a derived-key model, we give a black-box reduction showing that, if SARX is a PRF and keyed BLAKE3 is a SUF-CMA MAC, SARX-Vault achieves IND-CCA and INT-CTXT security. The reduction is standard and follows the Encrypt-then-MAC paradigm.
- We introduce an optional thermodynamic hardening mode: a public, deterministic post-processing function that applies an ARX-based random walk over a large scratch buffer to the Argon2id output. We prove that this mode does not weaken the security reduction and quantify how it increases the per-guess cost for offline password attacks.
- We present an experimental evaluation and preliminary cryptanalysis of SARX: keystreams pass NIST SP 800-22, Dieharder, PractRand, and TestU01 BigCrush with no reported failures or anomalies; we report performance figures; and we describe reduced-round behaviour and the ARX parameter-search metrics that guided the choice of round function.

The rest of the paper is organised as follows. Section 2 recalls notation and basic primitives. Section 3 defines SARX. Section 4 describes the SARX-Vault scheme. Section 5 gives the security model and reduction. Section 6 formalises thermodynamic hardening. Section 7 presents the experimental evaluation. Section 8 briefly compares SARX and SARX-Vault to related work, and Section 9 concludes.

2 Preliminaries and Notation

We use standard notation. Let $\{0,1\}^\ell$ denote the set of bit strings of length ℓ, and $\{0,1\}^*$ the set of finite bit strings. For a bit string X, we write $\mathrm{len}(X)$ for its length in bits and $|X|$ when the context is clear. Concatenation of byte or bit strings is written as $\|$. We write $U \leftarrow \{0,1\}^\ell$ for sampling U uniformly from $\{0,1\}^\ell$.

We denote by $\mathbb{Z}_{2^{64}}$ the ring of 64-bit words with addition modulo 2^{64}. Bitwise exclusive-or is written $\oplus$. Rotation of a 64-bit word x to the left by r bits is written $\mathrm{ROTL}_{64}(x, r)$. For a non-negative integer x, $\mathrm{LE64}(x)$ (resp. $\mathrm{BE64}(x)$) denotes the 8-byte little-endian (resp. big-endian) encoding of x.

We assume familiarity with symmetric-key primitives such as pseudorandom functions (PRFs), message authentication codes (MACs), and password-based key-derivation functions (KDFs). We briefly recall these in the context of our scheme and introduce the advantage notation used in the security reduction.

2.1 PRF and MAC assumptions

Let $F : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^m$ be a function family with key length κ and input length n. For a key $K \in \{0,1\}^\kappa$ we define $F_K(x) := F(K, x)$. A PRF adversary B is given oracle access to a function $O : \{0,1\}^n \to \{0,1\}^m$ and must distinguish whether $O = F_K$ for uniform K or $O = R$, where R is a truly random function. The PRF advantage is

$$\mathrm{Adv}_F^{\mathrm{PRF}}(B) := \left|\Pr[\, B^{F_K} = 1\,] - \Pr[\, B^R = 1\,]\right|.$$

Similarly, let $\mathsf{MAC}_K(M)$ be a MAC with key K and message M. A strong unforgeability (SUF-CMA) adversary B may query a tagging oracle $M \mapsto \mathsf{MAC}_K(M)$ adaptively, and must output a pair $(M^\star, T^\star)$ such that $T^\star$ is a valid tag for $M^\star$ and $(M^\star, T^\star)$ was not returned by the oracle on a previous query. The SUF-CMA advantage is

$$\mathrm{Adv}_{\mathsf{MAC}}^{\mathrm{MAC}}(B) := \Pr[B \text{ outputs a valid forgery}].$$

2.2 Password-based key derivation

SARX-Vault is designed for password-based encryption. We use Argon2id with fixed parameters and a 256-bit salt to derive an "outer" key string okm of length L bytes from a password and salt, and then derive internal keys from okm via simple splitting and public post-processing. In the security reduction we adopt a *derived-key model*: we treat the keystream key and MAC key as independent, uniformly random keys, and reduce the symmetric security of the scheme to the PRF and MAC assumptions above. Offline dictionary attacks are then governed by the password entropy and the cost of evaluating Argon2id per guess, which we discuss separately.

3 The SARX Stream Cipher

We now define SARX, the stream cipher used by SARX-Vault. SARX is a 256-bit-key, 64-bit-counter ARX construction that produces 32-byte keystream blocks. Its state consists of four 64-bit words.

3.1 Key and counter injection

Let $K \in \{0,1\}^{256}$ be a 256-bit key, viewed as four 64-bit words (k_0, k_1, k_2, k_3) in little-endian order. Let $\mathrm{ctr} \in \{0,1\}^{64}$ be a 64-bit counter. We define the initial state $(x_0, x_1, x_2, x_3) \in (\mathbb{Z}_{2^{64}})^4$ as:

$$\begin{aligned} x_0 &= k_0 \oplus (\mathrm{ctr} \cdot c), \\ x_1 &= k_1, \\ x_2 &= k_2, \\ x_3 &= k_3 \oplus \mathrm{ctr}, \end{aligned}$$

where c is a fixed 64-bit constant, e.g. $c =$ `0x9E3779B97F4A7C15`.

We also store the original state $(o_0, o_1, o_2, o_3) := (x_0, x_1, x_2, x_3)$ for a final feedforward step.

3.2 The ARX round function

Let $R = 8$ be the number of rounds. Each round consists of a "column" step and a "diagonal" step, operating on four 64-bit words. One round transforms (x_0, x_1, x_2, x_3) as follows:

Column step.

$$\begin{aligned} x_0 &:= x_0 + x_1, \\ x_3 &:= \mathrm{ROTL}_{64}(x_3 \oplus x_0, 27), \\ x_2 &:= x_2 + x_3, \\ x_1 &:= \mathrm{ROTL}_{64}(x_1 \oplus x_2, 31). \end{aligned}$$

Diagonal step.

$$\begin{aligned} x_0 &:= x_0 + x_2, \\ x_3 &:= \mathrm{ROTL}_{64}(x_3 \oplus x_0, 17), \\ x_1 &:= x_1 + x_3, \\ x_2 &:= \mathrm{ROTL}_{64}(x_2 \oplus x_1, 23). \end{aligned}$$

All additions are modulo 2^{64}; rotations and XORs are bitwise.

We apply this round function R times in sequence, updating (x_0, x_1, x_2, x_3).

3.3 Feedforward and block output

After R rounds we obtain $(y_0, y_1, y_2, y_3) := (x_0, x_1, x_2, x_3)$. We then compute a feedforward step:

$$(z_0, z_1, z_2, z_3) := (y_0 + o_0,\ y_1 + o_1,\ y_2 + o_2,\ y_3 + o_3) \pmod{2^{64}}.$$

The resulting 256-bit block is

$$\mathrm{Block}_K(\mathrm{ctr}) := \mathrm{LE64}(z_0) \,\|\, \mathrm{LE64}(z_1) \,\|\, \mathrm{LE64}(z_2) \,\|\, \mathrm{LE64}(z_3).$$

3.4 Stream generation with offset

SARX is used as a byte-addressable stream generator. We treat the infinite keystream as the concatenation of blocks:

$$S_K = \mathrm{Block}_K(0) \,\|\, \mathrm{Block}_K(1) \,\|\, \mathrm{Block}_K(2) \,\|\, \ldots$$

Each block has $B = 32$ bytes.

To generate a slice of keystream of length L bytes starting at byte offset $o \geq 0$:

1. Compute the starting block index $\mathrm{start} = \lfloor o/B \rfloor$ and the intra-block skip $\mathrm{skip} = o \bmod B$.
2. Initialise $\mathrm{ctr} := \mathrm{start}$ and an empty output buffer K.
3. While $\mathrm{len}(K) < L$:
    (a) Compute $\mathrm{Block}_K(\mathrm{ctr})$.
    (b) Append bytes from $\mathrm{Block}_K(\mathrm{ctr})[\mathrm{skip}..\,]$ to K.
    (c) Set $\mathrm{skip} := 0$ and increment ctr.
4. Truncate K to its first L bytes.

We denote the resulting function by $\mathsf{Stream}_K(o, L)$.

3.5 Design rationale for SARX

The SARX core is deliberately small and structured, with design choices guided by both implementation considerations and simple metrics on 64-bit ARX maps.

State size and layout. We use a four-word state (x_0, x_1, x_2, x_3) over $\mathbb{Z}_{2^{64}}$, for a total of 256 state bits. This aligns with 64-bit architectures and keeps the state compact compared to 16-word 32-bit designs such as Salsa20/ChaCha, while still allowing both "column" and "diagonal" mixing steps to move information between all words within a small number of rounds.

Counter injection. The 64-bit block counter ctr is injected into two words via

$$x_0 \leftarrow k_0 \oplus (\mathrm{ctr} \cdot c), \qquad x_3 \leftarrow k_3 \oplus \mathrm{ctr},$$

where c is a fixed odd constant (here c = `0x9E3779B97F4A7C15`). Mixing the counter into two positions via a multiplication and an XOR breaks trivial rotational and additive symmetries between counters and avoids degenerate cases where shifting the counter produces the same pre-round state, while keeping the rule simple enough to analyse.

Round structure and rotation constants. Each round consists of a "column" and a "diagonal" step, in the style of Salsa/ChaCha, but applied to a 4×64-bit state rather than a 4×4 matrix of 32-bit words. We considered a class of single-word ARX recurrences of the form

$$x \mapsto \mathrm{ROTL}_{64}\big((x + (x \ll a)) \oplus (x \gg b),\, c\big),$$

and, for a range of (a, b, c), measured:

- the average avalanche, defined as the expected number of flipped output bits when flipping one input bit, averaged over all bit positions and random inputs; and
- the distortion of Hamming distances, via the mean and standard deviation of $\rho(x, y) = d(f(x), f(y))/d(x, y)$ for random pairs (x, y), where d is the Hamming distance on $\{0, 1\}^{64}$.

We then embedded parameter triples with near-ideal avalanche (≈ 32 flipped bits out of 64) and mean ρ close to 1 with small variance into the four-word SARX round function, using the rotation pattern $(27, 31, 17, 23)$ and an 8-round core. This choice was selected as a simple instance that scored well under these metrics and did not exhibit obvious structural biases in preliminary experiments.

Feedforward and block size. A final feedforward step $(y_0, \ldots, y_3) \mapsto (y_0+o_0, \ldots, y_3+o_3)$, where $(o_0, \ldots, o_3)$ is the initial state, increases non-linearity and follows the design of Salsa/ChaCha. The 32-byte block output $\mathrm{Block}_K(\mathrm{ctr})$ is a natural match to the 4×64-bit state and to the keystream block size used in many existing ARX-based stream ciphers.

4 The SARX-Vault Scheme

We now describe SARX-Vault: a password-based AEAD construction that uses SARX for encryption and keyed BLAKE3 for authentication. A vault file consists of a fixed-size header, a MAC tag, and ciphertext.

4.1 Vault header format

A SARX-Vault header H is a fixed-length byte string encoding the following fields:

- A magic string, e.g. `"SARX"` (4 bytes).
- A version byte (1 byte).
- A 256-bit salt $\texttt{salt}_{32}$ for Argon2id (32 bytes).
- A 64-bit timestamp ts_ns in nanoseconds since the Unix epoch, encoded as BE64(ts_ns) (8 bytes).
- A 96-bit random nonce $\texttt{nonce}_{12}$ (12 bytes).
- Argon2id parameters:
  - *t_cost* (1 byte): time cost, in a fixed small range (e.g. $[1, 10]$).

– m_cost (1 byte): memory cost exponent, so memory is 2^{m_cost} KiB (e.g. $m\_cost = 17$ for 128 MiB).
– `lanes` (1 byte): number of parallel lanes (e.g. in $[1, 4]$).
– `kdf_id` (1 byte): KDF identifier (2 for Argon2id, 3 for Argon2id plus thermo hardening).

The header is treated as associated data (AD) for the MAC.

4.2 Key derivation

Given a password `pw` and header with salt $\texttt{salt}_{32}$ and Argon2id parameters ($t\_cost, m\_cost,$ `lanes`, `kdf_id`), SARX-Vault derives keys in two steps.

Argon2id layer. Let $b = \text{len}(\texttt{pw})$ (in bytes). Define

$$k_{\text{stream\_len}} = \max(32,\ \text{next multiple of 32 of } b), \quad L = k_{\text{stream\_len}} + 32.$$

We compute

$$\texttt{okm} := \text{Argon2id}(\texttt{pw}, \texttt{salt}_{32}, t = t\_cost, m = 2^{m\_cost}\text{KiB}, \ell = L, p = \texttt{lanes}),$$

and split:

$$k_{\text{stream}} := \texttt{okm}[0..k_{\text{stream\_len}}], \qquad k_{\text{mac}} := \texttt{okm}[k_{\text{stream\_len}}..L],$$

where k_{mac} is 32 bytes and used as a BLAKE3 key.

If $\texttt{kdf\_id} = 3$, an optional thermo-hardening step is applied to `okm` before the split; this is described in Section 6.

BLAKE3 keystream key. Independently of k_{stream} and k_{mac}, we derive the 256-bit keystream key for SARX as

$$\mathsf{base\_key32} := \text{BLAKE3\_XOF}(\texttt{pw} \,\|\, \text{BE64}(\text{ts\_ns}))[0..32].$$

We interpret base_key32 as (k_0, k_1, k_2, k_3) and use it as the SARX key.

In the security reduction we model this as producing an independent, uniform 256-bit key for SARX.

4.3 MAC computation

Let H be the header, C the ciphertext, and $\ell_C = \text{len}(C)$ the ciphertext length in bytes. We define $\texttt{len\_le} := \text{LE64}(\ell_C)$ and a fixed domain separator `dom_mac` := `"SARX-MAC-v1"`.

Given $k_{\text{mac}} \in \{0, 1\}^{256}$, the 32-byte MAC tag is

$$T := \text{BLAKE3\_KEYED}(k_{\text{mac}}, \texttt{dom\_mac} \,\|\, H \,\|\, \texttt{len\_le} \,\|\, C)[0..32].$$

4.4 Encryption

Given a password $\texttt{pw}$ and plaintext $P \in \{0,1\}^*$, the encryption algorithm proceeds as follows:

1. Sample ts_ns and $\texttt{nonce}_{12}$, and construct the header H with chosen Argon2id parameters.
2. Compute $\texttt{salt}_{32}$, e.g. $\texttt{salt}_{32} := \text{BLAKE3\_XOF}(\text{BE64}(\text{ts\_ns}) \,\|\, \texttt{nonce}_{12})[0..32]$, and populate the header.
3. Derive $\texttt{okm}$ via Argon2id, apply thermo hardening if $\texttt{kdf\_id} = 3$, and split to obtain $(k_{\text{stream}}, k_{\text{mac}})$.
4. Derive base_key32 from $(\texttt{pw}, \text{ts\_ns})$, interpret it as $(k_0, \ldots, k_3)$, and define the SARX key.
5. Generate keystream $K := \mathsf{Stream}_{\mathsf{base\_key32}}(0, \text{len}(P))$.
6. Compute ciphertext C as $C[i] = P[i] \oplus K[i]$ for all i.
7. Compute tag T via BLAKE3 keyed with k_{mac}.
8. Output the vault file $H \,\|\, T \,\|\, C$.

4.5 Decryption

Given a password $\texttt{pw}$ and a vault file $H \,\|\, T \,\|\, C$, the decryption algorithm:

1. Parses H and checks the magic, version, and header field ranges.
2. Recomputes $(k_{\text{stream}}, k_{\text{mac}})$ from $(\texttt{pw}, \texttt{salt}_{32})$ using the Argon2id parameters in H, applying thermo hardening if $\texttt{kdf\_id} = 3$.
3. Verifies the MAC by recomputing

$$T' := \text{BLAKE3\_KEYED}(k_{\text{mac}}, \texttt{dom\_mac} \,\|\, H \,\|\, \texttt{len\_le} \,\|\, C)[0..32],$$

   and comparing T' to T in constant time. If $T' \neq T$, the algorithm rejects with $\perp$.
4. If the MAC verifies, recomputes base_key32 from $(\texttt{pw}, \text{ts\_ns})$ in H and generates keystream K of length $\text{len}(C)$.
5. Outputs plaintext P as $P[i] = C[i] \oplus K[i]$.

5 Security Model and Reduction

We now analyse SARX-Vault in a derived-key model, where the SARX keystream key and the MAC key are assumed to be independent, uniform keys unknown to the adversary. We focus on the symmetric security guarantees; password entropy and KDF cost are treated separately.

Threat model for SARX-Vault

In the intended deployment, an adversary obtains one or more SARX-Vault files $H \parallel T \parallel C$ and knows all public header fields, including salts, timestamps, nonces, and Argon2id parameters. The adversary does not obtain the password or the derived keys and, in practice, does not have access to an online decryption oracle; our IND-CCA model is conservative in granting such an oracle. The primary practical threat is offline password guessing, in which the adversary iterates over candidate passwords, recomputes the derived keys from the header, and checks whether the tag verifies. Side-channel attacks and active attacks on an online decryption interface are outside the scope of this work.

5.1 Assumptions

Let $F : \{0,1\}^{256} \times \{0,1\}^{64} \to \{0,1\}^{256}$ denote the SARX block function, and for each K define $F_K(\mathrm{ctr}) := \mathrm{Block}_K(\mathrm{ctr})$. Let Stream_K be the byte-addressable stream generator defined in Section 3, which concatenates $F_K(0), F_K(1), \ldots$.

We make the following assumptions:

PRF assumption on SARX. F is a secure PRF: for any PPT adversary B_1, $\mathrm{Adv}_F^{\mathrm{PRF}}(B_1)$ is negligible in the key length.

MAC assumption on keyed BLAKE3. The keyed BLAKE3 instance used for tags in SARX-Vault is a secure SUF-CMA MAC: for any PPT adversary B_2, $\mathrm{Adv}_{\mathsf{MAC}}^{\mathrm{MAC}}(B_2)$ is negligible.

Derived-key model. In the analysis we treat the 256-bit keystream key $\mathsf{base\_key32}$ and the 256-bit MAC key k_{mac} as independent, uniformly random keys, and assume the adversary does not obtain the password or these keys. This abstracts away the KDF and reduces security to offline password guessing.

5.2 IND-CCA and INT-CTXT notions

We model SARX-Vault as an authenticated-encryption scheme with AD: $(\mathsf{Setup}, \mathsf{Enc}, \mathsf{Dec})$, where the header acts as associated data. In the derived-key model, Setup samples independent random keys (K_s, K_m) for SARX and the MAC. The encryption algorithm Enc corresponds to the real SARX-Vault encryption given (K_s, K_m), and Dec to decryption and verification.

We consider the usual IND-CCA game: an adversary has access to encryption and decryption oracles, submits a pair of equal-length plaintexts (P_0, P_1), receives a challenge ciphertext of P_b for uniform b, and must guess b, with the restriction that it may not query the exact challenge ciphertext to the decryption oracle.

Ciphertext integrity (INT-CTXT) requires that no PPT adversary with access to $(\mathsf{Enc}, \mathsf{Dec})$ can produce a new ciphertext that decrypts to a non-$\perp$ value with more than negligible probability.

5.3 IND-CCA reduction

We now state and prove the main reduction. For brevity we write $\mathrm{Adv}_{\text{SARX-Vault}}^{\text{IND-CCA}}(A)$ for the IND-CCA advantage of adversary A against SARX-Vault in the derived-key model.

Theorem 1. *Let A be a PPT adversary against SARX-Vault in the IND-CCA sense above. Then there exist PPT adversaries B_1 and B_2 such that*

$$\mathrm{Adv}^{\text{IND-CCA}}_{\text{SARX-Vault}}(A) \;\leq\; \mathrm{Adv}^{\text{PRF}}_{F}(B_1) \;+\; \mathrm{Adv}^{\text{MAC}}_{\mathsf{MAC}}(B_2).$$

In particular, if SARX is a secure PRF and keyed BLAKE3 is a secure MAC, then SARX-Vault is IND-CCA secure in the derived-key model.

Proof. We sketch the standard hybrid argument.

Hybrid $\mathbf{H}_0$. H_0 is the real IND-CCA experiment for SARX-Vault: keystreams are generated by Stream_{K_s}, and tags by MAC_{K_m} over $(H, \texttt{len\_le}, C)$.

Hybrid $\mathbf{H}_1$: replace SARX with a random function. In H_1 we replace Stream_{K_s} with a stream derived from a truly random function $R : \{0,1\}^{64} \to \{0,1\}^{256}$. The challenger samples R, and for each stream block index i uses $R(i)$ instead of $F_{K_s}(i)$, slicing as usual by offset. The MAC layer is unchanged.

If an adversary distinguishes H_0 from H_1 with advantage ϵ, we can build a PRF adversary B_1 for F by giving B_1 oracle access to O that is either F_{K_s} or a random function, and having B_1 simulate H_0 or H_1 for A by using O in place of F_{K_s} when generating keystream blocks. Thus

$$|\Pr[\,A \text{ succeeds in } \mathrm{H}_0\,] - \Pr[\,A \text{ succeeds in } \mathrm{H}_1\,]| \leq \mathrm{Adv}^{\text{PRF}}_{F}(B_1).$$

Hybrid $\mathbf{H}_2$: ideal one-time pad with MAC. Conditioned on H, K_m, and a random function R, the ciphertext C in H_1 is the XOR of the plaintext with a truly random stream, so it is information-theoretically one-time-pad encrypted. The only way the adversary can distinguish the challenge bit is through MAC forgeries that allow it to modify ciphertexts while retaining valid tags. Consider a conceptual hybrid H_2 where the MAC oracle rejects all ciphertexts not honestly generated by the encryption oracle. In H_2 the challenge ciphertext is perfectly hidden and A's IND-CCA advantage is 0.

If A can distinguish H_1 from H_2 with advantage δ, then we can build a MAC forger B_2 that runs A and simulates H_1 using its MAC oracle. Whenever A submits a ciphertext to the decryption oracle that is not an honest encryption oracle output but nevertheless verifies, B_2 outputs the corresponding MAC input and tag as a forgery. Therefore,

$$|\Pr[\,A \text{ succeeds in } \mathrm{H}_1\,] - \Pr[\,A \text{ succeeds in } \mathrm{H}_2\,]| \leq \mathrm{Adv}^{\text{MAC}}_{\mathsf{MAC}}(B_2).$$

Combining the hybrids and using that A's advantage in H_2 is 0 gives the claimed bound. □

5.4 Ciphertext integrity

By the standard Encrypt-then-MAC paradigm, Theorem 1 implies ciphertext integrity as well.

Theorem 2. *Under the MAC assumption on keyed BLAKE3, SARX-Vault achieves ciphertext integrity (INT-CTXT) in the derived-key model. In particular, the probability that a PPT adversary outputs a fresh ciphertext-tag pair that decrypts to a non-$\perp$ value is bounded by the SUF-CMA advantage against the MAC.*

The proof follows directly from the fact that any such ciphertext must have a valid tag under k_{mac} and was not generated by the encryption oracle, producing a MAC forgery; see e.g. Bellare and Namprempre's analysis of Encrypt-then-MAC.

5.5 Passwords and offline attacks

The reduction above treats (K_s, K_m) as independent uniform keys. In reality they are derived from a password via Argon2id and BLAKE3 using public salts and parameters from H. An attacker with a vault file but no password may mount an offline attack by iterating over candidate passwords, recomputing (K_s, K_m) for each, and checking whether the MAC verifies. The cost of such an attack is dominated by the cost of Argon2id (and, if enabled, thermo hardening) per guess, and multiplied by the dictionary size. This is orthogonal to the symmetric security of SARX-Vault and must be addressed by appropriate parameter choices and password policies.

6 Thermodynamic Hardening

We now formalise the optional thermodynamic hardening mode, which wraps the Argon2id output in a deterministic, public post-processing function to increase the per-guess cost of offline attacks. We show that this does not affect the security reduction (it is key-preserving in the derived-key model), and we give an independent cost analysis in terms of both operations and an approximate physical energy model.

6.1 Definition

Let $\texttt{okm} \in \{0,1\}^L$ be the outer key material produced by Argon2id. Thermo hardening computes a new key $\texttt{okm}' := H(\texttt{okm})$, where H is a deterministic function defined as follows:

1. **Seed derivation.** Compute a 32-byte seed via BLAKE3:

$$\texttt{seed} := \mathrm{BLAKE3\_XOF}(\texttt{"SARX-THERMO-SEED"} \parallel \texttt{okm})[0..32].$$

2. **Scratch buffer initialisation.** Allocate a scratch buffer of N 64-bit words (e.g. corresponding to 512 MiB) and fill it deterministically with pseudorandom data from BLAKE3 keyed by `seed` and a domain separator.

3. **Random walk.** Interpret `seed` as four 64-bit words (x_0, x_1, x_2, x_3), and perform W steps of an ARX-based random walk over the scratch buffer: at each step update (x_0, x_1, x_2, x_3) with a small ARX function and use a combination of (x_0, x_2, x_3) to select an index in the scratch buffer to read, mix, and write back.

4. **Mix extraction.** After W steps, compute a 32-byte mask $\mathsf{Mix}(\texttt{okm})$ by hashing a sampling of the final scratch buffer state via BLAKE3 with a domain separator.

5. **Output.** Define $\texttt{okm}' := \texttt{okm} \oplus \mathsf{Mix}(\texttt{okm})$, extending Mix periodically over the length of `okm`. The scratch buffer is then securely zeroed and discarded.

In practice, the same H is applied during encryption and decryption whenever `kdf_id` $= 3$, so the derived keys remain deterministic functions of the password and header.

We stress that thermo hardening does not add entropy; it increases the time and memory cost per evaluation of the KDF.

6.2 Bit-level energy floor

To give an additional quantitative perspective on cost, we consider a simple physical lower bound on the energy required for irreversible bit operations. For a single bit at temperature T (Kelvin) and confined to a region of characteristic radius R (meters), we take:

$$E_L(T) = k_B T \ln 2,$$
$$E_B(R) = \frac{\hbar c \ln 2}{2\pi R},$$

where E_L is the Landauer bound and E_B the Bekenstein bound, with k_B the Boltzmann constant, $\hbar$ the reduced Planck constant, and c the speed of light. We define the per-bit energy floor as

$$E_{\text{bit,min}}(T, R) := \max\{E_L(T), E_B(R)\}.$$

For concreteness, we evaluate $E_{\text{bit,min}}(T, R)$ at $T \approx 300\,\text{K}$ (room temperature) and a characteristic radius R on the order of $1\,\text{cm}$, representing a small region of a desktop CPU. This yields a lower bound on the Joules per bit required by any physical implementation performing irreversible bit erasures in that environment.

For a representative environment with $T = 300\,\text{K}$ and $R = 1\,\text{cm}$, we obtain

$$E_{\text{bit,min}}(300\text{ K}, 1\text{ cm}) \approx 2.87 \times 10^{-21}\text{ J/bit},$$

where the Landauer bound dominates. This serves as a reference floor: any physical implementation of irreversible bit erasures at these scales must consume at least this much energy per bit.

6.3 Empirical J/bit measurements

On a 6-core desktop CPU with TDP $\approx 65\,\text{W}$ and a $256\,\text{MiB}$ buffer, running 4 iterations of each workload, we measure:

- write-only: ≈ 4.69 J total, corresponding to 5.46×10^{-10} J/bit, i.e. an overhead of $\approx 1.9 \times 10^{11}$ over $E_{\text{bit,min}}$;
- read–modify–write: ≈ 4.93 J total, corresponding to 5.74×10^{-10} J/bit, i.e. an overhead of $\approx 2.0 \times 10^{11}$.

These numbers are consistent with the expectation that real hardware and software incur many orders of magnitude of overhead over the theoretical minimum $E_{\text{bit,min}}(T, R)$, even for simple bulk-memory patterns. We emphasise that this model is approximate (assuming constant power), and we use it only as a coarse baseline.

For each workload we measure the elapsed time t and approximate the energy used as $E \approx P \cdot t$, where P is an assumed average CPU power (e.g. $65\,\text{W}$ for a desktop CPU). Dividing by the number of logical bit erasures gives an empirical J/bit for write-heavy patterns. Comparing this to $E_{\text{bit,min}}(T, R)$ yields an overhead factor

$$\text{overhead} := \frac{\text{measured J/bit}}{E_{\text{bit,min}}(T, R)}.$$

In our experiments, the measured J/bit for write-only and read–modify–write workloads exceeded $E_{\text{bit,min}}$ by several orders of magnitude, as expected given the inefficiencies of real hardware and software. We do not claim precision in these estimates; they serve only as a coarse physical baseline.

6.4 KDF cost amplification experiments

We also directly compare the cost of baseline Argon2id and Argon2id-plus-thermo hardening using a simple brute-force simulation. We generate a set of random 16-byte passwords and a fixed salt, and for each password we compute:

- a baseline key via Argon2id (with fixed parameters), and
- a thermo-hard key via Argon2id followed by the scratch-buffer random walk and mixing described above.

For each mode we measure:

- total elapsed time over all guesses,
- approximate energy per guess $E_{\text{guess}} \approx P \cdot t / N_{\text{guesses}}$ (with an assumed power P),
- for the thermo-hard mode, an approximate number of bit erasures in the scratch walk (e.g. assuming ≈ 32 bits flipped per 64-bit word update), yielding an empirical J/bit.

We then report:

- the ratio of energy per guess between thermo-hard and baseline modes, and
- the overhead of thermo-hard J/bit versus $E_{\text{bit,min}}$.

In our reference configuration (e.g. scratch buffer of a few hundred MiB and several hundred thousand random-walk steps per guess), we observe that thermo hardening increases the per-guess cost by a multiplicative factor greater than 1 (often by several times) compared to pure Argon2id, and that the J/bit consumed in the scratch walk remains well above the physical floor $E_{\text{bit,min}}(T, R)$. The exact numbers depend on the chosen (N, W) and hardware, but this experiment confirms that thermo hardening acts as a tunable cost-amplification layer: increasing (N, W) reliably increases the per-guess work and energy.

6.5 Security neutrality and ARX design

The previous subsection addresses cost; from a cryptographic perspective we require that thermo hardening does not weaken the scheme. As shown in the security analysis, H can be modelled as a public, deterministic post-processing function on `okm`, keyed only by domain separators and `okm` itself via BLAKE3. Under the assumption that Argon2id outputs are indistinguishable from uniform and that BLAKE3 behaves as a PRF, replacing `okm` with $\texttt{okm}' = H(\texttt{okm})$ does not change the distribution of derived keys in a way that an adversary can exploit; the AEAD security reduction for SARX-Vault thus carries over unchanged.

Finally, the ARX update used in the random walk is chosen in the same spirit as the SARX core: we use simple add–rotate–xor operations on 64-bit words and avoid data-dependent memory access patterns beyond the indexing induced by (x_0, x_2, x_3). In separate experiments we scan single-word ARX recurrences of the form

$$x \mapsto \mathrm{ROTL}_{64}\big((x + (x \ll a)) \oplus (x \gg b),\, c\big),$$

measuring avalanche behaviour and distortion of Hamming distances to favour near-ideal diffusion. The chosen parameters for both SARX and the thermo random walk come from this class of candidates, ensuring that the ARX components used throughout the design exhibit strong bit-level mixing and do not introduce obvious structural weaknesses.

In a concrete experiment with $N = 256\,\mathrm{MiB}$ of scratch memory, $W = 500{,}000$ random-walk steps per guess, and $P = 80\,\mathrm{W}$ assumed power, we simulated 100 random password guesses. For baseline Argon2id we measured:

$$t_{\text{base}} \approx 1.57 \text{ s}, \quad E_{\text{base}} \approx 1.25 \times 10^2 \text{ J},$$

corresponding to ≈ 1.25 J/guess. With thermo hardening enabled we observed:

$$t_{\text{thermo}} \approx 2.49 \text{ s}, \quad E_{\text{thermo}} \approx 1.99 \times 10^2 \text{ J},$$

or ≈ 1.99 J/guess, i.e. a per-guess energy increase by a factor of about 1.6 over Argon2id alone.

For the thermo-hard run, the scratch random walk performed roughly 1.6×10^7 logical bit erasures per guess, giving an empirical J/bit of $\approx 1.25 \times 10^{-7}$ J/bit. Compared to $E_{\text{bit,min}}(300\text{ K}, 2\text{ cm}) \approx 2.87 \times 10^{-21}$ J/bit, this corresponds to an overhead of roughly 4.3×10^{13} in energy per bit erased. This confirms that thermo hardening is an aggressively wasteful, tunable cost amplifier: increasing (N, W) directly increases the per-guess work factor, while the symmetric security guarantees remain governed by the PRF and MAC assumptions.

7 Experimental Evaluation

We briefly report on our implementation and experimental evaluation of SARX and SARX-Vault. All tests were performed using a reference Rust implementation, compiled with optimisations enabled.

All SARX operations are expressed in terms of 64-bit additions, rotations, and XORs, with no data-dependent table lookups, making it straightforward to implement the core cipher in constant time on general-purpose CPUs. We rely on existing constant-time implementations of Argon2id and BLAKE3 and do not analyse microarchitectural side-channel leakage in this work.

7.1 Randomness testing

We generated long keystreams from SARX for randomly chosen keys and counters and subjected them to four standard statistical test suites:

- NIST SP 800-22 on multiple sequences of length 10^6 bits;
- the Dieharder battery on streams of size up to at least 2^{40}–2^{41} bytes;
- PractRand `stdin64` on multi-gigabyte keystream samples;
- TestU01 BigCrush on keystream interpreted as 32- and 64-bit words.

Across all four suites we observed no tests reported as failures or anomalies: p-value distributions remained within the ranges expected under the null hypothesis of uniform random input. As usual, these tests do not constitute a proof of cryptographic security, but they help rule out simple statistical biases and support modelling SARX as a pseudorandom generator in our reduction.

7.2 Performance benchmarks

We implemented SARX and SARX-Vault in Rust and benchmarked on a commodity desktop CPU (For testing, a Ryzen 5 3600 overclocked to 4.0ghz). A typical keystream benchmark encrypts 100 MB of data per run and averages over multiple runs.

A representative result for parallel keystream generation (12 hardware threads) is shown in Table 1. Here, KS denotes pure keystream generation throughput, ENC the throughput of XOR-encryption with SARX, and DEC the corresponding decrypt throughput.

The keystream produce 12,500 MB/s on 6 cores / 12 threads when ran alone.

Threads	KS (MB/s)	ENC (MB/s)	DEC (MB/s)
1	≈ 1460	≈ 845	≈ 990
2	≈ 1760	≈ 1495	≈ 1840
4	≈ 3250	≈ 2210	≈ 2070
6	≈ 4660	≈ 2650	≈ 2100
8	≈ 5390	≈ 3140	≈ 2100
12	≈ 7300	≈ 3640	≈ 2210

Table 1: Representative SARX throughput on a multi-core CPU, End-to-End with AEAD.

Thermo hardening adds a configurable overhead proportional to the scratch buffer size and walk length.

7.3 Reduced-round analysis

We performed preliminary reduced-round analysis of SARX by examining differential and linear behaviour of variants with fewer than 8 rounds. For 1–2 rounds, we observed predictable non-uniformities, high-probability differentials, and detectable linear biases, as expected for any short ARX network. For 3–4 rounds the observed biases declined quickly with the number of rounds. At the full 8 rounds we did not observe distinguishers beyond what would be expected from random sampling noise at the data volumes tested.

We do not claim a formal lower bound on the number of secure rounds, but these experiments suggest that 8 rounds offer a reasonable security margin over structurally weak reduced-round variants while preserving good performance.

8 Related Work

SARX belongs to the family of ARX-based stream ciphers pioneered by Salsa20 and its variants. ChaCha20, in particular, is widely deployed as a high-speed software cipher, and pairs with Poly1305 to form popular AEAD constructions such as ChaCha20-Poly1305. Compared to these designs, SARX uses a smaller, four-word 64-bit state and fewer rounds, aiming at a compact specification and high throughput, with security supported by empirical analysis and the PRF assumption.

On the password-based side, SARX-Vault follows the general pattern of combining a memory-hard KDF and an AEAD scheme for file encryption. Argon2id is the winner of the Password Hashing Competition and a widely recommended choice for password-based key derivation. Our use of Argon2id and keyed BLAKE3 in an Encrypt-then-MAC composition fits within established practice; the main novelty is the explicit definition of a stream-cipher–based vault format with a clear security model and optional cost-amplification via thermo hardening.

Thermo hardening can be seen as a structured way to add a tunable, memory-heavy post-processing layer on top of Argon2id, akin in spirit to constructions that layer additional hash- or memory-based work on top of a KDF to raise the cost of offline attacks. Our contribution is to formalise such a post-processing step, embed it in a concrete scheme, and show that it is neutral from the point of view of the symmetric security reduction.

9 Conclusion

We have presented SARX, a 256-bit ARX-based stream cipher, and SARX-Vault, a password-based AEAD construction that combines SARX, Argon2id, and keyed BLAKE3 in an Encrypt-then-MAC design. Under standard PRF and MAC assumptions, SARX-Vault achieves IND-CCA security and ciphertext integrity in a derived-key model. An optional thermo hardening mode provides a principled way to raise the per-guess cost of offline password attacks without weakening the symmetric security guarantees.

Our experimental evaluation shows that SARX passes widely used randomness test suites, including NIST SP 800-22, Dieharder, PractRand, and TestU01 BigCrush, and achieves multi-gigabyte-per-second throughput on commodity CPUs. These results support the suitability of SARX and SARX-Vault as building blocks for password-based vault and file encryption systems.

Future work includes deeper cryptanalysis of the SARX core, including higher-order differential and linear attacks and algebraic modelling, as well as exploration of implementation trade-offs and parameter choices for thermo hardening in various deployment scenarios.

Declarations

Use of AI-assisted technologies. The author(s) used AI-assisted tools solely to help with LaTeX typesetting and formatting, including equation layout, Greek symbols, algorithm pseudocode, and tables summarizing empirical results. The author(s) have carefully verified and take full responsibility for the content of this manuscript.

Declaration of interest. The author(s) declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Biryukov, D. Dinu, and D. Khovratovich. Argon2: The memory-hard function for password hashing and other applications. In *Proceedings of the IEEE European Symposium on Security and Privacy*, 2016.

[2] J. O'Connor et al. The BLAKE3 cryptographic hash and PRF. Specification and analysis, 2020. Available at `https://github.com/BLAKE3-team/BLAKE3`.

[3] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *ASIACRYPT 2000*, LNCS 1976, pp. 531–545. Springer, 2000.

[4] A. Rukhin et al. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST Special Publication 800-22, 2010.

[5] R. Brown. Dieharder: A random number test suite. Software and documentation, `https://webhome.phy.duke.edu/~rgb/General/dieharder.php`.

[6] C. Doty-Humphrey. PractRand: A suite of tests for random number generators. Software and documentation, `http://pracrand.sourceforge.net/`.

[7] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), 2007.

[8] D. J. Bernstein. Salsa20 specification. In *New Stream Cipher Designs*, LNCS 4986, pp. 84–97. Springer, 2008.

[9] D. J. Bernstein. ChaCha, a variant of Salsa20. Workshop Record of SASC 2008.

[10] M. Rogaway. Authenticated-encryption with associated-data. In *CCS 2002*, pp. 98–107. ACM, 2002.