An Innovative Lightweight Symmetric Encryption Algorithm Integrating Little OaldresPuzzle_Cryptic

The Components: NeoAlzette ARX Box, XorConstantRotation CSPRNG Hybrid8–Shadow, and a Key-Switch Layer

Twilight-Dream[1]

Independent Researcher, yujiang1187791459@gmail.com

Abstract.

Little OaldresPuzzle_Cryptic (LOPC) is specified as a parameterized 128-bit keyed block-transformation core built from three internal components: the NeoAlzette 64-bit ARX-box layer, the XorConstantRotation (XCR) Hybrid8–Shadow round-material generator, and a reversible Key Switch Layer. For every 128-bit data block, the core is invoked with a 128-bit key and a one-time public input `number_once`; the implementation supports counter-indexed block processing by assigning distinct `number_once` values to successive blocks. Authentication is not implemented inside the LOPC core: authenticated encryption must be obtained by composing the core with an external MAC or AEAD wrapper.

The paper gives a code-faithful specification of the current construction, including the diagonal NeoAlzette placement, XCR-derived round material, keyed switching, and the keyed bit-position tweak. For the NeoAlzette component, it provides local differential and linear MILP boxes, fixed-public arithmetic oracle interfaces, and a mathematical oracle-to-MILP backbone for MEDCP/MELCC-style single-characteristic searches. The current artifact additionally reports one-round 12-hour SCIP incumbent campaigns for differential and linear NeoAlzette characteristics; these are trace/oracle-consistent incumbents, not solver-certified optima. For XCR, it specifies the Hybrid8 byte layer, the zero-fill Shadow arithmetic operators, the non-padded polynomial-Weyl constant stream, and the exact component-oracle boundary; full pseudorandom-generator security of XCR is not claimed.

The manuscript separates component-local evidence from composition-level security claims. It proves round invertibility of the specified core and records the current component-analysis infrastructure, but it does not claim a complete AEAD mode, a full PRP/PRG proof for the LOPC composition, or complete endpoint differential/linear hull bounds unless the corresponding endpoint enumeration is explicitly complete. The repository also contains a separate OPC companion block-cipher artifact; Appendix B records its artifact boundary, but OPC is not analyzed as part of the LOPC security claims.

Keywords: ARX cryptography, lightweight symmetric encryption, NeoAlzette ARX-box, XorConstantRotation, round-material generation, counter-indexed block processing, keyed switching, shadow-boxed arithmetic, Hybrid8, MEDCP, MELCC, MILP trail search, endpoint HULL, implementation artifact

Contents

Technical Snapshot 6

I Introduction 7

1 Long-form Motivation and Related Work 7

2 LOPC as a Three-Component Construction 8

3 Design Problem and Main Contributions 9

4 Claim Boundary, Companion OPC Artifact, and Evidence Levels 10

5 Organization of the Paper 12

II Common Primitives 12

6 Word Space and Bit Numbering 12

7 Bit and Byte Accessor Functions 12

8 Boolean Word Operations 13

9 Modular Addition and Subtraction 14

10 Rotations 14

11 Zero-Filling Shifts 15

12 State and Joint Injection Values 15

13 Fixed-Public Arithmetic Relations 16

14 State Packing and Word Decomposition 16

15 Affine Constants, XOR Injection, and Phase Terms 16

16 Oracle Vocabulary for Differential and Linear Analysis 16

17 MILP Variable and Weight Conventions 17

18 Code-Faithful Pseudocode Convention 17

III Specification 17

19 Global LOPC State and Interface 17

20 Round Skeleton 17

21 Global LOPC Algorithm Specification **18**
21.1 LOPC counter-indexed block processing and external authentication boundary . . . 18
21.2 `LittleOaldresPuzzle_Cryptic.h` framework pseudocode . . . 18
21.3 Current per-round XCR-derived key-state generation . . . 20
21.4 Symmetric encryption specification . . . 21
21.5 Symmetric decryption specification . . . 21

22 Component A Specification: NeoAlzette **22**
22.1 Code-derived component boundary . . . 22
22.2 V6.5 implementation oracle . . . 22
22.3 Public constants and diffusion masks . . . 23
22.4 C/D injection equations . . . 24
22.5 Routed XOR and CD-derived addend interfaces . . . 24
22.6 Forward schedule . . . 25
22.7 Backward schedule and inversion . . . 25

23 Component B Specification: XCR CSPRNG **26**
23.1 Abbreviation dictionary and proof method for the XCR component . . . 26
23.2 Engineering state, public schedule, and analysis-oracle order . . . 28
23.3 Default initialization constant and initialization-root oracle . . . 28
23.4 Public `COUNTER_STEP` derivation and schedule oracle . . . 30
23.5 XCR round-constant stream generation . . . 31
23.6 Code-faithful XCR evaluation oracle . . . 32
23.7 Engineering primitive pseudocode behind the XCR oracles . . . 33
23.8 Hybrid8 byte layer . . . 36
23.9 Code-faithful XCR state transition . . . 37
23.10 StateIteration, GenerateSubKey128, and StateInitialize . . . 38

24 Component C Specification: Key Switch Layer **39**
24.1 Key Switch Layer: keyed switching and random bit tweak . . . 39

25 Decryption and Permutation Structure **40**

IV Design Rationale **40**

26 Global Design Rationale **40**

27 Component A Rationale: NeoAlzette **41**
27.1 Design Background . . . 41
27.2 Inheritance from Alzette . . . 41
27.3 Relation to the Prior Fallback Variant . . . 41
27.4 Fixed Public Subtraction . . . 41
27.5 Why the Constant Does Not Belong Only Inside the Injection . . . 42
27.6 Rotation Roles . . . 42
27.7 Bridge Selection Derivation . . . 42
27.8 First Bridge: 22 and 13 . . . 44
27.9 Second Bridge: 5 and 25 . . . 44
27.10 Why Not Reuse the Same Rotation Family . . . 44
27.11 Joint Boolean–Arithmetic Injection Semantics . . . 44
27.12 Implementation-Derived Schedule Reading . . . 47
27.13 Design Summary . . . 47
27.14 Quadratic Boolean Front Half and C/D injection . . . 47

28 Component B Rationale: XCR CSPRNG **49**
28.1 Operator-level rationale for the XCR H/S arithmetic 49

29 Component C Rationale: Key Switch Layer **50**
29.1 MixLinearTransform rationale 50
29.2 RandomBitTweak rationale 50

V Security Evaluation **50**

30 Claim Boundary **50**

31 Formal Security and Evidence Boundary of the Current LOPC Construction **50**

32 Standalone ARX Algebraic Security Analysis **51**
32.1 Matrix-analytic ARX complexity: what is kept and what is not 51
32.2 Standalone ARX Algebraic Complexity Microscope: The 8-bit Matrix Tower 52
32.3 Additive Differential Refinement via the Niu–Sun–Hu ARX Matrix Model 66

33 Component A Security: NeoAlzette **69**
33.1 Scope of the Component-A MILP Boxes 69
33.2 Binary and Bit-Vector MILP Boxes 70
33.3 Four Modular Addition/Subtraction MILP Boxes 70
33.4 Fixed-Public Arithmetic Call Sites 72
33.5 Joint-Injection Oracle Hook 72
33.6 How These Boxes Assemble the NeoAlzette Model 73
33.7 Result Status and MEDCP/MELCC Claim Boundary 73
33.8 NeoAlzette MILP Single-Characteristic Campaign Results 74

34 Component B Security: XCR CSPRNG **75**
34.1 Analysis oracles for XCR before MILP/SAT modeling 75
34.2 Exact differential-probability (DP) and linear-correlation (LC) transfer model for shadow operators 78
34.3 XCR factor-graph differential/linear model 79
34.4 XCR pseudorandom-generator (PRG) game boundary 80

35 Component C Security: Key Switch Layer **80**
35.1 MixLinearTransform propagation 80
35.2 Keyed choice effect 80
35.3 RandomBitTweak propagation 80
35.4 Branchless implementation boundary 81

36 Composition-Level Security Discussion **81**
36.1 Code-bound semantic object of the current LOPC round 81
36.2 NeoAlzette component theory after absorption 82
36.3 XCR component proof boundary after the rebuilt XCR section 82
36.4 Cryptographic Game Analysis of Encryption and Decryption Functions . 83
36.5 LOPC composition boundary 83

VI Software Implementation **84**

37 Implementation Benchmark Methodology **84**

38 Global C++20 Implementation Boundary **84**

39 Component A Implementation: NeoAlzette **85**

40 Component B Implementation: XCR **85**

41 Component C Implementation: Key Switch Layer **85**

42 Constants and Reproducibility **85**
42.1 Round-constant generation for XCR 85

43 Repository, Run Tests, and Artifact Notes **85**
43.1 Artifact and Reproducibility Notes 86

VII Hardware and Implementation Considerations **86**

44 Global Cost Model **86**

45 Component A Notes: NeoAlzette **86**

46 Component B Notes: XCR **86**

47 Component C Notes: Key Switch Layer **86**

48 Limitations **87**

VIII Conclusion **87**

49 Summary **87**

50 Component Summaries **87**

51 Future Work **87**

References **88**

IX Appendices **94**

A Imported NeoAlzette Oracle and MILP Backbone **94**
A.1 Purpose and mathematical scope 94
A.2 Notation and probability spaces 94
A.3 Modeling questions Q1 and Q2 95
A.4 Two-variable Q2 completion oracles: CDDT and CLAT 98
A.5 Fixed-public add/sub differential Q1 oracle 103
A.6 Fixed-public add/sub linear Q1 oracle 108
A.7 Fixed-public MILP interfaces and exact-log epigraph 113
A.8 Two-variable modular addition differential oracle 124
A.9 Two-variable modular addition linear oracle 125
A.10 Modular subtraction wrappers 125
A.11 Boolean and bit-vector MILP box derivations 125
A.12 ADD-Δ MILP derivation 126
A.13 ADD-L MILP derivation 126

A.14 Joint C/D injection oracle 127
A.15 QFH / Hessian / rank derivation 127
A.16 Top-level NeoAlzette MILP model assembly 128
A.17 MEDCP/MELCC search semantics 128
A.18 Endpoint HULL / Forest semantics 129
A.19 Mathematical pseudocode summary 131
A.20 Artifact traceability table 131

B Companion OPC Block-Cipher Artifact in the Repository 133

C Historical BPHT Note and Current Status 133

D Data Tables and Benchmark Tables 133

E MILP Campaign Artifact Summaries 134

F Implementation Code Listings 135

Technical Snapshot

Table 1: Technical snapshot of the reorganized LOPC manuscript.

Object	Little OaldresPuzzle_Cryptic (LOPC), a 128-bit symmetric block/permutation-style encryption core.
Block and key size	128-bit block interface and 128-bit key interface in the current C++ reference implementation.
State layout	Two 64-bit lanes, unpacked as four 32-bit words when the diagonal NeoAlzette layer is applied.
Component A	NeoAlzette: 64-bit ARX-box layer applied to (w_0, w_2) and (w_1, w_3).
Component B	XCR round-material generator: stateful Hybrid8–Shadow generator producing subkeys, branch choices, rotation amounts, and counter-indexed material from one-time public inputs.
Component C	Key Switch Layer: `MixLinearTransform_Forward/Backward` plus RandomBitTweak / Keyed Bit-Position Tweak.
Block-processing API	LOPC encrypts/decrypts one 128-bit block under a 128-bit key and a one-time public `number_once`; multi-block processing is counter-indexed by assigning distinct values to successive blocks. Authentication is external.
Current claim type	Component specifications, round invertibility, one-round 12-hour SCIP incumbent MEDCP/MELCC-style single-characteristic results for NeoAlzette, component-level oracles and reduced diagnostics for XCR Shadow arithmetic, and affine/keyed switching analysis for the Key Switch Layer.
Non-claims	No built-in authentication, no complete external AEAD/MAC composition proof, no full XCR PRG proof, and no complete endpoint differential hull or endpoint linear hull claim unless the corresponding HULL enumeration is explicitly complete.

Part I
Introduction

1 Long-form Motivation and Related Work

Lightweight symmetric cryptography is increasingly built from operations that are individually simple but difficult to analyse once they are composed: XOR, rotations, modular addition, modular subtraction, fixed constants, key-dependent words, and small scheduling or switching layers. ARX designs such as Salsa20 and ChaCha demonstrate why this style is attractive in software: the primitives are word-oriented, constant-time on common processors, and compact in constrained implementations [Ber05, Ber08, B+08a, B+08b, NL18]. The same design discipline also appears in lightweight and NIST-LWC-era constructions, where reproducible public constants, short instruction sequences, and explicit state boundaries are central engineering concerns [Nat18, DEMS21, TMC+25, BBdS+21, BP17]. In this manuscript, that engineering motivation is used conservatively: compact code is not treated as a security proof. Instead, the code is used as the object whose arithmetic, state boundaries, and generated round material must be specified before any differential, linear, MILP, HULL, or composition claim is discussed.

The difficulty is that the apparent simplicity of ARX/SRX code hides finite-state arithmetic. XOR and rotations are linear over $\mathbb{F}_2$; XOR by a constant is affine. Modular addition and modular subtraction are different: they introduce carry and borrow chains whose transition at bit $i+1$ depends on low-order data and on the current carry or borrow state. This is why ARX primitives are efficient to implement but nontrivial to analyse: a single source-level symbol such as $\boxplus$ or $\boxminus$ denotes a low-to-high state machine rather than an ordinary linear map. The cryptanalytic literature on modular addition reflects exactly this point. Lipmaa and Moriai gave efficient algorithms for differential properties of two-variable modular addition [LM02]; Wallén developed carry-function classifications and algorithms for linear approximations [Wal03a, Wal03b]; Schulte-Geers later connected two-variable addition to a quadratic Boolean representation through CCZ equivalence [SG13]. Subsequent ARX trail-search work incorporated modular-addition models into branch-and-bound, MILP, SAT, and related automatic frameworks [BV, BVC16, FWG+16, LLJW21, LWR16, SHW+14].

A second line is crucial for this paper: one-addend-fixed arithmetic. In an expression such as $x \mapsto x \boxplus K$, the operand K is not averaged over; it is fixed by a key, constant, subkey word, counter-derived value, or surrounding state. Miyano already observed that differential and linear probabilities of such additions depend on the addend [Miy98]; Machado supplied a direct algorithmic treatment on the differential side [Mac01]; Azimi et al. later gave bit-vector differential models for modular addition by a constant [ARS+20, ARS+22]. The fixed-addend branch is not a cosmetic specialization of the two-variable formula. It changes the probability space: the randomness is over the variable input only, while the bit pattern of K indexes the local carry/borrow transition family. The present manuscript therefore treats fixed-public arithmetic as its own modelling object and connects it to exact fixed-addend differential and linear oracle material [Twi26e, Twi26d].

The recent use of large ARX-based S-boxes and ARX-boxes further sharpens the need for this separation. The Alzette ARX-box showed that a 64-bit word-oriented component can be designed and analysed as a wide S-box-like object, but not by direct DDT/LAT enumeration [BBdS+20]. Instead, its relevant properties are evaluated through trail searches, MILP/SAT-style models, division-property analysis, fixed-constant experiments, and component-specific evidence. Related large-state ARX

S-box work follows the same general lesson: once the state is much larger than 4 or 8 bits, direct table enumeration becomes infeasible and one must analyse structured components through their arithmetic and finite-state representations [FFW23, XLW22, HXW22, LSL21, LNS+23, NSL+22]. This paper follows that philosophy for LOPC, but with a stronger emphasis on code-faithful component boundaries and fixed-public arithmetic interfaces.

Matrix and finite-state descriptions are therefore not decorative in this manuscript. They are a way to expose the object that compact ARX/SRX code hides. Additive-differential work on ARX constructions uses matrix-style transfer models to evaluate probability in structures involving modular addition and rotations [VMCP11, MKA+21, NSH23]. More generally, Mealy-machine and S-function viewpoints formalize the same phenomenon: modular addition, modular addition by a constant, and related sequential word operations can be represented by a product of local transition matrices, with many cryptanalytic properties taking the form of a boundary vector, a product of matrices, and a final boundary vector [NBHW25, Leu15]. The standalone matrix microscope later in this paper uses an 8-bit example not as a reduced security claim, but as a readable witness: even $z = x \boxplus_8 ((y \lll 2) \oplus k)$ expands into rotation matrices, affine constant-XOR matrices, carry-state transition matrices, and output reconstruction equations. The purpose is to make the ARX/SRX state explosion visible before the security-evaluation sections begin.

2 LOPC as a Three-Component Construction

Little OaldresPuzzle_Cryptic (LOPC) is the object specified in this paper. It is a 128-bit keyed block-transformation core whose current reference implementation operates on two 64-bit lanes, temporarily unpacks them into four 32-bit words for the diagonal ARX-box layer, and consumes XCR-derived round material through the block wrapper. LOPC is neither a standalone stream cipher nor a single isolated ARX-box. It is a three-component construction: Component A is NeoAlzette, Component B is the XorConstantRotation (XCR) Hybrid8–Shadow round-material generator, and Component C is the Key Switch Layer. The LOPC wrapper is the composition boundary that calls and connects these components; it is not counted as a fourth primitive component.

Component A, NeoAlzette, supplies the main ARX-box nonlinearity of the LOPC round. In the current state layout, the 128-bit block is unpacked as four 32-bit words and NeoAlzette is applied diagonally to (w_0, w_2) and (w_1, w_3). The design is influenced by the large-ARX-box lesson of Alzette, but it is not an Alzette instance: it introduces a joint Boolean–arithmetic injection layer, fixed-public arithmetic sites, and code-derived MILP/HULL interfaces. Its local differential and linear evidence is reported as MEDCP/MELCC-style single-characteristic material unless an endpoint HULL enumeration is explicitly complete [BBdS+20, LM02, Wal03a, SG13, LLJW21].

Component B, XCR, is the stateful round-material source. It derives subkeys, branch-choice bits, rotation amounts, and counter-indexed material from the master-key side and the public `number_once` input. It internally separates Hybrid8 H-rounds from zero-fill Shadow arithmetic S-rounds before final XOR/rotation diffusion. In this paper, XCR is analysed and specified as a round-material generator feeding LOPC. Its full pseudorandom-generator security is not proved here; the paper records its step relation, public schedule, component oracle boundary, and the assumption boundary under which the LOPC wrapper uses it. Salsa20 and ChaCha are cited only as canonical ARX stream-cipher background for why stateful word generators are attractive; LOPC does not inherit their security arguments [Ber05, B+08a, NL18].

Component C, the Key Switch Layer, is the keyed post-ARX switching/tweak layer.

For fixed XCR-derived round material, `MixLinearTransform_Forward/Backward` is a reversible affine/permutation switch on the repacked 64-bit lanes, and the RandomBitTweak / Keyed Bit-Position Tweak is a self-inverse keyed bit-position XOR injection. This component is not treated as a standalone nonlinear S-box and not treated as an independent source of randomness. Its role is to change the affine coordinate system and selected bit positions seen by the next round while preserving deterministic invertibility for every fixed generated key state.

Definition 1 (Three-component decomposition of LOPC)**.** The current LOPC paper uses the following component boundary:

$$\mathsf{LOPC} = \mathsf{Compose}(\mathsf{NeoAlzette}, \mathsf{XCR}, \mathsf{KeySwitch}).$$

Here NeoAlzette is the 64-bit ARX-box applied diagonally to the four 32-bit words of the 128-bit state; XCR is the Hybrid8–Shadow stateful generator used to derive round material; and KeySwitch is the deterministic reversible layer consisting of the MixLinearTransform forward/backward routines and the random bit tweak. This decomposition is a reading rule for the whole paper.

Lemma 1 (The Key Switch Layer is a separate component)**.** *The Key Switch Layer is not absorbed into* NeoAlzette *and is not a call to XCR. For fixed XCR-derived key-state words, it is a deterministic reversible lane transform.*

Proof. NeoAlzette acts on two 32-bit words and is called before repacking. XCR is called during key-state generation and does not transform the data block directly. The switching layer acts after the diagonal NeoAlzette calls on the two 64-bit data lanes, selecting one of four reversible branches and then applying the bit tweak. These are disjoint interfaces in the source code, so treating them as one primitive would hide a real composition boundary. □

3 Design Problem and Main Contributions

The central design problem is that the three LOPC components cannot be analysed honestly if their interfaces are flattened into independent black boxes. In NeoAlzette, the injection layer is deliberately a joint Boolean–arithmetic object:

$$\begin{aligned}\text{Joint Injection Layer} = {} & \text{Quadratic Boolean Front Half} \\ & + \text{Two-Variable Modular-Addition Back Half.}\end{aligned}$$

The front half produces coupled C/D-derived Boolean material from a source branch. One projection is routed through XOR into the opposite branch, while another projection becomes the non-constant operand of a subsequent modular addition. Therefore the XOR path and the arithmetic path are source-bound: their differential variables and linear masks must come from the same quadratic source oracle. Treating them as independent would create trails or approximations that are not present in the implementation. This is why the paper separates ordinary two-variable modular addition, fixed-public arithmetic, and source-constrained addition in the oracle and MILP models.

The contribution of this manuscript is organized as follows. First, it gives a code-faithful specification of the LOPC block-transformation core, including diagonal NeoAlzette placement, XCR-generated round material, keyed switching, and the keyed bit-position tweak. Second, it records a design rationale for the three-component construction, including the joint Boolean–arithmetic injection layer and the XCR round-material role. Third, it develops an algebraic complexity microscope for

ARX/SRX arithmetic, including an explicit 8-bit hand calculation of $z = x \boxplus_8 ((y \lll 2) \oplus k)$ that expands one line of source-level arithmetic into rotation matrices, affine constant-XOR matrices, carry-state transition matrices, and an output reconstruction vector. Fourth, it provides local differential and linear MILP boxes, fixed-public arithmetic oracle interfaces, joint-injection coupling constraints, semantic HULL/Forest boundaries, and implementation artifacts. Finally, it explicitly separates single-characteristic MEDCP/MELCC-style evidence from endpoint differential or linear hull claims.

4 Claim Boundary, Companion OPC Artifact, and Evidence Levels

LOPC is specified here as a block-transformation core. It is not claimed in this paper as a complete authenticated-encryption scheme, a proven PRP/PRG construction, or a fully endpoint-hull-evaluated primitive. Authentication, misuse resistance, and complete AEAD composition are outside the present claim boundary. The security evidence reported here is component-level and model-level: local oracle correctness, fixed-public arithmetic modelling, MILP-based single-characteristic searches, implementation-aligned constraints, and explicit separation between trail evidence and hull aggregation. The purpose of the paper is to make the construction precise, auditable, and analyzable, not to overstate the evidence beyond what was computed.

The repository also contains a companion OaldresPuzzle_Cryptic (OPC) block-cipher artifact. LOPC is the counter-indexed 128-bit core analysed in this manuscript, while OPC is a separate repository artifact. Appendix B records the OPC artifact boundary so that the repository structure is not hidden from the reader. OPC is not analysed in this paper and does not inherit any LOPC component claim, MEDCP/MELCC statement, or XCR assumption. The companion OPC artifact was designed with large-key and post-quantum-motivation concerns in mind, but this paper does not provide a quantum-security proof, a brute-force-resistance theorem, or a complete cryptanalysis of OPC. Those claims require an independent specification and security evaluation.

This manuscript fixes the current three-component engineering split: (i) the analysed NeoAlzette ARX-box, (ii) the XCR Hybrid8–Shadow round-material generator, and (iii) the Key Switch Layer consisting of `MixLinearTransform_Forward/Backward` and the key-state-dependent bit tweak. The LOPC block transform is the composition boundary joining these three components. Benchmark material is treated as implementation methodology rather than proof evidence. Empirical diagnostics, MEDCP/MELCC-style single-characteristic search, endpoint HULL enumeration, and full-system security are reported as separate evidence levels.

Table 2: Three-component evidence discipline adopted in this manuscript. The LOPC wrapper is the composition boundary combining these three components, not a fourth primitive component.

Component	Current evidence level	What is claimed in this paper
NeoAlzette ARX S-box	C++ implementation plus differential/linear MILP, local oracles, semantic HULL enumeration, and multi-core campaign runners; external V6.5 specification and analysis artifact.	Code-faithful permutation specification; exact local arithmetic modelling for the four add/sub cases; resource-limited trail/HULL evidence for the analysed ARX-box component.
XCR Hybrid8–Shadow CSPRNG	C++ implementation, Shadow-Boxed Arithmetic component theory, StateInitialize pseudocode, reduced diagnostics, and explicit PRG assumptions.	Engineering CSPRNG/key-schedule candidate. The deployed S-round uses zero-fill shadow carry/borrow operations; PRG-strength claims remain assumption-backed.
Key Switch Layer	Current MixLinearTransform forward/backward routines plus the random bit-tweak layer, driven by XCR-derived cf_i, α_i, β_i and subkey words.	A deterministic reversible switching/tweak component for each fixed generated key state. It contributes invertible branch selection and key-state-dependent affine/permutation maps, not standalone pseudorandomness.

Definition 2 (Code-faithful claim)**.** A statement is called *code-faithful* if the variables, operation order, word size, constants, and branch dependencies are identical to the C++ implementation that the statement names. A code-faithful claim may still be conditional or heuristic, but it must not silently analyse a different algorithm.

Definition 3 (Local model versus global search)**.** A *local model* specifies one operation such as two-variable modular addition, fixed-addend modular addition, XOR, rotation, byte substitution, or a shadow-boxed carry/borrow relation. A *global search* composes local models along a full schedule and uses MILP, SAT/SMT, finite-state transfer, or HULL enumeration to find or count characteristics. This paper does not conflate local exactness with global exhaustive proof unless the corresponding enumeration status is explicitly complete.

Lemma 2 (Evidence separation)**.** *The NeoAlzette MILP/HULL analysis, the XCR shadow-boxed CSPRNG assumptions, and the Key Switch Layer invertibility argument are logically separable. The LOPC wrapper is the composition boundary that connects them. In particular, a flaw or incompleteness in the XCR PRG proof does not invalidate the local NeoAlzette trail model or the deterministic inverse of the Key Switch Layer, and a resource-limited NeoAlzette MILP run does not by itself prove the full LOPC construction.*

Proof. The three component claims quantify over different objects. The NeoAlzette backend fixes a 64-bit ARX permutation and reasons about masks, differences, local operation weights, and semantic no-good enumeration. XCR is a stateful word generator whose current nonlinear middle layer is Hybrid8–Shadow arithmetic. The Key Switch Layer is a deterministic reversible layer once its generated key state $(sk_i, cf_i, \alpha_i, \beta_i)$ is fixed. The LOPC wrapper composes these components with subkey injection and round ordering. Since the interfaces between these objects are explicit, each proof obligation can be replaced by an ideal oracle, exact component theorem, or assumption while the others remain syntactically unchanged. Therefore the claims must be reported separately. □

5 Organization of the Paper

Part II fixes the common mathematical language: word spaces, bit numbering, Boolean operations, modular arithmetic, rotations, shifts, oracle vocabulary, and MILP weight conventions. The following specification parts define the LOPC round structure and its three components. The design-rationale part explains the three-component construction, the joint-injection layer, the XCR round-material role, and the Key Switch Layer. The standalone ARX/SRX matrix microscope makes the carry/borrow matrix explosion explicit for readers who are not already fluent in ARX analysis. The security-evaluation part then presents differential, linear, fixed-public, MILP, and HULL/Forest evidence boundaries. The appendices collect full oracle derivations, code-derived mathematical interfaces, reference listings, and artifact notes needed to reproduce or audit the construction.

Part II
Common Primitive Operations and Notation

This part is the common language layer for the whole LOPC manuscript. It is intentionally placed immediately after the introduction. The same word spaces, bit numbering, Boolean operations, modular arithmetic, rotations, shifts, oracle vocabulary, and MILP weight conventions are used later by NeoAlzette, XCR, the Key Switch Layer, the Shadow arithmetic material, and the standalone ARX matrix analysis. This prevents the later component sections from silently changing notation or probability spaces.

This section fixes the mathematical meaning of every primitive operation used later in the paper. The definitions are deliberately explicit because the implementation mixes arithmetic over $\mathbb{Z}/2^{32}\mathbb{Z}$, Boolean operations over $\mathbb{F}_2^{32}$, rotations, and zero-filling shifts. These are not interchangeable operations in ARX analysis.

6 Word Space and Bit Numbering

Definition 4 (32-bit word model and little-endian coordinates)**.** Let

$$\mathbb{W}_{32} = \mathbb{Z}/2^{32}\mathbb{Z}.$$

Every word $x \in \mathbb{W}_{32}$ is identified with its canonical integer representative $\mathrm{val}(x) \in \{0, \ldots, 2^{32} - 1\}$ and with its little-endian bit vector

$$x = (x_0, x_1, \ldots, x_{31}) \in \mathbb{F}_2^{32}, \qquad \mathrm{val}(x) = \sum_{i=0}^{31} x_i 2^i.$$

Bit index 0 is therefore the least significant bit. All algorithms in this paper are read over 32-bit words unless a packed 64-bit joint value is explicitly named.

7 Bit and Byte Accessor Functions

Definition 5 (Bit read/write functions)**.** For an n-bit word $X \in \mathbb{Z}/2^n\mathbb{Z}$ and an index $0 \le i < n$, the bit-read function is

$$\mathrm{ReadBit}(X, i) \;=\; \left\lfloor \frac{\mathrm{val}(X)}{2^i} \right\rfloor \bmod 2.$$

Thus $\mathrm{ReadBit}(X, i) = X_i$ under the little-endian convention of Definition 4. The bit-write function replaces exactly one coordinate and leaves all other coordinates unchanged:

$$\mathrm{val}(\mathrm{WriteBit}(X, i, b)) = \mathrm{val}(X) - \mathrm{ReadBit}(X, i)2^i + b2^i, \qquad b \in \{0, 1\}.$$

Equivalently, if $Y = \mathrm{WriteBit}(X, i, b)$, then

$$\mathrm{ReadBit}(Y, i) = b, \qquad \mathrm{ReadBit}(Y, j) = \mathrm{ReadBit}(X, j) \quad (j \neq i).$$

Definition 6 (Byte read/write functions)**.** For an n-bit word X, where $8 \mid n$, and a byte index $0 \leq j < n/8$, the byte-read function returns the little-endian byte

$$\mathrm{ReadByte}(X, j) \;=\; \sum_{t=0}^{7} \mathrm{ReadBit}(X, 8j + t)2^t \in \mathbb{Z}/2^8\mathbb{Z}.$$

For an 8-bit value B, the byte-write function replaces byte j by B and leaves the remaining bytes unchanged. Define $X^{(0)} = X$ and, for $0 \leq t < 8$,

$$X^{(t+1)} = \mathrm{WriteBit}(X^{(t)}, 8j + t, \mathrm{ReadBit}(B, t)).$$

Then

$$\mathrm{WriteByte}(X, j, B) = X^{(8)}.$$

The notation $X[i]$ or $X[j]$ is reserved for genuine arrays or tables. When the object is a word and the manuscript needs a bit or a byte, the pseudocode must use $\mathrm{ReadBit}(X, i)$, $\mathrm{WriteBit}(X, i, b)$, $\mathrm{ReadByte}(X, j)$, or $\mathrm{WriteByte}(X, j, B)$. This convention prevents bit and byte extraction from being misread as C++ array indexing.

8 Boolean Word Operations

Definition 7 (Boolean word operations over $\mathbb{F}_2$)**.** For $x, y \in \mathbb{W}_{32}$, bitwise XOR, AND, and OR are the word maps induced by the corresponding operations in $\mathbb{F}_2$:

$$(x \oplus y)_i = x_i + y_i \pmod 2, \qquad (x \wedge y)_i = x_i y_i, \qquad (x \vee y)_i = x_i + y_i + x_i y_i \pmod 2.$$

The word complement is width-limited:

$$(\neg x)_i = 1 + x_i \pmod 2, \qquad \neg x = x \oplus (2^{32} - 1).$$

Thus $\neg$ always complements exactly 32 bits. This convention matches the C++ type `std::uint32_t` used by the Core.

Equivalently, at the scalar bit level all Boolean operations are polynomial functions over the finite field $\mathbb{F}_2 = \{0, 1\}$. For $a, b \in \mathbb{F}_2$,

$$a \oplus b = a + b, \qquad a \wedge b = ab, \qquad a \vee b = a + b + ab, \qquad \neg a = 1 + a,$$

where both addition and multiplication are the field operations of $\mathbb{F}_2$. The OR identity follows from the Boolean-ring relation

$$a \vee b = 1 + (1 + a)(1 + b) = a + b + ab.$$

Hence XOR is linear over $\mathbb{F}_2$, NOT is affine over $\mathbb{F}_2$, and AND/OR are quadratic Boolean maps over $\mathbb{F}_2$. In particular, OR is not a new algebraic primitive beyond field addition and multiplication; it is the quadratic polynomial $a + b + ab$. For word operations, these scalar polynomials are applied independently to every coordinate i, so bitwise AND and OR have algebraic degree two as maps $\mathbb{F}_2^{64} \to \mathbb{F}_2^{32}$, while bitwise NOT has algebraic degree one as an affine map $\mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$.

9 Modular Addition and Subtraction

Definition 8 (Modular addition and subtraction recurrences)**.** For $x, y \in \mathbb{W}_{32}$, modular addition and subtraction are

$$x + y = (\mathrm{val}(x) + \mathrm{val}(y)) \bmod 2^{32}, \qquad x - y = (\mathrm{val}(x) - \mathrm{val}(y)) \bmod 2^{32}.$$

The bit-level meaning is fixed by the carry and borrow chains. For $z = x + y \bmod 2^{32}$, set $c_0 = 0$ and, for $0 \le i < 32$,

$$\begin{aligned} z_i &= x_i \oplus y_i \oplus c_i, \\ c_{i+1} &= (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i). \end{aligned}$$

The outgoing carry c_{32} is discarded by the reduction modulo 2^{32}. Thus the i-th output bit is not a function only of (x_i, y_i); it also depends on the complete lower-bit carry history generated by positions $0, \ldots, i-1$.

For $z = x - y \bmod 2^{32}$, set $b_0 = 0$ and, for $0 \le i < 32$,

$$\begin{aligned} z_i &= x_i \oplus y_i \oplus b_i, \\ b_{i+1} &= ((\neg x_i) \wedge y_i) \vee ((\neg x_i) \wedge b_i) \vee (y_i \wedge b_i). \end{aligned}$$

Equivalently, subtraction may be expressed as

$$x - y = x + (\neg y) + 1 \pmod{2^{32}},$$

but the paper keeps the borrow-chain notation when the value-domain operation is written as a subtraction in the Core. The outgoing borrow b_{32} is discarded by the same word-size convention as the outgoing carry.

When one operand is a public constant K, the same recurrences are used with $y_i = K_i$ fixed. This specialization is still not a linear layer: constant bits select different carry or borrow transitions, and therefore change exact differential probabilities and exact linear correlations. The MILP backends in Part II model fixed-public add/sub operations with dedicated constant-addend operators for precisely this reason. Modular addition and subtraction are therefore the main arithmetic source of differential and linear weight in the round, together with the nonlinear Boolean injection maps.

10 Rotations

Definition 9 (32-bit rotations)**.** For any integer r, the implementation reduces r modulo 32. The left and right rotations are permutations of bit positions:

$$(\mathrm{x} \lll^{\mathrm{r}})_i = x_{(i-r) \bmod 32}, \qquad (\mathrm{x} \ggg_{\mathrm{r}})_i = x_{(i+r) \bmod 32}.$$

Equivalently, in C++ word syntax,

$$\mathrm{x} \lll^{\mathrm{r}} = ((x \ll r) \vee (x \gg (32 - r))) \bmod 2^{32}$$

after reducing r modulo 32, with the special case $r = 0$ returning x. The rotr definition is analogous with the directions exchanged. The shifts in this expression are the zero-filling unsigned word shifts defined below, not arithmetic signed shifts. For $0 < r < 32$, the same rotation can be written directly in the integer representative as

$$\begin{aligned} \mathrm{val}(\mathrm{x} \lll^{\mathrm{r}}) &= \big(2^r \mathrm{val}(x) \bmod 2^{32}\big) + \left\lfloor \frac{\mathrm{val}(x)}{2^{32-r}} \right\rfloor, \\ \mathrm{val}(\mathrm{x} \ggg_{\mathrm{r}}) &= \left\lfloor \frac{\mathrm{val}(x)}{2^r} \right\rfloor + 2^{32-r} \big(\mathrm{val}(x) \bmod 2^r\big). \end{aligned}$$

The first term keeps the shifted-in-place bits, and the second term wraps the discarded end bits to the other side. Rotations are linear over $\mathbb{F}_2^{32}$ and preserve Hamming weight.

11 Zero-Filling Shifts

Definition 10 (Zero-filling unsigned shifts)**.** The implementation also uses zero-filling shifts in injection pre-mix expressions. These are logical unsigned shifts on 32-bit words. They are defined for $0 \le s < 32$; shifts by the word size or more are outside the primitive operation used in this paper and are not needed by the implemented Core.

In the integer representative, the left shift is multiplication by 2^s followed by the 32-bit word reduction:

$$\mathrm{val}(x \ll s) = 2^s \mathrm{val}(x) \bmod 2^{32} = \sum_{i=0}^{31-s} x_i 2^{i+s}.$$

When $s > 0$, the bits $x_{32-s}, \ldots, x_{31}$ are the terms that would land at powers $2^{32}, \ldots, 2^{31+s}$; they are discarded by the reduction modulo 2^{32}. For $s = 0$ no bit is discarded. Equivalently,

$$(x \ll s)_i = \begin{cases} x_{i-s}, & i \ge s, \\ 0, & i < s, \end{cases}$$

so the low s output bits are zero-filled.

The right shift is integer division by 2^s with the fractional part discarded:

$$\mathrm{val}(x \gg s) = \left\lfloor \frac{\mathrm{val}(x)}{2^s} \right\rfloor = \sum_{i=s}^{31} x_i 2^{i-s}.$$

The floor is required because $\mathrm{val}(x)/2^s$ need not be an integer. The fractional part is exactly the contribution of the discarded low bits $x_0, \ldots, x_{s-1}$:

$$\frac{\mathrm{val}(x)}{2^s} = \sum_{i=s}^{31} x_i 2^{i-s} + \sum_{i=0}^{s-1} x_i 2^{i-s}.$$

The second sum lies in $[0, 1)$, and taking the floor removes it. This is the mathematical form of C++ unsigned right shift: it drops the low s bits and does not propagate a sign bit. Equivalently,

$$(x \gg s)_i = \begin{cases} x_{i+s}, & i + s < 32, \\ 0, & i + s \ge 32. \end{cases}$$

Thus the high s output bits are zero-filled.

These shifts are not rotations. In particular, $B \ll 9$ discards the top nine bits and injects zeros into the low nine positions, while $A \gg 9$ discards the low nine bits and injects zeros into the high nine positions. The distinction matters in the B-to-A and A-to-B injection bridge-state terms.

12 State and Joint Injection Values

Definition 11 (State and local joint-injection value)**.** The full box state is a pair $(A, B) \in \mathbb{W}_{32}^2$. Each local injection function returns a pair $(C, D) \in \mathbb{W}_{32}^2$. In the implementation and in the MILP analysis, the two components are used differently: one component is part of a zero-weight XOR/rotation injection into the opposite branch, while a derived word from the pair contributes to a modular-addition operand. When the analysis describes a joint injection output, it packs the two 32-bit components into a 64-bit bookkeeping value only for reporting; the Core itself operates on two 32-bit words.

13 Fixed-Public Arithmetic Relations

Definition 12 (Fixed-public arithmetic maps)**.** For a fixed word constant $K \in \mathbb{Z}/2^n\mathbb{Z}$, define

$$A_{+K}(X) = X \boxplus_n K, \qquad A_{-C}(X) = X - C \pmod{2^n}.$$

The constant K is public and fixed for the local relation. The probability space is therefore over X only, not over a sampled pair (X, K). This convention is used by the fixed-public subtraction sites of NeoAlzette and by every fixed-operand oracle later in the manuscript.

Remark 1 (Fixed-public arithmetic is not an averaged add-box). A fixed-public relation $X \mapsto X \boxplus K$ or $X \mapsto X \boxminus K$ cannot be replaced by the averaged two-variable relation $(X, Y) \mapsto X \boxplus Y$. The averaged relation samples the second operand; the fixed-public relation conditions the carry or borrow chain on the concrete bit pattern of K. The dedicated fixed-public oracles in Part V and Appendix A exist precisely to preserve this probability space.

14 State Packing and Word Decomposition

A LOPC block is written as two 64-bit lanes

$$\text{Block128} = (L_0, L_1),$$

and equivalently as four 32-bit words

$$(w_0, w_1, w_2, w_3), \qquad L_0 = w_0 \| w_1, \quad L_1 = w_2 \| w_3.$$

The diagonal NeoAlzette layer uses the pairs (w_0, w_2) and (w_1, w_3). The notation $\|$ denotes the implementation packing convention; whenever bit-level analysis is required, the little-endian bit coordinate convention from Section 6 is used inside each word.

15 Affine Constants, XOR Injection, and Phase Terms

XOR by a fixed word constant is affine over $\mathbb{F}_2$. In XOR-differential propagation, a fixed XOR constant cancels from both sides of the pair. In linear-correlation propagation, the same constant contributes a deterministic phase or sign term. This distinction is used for NeoAlzette whitening constants, XCR-derived round material, and the Keyed Bit-Position Tweak.

16 Oracle Vocabulary for Differential and Linear Analysis

A *differential oracle* returns feasibility and differential weight for a local relation. A *linear oracle* returns signed Walsh numerator, correlation, or a conservative linear weight for a local mask relation. A *single characteristic* fixes all intermediate local transitions. An *endpoint hull* aggregates all characteristics with the same external endpoint. These terms are kept separate throughout the paper.

MEDCP/MELCC boundary. For wide ARX-box analysis, this manuscript follows the Alzette-style distinction between characteristic bounds and endpoint hull quantities [BBdS+20]. The maximum expected differential characteristic probability (MEDCP) is a single-differential-characteristic quantity. The maximum expected absolute linear characteristic correlation (MELCC) is a single-linear-characteristic quantity. An endpoint

differential hull probability aggregates all differential characteristics with the same external endpoint. An endpoint linear hull correlation aggregates signed linear-characteristic contributions with the same external masks. Consequently, NeoAlzette component searches are described as MEDCP/MELCC-style single-characteristic evidence unless a complete endpoint HULL enumeration is explicitly stated.

17 MILP Variable and Weight Conventions

MILP boxes use binary variables for bits, selector variables for local cases, and weight variables for differential or linear objective contributions. An epigraph variable is used when an exact nonlinear weight, such as $n - \log_2 |N|$, is represented by tangent cuts or an outer approximation. A resource-limited solver incumbent is not an optimality certificate unless the solver status proves optimality. This convention is used for the reported NeoAlzette one-round differential and linear experiments.

18 Code-Faithful Pseudocode Convention

The pseudocode in this paper follows the reference C++ operation order. Names may be mathematical, but data dependencies, word widths, fixed constants, inverse order, and recomputed branch-dependent values follow the implementation. When a later section says that an oracle is code-faithful, it means that the oracle preserves the same deterministic word-assignment graph as the referenced C++ routine.

Part III
Specification

19 Global LOPC State and Interface

The current implementation uses a 128-bit block interface $\mathsf{Block128} = (\mathsf{lane}_0, \mathsf{lane}_1)$, a 128-bit key interface $\mathsf{Key128} = (\mathsf{key}_0, \mathsf{key}_1)$, and a nonce-like public input named `number_once`. Inside the round function the two 64-bit lanes are unpacked as four 32-bit words (w_0, w_1, w_2, w_3), where $\mathsf{lane}_0 = w_0 \| w_1$ and $\mathsf{lane}_1 = w_2 \| w_3$. The number of internal cryptographic rounds is controlled by the `rounds` parameter.

20 Round Skeleton

Algorithm 1 LOPC round skeleton

Require: state lanes (L_0, L_1), key (K_0, K_1), nonce input N, round index r

1: Generate XCR-derived round material: subkey words, choice function, and rotation amounts.
2: XOR the two subkey words into (L_0, L_1).
3: Unpack (L_0, L_1) into four 32-bit words (w_0, w_1, w_2, w_3).
4: Apply diagonal NeoAlzette: NeoAlzette(w_0, w_2) and NeoAlzette(w_1, w_3).
5: Pack the four words back into two 64-bit lanes.
6: Apply `MixLinearTransform_Forward`.
7: Apply RandomBitTweak / Keyed Bit-Position Tweak.

21 Global LOPC Algorithm Specification

The current `LittleOaldresPuzzle_Cryptic` implementation is a 128-bit block transform driven by a persistent two-instance XCR key-schedule generator. The data state consists of two 64-bit lanes, while each round temporarily unpacks the state into four 32-bit words for two diagonal NeoAlzette calls. The implementation should not be described as a generic stream cipher wrapper: its encryption and decryption cores are explicit reversible block transformations, and its XCR objects are consumed only to derive round-dependent subkeys, switching choices, and bit-tweak locations.

21.1 LOPC counter-indexed block processing and external authentication boundary

For round count R, constructor seed s, 128-bit key K, and one-time public input N, the LOPC core is specified as a deterministic block transformation

$$E^{(R)}_{s,K,N} : \{0,1\}^{128} \longrightarrow \{0,1\}^{128}.$$

In the current C++ core, `number_once` is a 64-bit public input to `EncryptionCoreFunction(data,key,number_once)` and `DecryptionCoreFunction(data,key,number_once)`. The multi-block wrapper uses the block index as this public input, giving the default counter-indexed processing rule $N_j = j$ for successive 128-bit blocks. Other callers may use a different injective counter schedule, but for a fixed seed and key the value supplied as `number_once` must be unique for every encrypted block. The same value must be supplied for decryption. Reusing `number_once` under the same seed and key is outside the current claim boundary.

This is a counter-indexed block transformation, not a proof of standard CTR-mode security. In particular, LOPC does not implement authentication internally. For authenticated encryption, the LOPC core must be composed with an external MAC or AEAD wrapper, for example an encrypt-then-MAC construction over associated data, the sequence of `number_once` values, and ciphertext blocks. This manuscript specifies the LOPC core and its counter-indexed API; it does not prove the external MAC/AEAD composition.

21.2 `LittleOaldresPuzzle_Cryptic.h` framework pseudocode

The header is used here as a framework contract, not as an implementation-language listing. It tells the reader which objects persist across a session, which calls are top-level sessions, which routines are private round machinery, and where the XCR generators are reset. In plain engineering terms: the object owns two XCR generators, consumes them while a session is running, and restores them at the end of each top-level call. The mathematical consequence is that a public call is deterministic for fixed inputs, while the round material inside that call is still generated by evolving state.

The basic types are

$$\mathsf{Key128} = \mathbb{Z}_{2^{64}} \times \mathbb{Z}_{2^{64}}, \qquad \mathsf{Block128} = \mathbb{Z}_{2^{64}} \times \mathbb{Z}_{2^{64}}.$$

For a constructor seed $s \in \mathbb{Z}_{2^{64}}$, the two persistent XCR objects are initialized as

$$G_L \leftarrow \mathsf{XCR}(s), \qquad G_R \leftarrow \mathsf{XCR}(\neg s \oplus \mathrm{rotl}_{64}(s, 32)).$$

They are object fields. They are not recreated inside every round.

Algorithm 2 Header-level framework extracted from the LOPC header

1: **type** Key128 $= (u_0, u_1) \in \mathbb{Z}_{2^{64}}^2$
2: **type** Block128 $= (v_0, v_1) \in \mathbb{Z}_{2^{64}}^2$
3: **record** KeyState:
    subkey $\in$ Key128
    choice_function $\in \{0, 1, 2, 3\}$
    bit_rotation_amount_a, bit_rotation_amount_b $\in \{0, \ldots, 63\}$
4: **object fields**:
    constructor seed s, round count r, persistent generators G_L, G_R, array KeyStates$[0 \ldots r-1]$
5: **constructor** Init(s, r):
    set $G_L \leftarrow$ XCR(s)
    set $G_R \leftarrow$ XCR$(\neg s \oplus \mathrm{rotl}_{64}(s, 32))$
    allocate KeyStates of length r
6: **public session** SingleRoundEncryption(M, K, N):
    $C \leftarrow$ EncryptionCoreFunction(M, K, N)
    ResetPRNG(); **return** C
7: **public session** SingleRoundDecryption(C, K, N):
    $M \leftarrow$ DecryptionCoreFunction(C, K, N)
    ResetPRNG(); **return** M
8: **public session** MultipleRoundsEncryption$(M[0..m-1], K[0..\ell-1])$:
9: **for** $j = 0$ to $m - 1$ **do**
10: $C[j] \leftarrow$ EncryptionCoreFunction$(M[j], K[j \bmod \ell], j)$
11: **end for**
12: ResetPRNG(); **return** $C[0..m-1]$
13: **public session** MultipleRoundsDecryption$(C[0..m-1], K[0..\ell-1])$:
14: **for** $j = 0$ to $m - 1$ **do**
15: $M[j] \leftarrow$ DecryptionCoreFunction$(C[j], K[j \bmod \ell], j)$
16: **end for**
17: ResetPRNG(); **return** $M[0..m-1]$
18: **public session** GenerateSubkey_WithUseEncryption(K, L):
    use the seeded secondary generator $\mathsf{MT}_{64}(K_0 \oplus K_1 \oplus L)$ to create nonce blocks
    feed each nonce block through EncryptionCoreFunction, xor-fold the outputs, then reset G_L, G_R
19: **public session** GenerateSubkey_WithUseDecryption(K, L):
    same construction, but using DecryptionCoreFunction
20: **reset routine** ResetPRNG():
    G_L.Seed(s)
    G_R.Seed$(\neg s \oplus \mathrm{rotl}_{64}(s, 32))$
21: **private round machinery**:
    GenerateAndStoreKeyStates(K, N)
    MixLinearTransform_Forward$(L_0, L_1,$ KeyState$)$
    MixLinearTransform_Backward$(L_0, L_1,$ KeyState$)$
    EncryptionCoreFunction(M, K, N), DecryptionCoreFunction(C, K, N)

Definition 13 (LOPC header-level session)**.** A header-level session is one completed invocation of one of the following public routines:

- SingleRoundEncryption or SingleRoundDecryption;
- MultipleRoundsEncryption or MultipleRoundsDecryption;
- GenerateSubkey_WithUseEncryption or GenerateSubkey_WithUseDecryption.

The session begins with the current member states (G_L, G_R), consumes these states through the private round machinery, and ends by calling ResetPRNG.

Lemma 3 (Deterministic public-session boundary)**.** *Fix the constructor seed s, the round count r, the input data, the key material, and the nonce/counter input. Each header-level session is deterministic, and after the session returns the member generators satisfy*

$$G_L = \mathsf{XCR}(s), \qquad G_R = \mathsf{XCR}(\neg s \oplus \mathrm{rotl}_{64}(s, 32)).$$

Inside the session, however, G_L and G_R evolve continuously and are not round-local random oracles.

Proof. The private encryption, decryption, and subkey-generation routines are deterministic functions of their inputs and of the two member XCR states. Every header-level session listed above finishes by invoking ResetPRNG, which reseeds the left generator with s and the right generator with the paired seed $\neg s \oplus \mathrm{rotl}_{64}(s, 32)$. Therefore repeated top-level calls with the same public inputs start from the same pair of XCR states and return the object to the same pair of XCR states. This reset invariant does not imply independence of round material inside a session: during the body of the call, the private key-state generation routine advances the same persistent generators across the round sequence. □

Remark 2 (Why this framework matters)*.* The framework separates three objects that should not be conflated: the public session wrapper, the persistent XCR key-schedule state, and the reversible block-core routines. The security discussion therefore treats the reset rule as an implementation invariant, while the cryptographic proof burden remains on the XCR-generated round material, the NeoAlzette layer, and the key-switch layer used inside the private core.

21.3 Current per-round XCR-derived key-state generation

For each round i, the C++ method `GenerateAndStoreKeyStates` calls both persistent XCR generators exactly once. The left and right public inputs are domain-separated by the round index:

$$input_L = number_once \oplus i, \qquad input_R = (number_once \oplus (i \ll 1)) \oplus (i \gg 1).$$

If

$$G_L(input_L) = (a, b), \qquad G_R(input_R) = (c, d),$$

then the round key state is

$$\begin{aligned} sk_i^{(0)} &= (K_0 + a) \oplus \mathrm{rotr}_{64}(c, i \bmod 64), \\ sk_i^{(1)} &= (K_1 - b) \oplus \mathrm{rotr}_{64}(d, (i+1) \bmod 64), \\ cf_i &= (a \oplus b \oplus c \oplus d) \bmod 4, \\ rpool_i &= (a \oplus b) \oplus (c \oplus d) \oplus \mathrm{rotl}_{64}(sk_i^{(0)}, 1) \oplus \mathrm{rotl}_{64}(sk_i^{(1)}, 3), \\ \alpha_i &= rpool_i \bmod 64, \qquad \beta_i = (rpool_i \gg 6) \bmod 64. \end{aligned}$$

The chosen form performs the modular addition/subtraction with the key word first and only then applies the rotated XOR shell. This keeps the ARX key-mixing boundary separate from the XCR-derived material. The rejected alternative, namely mixing XCR words before the carry/borrow chain, would entangle the XCR structure with the modular core and would require a broader key-schedule model.

Algorithm 3 Current code-faithful LOPC round-key-state generation

Require: key (K_0, K_1), public *number_once*, persistent XCR objects G_L, G_R, round count r

for $i = 0$ to $r - 1$ **do**
  $input_L \leftarrow number_once \oplus i$
  $input_R \leftarrow (number_once \oplus (i \ll 1)) \oplus (i \gg 1)$
  $(a, b) \leftarrow G_L.\textsc{GenerateSubKey128}(input_L)$
  $(c, d) \leftarrow G_R.\textsc{GenerateSubKey128}(input_R)$
  $sk_i^{(0)} \leftarrow (K_0 \boxplus_{64} a) \oplus \mathrm{rotr}_{64}(c, i \bmod 64)$
  $sk_i^{(1)} \leftarrow (K_1 \boxminus_{64} b) \oplus \mathrm{rotr}_{64}(d, (i+1) \bmod 64)$
  $cf_i \leftarrow (a \oplus b \oplus c \oplus d)\ \&\ 3$
  $rpool \leftarrow (a \oplus b) \oplus (c \oplus d) \oplus \mathrm{rotl}_{64}(sk_i^{(0)}, 1) \oplus \mathrm{rotl}_{64}(sk_i^{(1)}, 3)$
  $\alpha_i \leftarrow rpool\ \&\ 63; \quad \beta_i \leftarrow (rpool \gg 6)\ \&\ 63$
end for
return $\{(sk_i, cf_i, \alpha_i, \beta_i)\}_{i=0}^{r-1}$

21.4 Symmetric encryption specification

Let a 128-bit block be (L_0, L_1), where each lane is a 64-bit word. A round first xors the two subkey lanes, then unpacks

$$L_0 = (w_0 \parallel w_1), \qquad L_1 = (w_2 \parallel w_3),$$

where each w_j is 32 bits. NeoAlzette is applied diagonally to (w_0, w_2) and (w_1, w_3), after which the result is repacked into two lanes. The keyed switching layer then applies one of four reversible operations selected by cf_i, and a final single-bit tweak toggles one bit in each 64-bit lane.

Algorithm 4 Current code-faithful LOPC encryption core

Require: block (L_0, L_1), key (K_0, K_1), public $number_once$, round count r
Generate key states by Algorithm 3
for $i = 0$ to $r - 1$ **do**
  $(L_0, L_1) \leftarrow (L_0 \oplus sk_i^{(0)},\ L_1 \oplus sk_i^{(1)})$
  unpack $L_0 = (w_0 \parallel w_1)$ and $L_1 = (w_2 \parallel w_3)$
  $(w_0, w_2) \leftarrow \mathrm{NeoAlzette}(w_0, w_2)$
  $(w_1, w_3) \leftarrow \mathrm{NeoAlzette}(w_1, w_3)$
  repack $L_0 = (w_0 \parallel w_1)$ and $L_1 = (w_2 \parallel w_3)$
  $(L_0, L_1) \leftarrow \mathrm{SwitchForward}_{cf_i, sk_i, \beta_i}(L_0, L_1)$
  $L_0 \leftarrow L_0 \oplus (1 \ll \alpha_i)$
  $L_1 \leftarrow L_1 \oplus (1 \ll (63 - \alpha_i))$
end for
return (L_0, L_1)

21.5 Symmetric decryption specification

Decryption uses the same generated key-state sequence and walks the rounds in reverse. The bit tweak is self-inverse, the keyed switching layer uses the inverse branch, NeoAlzette backward is applied to the same diagonal pairs, and the subkey xors are undone last.

Algorithm 5 Current code-faithful LOPC decryption core

Require: block (L_0, L_1), key (K_0, K_1), public $number_once$, round count r
Generate the same key states by Algorithm 3
for $i = r - 1$ down to 0 **do**
  $L_0 \leftarrow L_0 \oplus (1 \ll \alpha_i)$
  $L_1 \leftarrow L_1 \oplus (1 \ll (63 - \alpha_i))$
  $(L_0, L_1) \leftarrow \mathrm{SwitchBackward}_{cf_i, sk_i, \beta_i}(L_0, L_1)$

6: unpack $L_0 = (w_0 \parallel w_1)$ and $L_1 = (w_2 \parallel w_3)$
7: $(w_1, w_3) \leftarrow \text{NeoAlzette}^{-1}(w_1, w_3)$
8: $(w_0, w_2) \leftarrow \text{NeoAlzette}^{-1}(w_0, w_2)$
9: repack $L_0 = (w_0 \parallel w_1)$ and $L_1 = (w_2 \parallel w_3)$
10: $(L_0, L_1) \leftarrow (L_0 \oplus sk_i^{(0)},\ L_1 \oplus sk_i^{(1)})$
11: **end for**
12: **return** (L_0, L_1)

Lemma 4 (Round invertibility)**.** *For a fixed generated key state, one LOPC encryption round is a permutation of* $(\mathbb{Z}_{2^{64}})^2$*, and the decryption round above is its inverse.*

Proof. The subkey xor is self-inverse. Unpack and repack are bit reinterpretations. Both diagonal NeoAlzette calls are permutations by the NeoAlzette inverse proof. The switching layer branches are individually reversible: xor by subkey is self-inverse, complement+xor is self-inverse, and the rotation branches invert by rotating in the opposite direction. The single-bit tweak is xor by a unit word and is self-inverse. Applying these inverses in reverse order gives Algorithm 5. □

22 Component A Specification: NeoAlzette

This section is the code-faithful executable specification of the NeoAlzette component used by LOPC. The high-level diagonal call in the LOPC round skeleton is only an orientation. The normative component specification is stated here directly: the constants, diffusion masks, C/D injection equations, routed addends, and forward/backward schedules have been extracted from the current implementation and rewritten as mathematical component definitions.

22.1 Code-derived component boundary

Definition 14 (NeoAlzette component boundary)**.** NeoAlzette acts on a pair of 32-bit words

$$(A, B) \in \mathbb{W}_{32}^2,$$

using the word, bit-coordinate, Boolean-operation, rotation, zero-filling shift, and modular-arithmetic conventions of Part II. It is the Component A ARX-box inside the LOPC round; it is not the whole LOPC construction, not the XCR generator, and not the KeySwitchLayer.

Remark 3 (Specification source)*.* The compact diagonal call in the LOPC round skeleton is only an orientation. The component specification below is extracted from the current C++ forward/backward implementation: constants, diffusion masks, the two C/D injection circuits, CD-derived addends, and the exact forward/backward schedule are stated here as mathematical objects. Common primitive operations are not redefined here.

22.2 V6.5 implementation oracle

Definition 15 (V6.5 implementation oracle)**.** The code-faithful NeoAlzette object analyzed in this paper is

$$\mathcal{O}_{\mathrm{NA}}^{6.5} = \big(RC, \mathsf{DM}_0^{\mathrm{NA}}, \mathsf{DM}_1^{\mathrm{NA}}, \mathsf{Inj}_{B\to A}^{\mathrm{NA}}, \mathsf{Inj}_{A\to B}^{\mathrm{NA}}, \mathsf{Sched}_{\mathrm{fwd}}^{\mathrm{NA}}, \mathsf{Sched}_{\mathrm{bwd}}^{\mathrm{NA}}\big).$$

Here RC is the public constant array, $\mathsf{DM}_0^{\mathrm{NA}}$ and $\mathsf{DM}_1^{\mathrm{NA}}$ are the two diffusion-mask maps, $\mathsf{Inj}_{B\to A}^{\mathrm{NA}}$ and $\mathsf{Inj}_{A\to B}^{\mathrm{NA}}$ are the oriented C/D injection maps, and $\mathsf{Sched}_{\mathrm{fwd}}^{\mathrm{NA}}$, $\mathsf{Sched}_{\mathrm{bwd}}^{\mathrm{NA}}$ are the forward and backward schedules.

22.3 Public constants and diffusion masks

Definition 16 (Public constant array)**.** The component uses a 16-word public constant array $RC[0], \ldots, RC[15] \in \mathbb{W}_{32}$. The current schedule uses $RC[0]$, $RC[1]$, $RC[4]$, $RC[5]$, $RC[6]$, $RC[9]$, $RC[10]$, and $RC[11]$ directly; the injection circuits also use rotated views of $RC[2]$, $RC[3]$, $RC[7]$, $RC[8]$, $RC[12]$, and $RC[13]$.

Index	Hex word	source comment
0	`0x16B2C40B`	Fibonacci-derived prefix
1	`0xC117176A`	Fibonacci-derived prefix
2	`0x0F9A2598`	Fibonacci-derived prefix
3	`0xA1563ACA`	Fibonacci-derived prefix
4	`0x243F6A88`	hexadecimal expansion of π
5	`0x85A308D3`	hexadecimal expansion of π
6	`0x13198A2E`	hexadecimal expansion of π
7	`0x03707344`	hexadecimal expansion of π
8	`0x9E3779B9`	hexadecimal expansion of the golden ratio
9	`0x7F4A7C15`	hexadecimal expansion of the golden ratio
10	`0xF39CC060`	hexadecimal expansion of the golden ratio
11	`0x5CEDC834`	hexadecimal expansion of the golden ratio
12	`0xB7E15162`	hexadecimal expansion of e
13	`0x8AED2A6A`	hexadecimal expansion of e
14	`0xBF715880`	hexadecimal expansion of e
15	`0x9CF4F3C7`	hexadecimal expansion of e

Definition 17 (Dynamic diffusion masks)**.** For $x \in \mathbb{W}_{32}$, define $\mathsf{DM}_0^{\mathrm{NA}}(x)$ by

$$\begin{aligned} v_0 &= x, \\ v_1 &= v_0 \oplus \mathrm{v}_0 \lll^{2}, \\ v_2 &= v_0 \oplus \mathrm{v}_1 \lll^{17}, \\ v_3 &= v_0 \oplus \mathrm{v}_2 \lll^{4}, \\ v_4 &= v_3 \oplus \mathrm{v}_3 \lll^{24}, \\ \mathsf{DM}_0^{\mathrm{NA}}(x) &= v_2 \oplus \mathrm{v}_4 \lll^{7}. \end{aligned}$$

Similarly, define $\mathsf{DM}_1^{\mathrm{NA}}(x)$ by

$$\begin{aligned} u_0 &= x, \\ u_1 &= u_0 \oplus \mathrm{u}_0 \ggg_{2}, \\ u_2 &= u_0 \oplus \mathrm{u}_1 \ggg_{17}, \\ u_3 &= u_0 \oplus \mathrm{u}_2 \ggg_{4}, \\ u_4 &= u_3 \oplus \mathrm{u}_3 \ggg_{24}, \\ \mathsf{DM}_1^{\mathrm{NA}}(x) &= u_2 \oplus \mathrm{u}_4 \ggg_{7}. \end{aligned}$$

Both maps are linear over $\mathbb{F}_2$. They feed the C/D injection circuits and therefore influence both the routed XOR injection and the later CD-derived modular-addition operands.

Definition 18 (Rotated constant views used by the injectors)**.** The B-to-A injector uses

$$\kappa_7^B = \mathrm{RC}[7] \ggg_{24}, \qquad \kappa_8^B = \mathrm{RC}[8] \ggg_{24}, \qquad \kappa_{13}^B = \mathrm{RC}[13] \ggg_{24},$$

and the A-to-B injector uses

$$\kappa_2^A = \mathrm{RC}[2] \lll^{8}, \qquad \kappa_3^A = \mathrm{RC}[3] \lll^{8}, \qquad \kappa_{12}^A = \mathrm{RC}[12] \lll^{8}.$$

The two mask constants are

$$\mu_7^B = \mathsf{DM}_0^{\mathrm{NA}}(RC[7]), \qquad \mu_2^A = \mathsf{DM}_1^{\mathrm{NA}}(RC[2]).$$

These are public derived constants, not additional secret material.

22.4 C/D injection equations

Definition 19 (B-to-A C/D injection)**.** For a control word $B \in \mathbb{W}_{32}$, let

$$\begin{aligned}
\rho_B &= \mathrm{B} \ggg_{24}, & m_B &= \mathsf{DM}_0^{\mathrm{NA}}(B),\\
\widehat{m}_B &= \mathrm{m_B} \ggg_{24} \oplus \mu_7^B, & m'_B &= \mathrm{m_B} \ggg_5,\\
\xi_B &= \rho_B \oplus m_B, & \eta_B &= B \oplus m_B,\\
\upsilon_B &= \rho_B \oplus \widehat{m}_B, & \tau_B &= \mathrm{B} \ggg_{19} \oplus (B \ll 9).
\end{aligned}$$

Define the three quadratic carrier views and their shared part by

$$\begin{aligned}
q^B_{\mathrm{state}} &= \kappa_7^B \oplus \neg(B \wedge m_B),\\
q^B_{\mathrm{comp}} &= \rho_B \oplus B \oplus \kappa_8^B \oplus \neg(\rho_B \vee m'_B),\\
q^B_{\mathrm{bridge}} &= \tau_B \oplus B \oplus \kappa_{13}^B \oplus \neg(\tau_B \wedge \widehat{m}_B),\\
q^B_{\mathrm{shared}} &= q^B_{\mathrm{state}} \oplus q^B_{\mathrm{comp}}.
\end{aligned}$$

The cross and anti-coupling words are

$$\begin{aligned}
\chi_B &= (B \oplus m'_B) \wedge \mathrm{m_B} \oplus \widehat{\mathrm{m}}_{\mathrm{B}} \ggg_7,\\
\alpha_B &= ((\eta_B \gg 3) \oplus (\upsilon_B \gg 5) \oplus m'_B) \wedge (B \oplus \xi_{\mathrm{B}} \ggg_{11}).
\end{aligned}$$

Then

$$\begin{aligned}
C^{B\to A} &= q^B_{\mathrm{shared}} \oplus \mathrm{q}^{\mathrm{B}}_{\mathrm{comp}} \ggg_5 \oplus \mathrm{q}^{\mathrm{B}}_{\mathrm{comp}} \ggg_{11} \oplus \alpha_B,\\
D^{B\to A} &= q^B_{\mathrm{shared}} \oplus \mathrm{q}^{\mathrm{B}}_{\mathrm{state}} \ggg_5 \oplus \mathrm{q}^{\mathrm{B}}_{\mathrm{bridge}} \ggg_{13} \oplus \chi_B \oplus \alpha_B.
\end{aligned}$$

We write $(C^{B\to A}, D^{B\to A}) = \mathsf{Inj}^{\mathrm{NA}}_{B\to A}(B)$.

Definition 20 (A-to-B C/D injection)**.** For a control word $A \in \mathbb{W}_{32}$, let

$$\begin{aligned}
\rho_A &= \mathrm{A} \lll^8, & m_A &= \mathsf{DM}_1^{\mathrm{NA}}(A),\\
\widehat{m}_A &= \mathrm{m_A} \lll^8 \oplus \mu_2^A, & m'_A &= \mathrm{m_A} \ggg_5,\\
\xi_A &= \rho_A \oplus m_A, & \eta_A &= A \oplus m_A,\\
\upsilon_A &= \rho_A \oplus \widehat{m}_A, & \tau_A &= \mathrm{A} \lll^{19} \oplus (A \gg 9).
\end{aligned}$$

Define

$$\begin{aligned}
q^A_{\mathrm{state}} &= \kappa_2^A \oplus \neg(A \vee m_A),\\
q^A_{\mathrm{comp}} &= \rho_A \oplus A \oplus \kappa_3^A \oplus \neg(\rho_A \wedge m'_A),\\
q^A_{\mathrm{bridge}} &= \tau_A \oplus A \oplus \kappa_{12}^A \oplus \neg(\tau_A \vee \widehat{m}_A),\\
q^A_{\mathrm{shared}} &= q^A_{\mathrm{state}} \oplus q^A_{\mathrm{comp}}.
\end{aligned}$$

The cross and anti-coupling words are

$$\begin{aligned}
\chi_A &= (A \oplus m'_A) \wedge \mathrm{m_A} \oplus \widehat{\mathrm{m}}_{\mathrm{A}} \lll^{13},\\
\alpha_A &= ((\eta_A \ll 3) \oplus (\upsilon_A \ll 5) \oplus m'_A) \vee (A \oplus \xi_{\mathrm{A}} \lll^{11}).
\end{aligned}$$

Then

$$\begin{aligned}
C^{A\to B} &= q^A_{\mathrm{shared}} \oplus \mathrm{q}^{\mathrm{A}}_{\mathrm{comp}} \lll^5 \oplus \mathrm{q}^{\mathrm{A}}_{\mathrm{comp}} \lll^{11} \oplus \alpha_A,\\
D^{A\to B} &= q^A_{\mathrm{shared}} \oplus \mathrm{q}^{\mathrm{A}}_{\mathrm{state}} \lll^5 \oplus \mathrm{q}^{\mathrm{A}}_{\mathrm{bridge}} \lll^{13} \oplus \chi_A \oplus \alpha_A.
\end{aligned}$$

We write $(C^{A\to B}, D^{A\to B}) = \mathsf{Inj}^{\mathrm{NA}}_{A\to B}(A)$.

22.5 Routed XOR and CD-derived addend interfaces

Definition 21 (B-to-A routed outputs)**.** Given $(C^{B\to A}, D^{B\to A}) = \mathsf{Inj}^{\mathrm{NA}}_{B\to A}(B)$, define

$$\begin{aligned}
\mathsf{Xor}^{B\to A}_{\mathrm{NA}}(B) &= \mathrm{B} \lll^{24} \oplus \mathrm{C}^{\mathrm{B\to A}} \lll^{16} \oplus \mathrm{B} \lll^8,\\
\mathsf{CD}^{B\to A}_0(B) &= (C^{B\to A} \ll 2) \oplus (D^{B\to A} \gg 2),\\
\mathsf{CD}^{B\to A}_1(B) &= (C^{B\to A} \gg 5) \oplus (D^{B\to A} \ll 5),\\
\mathsf{Add}^{B\to A}_{\mathrm{NA}}(B) &= \mathsf{CD}^{\mathrm{B\to A}}_0(\mathrm{B}) \lll^{31} \oplus \mathsf{CD}^{\mathrm{B\to A}}_1(\mathrm{B}) \lll^{17} \oplus RC[0].
\end{aligned}$$

Definition 22 (A-to-B routed outputs)**.** Given $(C^{A\to B}, D^{A\to B}) = \mathsf{Inj}^{\mathrm{NA}}_{A\to B}(A)$, define

$$\begin{aligned}
\mathsf{Xor}^{A\to B}_{\mathrm{NA}}(A) &= \mathrm{A} \ggg_{24} \oplus \mathrm{D}^{\mathrm{A\to B}} \ggg_{16} \oplus \mathrm{A} \ggg_{8},\\
\mathsf{CD}^{A\to B}_{0}(A) &= (C^{A\to B} \gg 3) \oplus (D^{A\to B} \ll 3),\\
\mathsf{CD}^{A\to B}_{1}(A) &= (C^{A\to B} \ll 1) \oplus (D^{A\to B} \gg 1),\\
\mathsf{Add}^{A\to B}_{\mathrm{NA}}(A) &= \mathsf{CD}^{A\to B}_{0}(A) \oplus \mathsf{CD}^{A\to B}_{1}(A) \oplus RC[5].
\end{aligned}$$

22.6 Forward schedule

Definition 23 (V6.5 forward schedule)**.** The forward schedule $\mathsf{Sched}^{\mathrm{NA}}_{\mathrm{fwd}}$ maps (A_0, B_0) to (A_6, B_6) by the ordered assignments

$$\begin{array}{rcll}
B_1 &=& B_0 \boxminus RC[1], & \text{fixed-public subtraction},\\
(C_0, D_0) &=& \mathsf{Inj}^{\mathrm{NA}}_{B\to A}(B_1), & \text{B-to-A C/D injection},\\
A_1 &=& A_0 \oplus \mathsf{Xor}^{B\to A}_{\mathrm{NA}}(B_1), & \text{routed XOR injection},\\
A_2 &=& A_1 \boxplus \mathsf{Add}^{B\to A}_{\mathrm{NA}}(B_1), & \text{CD-derived variable-addend addition},\\
B_2 &=& B_1 \oplus \mathrm{A}_2 \lll^{22} \oplus RC[4], & \text{first bridge line},\\
A_3 &=& A_2 \oplus \mathrm{B}_2 \lll^{13}, & \text{second bridge line},\\
A_4 &=& A_3 \boxminus RC[6], & \text{fixed-public subtraction},\\
(C_1, D_1) &=& \mathsf{Inj}^{\mathrm{NA}}_{A\to B}(A_4), & \text{A-to-B C/D injection},\\
B_3 &=& B_2 \oplus \mathsf{Xor}^{A\to B}_{\mathrm{NA}}(A_4), & \text{routed XOR injection},\\
B_4 &=& B_3 \boxplus \mathsf{Add}^{A\to B}_{\mathrm{NA}}(A_4), & \text{CD-derived variable-addend addition},\\
A_5 &=& A_4 \oplus \mathrm{B}_4 \lll^{5} \oplus RC[9], & \text{final A bridge},\\
B_5 &=& B_4 \oplus \mathrm{A}_5 \lll^{25}, & \text{final B bridge},\\
A_6 &=& A_5 \oplus RC[10], & \text{whitening},\\
B_6 &=& B_5 \oplus RC[11]. & \text{whitening}
\end{array}$$

The output is $\mathsf{Sched}^{\mathrm{NA}}_{\mathrm{fwd}}(A_0, B_0) = (A_6, B_6)$.

Algorithm 6 NeoAlzette V6.5 forward schedule

Require: two 32-bit words (A, B)

$B \leftarrow B \boxminus RC[1]$
$(C, D) \leftarrow \mathsf{Inj}^{\mathrm{NA}}_{B\to A}(B)$
$A \leftarrow A \oplus \mathsf{Xor}^{B\to A}_{\mathrm{NA}}(B)$
$A \leftarrow A \boxplus \mathsf{Add}^{B\to A}_{\mathrm{NA}}(B)$
$B \leftarrow B \oplus \mathrm{A} \lll^{22} \oplus RC[4]$
$A \leftarrow A \oplus \mathrm{B} \lll^{13}$
$A \leftarrow A \boxminus RC[6]$
$(C, D) \leftarrow \mathsf{Inj}^{\mathrm{NA}}_{A\to B}(A)$
$B \leftarrow B \oplus \mathsf{Xor}^{A\to B}_{\mathrm{NA}}(A)$
$B \leftarrow B \boxplus \mathsf{Add}^{A\to B}_{\mathrm{NA}}(A)$
$A \leftarrow A \oplus \mathrm{B} \lll^{5} \oplus RC[9]$
$B \leftarrow B \oplus \mathrm{A} \lll^{25}$
$A \leftarrow A \oplus RC[10]$; $B \leftarrow B \oplus RC[11]$
return (A, B)

22.7 Backward schedule and inversion

Algorithm 7 NeoAlzette V6.5 backward schedule

Require: two 32-bit words (A, B)

$B \leftarrow B \oplus RC[11]$; $A \leftarrow A \oplus RC[10]$

2: $B \leftarrow B \oplus \mathrm{A} \lll^{25}$
3: $A \leftarrow A \oplus \mathrm{B} \lll^{5} \oplus RC[9]$
4: $(C, D) \leftarrow \mathsf{Inj}^{\mathrm{NA}}_{A \to B}(A)$
5: $B \leftarrow B \boxminus \mathsf{Add}^{A \to B}_{\mathrm{NA}}(A)$
6: $B \leftarrow B \oplus \mathsf{Xor}^{A \to B}_{\mathrm{NA}}(A)$
7: $A \leftarrow A \boxplus RC[6]$
8: $A \leftarrow A \oplus \mathrm{B} \lll^{13}$
9: $B \leftarrow B \oplus \mathrm{A} \lll^{22} \oplus RC[4]$
10: $(C, D) \leftarrow \mathsf{Inj}^{\mathrm{NA}}_{B \to A}(B)$
11: $A \leftarrow A \boxminus \mathsf{Add}^{B \to A}_{\mathrm{NA}}(B)$
12: $A \leftarrow A \oplus \mathsf{Xor}^{B \to A}_{\mathrm{NA}}(B)$
13: $B \leftarrow B \boxplus RC[1]$
14: **return** (A, B)

Proposition 1 (NeoAlzette V6.5 inversion)**.** *For fixed public constants,*

$$\mathsf{Sched}^{\mathrm{NA}}_{\mathrm{bwd}} \circ \mathsf{Sched}^{\mathrm{NA}}_{\mathrm{fwd}} = \mathrm{id}_{\mathbb{W}^2_{32}}, \qquad \mathsf{Sched}^{\mathrm{NA}}_{\mathrm{fwd}} \circ \mathsf{Sched}^{\mathrm{NA}}_{\mathrm{bwd}} = \mathrm{id}_{\mathbb{W}^2_{32}}.$$

Proof. Whitening is XOR by a public word. The bridge and routed-injection assignments are triangular XOR updates and are undone once their source branch is restored. The CD-derived additions are undone by modular subtraction after the same branch-dependent addend has been recomputed. Fixed-public subtractions are undone by fixed-public additions. The backward schedule applies these inverse assignments in reverse order. □

Remark 4 (Implementation facts absorbed into the specification)*.* The earlier engineering appendix has been removed. Constants, masks, injection equations, routed addends, and forward/backward schedules are now stated in the component specification itself. Engineering file names and C++ temporary names are not used as mathematical references.

23 Component B Specification: XCR CSPRNG

In this section, XCR means *XorConstantRotation*, and CSPRNG means *cryptographically secure pseudorandom number generator*. The section is now written in the order required for later solver work. We first expose the public schedule, including `COUNTER_STEP`; then we define the code-faithful evaluation relation and the analysis-oracle layer; then we attach the Shadow S-round and Hybrid8 H-round oracle families; only after that do we discuss MILP/SAT/SMT as future exports of the oracle relation. The last part states the real-or-random assumption for the full stateful generator.

23.1 Abbreviation dictionary and proof method for the XCR component

The XCR chapter uses several short names because the implementation names are long and the algebraic objects are repeated many times. This dictionary is part of the specification. After this point, every abbreviation in the engineering description or in the algebraic analysis refers to the object listed here.

XCR. *XorConstantRotation.* This is the stateful generator implemented by the files `XorConstantRotation.h` and `XorConstantRotation.cpp`. In LOPC it is used to generate round material and key-switch material; it is not the block transform itself.

CSPRNG. *Cryptographically secure pseudorandom number generator.* The word "secure" is used as a design target and as an assumption boundary for the full stateful generator, not as an unconditional theorem in this paper.

PRG. *Pseudorandom generator.* The PRG game at the end of this section measures whether XCR outputs can replace uniform 128-bit strings for the stated use.

ROR. *Real-or-random.* This is the standard distinguishing game in which an adversary sees either real generator outputs or independent uniform outputs.

ARX. *Addition–Rotation–XOR.* XCR uses XORs and rotations directly. Its deployed S-round does not use native CPU modular addition; it uses the shadow-boxed arithmetic maps defined below.

XOR. Bitwise exclusive-or over $\mathbb{F}_2$. In the algebraic sections it is a linear map unless one operand is a constant, in which case the homogeneous lifted form is affine.

H-round. The *Hybrid8 round.* It is the byte-local branch selected by `(counter & 3)==0`. It applies `BigS64`, which is built from the two 8-bit substitutions S_A, S_B.

S-round. The *Shadow round.* It is the 64-bit word branch selected when the counter branch is not the H-round. It calls `shadow_carry64` and `shadow_borrow64`.

LTAM. *Linear tweak-and-mask map.* The implementation has a left form `ltam_l8` and a right form `ltam_r8`; both are 8-bit XOR/shift/AND maps parameterized by a public byte constant.

BTM. *Byte tweak mix.* In the code, `btm8(a,k)` means `ltam_r8(ltam_l8(a,k),k)`.

LCA. *Lifted carry-add byte map.* This is `lifted_carry_add8`; it is a byte-level bounded carry-source map used inside Hybrid8.

LBS. *Lifted borrow-subtract byte map.* This is `lifted_borrow_sub8`; it is the borrow-side byte analogue of LCA.

DP. *Differential probability.* In this section DP always means an XOR-difference probability for the specified Boolean relation, with the denominator stated explicitly.

LC. *Linear correlation.* LC is a signed Walsh correlation. The sign is part of the object; absolute values are taken only after the signed sum if a bound requires it.

ANF. *Algebraic normal form.* ANF is the polynomial representation of a Boolean function over $\mathbb{F}_2$. It is used for algebraic degree and lifted monomial-coordinate arguments, not as a solver certificate.

MILP. *Mixed-integer linear programming.* The NeoAlzette component has a dedicated MILP/HULL backend. The XCR component in this paper does not claim a completed MILP campaign.

SAT/SMT. *Boolean satisfiability* and *satisfiability modulo theories.* They are possible encodings of the same Boolean/factor relations, but no XCR SAT or SMT search certificate is reported here.

Remark 5 (Solver-free proof policy for XCR). The XCR chapter does not hide a missing MILP, SAT, or SMT experiment behind notation. Its mathematical layer is different: shadow-boxed addition and subtraction are analysed by exact finite-state transfer formulas; the code step is written as an explicit composition of affine factors, byte factors, and shadow factors; and the full generator is separated into a component-theory part and an explicit PRG assumption. Thus XCR has mathematical theory in this paper, but it does not have a claimed solver optimality certificate.

23.2 Engineering state, public schedule, and analysis-oracle order

The XCR component is implemented as a stateful four-lane word generator. The paper does not start from a MILP file because there is no honest MILP file until the implementation-level step relation has been named. The right order is: public state and schedule, code-faithful evaluation oracle, component analysis oracles, factor-graph oracle, and only then future solver exports.

Definition 24 (XCR public state and production call)**.** Let

$$S = (w, x, y, z) \in (\mathbb{W}_{64})^4, \qquad C \in \mathbb{Z}/2^{64}\mathbb{Z}, \qquad n \in \mathbb{W}_{64}.$$

Here S is the internal lane state, C is the public counter used by the round-constant schedule, and n is the per-call public input called `number_once` in the implementation. A public production call mutates the state, returns

$$O = (x' \oplus y',\ z' \oplus w') \in (\mathbb{W}_{64})^2,$$

then advances the public counter. Thus the semantic object is not a stateless map $\mathbb{W}_{64} \to \mathbb{W}_{128}$; it is a deterministic transition over (S, C) with visible output.

Definition 25 (XCR step decomposition)**.** For fixed n and C, define

$$\mathrm{Step}_{\mathrm{XCR}}(S, n, C) = (S', O, C') \qquad \text{(XCR-Step)}$$

by the composition

$$\mathrm{Step}_{\mathrm{XCR}} = \mathrm{CounterAdvance}_{\kappa} \circ \mathrm{Extract}_{x\oplus y, z\oplus w} \circ L_{\mathrm{xor/rot}} \circ N_{\tau(C)} \circ A_{n,C}.$$

The map $A_{n,C}$ is the XOR/rotation rebinding and round-constant injection layer; $N_{\tau(C)}$ is either the Hybrid8 H-round nonlinear layer or the Shadow S-round nonlinear layer; $L_{\mathrm{xor/rot}}$ is the final XOR/rotation diffusion; and $\mathrm{CounterAdvance}_{\kappa}$ sends C to $C + \kappa$ modulo 2^{64}.

Remark 6 (Why this order matters). The equation above is the object that a later MILP, SAT, or SMT model must encode. The four shadow calls inside one S-round are only factors of $N_{\tau(C)}$. They are not the whole XCR step, and their marginal probabilities cannot be multiplied and called an exact XCR round probability without shared-lane compatibility constraints.

23.3 Default initialization constant and initialization-root oracle

The no-argument XCR constructor does not begin from an unnamed magic value. It begins from a public, reproducibly searched 64-bit seed-root constant and then calls `StateInitialize`. This constant is not a secret key. It is a deterministic default root used only when the caller does not provide an explicit seed. The explicit-seed constructor and `Seed(seed)` replace this value by the caller's seed before running the same initialization oracle.

Definition 26 (Default initialization constant)**.** Let

$$\theta = \texttt{0xADB136136669D153}.$$

The implementation names this value as

$$\theta = \texttt{XCR_CSPRNG_DEFAULT_INITIALIZE_CONSTANT}.$$

The no-argument constructor initializes

$$(w, x, y, z, C) = (\theta, 0, 0, 0, \kappa)$$

and then executes `StateInitialize`. An explicit call `Seed(seed)` instead initializes

$$(w, x, y, z, C) = (seed, 0, 0, 0, \kappa)$$

and executes the same initialization routine.

Definition 27 (Public search predicate for the default root)**.** The source comment records a deterministic public search with seed 20260221. The search samples a 64-bit candidate by choosing exactly sixteen one-bits in the low half and exactly sixteen one-bits in the high half. A candidate u is accepted only if it satisfies the following finite predicates:

$$\begin{aligned}
&\mathrm{MR}_{64}(u) = 1, \qquad \mathrm{wt}(u) = 32,\\
&\mathrm{wt}(u_{\mathrm{lo}}) = \mathrm{wt}(u_{\mathrm{hi}}) = 16,\\
&\mathrm{SmallModFilter}(u) = 1,\\
&\mathrm{UniformHalf}_{32}(u_{\mathrm{lo}}) = 1, \qquad \mathrm{UniformHalf}_{32}(u_{\mathrm{hi}}) = 1.
\end{aligned}$$

Here MR_{64} is the deterministic Miller–Rabin test for 64-bit integers using the bases embedded in the source comment; $u_{\mathrm{lo}} = u \bmod 2^{32}$ and $u_{\mathrm{hi}} = \lfloor u/2^{32} \rfloor$. The half-uniformity predicate checks balanced Hamming weight, bounded runs, nibble and byte local balance, sliding-window balance, and anti-self-similarity under selected rotations and bit reversal.

Lemma 5 (Default-root search certificate)**.** *The current default root*

$$\theta = \texttt{0xADB136136669D153}$$

is the recorded accepted candidate found after 493 trials of the public search seeded by 20260221*. It satisfies*

$$\begin{aligned}
&\mathrm{MR}_{64}(\theta) = 1, \qquad \mathrm{wt}(\theta) = 32,\\
&\theta_{\mathrm{lo}} = \texttt{0x6669D153}, \qquad \theta_{\mathrm{hi}} = \texttt{0xADB13613},\\
&\mathrm{wt}(\theta_{\mathrm{lo}}) = \mathrm{wt}(\theta_{\mathrm{hi}}) = 16,\\
&\mathrm{maxrun}(\theta_{\mathrm{lo}}) = 3, \qquad \mathrm{maxrun}(\theta_{\mathrm{hi}}) = 4.
\end{aligned}$$

Proof. All assertions are finite checks over a 64-bit integer. The source embeds the search script and records the accepted value, trial count, primality result, half weights, and run bounds. Re-evaluating the predicates on θ gives exactly the displayed values. This proves only public reproducibility of the default constant; it is not a randomness or hardness theorem. □

Definition 28 (Initialization-root oracle)**.** Define the deterministic initialization-root oracle

$$\mathcal{O}_{\mathrm{init}}^{\mathrm{XCR}}(s) = (S_*, C_*)$$

by executing `StateInitialize` from

$$(w, x, y, z, C) = (s, 0, 0, 0, \kappa)$$

with the implementation's branch-free correction replacing $s = 0$ by 1. The no-argument constructor uses $s = \theta$, while an explicit seed path uses the caller-supplied s. The oracle returns the initialized state $S_* = (w_*, x_*, y_*, z_*)$ and resets the public production counter to $C_* = \kappa$.

Remark 7 (Initialization constant versus secret material)*.* The default constant θ, the counter step κ, and the round-constant table are public schedule material. They must be modeled by an analysis oracle, but they must not be described as hidden key entropy. Secret or user-provided entropy enters XCR only through the explicit seed path before $\mathcal{O}_{\mathrm{init}}^{\mathrm{XCR}}$ is applied.

23.4 Public `COUNTER_STEP` derivation and schedule oracle

Definition 29 (Public counter step)**.** Let

$$N_c = 300, \qquad \kappa = \texttt{0xC8522A96E53AF749}.$$

The implementation names κ as `COUNTER_STEP`; it has popcount$(\kappa) = 32$,

$$\kappa \bmod 300 = 241, \qquad \gcd(\kappa, 300) = 1, \qquad \kappa \equiv 1 \pmod 4.$$

The production counter recurrence is

$$C_0 = \kappa, \qquad C_{t+1} = C_t + \kappa \pmod{2^{64}}.$$

For table-index reasoning we also use the residue projection $\rho_t = C_t \bmod 300$.

Definition 30 (Canonical derivation of κ)**.** The public derivation embedded with the implementation starts from $N_c = 300$. Let

$$p_{\max} = \max \text{PrimeFactors}(300) = 5,$$

$$\sigma = \text{previousPrime}\left(\frac{300}{2p_{\max}}\right) = \text{previousPrime}(30) = 29,$$

and

$$\rho = \sigma^2 \bmod 300 = 29^2 \bmod 300 = 241.$$

The small factor is chosen as the largest 21-bit prime P satisfying

$$P \equiv \sigma \pmod{300}, \qquad \text{popcount}(P) = 11.$$

For the current artifact,

$$P = \texttt{0x1F9289} = 2069129.$$

The large factor Q is the first prime above the public anchor that satisfies the documented bit-profile filters and the final κ-filter. In the current implementation,

$$Q = \texttt{0x658459A0EC1} = 6976194612929,$$

and

$$\kappa = PQ = \texttt{0xC8522A96E53AF749}.$$

This derivation is public schedule engineering: it does not add a secret and it is not used as a cryptographic hardness assumption.

Lemma 6 (Table-residue full-cycle property)**.** *In the residue abstraction $\mathbb{Z}_{300}$, the recurrence*

$$\rho_{t+1} = \rho_t + \kappa \pmod{300}$$

visits all 300 residues before repeating.

Proof. The additive order of κ in $\mathbb{Z}_{300}$ is $300/\gcd(\kappa, 300)$. Since $\gcd(\kappa, 300) = 1$, the order is 300. Hence the residue walk is a single full cycle over the round-constant table indices. □

Corollary 1 (H/S selector period)**.** *The branch selector*

$$\tau(C) = \begin{cases} H, & C \bmod 4 = 0, \\ S, & C \bmod 4 \neq 0 \end{cases}$$

has public period four along production calls. Since $C_0 = \kappa \equiv 1 \pmod 4$ and $\kappa \equiv 1 \pmod 4$, the call sequence is

$$S, S, S, H, S, S, S, H, \ldots$$

when counted from the first production call.

Proof. Before the t-th call, $C_t \equiv (t+1) \pmod 4$. The H-round condition is $C_t \equiv 0 \pmod 4$, which is equivalent to $t+1 \equiv 0 \pmod 4$. □

Definition 31 (Schedule oracle)**.** Let $T[0], \ldots, T[299]$ be the public round-constant table. The schedule oracle is

$$\mathcal{O}_{\text{sched}}(n, C) = (RC_0, RC_1, RC_2, RC_3, \tau),$$

where

$$\begin{aligned} RC_0 &= T[n \bmod 300], \\ RC_1 &= T[C \bmod 300], \\ RC_2 &= T[(n + C) \bmod 300], \\ RC_3 &= T[(n \oplus \operatorname{rotl}_3(n \oplus C)) \bmod 300], \end{aligned}$$

and $\tau = H$ if $C \bmod 4 = 0$, otherwise $\tau = S$.

Remark 8 (Counter modeling boundary). The raw implementation counter is a 64-bit word and evolves modulo 2^{64}. The residue walk modulo 300 is a table-index abstraction used by bounded trail analysis and schedule reasoning. A solver model must state which object it encodes.

23.5 XCR round-constant stream generation

The public XCR table $T[0], \ldots, T[299]$ is partly anchored and partly generated. The first four table words are manual public anchors. The remaining table words are filled by the generated stream printed by `GenerateAndDisplay_XorConstantRotation_RoundConstant.py`. This subsection is the normative mathematical reading of that script; the full script is kept in Appendix F.

Let $\operatorname{frac}(x) = x - \lfloor x \rfloor$. The generator fixes decimal precision dps $= 100$ and uses the constants

$$e, \quad \pi, \quad \varphi = \frac{1+\sqrt{5}}{2}, \quad \sqrt{2}, \quad \sqrt{3}, \quad \gamma, \quad \delta, \quad \rho,$$

where γ is Euler's constant, δ is Feigenbaum's constant as printed in the script, and ρ is the plastic number. For $n \geq 1$, define

$$F(n) = \operatorname{frac}(en + \pi n^2 + \varphi n^3 + \sqrt{2} n^4 + \sqrt{3} n^5 + \gamma n^6 + \delta n^7 + \rho n^8).$$

Algorithm 8 GenerateXCRConstantStream

Require: precision dps $= 100$, sample count 150
Ensure: generated 64-bit word stream G

```
set the high-precision constants e, π, φ, √2, √3, γ, δ, ρ
B ← empty bit string
for n = 1 to 150 do
    u_n ← ⌊F(n) · 2^128⌋
    b_n ← BinaryWithoutLeadingPadding(u_n)
    append b_n to B
end for
interpret B as one binary integer and print its hexadecimal expansion
split that hexadecimal expansion into consecutive 64-bit words
return the resulting word stream G
```

Remark 9 (Non-padded bitstream). The current script uses `format(int(fractional_part*2**128),'b')`. It therefore concatenates non-padded binary encodings of the scaled 128-bit fractional integers. The stream length is artifact-defined by the script. It is not $150 \cdot 128$ bits unless fixed-width padding such as `'0128b'` is explicitly enabled. Re-evaluating the current script gives a 19060-bit concatenated stream; its first generated 64-bit word is `0xc7fed6d75df59ae9`, which appears as $T[4]$ after the four manual anchors. This is the table alignment used by the present artifact.

Remark 10 (Historical generator note). The script preserves an older commented-out generator based on a more exploratory analytic expression. That version is retained as a historical note because it was more ad hoc and less reproducible for the desired public constant discipline. The current V2 polynomial Weyl generator fixes the precision, constants, sample count, and stream-extraction rule. This is public reproducibility and constant-discipline evidence, not a randomness proof.

23.6 Code-faithful XCR evaluation oracle

Definition 32 (Evaluation oracle)**.** The XCR evaluation oracle is the deterministic map

$$\mathcal{O}_{\mathrm{eval}}^{\mathrm{XCR}}(S, n, C) = (S', O, C')$$

obtained by one execution of `PermutationARX(number_once)`, output extraction $O = (x' \oplus y', z' \oplus w')$, and counter update $C' = C + \kappa \pmod{2^{64}}$. It is a semantic oracle, not a cryptanalytic probability oracle.

Algorithm 9 Code-faithful XCR evaluation oracle for one production call

Require: state lanes (w, x, y, z), public input n, public counter C
$(RC_0, RC_1, RC_2, RC_3, \tau) \leftarrow \mathcal{O}_{\mathrm{sched}}(n, C)$
$ww \leftarrow x \oplus n$; $xx \leftarrow y \oplus \mathrm{rotl}_9(n)$
$yy \leftarrow z \oplus \mathrm{rotl}_{27}(n)$; $zz \leftarrow w \oplus \mathrm{rotl}_{43}(n)$
$z_0 \leftarrow xx \oplus RC_0$; $w_0 \leftarrow yy \oplus RC_1$; $x_0 \leftarrow zz \oplus RC_2$; $y_0 \leftarrow ww \oplus RC_3$
if $\tau = H$ **then**
    $yy_1 \leftarrow \mathrm{BigS}_{64}(yy \oplus x_0, 0)$
    $zz_1 \leftarrow \mathrm{BigS}_{64}(zz \oplus y_0, 8)$
    $ww_1 \leftarrow \mathrm{BigS}_{64}(ww \oplus z_0, 16)$
    $xx_1 \leftarrow \mathrm{BigS}_{64}(xx \oplus w_0, 24)$
else
    $yy_1 \leftarrow yy \boxplus_{64}^{\mathrm{sh,zf}} x_0$; $zz_1 \leftarrow zz \boxminus_{64}^{\mathrm{sh,zf}} y_0$
    $ww_1 \leftarrow ww \boxplus_{64}^{\mathrm{sh,zf}} z_0$; $xx_1 \leftarrow xx \boxminus_{64}^{\mathrm{sh,zf}} w_0$
end if
$x' \leftarrow x_0 \oplus \mathrm{rotl}_7(xx_1) \oplus \mathrm{rotl}_{19}(yy_1) \oplus zz_1$
$y' \leftarrow y_0 \oplus \mathrm{rotl}_{11}(yy_1) \oplus \mathrm{rotl}_{23}(zz_1) \oplus ww_1$
$z' \leftarrow z_0 \oplus \mathrm{rotl}_{17}(zz_1) \oplus \mathrm{rotl}_{29}(ww_1) \oplus xx_1$
$w' \leftarrow w_0 \oplus \mathrm{rotl}_{13}(ww_1) \oplus \mathrm{rotl}_{31}(xx_1) \oplus yy_1$
$O \leftarrow (x' \oplus y', z' \oplus w')$; $C' \leftarrow C + \kappa \pmod{2^{64}}$
return $((w', x', y', z'), O, C')$

Lemma 7 (Code-faithfulness of the evaluation oracle)**.** *Algorithm 9 is operator-by-operator identical to the current* `PermutationARX` *and* `StateIteration` *order.*

Proof. The algorithm first performs the same XOR/rotation rebinding of n into (ww, xx, yy, zz), then derives the same four table constants, then writes the same premixed lanes (z_0, w_0, x_0, y_0). The branch predicate is the public condition $(C \,\&\, 3) = 0$. The H-branch calls the same four `BigS64` expressions; the S-branch calls the same four

shadow carry/borrow expressions. The final four lane equations and the output view $(x' \oplus y', z' \oplus w')$ match the implementation order, and the counter is advanced only after output extraction. □

Theorem 1 (Evaluation-oracle soundness)**.** *For every input triple (S, n, C), the evaluation oracle $\mathcal{O}_{\mathrm{eval}}^{\mathrm{XCR}}(S, n, C)$ returns exactly the state, output, and next counter produced by one public XCR production call. The statement is about the current source-level transition order.*

Proof. The preceding lemma gives equality of every intermediate word operation. Since fixed-width XOR, rotation, table lookup, byte maps, shadow maps, and counter addition are deterministic total operations on finite domains, equality of the ordered operations implies equality of the composed transition. □

23.7 Engineering primitive pseudocode behind the XCR oracles

The analysis oracles below are not abstract black boxes. They are attached to the exact helper functions that appear in `XorConstantRotation.cpp`. This subsection first gives the XCR-specific arithmetic-operator notation, and then keeps the code-faithful pseudocode that implements those operators. The operator notation is used by the DP, LC, SAT, SMT, and MILP interfaces; the pseudocode remains as the implementation audit view.

Definition 33 (XCR engineered arithmetic operators)**.** XCR uses four non-standard ARX-like arithmetic operators. They are deterministic word maps, not ordinary modular addition or subtraction. The lower index gives the word size, and the upper tag gives the operator family:

$$\boxplus_8^{\mathrm{lift}}, \qquad \boxminus_8^{\mathrm{lift}}, \qquad \boxplus_{64}^{\mathrm{sh,zf}}, \qquad \boxminus_{64}^{\mathrm{sh,zf}}.$$

The lifted byte operators are the Hybrid8 byte helpers. The shadow zero-fill word operators are the S-round shadow carry/borrow helpers. The tag lift means that byte-local carry or borrow source material is lifted through the distances $1, 2, 4$ inside one byte. The tag sh, zf means a shadow carry/borrow source is updated twice and projected back through zero-fill shifts. Neither family is associative, and neither family should be replaced by the ordinary operators $\boxplus_n$ or $\boxminus_n$.

Definition 34 (Lifted byte addition and subtraction operators)**.** For $a, k \in \mathbb{F}_2^8$, define the lifted byte-add operator $a \boxplus_8^{\mathrm{lift}} k$ as follows:

$$\begin{aligned} p &= a \oplus k, & g &= a \wedge k, \\ n_1 &= p \gg 1, & n_2 &= (p \gg 2) \oplus (p \gg 3), \\ w_1 &= g, & w_2 &= w_1 \wedge n_1, & w_4 &= w_2 \wedge n_2, \end{aligned}$$

$$a \boxplus_8^{\mathrm{lift}} k = p \oplus (w_1 \ll 1) \oplus (w_2 \ll 2) \oplus (w_4 \ll 4),$$

with every shift interpreted as the eight-bit zero-fill shift of Part II. Define the lifted byte-subtract operator $a \boxminus_8^{\mathrm{lift}} k$ by

$$\begin{aligned} p &= a \oplus k, & h &= (\neg_8 a) \wedge k, \\ r &= \neg_8 p, & n_1 &= r \gg 1, & n_2 &= (r \gg 2) \oplus (r \gg 3), \\ w_1 &= h, & w_2 &= w_1 \wedge n_1, & w_4 &= w_2 \wedge n_2, \end{aligned}$$

$$a \boxminus_8^{\mathrm{lift}} k = p \oplus (w_1 \ll 1) \oplus (w_2 \ll 2) \oplus (w_4 \ll 4).$$

The lifted addition side starts from the generate source $a \wedge k$. The lifted subtraction side starts from the oriented borrow source $(\neg_8 a) \wedge k$. Both maps keep the carry or borrow source inside the byte; no inter-byte carry or borrow is exported.

Definition 35 (LTAM, BTM, Hybrid8, and BigS)**.** For $a, k \in \mathbb{F}_2^8$, define

$$\mathrm{LTAM}_{L,8}(a,k) = a \oplus ((a \ll 1) \wedge k) \oplus ((a \ll 2) \wedge k) \oplus ((a \ll 4) \wedge k),$$

$$\mathrm{LTAM}_{R,8}(a,k) = a \oplus ((a \gg 1) \wedge k) \oplus ((a \gg 2) \wedge k) \oplus ((a \gg 4) \wedge k),$$

and

$$\mathrm{BTM}_8(a,k) = \mathrm{LTAM}_{R,8}(\mathrm{LTAM}_{L,8}(a,k),k).$$

The two Hybrid8 byte substitutions are the code-faithful compositions

$$S_A(x) = \big(\mathrm{BTM}_8((\mathrm{BTM}_8(x,\texttt{0x2b}) \boxminus_8^{\mathrm{lift}} \texttt{0x3f}),\texttt{0x2b})\big) \boxplus_8^{\mathrm{lift}} \texttt{0x3f},$$
$$S_B(x) = \mathrm{BTM}_8(\mathrm{BTM}_8((x \boxplus_8^{\mathrm{lift}} \texttt{0x3f}),\texttt{0x2b}) \boxminus_8^{\mathrm{lift}} \texttt{0x3f},\texttt{0x2b}).$$

For a 64-bit word u, byte j is read using the common accessor $\mathrm{ReadByte}(u,j)$. The output byte of the BigS layer is defined by

$$\mathrm{ReadByte}(\mathrm{BigS}_{64}(u,o),j) = \begin{cases} S_A(\mathrm{ReadByte}(u,j)), & (o+j) \bmod 2 = 0, \\ S_B(\mathrm{ReadByte}(u,j)), & (o+j) \bmod 2 = 1. \end{cases}$$

Algorithm 10 Code-faithful Hybrid8 helpers used by the H-round

function LCA8(a,k)
  $p \leftarrow a \oplus k$; $g \leftarrow a \wedge k$
  $n_1 \leftarrow p \gg 1$; $n_2 \leftarrow (p \gg 2) \oplus (p \gg 3)$
  $w_1 \leftarrow g$; $w_2 \leftarrow w_1 \wedge n_1$; $w_4 \leftarrow w_2 \wedge n_2$
  return $p \oplus (w_1 \ll 1) \oplus (w_2 \ll 2) \oplus (w_4 \ll 4)$ truncated to 8 bits
end function

function LBS8(a,k)
  $p \leftarrow a \oplus k$; $h \leftarrow (\neg_8 a) \wedge k$; $r \leftarrow \neg_8 p$
  $n_1 \leftarrow r \gg 1$; $n_2 \leftarrow (r \gg 2) \oplus (r \gg 3)$
  $w_1 \leftarrow h$; $w_2 \leftarrow w_1 \wedge n_1$; $w_4 \leftarrow w_2 \wedge n_2$
  return $p \oplus (w_1 \ll 1) \oplus (w_2 \ll 2) \oplus (w_4 \ll 4)$ truncated to 8 bits
end function

function Hybrid8SA(x)
  $x \leftarrow \mathrm{BTM}_8(x,\texttt{0x2b})$
  $x \leftarrow x \boxminus_8^{\mathrm{lift}} \texttt{0x3f}$
  $x \leftarrow \mathrm{BTM}_8(x,\texttt{0x2b})$
  $x \leftarrow x \boxplus_8^{\mathrm{lift}} \texttt{0x3f}$
  return x
end function

function Hybrid8SB(x)
  $x \leftarrow x \boxplus_8^{\mathrm{lift}} \texttt{0x3f}$
  $x \leftarrow \mathrm{BTM}_8(x,\texttt{0x2b})$
  $x \leftarrow x \boxminus_8^{\mathrm{lift}} \texttt{0x3f}$
  $x \leftarrow \mathrm{BTM}_8(x,\texttt{0x2b})$
  return x
end function

function BigS64(u,o)
  $r \leftarrow 0$
  for $j = 0$ to 7 **do**

```
        b ← ReadByte(u, j)
        if (o + j) mod 2 = 0 then
            b′ ← Hybrid8SA(b)
        else
            b′ ← Hybrid8SB(b)
        end if
        r ← WriteByte(r, j, b′)
    end for
    return r
end function
```

Definition 36 (Hybrid8 engineering analysis oracle)**.** The Hybrid8 engineering oracle is the tuple

$$\mathcal{O}_{\mathrm{H,eng}} = (S_A, S_B, \mathrm{BigS}_{64}, \mathcal{D}_A, \mathcal{D}_B, \mathcal{L}_A, \mathcal{L}_B),$$

where S_A, S_B are the byte maps above, $\mathcal{D}_s$ is the exact byte DDT

$$\mathcal{D}_s(\alpha, \gamma) = \#\{x \in \mathbb{F}_2^8 : S_s(x \oplus \alpha) \oplus S_s(x) = \gamma\},$$

and $\mathcal{L}_s$ is the exact signed byte LAT

$$\mathcal{L}_s(\lambda, \nu) = \sum_{x \in \mathbb{F}_2^8} (-1)^{\lambda \cdot x \oplus \nu \cdot S_s(x)}.$$

This oracle is what a future H-round MILP/SAT/SMT model must consume for byte nonlinear behavior.

Definition 37 (Shadow zero-fill word addition and subtraction operators)**.** For $a, b \in \mathbb{F}_2^{64}$, define the shadow zero-fill addition operator $a \boxplus_{64}^{\mathrm{sh,zf}} b$ by

$$\begin{aligned} p &= a \oplus b, & c^{(0)} &= a \wedge b, & q^{(0)} &= p, \\ c^{(1)} &= c^{(0)} \oplus (q^{(0)} \wedge (c^{(0)} \ll 1)), & q^{(1)} &= q^{(0)} \wedge (q^{(0)} \ll 1), & & \\ c^{(2)} &= c^{(1)} \oplus (q^{(1)} \wedge (c^{(1)} \ll 2)), & a \boxplus_{64}^{\mathrm{sh,zf}} b &= p \oplus (c^{(2)} \ll 1) \oplus (c^{(2)} \ll 2). & & \end{aligned}$$

Define the shadow zero-fill subtraction operator $a \boxminus_{64}^{\mathrm{sh,zf}} b$ by

$$\begin{aligned} p &= a \oplus b, & d^{(0)} &= (\neg_{64} a) \wedge b, & q^{(0)} &= \neg_{64} p, \\ d^{(1)} &= d^{(0)} \oplus (q^{(0)} \wedge (d^{(0)} \ll 1)), & q^{(1)} &= q^{(0)} \wedge (q^{(0)} \ll 1), & & \\ d^{(2)} &= d^{(1)} \oplus (q^{(1)} \wedge (d^{(1)} \ll 2)), & a \boxminus_{64}^{\mathrm{sh,zf}} b &= p \oplus (d^{(2)} \ll 1) \oplus (d^{(2)} \ll 2). & & \end{aligned}$$

All shifts are 64-bit zero-fill shifts. These operators are the deployed Model-A shadow-boxed word maps. They secretly inject a bounded carry or borrow source into the XOR parity word through the projected terms at offsets one and two. The injection is bounded: it is not a full ripple carry or borrow chain, and no cyclic wraparound is used.

Algorithm 11 Code-faithful shadow helpers used by the S-round

```
function ShadowCarry64(a, b)
    p ← a ⊕ b; c ← a ∧ b; q ← p
    c ← c ⊕ (q ∧ (c ≪ 1))
    q ← q ∧ (q ≪ 1)
    c ← c ⊕ (q ∧ (c ≪ 2))
    return p ⊕ (c ≪ 1) ⊕ (c ≪ 2) truncated to 64 bits
end function
```

function ShadowBorrow64(a, b)
  $p \leftarrow a \oplus b$; $c \leftarrow (\neg_{64} a) \wedge b$; $q \leftarrow \neg_{64} p$
  $c \leftarrow c \oplus (q \wedge (c \ll 1))$
  $q \leftarrow q \wedge (q \ll 1)$
  $c \leftarrow c \oplus (q \wedge (c \ll 2))$
  return $p \oplus (c \ll 1) \oplus (c \ll 2)$ truncated to 64 bits
end function

Lemma 8 (Shadow helper equivalence)**.** *The code-faithful helper names and the operator notation denote the same maps:*

$$\mathrm{ShadowCarry}_{64}(a, b) = a \boxplus_{64}^{\mathrm{sh,zf}} b, \qquad \mathrm{ShadowBorrow}_{64}(a, b) = a \boxminus_{64}^{\mathrm{sh,zf}} b.$$

The implementation variable c represents the bounded source word after two updates.

Proof. For carry, the implementation sets $p = a \oplus b$, initializes the source as $a \wedge b$, applies the one-position update, updates the propagate-run mask, applies the two-position update, and projects the final source at offsets one and two. This is the zero-fill shadow addition definition. For borrow, the only semantic changes are the oriented borrow-generate source $(\neg_{64} a) \wedge b$ and the borrow-propagate word $\neg_{64}(a \oplus b)$; the same update and projection graph gives zero-fill shadow subtraction. □

Definition 38 (Shadow64 engineering analysis oracle)**.** The Shadow64 engineering oracle is

$$\mathcal{O}_{\mathrm{sh64,eng}} = (\boxplus_{64}^{\mathrm{sh,zf}}, \boxminus_{64}^{\mathrm{sh,zf}}, \mathcal{O}_{\mathrm{sh}}^{\Delta,+}, \mathcal{O}_{\mathrm{sh}}^{\Delta,-}, \mathcal{O}_{\mathrm{sh}}^{L,+}, \mathcal{O}_{\mathrm{sh}}^{L,-}).$$

The first two entries are exact evaluators. The last four entries are the exact DP and signed-LC component oracles inherited from Model-A zero-fill shadow-boxed arithmetic. A future S-round solver model must connect these component oracles through the shared XCR lane equations rather than multiplying their marginal scores in isolation.

Theorem 2 (Engineering-oracle soundness for H and S primitives)**.** *The byte oracle* $\mathcal{O}_{\mathrm{H,eng}}$ *and the word oracle* $\mathcal{O}_{\mathrm{sh64,eng}}$ *describe exactly the nonlinear helper calls used by the current XCR H-round and S-round.*

Proof. Algorithm 10 follows the implementation order of `hybrid8_sa`, `hybrid8_sb`, and `bigs64`; Algorithm 11 follows `shadow_carry64` and `shadow_borrow64`. The DDT/LAT components are finite enumerations of the displayed byte maps, and the shadow DP/LC components are the Model-A zero-fill transfers for the displayed word maps. Therefore every nonlinear helper call in `PermutationARX` has a matching semantic evaluator and a matching analysis-oracle interface. □

23.8 Hybrid8 byte layer

Definition 39 (Hybrid8 byte maps)**.** For an 8-bit word x, define

$$\mathrm{LTAM}_{L,8}(x, k) = x \oplus ((x \ll 1) \wedge k) \oplus ((x \ll 2) \wedge k) \oplus ((x \ll 4) \wedge k),$$

$$\mathrm{LTAM}_{R,8}(x, k) = x \oplus ((x \gg 1) \wedge k) \oplus ((x \gg 2) \wedge k) \oplus ((x \gg 4) \wedge k),$$

and $\mathrm{BTM}_8(x, k) = \mathrm{LTAM}_{R,8}(\mathrm{LTAM}_{L,8}(x, k), k)$. With LCA_8 and LBS_8 denoting the byte lifted carry and borrow maps,

$$S_A(x) = \big(\mathrm{BTM}_8((\mathrm{BTM}_8(x, \texttt{0x2b}) \boxminus_8^{\mathrm{lift}} \texttt{0x3f}), \texttt{0x2b})\big) \boxplus_8^{\mathrm{lift}} \texttt{0x3f},$$
$$S_B(x) = \mathrm{BTM}_8(\mathrm{BTM}_8((x \boxplus_8^{\mathrm{lift}} \texttt{0x3f}), \texttt{0x2b}) \boxminus_8^{\mathrm{lift}} \texttt{0x3f}, \texttt{0x2b}).$$

The 64-bit BigS_{64} applies $S_A, S_B, S_A, S_B, \ldots$ over the eight bytes with a public byte offset.

Remark 11 (Engineering reading). The H-round is the byte hammer. The S-round is the shadow arithmetic hammer. The public counter decides which hammer is used. Shadow arithmetic theorems analyze the S-round components; they do not automatically analyze Hybrid8.

23.9 Code-faithful XCR state transition

Definition 40 (XCR step object)**.** Let $S = (w, x, y, z, c) \in (\mathbb{W}_{64})^4 \times \mathbb{Z}_{2^{64}}$. For public input n, define

$$\mathrm{Step}_{\mathrm{XCR}}(S, n) = (S', O), \qquad O = (x' \oplus y', z' \oplus w'),$$

where (w', x', y', z') are the lanes after `PermutationARX`; the returned state has $c' = c +$ `COUNTER_STEP` $(\mathrm{mod}\ 2^{64})$.

Algorithm 12 Current XCR `PermutationARX(number_once)`

Require: lanes (w, x, y, z), counter c, public input n, table $RC[0..299]$.

$ww \leftarrow x \oplus n$; $xx \leftarrow y \oplus \mathrm{rotl}_{64}(n, 9)$
$yy \leftarrow z \oplus \mathrm{rotl}_{64}(n, 27)$; $zz \leftarrow w \oplus \mathrm{rotl}_{64}(n, 43)$
$RC_0 \leftarrow RC[n \bmod 300]$; $RC_1 \leftarrow RC[c \bmod 300]$
$RC_2 \leftarrow RC[(n + c) \bmod 300]$
$RC_3 \leftarrow RC[(n \oplus \mathrm{rotl}_{64}(n \oplus c, 3)) \bmod 300]$
$z \leftarrow xx \oplus RC_0$; $w \leftarrow yy \oplus RC_1$; $x \leftarrow zz \oplus RC_2$; $y \leftarrow ww \oplus RC_3$
if $(c \ \&\ 3) = 0$ **then**
    $yy \leftarrow \mathrm{BigS}_{64}(yy \oplus x, 0)$; $zz \leftarrow \mathrm{BigS}_{64}(zz \oplus y, 8)$
    $ww \leftarrow \mathrm{BigS}_{64}(ww \oplus z, 16)$; $xx \leftarrow \mathrm{BigS}_{64}(xx \oplus w, 24)$
else
    $yy \leftarrow yy \boxplus_{64}^{\mathrm{sh,zf}} x$; $zz \leftarrow zz \boxminus_{64}^{\mathrm{sh,zf}} y$; $ww \leftarrow ww \boxplus_{64}^{\mathrm{sh,zf}} z$; $xx \leftarrow xx \boxminus_{64}^{\mathrm{sh,zf}} w$
end if
$x \leftarrow x \oplus \mathrm{rotl}_{64}(xx, 7) \oplus \mathrm{rotl}_{64}(yy, 19) \oplus zz$
$y \leftarrow y \oplus \mathrm{rotl}_{64}(yy, 11) \oplus \mathrm{rotl}_{64}(zz, 23) \oplus ww$
$z \leftarrow z \oplus \mathrm{rotl}_{64}(zz, 17) \oplus \mathrm{rotl}_{64}(ww, 29) \oplus xx$
$w \leftarrow w \oplus \mathrm{rotl}_{64}(ww, 13) \oplus \mathrm{rotl}_{64}(xx, 31) \oplus yy$

Algorithm 13 Current XCR public production call

function StateIteration(n)
    PermutationARX(n)
    $out \leftarrow (x \oplus y,\ z \oplus w)$
    $c \leftarrow c +$ `COUNTER_STEP` $(\mathrm{mod}\ 2^{64})$
    return *out*
end function

function GenerateSubKey128(n)
    return StateIteration(n)
end function

function operator()(n)
    return StateIteration(n)
end function

Lemma 9 (Counter timing)**.** *The production call mutates lanes, extracts* $(x \oplus y, z \oplus w)$*, and only then advances the public counter.*

Proof. This is the order in `StateIteration`: `PermutationARX`, construct `out`, increment `counter`, return `out`. □

Lemma 10 (Affine shell and nonlinear middle)**.** *Once n, c, and public constants are fixed, rebinding, constant injection, final diffusion, and output extraction are affine over $\mathbb{F}_2$. Nonlinearity occurs only in the selected H-round byte substitutions or S-round shadow operations.*

Proof. XOR by fixed words, rotations, and XOR projections are affine. Hybrid8 contains nonlinear byte maps, and shadow operations contain AND-generated source terms. The current core permutation has no native state-lane modular add/sub. □

23.10 StateIteration, GenerateSubKey128, and StateInitialize

`StateInitialize` is not a hidden primitive. It is a deterministic GGM-like two-leaf warm-up wrapper around `StateIteration`. The proof focus therefore stays on the iterated XCR step.

Algorithm 14 GGM-like local routine `GGM64Rounds(seed64)`

$out64 \leftarrow seed64$
for $round = 0$ to 1 **do**
  $next64 \leftarrow 0$
  for $bit = 0$ to 63 **do**
    $in \leftarrow (\texttt{WARMUP} \ll 48) \oplus (out64 \ll 32) \oplus (out64 \ll 16) \oplus (round \ll 8) \oplus bit$
    $out128 \leftarrow \textsc{StateIteration}(in)$
    $next64 \leftarrow (next64 \ll 1) \oplus \mathrm{ReadBit}(out128, 127)$
  end for
  $out64 \leftarrow next64$
end for
return $out64$

Algorithm 15 Current XCR `StateInitialize()`

Require: constructor or `Seed` set $(w, x, y, z, c) = (seed, 0, 0, 0, \texttt{COUNTER_STEP})$.
force $w \neq 0$ using the implementation's branch-free correction
$backup \leftarrow w$
$left_input \leftarrow backup \oplus RC[299]$; $right_input \leftarrow backup \oplus RC[298]$
$left_leaf \leftarrow \text{GGM64}\textsc{Rounds}(left_input)$; $(l_x, l_y, l_z) \leftarrow (x, y, z)$
reset $(x, y, z, w, c) \leftarrow (0, 0, 0, backup, \texttt{COUNTER_STEP})$
$right_leaf \leftarrow \text{GGM64}\textsc{Rounds}(right_input)$; $(r_x, r_y, r_z) \leftarrow (x, y, z)$
reset $(x, y, z, w, c) \leftarrow (0, 0, 0, backup, \texttt{COUNTER_STEP})$
$x \leftarrow l_x \oplus r_x$; $y \leftarrow l_y \oplus r_y$; $z \leftarrow l_z \oplus r_z$
$w \leftarrow backup \oplus left_leaf \oplus right_leaf$

Lemma 11 (Initialization reduces to StateIteration)**.** *Every nonlinear call inside* `StateInitialize` *is a call to* `StateIteration`*; the remaining actions are XOR, assignment, and reset-to-root.*

Proof. The two leaf routines generate bits by repeated `StateIteration` calls. Between leaves the object is reset to the same root state. The final initialized state is formed by XOR-folding the two leaf states and leaf outputs. □

Remark 12 (GGM-like boundary). "GGM-like" describes the two-branch expansion shape. It is not an unconditional GGM theorem for XCR.

24 Component C Specification: Key Switch Layer

24.1 Key Switch Layer: keyed switching and random bit tweak

The third component of the current LOPC split is the *Key Switch Layer*. It is not the NeoAlzette S-box and it is not the XCR generator. It is the deterministic reversible layer that consumes the XCR-derived key-state fields

$$(sk_i^{(0)}, sk_i^{(1)}, cf_i, \alpha_i, \beta_i)$$

after the two diagonal NeoAlzette calls. In the C++ code this component appears as two sublayers:

`MixLinearTransform_Forward`, `MixLinearTransform_Backward`,
`Random Bit Tweak Forward`, `Random Bit Tweak Backward`.

For each fixed generated key state it is an affine/permutation layer over the two 64-bit data lanes. It should therefore be analysed as a code-bound reversible switching layer, not as a separate CSPRNG.

Definition 41 (Keyed switching map)**.** For a fixed key state $K_i = (sk_i^{(0)}, sk_i^{(1)}, cf_i, \beta_i)$, define $\mathrm{SwitchForward}_{K_i} : (\mathbb{W}_{64})^2 \to (\mathbb{W}_{64})^2$ by

$$\begin{aligned}
cf_i = 0 &: (L_0, L_1) \leftarrow (L_0 \oplus sk_i^{(0)}, L_1 \oplus sk_i^{(1)}),\\
cf_i = 1 &: (L_0, L_1) \leftarrow (\neg L_0 \oplus sk_i^{(0)}, \neg L_1 \oplus sk_i^{(1)}),\\
cf_i = 2 &: (L_0, L_1) \leftarrow (\mathrm{rotl}_{64}(L_0, \beta_i), \mathrm{rotl}_{64}(L_1, \beta_i)),\\
cf_i = 3 &: (L_0, L_1) \leftarrow (\mathrm{rotr}_{64}(L_0, \beta_i), \mathrm{rotr}_{64}(L_1, \beta_i)).
\end{aligned}$$

The backward map uses the same first two branches and the opposite rotation directions for the last two branches.

Algorithm 16 Key Switch Layer: `MixLinearTransform_Forward`

Require: lanes (L_0, L_1) and generated key state $(sk^{(0)}, sk^{(1)}, cf, \beta)$

$A_0 \leftarrow L_0 \oplus sk^{(0)}$; $A_1 \leftarrow L_1 \oplus sk^{(1)}$
$B_0 \leftarrow \neg L_0 \oplus sk^{(0)}$; $B_1 \leftarrow \neg L_1 \oplus sk^{(1)}$
$C_0 \leftarrow \mathrm{rotl}_{64}(L_0, \beta)$; $C_1 \leftarrow \mathrm{rotl}_{64}(L_1, \beta)$
$D_0 \leftarrow \mathrm{rotr}_{64}(L_0, \beta)$; $D_1 \leftarrow \mathrm{rotr}_{64}(L_1, \beta)$
Select $(A_0, A_1), (B_0, B_1), (C_0, C_1)$, or (D_0, D_1) according to $cf \bmod 4$ by constant-time equality masks.
return selected pair.

Algorithm 17 Key Switch Layer: `MixLinearTransform_Backward`

Require: lanes (L_0, L_1) and generated key state $(sk^{(0)}, sk^{(1)}, cf, \beta)$

$A_0 \leftarrow L_0 \oplus sk^{(0)}$; $A_1 \leftarrow L_1 \oplus sk^{(1)}$
$B_0 \leftarrow \neg L_0 \oplus sk^{(0)}$; $B_1 \leftarrow \neg L_1 \oplus sk^{(1)}$
$C_0 \leftarrow \mathrm{rotr}_{64}(L_0, \beta)$; $C_1 \leftarrow \mathrm{rotr}_{64}(L_1, \beta)$
$D_0 \leftarrow \mathrm{rotl}_{64}(L_0, \beta)$; $D_1 \leftarrow \mathrm{rotl}_{64}(L_1, \beta)$
Select $(A_0, A_1), (B_0, B_1), (C_0, C_1)$, or (D_0, D_1) according to $cf \bmod 4$ by constant-time equality masks.
return selected pair.

Definition 42 (Random bit tweak)**.** For a fixed generated tweak index $\alpha_i \in \{0, \ldots, 63\}$, define

$$\mathrm{Tweak}_{\alpha_i}(L_0, L_1) = \big(L_0 \oplus (1 \ll \alpha_i),\ L_1 \oplus (1 \ll (63 - \alpha_i))\big).$$

The forward and backward tweak maps are the same map.

Lemma 12 (Key Switch Layer invertibility)**.** *For every fixed generated key state* $(sk_i^{(0)}, sk_i^{(1)}, cf_i, \alpha_i, \beta_i)$*, the composition*

$$\mathrm{Tweak}_{\alpha_i} \circ \mathrm{SwitchForward}_{K_i}$$

is a permutation of $(\mathbb{W}_{64})^2$*. Its inverse is*

$$\mathrm{SwitchBackward}_{K_i} \circ \mathrm{Tweak}_{\alpha_i} .$$

Proof. The tweak is xor by two fixed unit words and is self-inverse. For the switching map, branch $cf = 0$ is xor by the two fixed subkey words and is self-inverse. Branch $cf = 1$ is complement followed by xor by a fixed subkey word; applying the same branch twice gives the identity because $\neg(\neg L \oplus s) \oplus s = L$. Branch $cf = 2$ is rotation left by β, whose inverse is rotation right by β. Branch $cf = 3$ is rotation right by β, whose inverse is rotation left by β. Therefore Algorithm 17 inverts Algorithm 16, and the self-inverse tweak completes the stated inverse. □

Matrix interpretation. For fixed key-state values, the Key Switch Layer lies inside the same algebraic matrix view used elsewhere in this paper. The xor branches are homogeneous affine binary matrices on lifted vectors $(L_0, L_1, 1)$, the complement-xor branch is also affine because $\neg L = L \oplus \mathbf{1}$, and the rotation branches are block-diagonal permutation matrices. The random bit tweak is a final affine translation by two unit masks. The layer is nonlinear only with respect to the generated control variables $(cf_i, \alpha_i, \beta_i)$; once those variables are fixed, the data path is exactly reversible and matrix-representable.

25 Decryption and Permutation Structure

Decryption applies the inverse layers in reverse order: RandomBitTweak is undone by the same XOR operation, `MixLinearTransform_Backward` inverts the selected branch, NeoAlzette backward inverts the diagonal ARX-box calls, and round-key XOR is self-inverse. This section keeps the code-bound correctness material in the security Part and records the specification-level inverse order here.

Part IV
Design Rationale

26 Global Design Rationale

The three components serve different roles. NeoAlzette handles state nonlinearity through a diagonal ARX-box layer. XCR handles round-material evolution through a stateful Hybrid8–Shadow generator. The Key Switch Layer changes the affine/permutation geometry of each fixed round and injects an asymmetric keyed bit-position constant. These roles are separated throughout the rest of the paper.

The round order follows this component boundary. XCR first supplies evolving round material; the LOPC wrapper then binds that material to the two 64-bit data lanes through subkey injection. The state is unpacked so that NeoAlzette supplies the main 32-bit-branch ARX-box nonlinearity, and the repacked 64-bit lanes are finally passed through the Key Switch Layer before the next round. In this reading, XCR is the upstream stream-generator/key-material source, NeoAlzette is the block-core nonlinear layer, and the Key Switch Layer is the post-ARX keyed affine/tweak layer. The design rationale below explains why those responsibilities are kept distinct instead of merging them into one opaque round formula.

27 Component A Rationale: NeoAlzette

This section imports the V6.5 anti-bypass engineering rationale into the design-rationale part of the LOPC paper. The primitive definitions used by the arguments live in Part II; this section explains why the component is arranged in the implemented order.

27.1 Design Background

The relevant failure mode is an internal zero-window bypass. This is not the trivial statement that an external all-zero difference maps to an all-zero output difference. It is the more specific phenomenon in which the controlling branch is forced to zero exactly at the injection point, allowing the local injection to be skipped, and is then restored only after the local work is already over. The schedule below is arranged so that any successful bypass attempt must also remain compatible with the derived modular-addition operand and with the later bridge feedback.

The intended design principle is therefore:

$$\begin{aligned}&\text{source branch} \to \text{joint injection} \to \text{XOR routing}\\ &\to \text{derived modular addition} \to \text{bridge feedback}.\end{aligned}$$

The injection layer is not an isolated XOR gadget. It is a joint source for both the XOR injection material and the subsequent arithmetic operand.

27.2 Inheritance from Alzette

The schedule inherits the ARX rhythm of Alzette, not its implementation economy. The important part of the Alzette spirit is the alternation of addition, rotation, XOR, and constant disturbance. NeoAlzette keeps that rhythm, but adds a fixed public subtraction at the beginning of each half-round, branch-derived injection into the opposite branch, and an injection-derived modular-addition operand. In short:

$$\begin{aligned}&\text{fixed subtraction} \to \text{joint injection} \to \text{derived addition}\\ &\to \text{bridge feedback} \to \text{mirror half-round}.\end{aligned}$$

27.3 Relation to the Prior Fallback Variant

This backup schedule is not a cosmetic restatement of the previous fallback version. The earlier variant let the post-addition intermediate state feed the bridge and the injection. That was still vulnerable to a low-cost source collapse: a trail search could keep the relevant intermediate branch inactive and let the local injection contribute no effective cost.

The present version reverses that dependency. The source branch is injected first, and the injection output then generates the modular-addition operand. The schedule therefore becomes:

$$\text{injection source} \to \text{XOR injection} + \text{injection-derived addend}.$$

This is the essential correction. The injection layer is no longer a local XOR-only patch that can be skipped before the arithmetic path is touched.

27.4 Fixed Public Subtraction

The steps $B \leftarrow B - \mathrm{RC}[1]$ and $A \leftarrow A - \mathrm{RC}[6]$ are not decorative whitening. They introduce fixed borrow chains, so the bit order becomes a real coordinate anchor rather

than a rotation-covariant convenience. A trail search that tries to erase activity in the injection window must still remain compatible with the borrow propagation induced by the public subtraction.

This is the practical role of the fixed subtraction: it does not make zero activity impossible, but it makes a zero-window condition depend on the concrete borrow-chain relation induced by the fixed public constant. In this precise sense the zero window is not a free averaged-addition event.

27.5 Why the Constant Does Not Belong Only Inside the Injection

Earlier variants tried to hide more complexity inside the local injection function alone. That is not enough. If a trail search can avoid entering the injection window, then a complicated local function never gets touched. The anti-bypass schedule therefore makes the injection output feed the later modular addition as well. This turns the injection layer from a local patch into a global path constraint.

27.6 Rotation Roles

The rotation budget is partitioned by role rather than by symmetry. The four bridge numbers are not hidden in the sleeves of the schedule; they are the output of an explicit engineering filter against zero-window bypass and bridge template reuse:

$$\{31, 17, 22, 13, 24, 16, 8, 5, 25\}.$$

The roles are:

Rotation set	Role
$31, 17$	derived modular-addition rotations on the B-to-A side
$22, 13$	first cross-branch feedback bridge
$24, 16, 8$	local injection routing and diffusion on the B-side
$5, 25$	second cross-branch feedback bridge

The zero-filling shifts inside $\mathsf{Inj}^{\mathrm{NA}}_{B\to A}$ and $\mathsf{Inj}^{\mathrm{NA}}_{A\to B}$ are local Boolean expansion operators; they are not bridge rotations.

The bridge selection logic is therefore part of the specification, not an undocumented implementation habit. The numbers are written out because the paper must show where they come from. This is a design heuristic supported by component screening and implementation audit; it is not claimed as a proof of optimality over all possible rotation schedules. Complete candidate logs, when reported, belong to the artifact layer rather than to this design rationale paragraph.

Candidate family	Design concern	Status in this rationale
Repeated bridge template	reusable low-cost trail shape	rejected design pattern
Quarter-turn positions	byte/word-periodic windows	excluded from the first bridge
First bridge with odd sum	entrance-bypass resistance	retained after screening
Second bridge in a distinct family	tail feedback diversity	retained as secondary feedback

27.7 Bridge Selection Derivation

For the first bridge, the implementation order is $(22, 13)$, while the rotation-pair screening is read on the unordered pair $\{13, 22\}$. The key structural quantity is the bridge sum:

$$(22 + 13) \bmod 32 = 35 \bmod 32 = 3.$$

Because $32 = 2^5$, the condition

$$\gcd\big(((R_0 + R_1) \bmod 32), 32\big) = 1$$

is equivalent here to asking that $(R_0 + R_1) \bmod 32$ be odd. That matters because large common divisors would leave periodic rotation subspaces in which expressions such as

$$\Delta B \oplus \Delta \mathrm{B} \lll^{\mathrm{R_0+R_1}}$$

can cancel too cheaply. So gcd = 1 is only the first gate, not the whole story.

The actual first-bridge screening rule is stricter:

- exclude the quarter-turn positions $8, 16, 24$;
- prefer a moderate circular distance

  $$\mathbf{CircularDistance}(r_1, r_2) = \min\big((r_2 - r_1) \bmod 32,\ (r_1 - r_2) \bmod 32\big),$$

  with $5 \le \mathbf{CircularDistance}(r_1, r_2) \le 11$;
- prefer one rotation in $[1, 15]$ and the other in $[17, 31]$;
- verify the candidate through differential and linear MILP screening;
- reject any pair that induces an obvious zero-window injection bypass.

For the chosen unordered pair $\{13, 22\}$,

$$\mathbf{CircularDistance}(13, 22) = 9, \qquad (13 + 22) \bmod 32 = 3.$$

That is why the first bridge is kept. It is not kept because the numbers look pretty; it is kept because they pass the documented component-level engineering screening filter.

For the second bridge, the ordered pair is $(5, 25)$. Its basic quantities are

$$5 + 25 = 30, \qquad \gcd(30, 32) = 2, \qquad 25 - 5 = 20.$$

So the second bridge is a controlled secondary periodic structure, not a clone of the first bridge. Its circular distance is

$$\mathbf{CircularDistance}(5, 25) = \min(20, 12) = 12.$$

That places it outside the short-distance entry window used for the first bridge and keeps it from repeating the first bridge's template family.

The two bridges are intentionally different:

- first bridge: entrance feedback bridge intended to reduce low-cost entrance bypasses, gcd = 1, short circular distance;
- second bridge: controlled secondary feedback, gcd = 2, different template family.

The order matters. If the first bridge were also given gcd = 2, the trail search could start from a period-2 entrance subspace, and the later gcd = 1 choice would only be repairing an already weakened state. The chosen order is therefore:

$$\text{entrance feedback bridge} \rightarrow \text{controlled secondary feedback}.$$

The mirrored distance windows are also deliberate. The first bridge uses the short circular window 5–11; the second bridge uses the mirrored raw-distance window 20–26. These are not unrelated rules. They are the two sides of the same engineering distance policy. The pair $(5, 25)$ sits at the conservative edge of that mirrored window, is still coprime at the individual-word level, and does not repeat the $(22, 13)$ bridge family.

27.8 First Bridge: 22 and 13

The first bridge therefore follows directly from the derivation above. In implementation order it is

$$B\oplus = \mathrm{A} \lll^{22} \oplus \mathrm{RC}[4], \qquad A\oplus = \mathrm{B} \lll^{13}.$$

This is the entrance feedback bridge intended to reduce low-cost entrance bypasses.

27.9 Second Bridge: 5 and 25

The second bridge is selected in the same visible style. In implementation order it is

$$A\oplus = \mathrm{B} \lll^{5} \oplus \mathrm{RC}[9], \qquad B\oplus = \mathrm{A} \lll^{25}.$$

This is not a copy of the first bridge. It is a different feedback pattern for the tail half of the schedule.

The reason it is $(5, 25)$ rather than another $(22, 13)$-style copy is that the second bridge is meant to be a controlled secondary periodic structure:

$$5 + 25 = 30, \qquad \gcd(30, 32) = 2, \qquad \mathbf{CircularDistance}(5, 25) = 12.$$

That puts it outside the short entry window and prevents bridge-template isomorphism. In plain terms, the first bridge locks the entrance; the second bridge changes the template family instead of repeating it.

27.10 Why Not Reuse the Same Rotation Family

If the same bridge template is used twice, trail search can learn one low-cost route and reuse it. The construction therefore keeps the first bridge and the second bridge structurally distinct. The first one reduces low-cost entrance bypasses more aggressively; the second one is a different feedback pattern that prevents the two halves of the schedule from collapsing into the same trail family.

The design rule is therefore:

- first bridge: entrance feedback bridge, gcd= 1, short circular distance, and full screening against obvious periodic cancellation;
- second bridge: controlled secondary feedback, gcd= 2, different template family, and no reuse of the first bridge's low-cost trail shape.

27.11 Joint Boolean–Arithmetic Injection Semantics

Definition 43 (Joint-injection design identity)**.** The design identity used for the NeoAlzette V6.5 injection layer is

$$\boxed{\begin{aligned}\text{Joint Injection Layer} &= \text{Quadratic Boolean Front Half} \\ &\quad + \text{ Modular Addition Back Half (Two-Variable).}\end{aligned}}$$

Equivalently, the layer is a joint Boolean–arithmetic injection: the quadratic Boolean front half produces the coupled C/D material, while the modular-addition back half is the ordinary two-variable ARX addition whose non-constant addend is source-constrained by that same C/D material. This equation is a design identity, not a fourth component split: both halves belong to one source-bound injection layer.

The local injection maps are not detached XOR patches. Each oriented call is read as a joint Boolean–arithmetic object: a quadratic Boolean front half first extracts two coupled words from the source branch, and the following arithmetic back half consumes one CD-derived word as the non-constant operand of a standard two-variable modular addition. In other words, the implemented slice is not only

$$T \leftarrow T \oplus X_s(S),$$

but rather

$$T \leftarrow \big(T \oplus X_s(S)\big) \boxplus \big(Y_s(S) \oplus K_s\big),$$

where S is the source branch, T is the opposite target branch, X_s is the routed-XOR word, Y_s is the CD-derived addend word, and K_s is the public round constant used at that addition site. The modular addition itself is the usual two-variable ARX addition box. What is special is that its second operand is source-constrained by the same injection event that already supplied the XOR-routed word.

For the two implemented orientations this reading is concrete. If $(C_0, D_0) = \mathsf{Inj}^{\mathrm{NA}}_{B\to A}(B)$, then

$$\begin{aligned}
X_{B\to A}(B) &= \mathrm{B} \lll^{24} \oplus \mathrm{C}_0 \lll^{16} \oplus \mathrm{B} \lll^{8},\\
CD_0^{B\to A}(B) &= (C_0 \ll 2) \oplus (D_0 \gg 2),\\
CD_1^{B\to A}(B) &= (C_0 \gg 5) \oplus (D_0 \ll 5),\\
Y_{B\to A}(B) &= \mathrm{CD}_0^{\mathrm{B}\to\mathrm{A}}(\mathrm{B}) \lll^{31} \oplus \mathrm{CD}_1^{\mathrm{B}\to\mathrm{A}}(\mathrm{B}) \lll^{17}.
\end{aligned}$$

The subsequent arithmetic step is

$$A \leftarrow \big(A \oplus X_{B\to A}(B)\big) \boxplus \big(Y_{B\to A}(B) \oplus \mathrm{RC}[0]\big).$$

If $(C_1, D_1) = \mathsf{Inj}^{\mathrm{NA}}_{A\to B}(A)$, then

$$\begin{aligned}
X_{A\to B}(A) &= \mathrm{A} \ggg_{24} \oplus \mathrm{D}_1 \ggg_{16} \oplus \mathrm{A} \ggg_{8},\\
CD_0^{A\to B}(A) &= (C_1 \gg 3) \oplus (D_1 \ll 3),\\
CD_1^{A\to B}(A) &= (C_1 \ll 1) \oplus (D_1 \gg 1),\\
Y_{A\to B}(A) &= CD_0^{A\to B}(A) \oplus CD_1^{A\to B}(A),
\end{aligned}$$

and the subsequent arithmetic step is

$$B \leftarrow \big(B \oplus X_{A\to B}(A)\big) \boxplus \big(Y_{A\to B}(A) \oplus \mathrm{RC}[5]\big).$$

This realizes the design identity Joint Injection Layer = QFH+Two-Variable Modular-Addition Back Half. The phrase "back half" should not be read as a fourth component or as a new addition primitive; it is the ordinary two-variable modular-addition box whose operand is fed by the packed injection oracle.

27.11.1 Why the C/D outputs are staggered

The C/D words are deliberately not collapsed into one mixed injection word. On the $B \to A$ side, the XOR route uses C_0 directly, while the modular-addition operand uses two different shifted mixtures of C_0 and D_0. On the $A \to B$ side, the XOR route uses D_1 directly, while the modular-addition operand again uses two different shifted mixtures of C_1 and D_1. This staggering forces two downstream operations to see different projections of the same quadratic source computation.

If both downstream paths were fed only by a single shared value such as $C \oplus D$, then a trail search could try to collapse the injection into one free local output and pay

only for nullifying that one projection. The V6.5 schedule instead exposes two coupled channels: the XOR channel and the addend channel. They are distinct outputs, but they are not independent variables, because they share the same source word, diffusion masks, quadratic carriers, cross-coupling terms, and anti-coupling terms. A model that assigns them independent local differences or masks is therefore not modeling the implemented component.

27.11.2 Differential consequence

For either orientation $s \in \{B \to A, A \to B\}$, write the packed oracle as

$$\Phi_s(x) = \big(X_s(x), Y_s(x)\big) : \mathbb{F}_2^{32} \to \mathbb{F}_2^{64}.$$

Because the Quadratic Boolean Front Half has degree at most two, for a fixed source difference Δ_s its derivative is affine:

$$D_{\Delta_s}\Phi_s(z) = \Phi_s(z) \oplus \Phi_s(z \oplus \Delta_s) = M_{s,\Delta_s} z \oplus c_{s,\Delta_s}.$$

Thus the pair of downstream differences is not chosen freely. It must satisfy

$$(\delta_X, \delta_Y) \in c_{s,\Delta_s} + \operatorname{im} M_{s,\Delta_s}.$$

After the XOR route, the first input difference of the following modular addition is $\Delta_T \oplus \delta_X$, while the CD-derived addend difference is δ_Y. The local differential cost of a selected single characteristic is therefore of the form

$$W^{\Delta}_{\mathrm{joint},s} = \operatorname{rank}(M_{s,\Delta_s}) + W^{\Delta}_{\boxplus}(\Delta_T \oplus \delta_X, \delta_Y \mapsto \Delta_{T'}),$$

subject to the affine-image membership above. The addition box is standard; the coupling is in the fact that δ_X and δ_Y come from the same derivative image of Φ_s. Splitting them would admit differential paths whose XOR injection and modular-addition operand cannot be produced by any real source branch.

27.11.3 Linear consequence

The same coupling appears on the linear side. Let γ be the output mask of the following modular addition, and let λ, η be the two masks on its first and second input operands. The mask λ is also the mask applied to the XOR-routed word X_s, and η is also the mask applied to the CD-derived addend word Y_s. Therefore the packed output mask of the quadratic oracle is

$$\beta = \operatorname{pack}(\lambda, \eta).$$

For this β, the scalar quadratic phase $x \mapsto \langle \beta, \Phi_s(x) \rangle$ has an alternating Hessian contraction $S_{s,\beta}$. If α denotes the residual source-side mask that must be absorbed by the injector, then nonzero Walsh support requires

$$\alpha \in \alpha^0_{s,\beta} + \operatorname{im} S_{s,\beta},$$

and the local single-characteristic weight has the form

$$W^{L}_{\mathrm{joint},s} = \frac{1}{2}\operatorname{rank}(S_{s,\beta}) + W^{L}_{\boxplus}(\gamma; \lambda, \eta),$$

up to the deterministic phase contributed by public constants. The important point is that λ and η are simultaneously addition masks and packed injection-output masks. Allowing them to be chosen independently of the Walsh-support condition would create false low-weight linear characteristics.

27.11.4 Design summary

The joint injection layer is therefore a deliberate anti-bypass mechanism. The Quadratic Boolean Front Half creates a coupled pair of words. The XOR route uses one projection of that pair, and the modular-addition back half uses a staggered CD-derived projection as an addend. This forces any differential or linear trail to pay for one joint source event and one source-constrained addition event, rather than treating the XOR patch and the addition operand as two unrelated free variables.

27.12 Implementation-Derived Schedule Reading

The implemented schedule makes three design facts explicit. First, the source branch is preconditioned by fixed-public subtraction before it is consumed by an injector: $B \boxminus RC[1]$ feeds $\mathsf{Inj}^{\mathrm{NA}}_{B\to A}$, and after the first bridge $A \boxminus RC[6]$ feeds $\mathsf{Inj}^{\mathrm{NA}}_{A\to B}$. Second, each injector feeds two downstream paths: a zero-weight XOR routing path and a CD-derived modular-addition path. Third, the inverse schedule recomputes the same oriented injection from the restored control branch before undoing the corresponding modular addition and XOR routing. Thus the injector does not need to be invertible as a standalone map; reversibility comes from the triangular schedule around it.

This reading explains why the component is specified as a schedule rather than as a black-box formula. The security model must preserve the ordering

$$\begin{aligned}&\text{fixed-public subtraction} \to \text{oriented C/D injection}\\ &\to \text{routed XOR and CD-derived addition} \to \text{bridge feedback.}\end{aligned}$$

If this order is changed, the zero-window bypass argument and the inverse recomputation argument no longer refer to the implemented component.

27.13 Design Summary

The essential point is simple. The injection layer no longer behaves as an isolated XOR patch. It feeds both the XOR bridge and the modular-addition operand. The fixed subtraction anchors the bit order through borrow propagation, and the two bridges prevent the two halves of the state from remaining cleanly separable. This is the anti-bypass reading of the implemented round, and it is the reading that should be used in the paper text and in the corresponding MILP modeling.

27.14 Quadratic Boolean Front Half and C/D injection

The following material from the earlier LOPC draft is retained as the compact orientation to the Quadratic Boolean Front Half. Full derivations and the oracle/MILP bridge remain in Appendix A.

27.14.1 Quadratic Boolean Front Half and C/D injection

Definition 44 (Unified NAND/NOR carrier)**.** Let $\mu(u, v) = u \odot v$ be coordinatewise multiplication on $\mathbb{F}_2^{32}$. For $\varepsilon \in \mathbb{F}_2$, define

$$\Pi_\varepsilon(u, v) = \mathbf{1} \oplus \mu(u, v) \oplus \varepsilon(u \oplus v).$$

Thus $\Pi_0(u, v) = \neg(u \wedge v)$ and $\Pi_1(u, v) = \neg(u \vee v)$. For affine views $U(x) = Lx \oplus a$, $Z(x) = Mx \oplus b$, and $\ell(x) = Nx \oplus c$, define

$$Q_\varepsilon[U, Z; \ell](x) = \ell(x) \oplus \Pi_\varepsilon(U(x), Z(x)).$$

The C/D injection layer is built from these quadratic carriers, linear diffusion-mask views, cross-carrier products, anti-coupling products, shifts, rotations, and output XOR wiring.

Lemma 13 (Second-derivative skeleton of one carrier)**.** *For $Q = Q_\varepsilon[U, Z; \ell]$, its second Boolean derivative satisfies*

$$\mathcal{H}_Q(u, v) = Q(0) \oplus Q(u) \oplus Q(v) \oplus Q(u \oplus v) = Lu \odot Mv \oplus Lv \odot Mu.$$

Consequently, affine constants, affine output terms, complements, and NAND/NOR polarity move cosets and signs but do not change the bilinear skeleton.

Proof. All affine terms cancel in a second Boolean derivative. Expanding the coordinate product $(Lx + a)(Mx + b)$ at $x = 0, u, v, u \oplus v$ and XORing the four values removes constant and linear terms. The remaining terms are exactly $(Lu)(Mv)$ and $(Lv)(Mu)$ coordinatewise. The polarity term $\varepsilon(U \oplus Z)$ is affine, so it also cancels. □

Definition 45 (Packed joint injection map)**.** For either direction $s \in \{B \to A, A \to B\}$, define the packed map

$$\Phi_s : \mathbb{F}_2^{32} \longrightarrow \mathbb{F}_2^{64}, \qquad \Phi_s(x) = X_s(x) \,\|\, Y_s(x).$$

Here X_s is the routed-XOR word, and Y_s is the CD-derived addend word consumed by the following two-variable modular addition, up to the public round constant at that addition site. The Quadratic Boolean Front Half is the common source computation that produces the coupled C/D material; the modular-addition back half is the standard ARX addition whose second input is constrained by that material. Thus the relevant local object is the composition

$$T \longmapsto \big(T \oplus X_s(x)\big) \boxplus \big(Y_s(x) \oplus K_s\big),$$

not two unrelated local transitions. A model that splits X_s and Y_s into independent free variables is analyzing a different component.

Theorem 3 (Primal-dual rank principle for the joint injection)**.** *Assume a packed injection $H : \mathbb{F}_2^{32} \to \mathbb{F}_2^{64}$ has algebraic degree at most two. For every nonzero difference $\Delta \in \mathbb{F}_2^{32}$, the derivative is affine:*

$$D_\Delta H(x) = H(x) \oplus H(x \oplus \Delta) = M_\Delta x \oplus c_\Delta.$$

The supported output differences form the affine image $c_\Delta + \operatorname{im} M_\Delta$, and each supported difference has probability $2^{-\operatorname{rank} M_\Delta}$. Dually, for an output mask $\beta \in (\mathbb{F}_2^{64})^*$, the scalar quadratic phase $x \mapsto \langle \beta, H(x) \rangle$ has an alternating Hessian matrix S_β. Nonzero Walsh correlations have magnitude $2^{-\operatorname{rank}(S_\beta)/2}$, and the compatible input masks form one affine coset of $\operatorname{im} S_\beta$.

Proof. A first derivative of a degree-two vectorial Boolean map is affine, and its linear part is the bilinear slice of the vector Hessian. An affine map of rank r has image size 2^r, and each image point has 2^{32-r} preimages, giving probability 2^{-r}. For the linear statement, contracting the vector Hessian by β yields the alternating bilinear form of a scalar quadratic function. Standard quadratic Walsh theory gives zero outside the compatible affine mask coset and normalized magnitude $2^{-r/2}$ on that coset. □

Corollary 2 (Differential and linear injection constraints are different objects)**.** *The differential backend charges an image-rank cost $\operatorname{rank} M_\Delta$. The linear backend charges a Walsh-rank cost $\operatorname{rank}(S_\beta)/2$ and must also track signs and support cosets. These two rank objects are related by the same quadratic tensor, but they are not interchangeable.*

Remark 13 (Rank audit boundary). The QBF rank audit supports the design claim that the injection is a nontrivial coupled quadratic layer. It is component design evidence. It is not, by itself, a pseudorandom-function proof, a full S-box proof, or a LOPC composition proof.

28 Component B Rationale: XCR CSPRNG

28.1 Operator-level rationale for the XCR H/S arithmetic

The XCR specification introduced four XCR-specific arithmetic operators in Definitions 33, 34, and 37. This rationale uses those operators instead of the implementation helper names. The Hybrid8 H-round uses the byte-local lifted operators $\boxplus_8^{\mathrm{lift}}$ and $\boxminus_8^{\mathrm{lift}}$. The deployed S-round uses the zero-fill shadow word operators $\boxplus_{64}^{\mathrm{sh,zf}}$ and $\boxminus_{64}^{\mathrm{sh,zf}}$. The code-faithful helper names LCA8, LBS8, ShadowCarry64, and ShadowBorrow64 remain in the pseudocode only as implementation audit names.

Remark 14 (Boundary policy for the deployed XCR arithmetic). The deployed XCR instance does not use the cyclic or prime-lift experimental branches in its S-round. Its word-level shadow arithmetic is exactly the zero-fill family

$$\boxplus_{64}^{\mathrm{sh,zf}}, \qquad \boxminus_{64}^{\mathrm{sh,zf}}.$$

The cyclic and prime-lift variants remain design-space material and must not be silently substituted into the LOPC/XCR security discussion.

Definition 46 (Operational character of the XCR engineered operators)**.** The lifted byte operators

$$\boxplus_8^{\mathrm{lift}}, \qquad \boxminus_8^{\mathrm{lift}}$$

are byte-local bounded carry/borrow-source lifts. They create arithmetic-shaped byte diffusion by propagating source material across offsets $1, 2, 4$ inside one byte, but they export no inter-byte carry or borrow.

The shadow word operators

$$\boxplus_{64}^{\mathrm{sh,zf}}, \qquad \boxminus_{64}^{\mathrm{sh,zf}}$$

are word-level bounded source operators. They start from the XOR parity word and inject a bounded carry or borrow source back into that parity word through two zero-fill projections. In informal engineering language, the shadow addition/subtraction "sneaks in" a short carry/borrow effect, but the effect is deliberately bounded: it is not a full ripple-carry modular addition or subtraction and it has no wraparound.

The complement-duality, six-position locality, local-state-size, and fixed-operand triangularity statements are modeling consequences of these operators. They are used by the security and oracle layer, so their formal statements are placed in Section 34.1 rather than treated as design motivation.

Remark 15 (Engineering reading). The deployed XCR S-round does not call CPU addition or CPU subtraction. It calls the bounded Boolean operators $\boxplus_{64}^{\mathrm{sh,zf}}$ and $\boxminus_{64}^{\mathrm{sh,zf}}$. They look arithmetic-shaped because they are built around parity plus carry/borrow-source injection, but their source propagation is intentionally capped at the short zero-fill window described above.

XCR is stateful because LOPC uses it as an evolving round-material generator. The current wrapper uses two XCR instances during key-state generation, folds generated words into subkeys with ordinary modular add/sub and rotations, separates lifted-operator Hybrid8 H-rounds from zero-fill shadow-operator S-rounds, and treats the final XOR/rotation diffusion as part of the XCR step rather than an optional post-processing layer.

29 Component C Rationale: Key Switch Layer

29.1 MixLinearTransform rationale

`MixLinearTransform_Forward/Backward` gives a key-dependent affine/permutation switch. For each fixed generated key state it selects a reversible branch on a 64-bit lane, changing the active propagation geometry from round to round. It should therefore be analysed as keyed switching composed with the surrounding nonlinear and key-generation components, not as a standalone nonlinear S-box.

The layer is placed after the diagonal NeoAlzette calls because it acts on the repacked 64-bit lanes rather than on the internal 32-bit ARX-box branches. Its design role is not to add an independent nonlinear weight, but to change the affine coordinate system seen by the next round and to make the same NeoAlzette component appear under different lane geometries across generated key states.

29.2 RandomBitTweak rationale

RandomBitTweak injects a key-state-dependent bit-position constant after the switching layer. It is self-inverse and cheap. For XOR differentials a fixed XOR constant cancels, while for linear correlations it contributes a phase/sign term; the selected positions are driven by XCR-derived round material.

The tweak positions are derived from the same round-material state that selects the switching branch. The layer is therefore a keyed scheduling/tweak component attached to the post-ARX lane transform, not an independent randomness source and not an S-box replacement. Its rationale is to disturb overly regular fixed bit positions between rounds while keeping the inverse operation exact and inexpensive.

Part V
Security Evaluation

30 Claim Boundary

This paper distinguishes single-characteristic results, endpoint differential or linear hulls, component-oracle results, fixed-key or fixed-round experiments, and full-construction security claims. Current NeoAlzette MILP results are resource-limited single-trail incumbents unless solver optimality or a complete enumeration certificate is stated. Current XCR analysis is component-level and reduced-width diagnostic unless the complete 64-bit Hybrid8–Shadow composition is analysed. The Key Switch Layer is a keyed affine/permutation switch plus bit-position tweak, not an S-box.

31 Formal Security and Evidence Boundary of the Current LOPC Construction

This section replaces the earlier proof sketch with a source-audited proof layer. The goal is not to claim that every part of the construction is already proven as a complete cipher. The goal is sharper: define the exact mathematical objects implemented by the current C++ code, state the component theorems that actually follow from the NeoAlzette and Shadow-Boxed Arithmetic artifacts, and mark the remaining composition assumptions. In particular, MILP incumbents remain resource-limited

evidence unless the solver or enumeration certificate says complete, and XCR remains a stateful Hybrid8–Shadow generator whose full PRG security is an explicit future proof obligation.

32 Standalone ARX Algebraic Security Analysis

This section is deliberately separated from the XCR component proof and from the KeySwitchLayer proof. It collects the general ARX algebraic analysis layer: carry-state matrix transfer, explicit small-width matrix interpretation, and additive-differential matrix refinement. The component sections may instantiate these tools, but the tools themselves describe ARX algebra and evidence boundaries rather than proving the full LOPC construction.

32.1 Matrix-analytic ARX complexity: what is kept and what is not

The previous draft had a useful instinct–ARX behavior should be explained through state-dependent linear algebra–but it overstated what such matrices prove. The corrected presentation follows the matrix view of modular-addition correlations, the log-time differential analysis of ordinary addition, CDDT/CLAT-based ARX trail search, and the general Mealy-machine formula for chunked cryptanalytic properties [Wal03a, LM02, LLJW21, NBHW25]. The corrected statement is the following: ARX nonlinearity comes from carry/borrow state; cryptanalytic properties are evaluated by transfer matrices or finite-state machines; and full security is a trail/hull/composition question, not a consequence of the existence of a large matrix.

Definition 47 (Property transfer form)**.** Let F be a word operation scanned in chunks with hidden state s_i. A differential, linear, integral, or related cryptanalytic property that factors by chunks can be written in the form

$$bM_{n-1}M_{n-2}\cdots M_0 a,$$

where a encodes the initial hidden state, b encodes terminal observation, and M_i is the local transfer matrix determined by the visible difference, mask, or observation at coordinate i. This is the Mealy-machine/S-function view of word operations.

Lemma 14 (Ordinary modular addition is not globally affine)**.** *The map $(x, y) \mapsto x + y \bmod 2^n$ is not an affine map over $\mathbb{F}_2^{2n}$ for $n \geq 2$, but for a fixed carry state its next output bit is affine in the current input bits.*

Proof. For bit i, the output is $z_i = x_i \oplus y_i \oplus c_i$, while the next carry is

$$c_{i+1} = x_i y_i \oplus x_i c_i \oplus y_i c_i.$$

The monomial $x_i y_i$ proves non-affineness. Once c_i is fixed, the emitted output bit is affine, but the hidden carry update remains a nonlinear state transition. Thus exact analysis must track carry states or an equivalent transfer relation. □

Theorem 4 (Matrix reading of ARX complexity)**.** *The differential or linear analysis of an ARX component is exact only when the transfer object includes the carry/borrow state required by the operation. For ordinary two-variable modular addition, this recovers the Lipmaa–Moriai differential conditions and the Wallén/Schulte–Geers linear transfer view. For fixed-addend addition, the state space changes because the actual constant bits enter every local transition, as already emphasized by addend-dependency work and fixed-addend recurrence models [Miy98, Mac01, Twi26e, Twi26d]. For Shadow-Boxed Arithmetic, a different six-position local relation is required; ordinary modular-addition matrices do not apply by relabeling.*

Proof. Ordinary two-variable addition has one native carry chain in each execution. XOR-differential analysis compares two executions and reduces to a carry-pair relation; linear analysis evaluates signed sums over carry states. In the fixed-addend map $x \mapsto x + c$, the sampled probability space is one word rather than two words, and the bit pattern of c changes the transfer matrix at each coordinate. Shadow-Boxed Arithmetic replaces the unbounded native carry/borrow chain with two bounded source-propagation updates and a two-offset projection. Its coordinate depends on the current pair and the five previous pairs, so the exact binary-coordinate oracle uses a 1024-state two-variable history or a 32-state fixed-operand history. The transfer state is therefore operation-specific. □

Engineering interpretation. The matrix story is retained, but sharpened. It says why ARX is hard to analyse: local affine-looking output bits are coupled by hidden nonlinear carry/borrow state, fixed constants change the probability space, and trail/hull search must sum or bound many compatible internal routes. It does not say that a cipher is secure merely because its matrix representation is large.

32.2 Standalone ARX Algebraic Complexity Microscope: The 8-bit Matrix Tower

Disclaimer. This subsection is deliberately *not* a stand-alone security proof. It is an algebraic microscope. Its purpose is to translate ARX/SRX operations into exact linear-algebraic objects so that the reader can see where the combinatorial explosion enters when modular addition and subtraction are written without hiding the carry or borrow state. The 8-bit model is used because it is the largest scale at which the complete exact objects can still be displayed, rendered, and inspected by a human reader. The same structures underlie the 64-bit ARX-box layer and the 128-bit LOPC state, but at those widths the exact matrices are far beyond direct visualization.

Structure of the matrix tower. The microscope follows an atomize-before-compose route. We do not first hide the whole construction inside a black-box permutation F. We first tear apart the primitive operations themselves and then compose the resulting matrix objects.

1. **Level 0: arithmetic atomization.** XOR and rotations are ordinary linear or affine matrices, but modular addition and subtraction split into carry/borrow transition families. With one operand fixed, these families split again into constant-indexed singular subfamilies.

2. **Level 1: state permutation matrix.** After the atoms are fixed, every finite permutation can be represented by a state permutation matrix; round composition becomes matrix multiplication.

3. **Level 2: differential and linear induced operators.** The DDT and LAT are projections of the permutation matrix through XOR-translation and Fourier/Hadamard bases.

4. **Level 3: carry/borrow nested matrix products.** The visible permutation matrix is generated by selector-driven products of small carry/borrow transition matrices, or equivalently by piecewise-affine slices, Boolean Jacobian families, and lifted monomial matrices.

Together these layers form the *matrix tower* of an ARX/SRX design.

32.2.1 Level 0: arithmetic atomization before composition

The first reviewer-facing question is not whether an entire ARX/SRX round can be named F and then assigned a matrix. The sharper question is whether the individual source-level operations already have exact matrix meanings. The answer is yes, but addition and subtraction do not give a single visible-coordinate matrix. They give matrix *families* indexed by carry or borrow decisions, and fixed operands split those families according to the constant bit pattern.

Visible linear atoms. For a word in $\mathbb{F}_2^n$, rotations are ordinary permutation matrices:

$$X \lll r = P_{\lll r} X, \qquad X \ggg r = P_{\ggg r} X.$$

XOR of two words is linear,

$$X \oplus Y = \begin{bmatrix} I_n & I_n \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix},$$

and XOR by a public constant is affine. In homogeneous coordinates it becomes a genuine matrix,

$$\begin{bmatrix} X \oplus a \\ 1 \end{bmatrix} = \begin{bmatrix} I_n & a \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X \\ 1 \end{bmatrix}.$$

These atoms are easy: they stay inside the visible coordinate space.

Two-variable addition as an incidence object and as a branch permutation. The two-variable addition map

$$A_+(X, Y) = X \boxplus_n Y$$

can be written as a $2^n \times 2^{2n}$ incidence matrix if one records only the output word. This matrix is not square because two input words collapse to one output word. If the operation is embedded as a reversible branch update, for example

$$U_+(X, Y) = (X \boxplus_n Y,\ Y)$$

or

$$L_+(X, Y) = (X,\ X \boxplus_n Y),$$

then the corresponding matrix is a $2^{2n} \times 2^{2n}$ permutation matrix. Thus even a single addition can be written exactly as a matrix object, but the natural matrix dimension is exponential in the number of visible state bits.

At the bit level, the same operation is generated by four carry-state matrices. Put

$$E_0 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \qquad E_1 = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \qquad I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Then the addition carry atom is

$$T^+_{00} = E_0, \qquad T^+_{01} = I_2, \qquad T^+_{10} = I_2, \qquad T^+_{11} = E_1. \tag{1}$$

The matrices E_0 and E_1 are rank-one idempotent projection operators: they map both incoming carry states to carry 0 or carry 1, respectively. They are singular. The identity cases preserve the carry state. Thus a two-variable addition alternates, bit by bit, between full-rank carry propagation and singular carry forcing.

Two-variable subtraction as a distinct borrow atom. Subtraction is not addition with a sign painted on top. For

$$B_-(X,Y) = X \boxminus_n Y,$$

the borrow update uses the bit-local family

$$T^-_{00} = I_2, \qquad T^-_{01} = E_1, \qquad T^-_{10} = E_0, \qquad T^-_{11} = I_2. \tag{2}$$

The singular operators occur in different locations from addition. More importantly, the two inputs are not symmetric. The update for $X \boxminus_n Y$ uses $T^-_{x_i y_i}$, while the update for $Y \boxminus_n X$ uses $T^-_{y_i x_i}$. Swapping operands changes which singular projection is selected.

Fixed-addend addition: constant-indexed splitting. Now freeze one operand and consider $X \mapsto X \boxplus_n K$. The local atom is no longer the universal two-variable family $T^+_{x_i y_i}$; it becomes

$$T^+_{x_i K_i}.$$

Therefore the matrix family splits according to the constant bit:

K_i	$x_i = 0$	$x_i = 1$
0	E_0	I_2
1	I_2	E_1

(3)

A zero bit of the constant exposes the subfamily $\{E_0, I_2\}$, while a one bit exposes $\{I_2, E_1\}$. Thus the bit pattern of K is not a harmless parameter. It indexes which singular carry-forcing operators can occur at each coordinate. Long zero-runs and long one-runs of the constant create different carry-transition landscapes. This is the matrix version of the fixed-public/addend-dependence warning used later by the fixed-addend oracles.

Fixed subtraction splits in two different ways. For $X \mapsto X \boxminus_n K$, the local atom is $T^-_{x_i K_i}$, hence

K_i	$x_i = 0$	$x_i = 1$
0	I_2	E_0
1	E_1	I_2

(4)

For $X \mapsto K \boxminus_n X$, the local atom is instead $T^-_{K_i x_i}$, hence

K_i	$x_i = 0$	$x_i = 1$
0	I_2	E_1
1	E_0	I_2

(5)

The two tables differ by exchanging E_0 and E_1. This is the exact algebraic reason why fixed-subtrahend and fixed-minuend subtraction cannot be merged into one generic fixed-constant subtraction box. The matrix family itself splits in a direction-sensitive way.

Theorem 5 (Constant-operand matrix splitting)**.** *For fixed-operand modular arithmetic, the local carry/borrow transfer family is a constant-indexed bundle of full-rank and rank-deficient matrices. Fixed-addend addition splits according to Eq.* (3)*; fixed-subtrahend subtraction splits according to Eq.* (4)*; fixed-minuend subtraction splits according to Eq.* (5)*. In particular, fixed constants cannot be averaged away without changing the transition object.*

Proof. Equations (1) and (2) list the complete two-variable local families. Freezing an operand simply restricts the first or second index of the relevant family to the fixed constant bit K_i. Substituting $K_i = 0$ and $K_i = 1$ gives the three displayed splitting tables. Since E_0 and E_1 have rank one while I_2 has rank two, each fixed-operand arithmetic operation is a constant-indexed mixture of full-rank propagation and singular projection. The two subtraction placements differ because $T^-_{01} = E_1$ and $T^-_{10} = E_0$ are not equal. □

Composition after atomization. Only after this split do we compose an ARX/SRX round. A round matrix family is not merely P_F for an unexplained black-box F; it is generated from products of visible linear atoms and constant-indexed arithmetic matrix families:

$$\mathcal{M}_{\mathrm{round}} \subseteq \left\{ \prod_t \widetilde{X}(a_t)\, \widetilde{R}_{r_t}\, \widetilde{A}_{\sigma_t}(K^{(t)}, h^{(t)}) \;:\; h^{(t)} \in \mathcal{H}_{\sigma_t, K^{(t)}} \right\}, \tag{6}$$

where $\sigma_t \in \{\boxplus, \boxminus\}$, $h^{(t)}$ is the selected carry or borrow path, and $\mathcal{H}_{\sigma_t, K^{(t)}}$ is the constant-dependent path set. This is the intended order of the microscope: first split the arithmetic atoms, then multiply them into the round, and only then project the result into differential or linear evidence.

32.2.2 Worked 8-bit matrix wall: one ARX/SRX-looking line by hand

The previous paragraphs explain that addition and subtraction split into carry/borrow matrix families. We now force one ordinary-looking 8-bit line onto the page and compute it by hand. The point is pedagogical: a reader who does not yet have ARX/SRX intuition can see that the source expression is short only because the carry chain is hidden.

Consider

$$z = f_k(x, y) = x \boxplus_8 \big((y \lll 2) \oplus k\big). \tag{7}$$

For the worked instance take

$$x = \mathtt{10110110}_2 = \mathtt{0xb6}, \qquad y = \mathtt{01101001}_2 = \mathtt{0x69}, \qquad k = \mathtt{11001010}_2 = \mathtt{0xca}.$$

Unless stated otherwise, the column vectors below are written in little-endian bit order, so that the first coordinate is bit 0. Thus

$$\mathbf{x} = \begin{bmatrix} 0\\1\\1\\0\\1\\1\\0\\1 \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} 1\\0\\0\\1\\0\\1\\1\\0 \end{bmatrix}, \qquad \mathbf{k} = \begin{bmatrix} 0\\1\\0\\1\\0\\0\\1\\1 \end{bmatrix}.$$

The rotation is already a visible binary matrix. The 8-bit left rotation by two positions is the permutation matrix

$$R_{\lll 2} = \begin{bmatrix} 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0 \end{bmatrix}.$$

Therefore

$$R_{\lll 2}\mathbf{y} = \begin{bmatrix} 1\\0\\1\\0\\0\\1\\0\\1 \end{bmatrix} \quad \Longleftrightarrow \quad y \lll 2 = \mathtt{10100101}_2 = \mathtt{0xa5}.$$

The constant XOR is affine; in homogeneous coordinates it is a binary matrix. The affine map $u \mapsto u \oplus k$ becomes

$$\widetilde{X}(\mathbf{k}) = \left[\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right].$$

The affine multiplication is now completely explicit:

$$\begin{aligned} \begin{bmatrix} \mathbf{q} \\ 1 \end{bmatrix} &= \widetilde{X}(\mathbf{k}) \begin{bmatrix} R_{\lll 2}\mathbf{y} \\ 1 \end{bmatrix} \\ &= \left[\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix}. \end{aligned} \tag{8}$$

Therefore

$$\mathbf{q} = (R_{\lll 2}\mathbf{y}) \oplus \mathbf{k} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \Longleftrightarrow \quad q = (y \lll 2) \oplus k = \texttt{01101111}_2 = \texttt{0x6f}. \tag{9}$$

At this point the high-level code has only performed a rotation and a constant XOR. The rotation is linear and the constant XOR is affine after homogenisation. This does *not* mean that the next symbol can be simulated by ordinary matrix addition or subtraction over the original eight bit coordinates. Pure XOR-linear matrix arithmetic can express maps such as $x \mapsto Rx$, $(x, y) \mapsto x \oplus y$, and $x \mapsto x \oplus k$; it cannot express $x \boxplus_8 q$ as one fixed 8×16 binary linear matrix in (x, q). For instance, already

$$(x \boxplus_8 q)_1 = x_1 \oplus q_1 \oplus x_0 q_0,$$

which contains the product term $x_0 q_0$. Therefore the matrix microscope must mix ordinary matrices with a small amount of Boolean algebra: the data bits first select carry-state matrices, and those selected matrices are then multiplied in the usual way. The explosion starts precisely at this boundary.

The addition is a product of selected binary carry matrices. Using the addition atoms from Eq. (1), the selected sequence for $x \boxplus_8 q$ is

i	x_i	q_i	$T^+_{x_i q_i}$	incoming e_{c_i}
0	0	1	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix}$
1	1	1	$\begin{bmatrix}0&0\\1&1\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix}$
2	1	1	$\begin{bmatrix}0&0\\1&1\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$
3	0	1	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$
4	1	0	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$
5	1	1	$\begin{bmatrix}0&0\\1&1\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$
6	0	1	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$
7	1	0	$\begin{bmatrix}1&0\\0&1\end{bmatrix}$	$\begin{bmatrix}0\\1\end{bmatrix}$

The table is not merely notation. The carry state is obtained by literal matrix-vector multiplication, one bit at a time:

$$
\begin{aligned}
e_{c_0} &= \begin{bmatrix}1\\0\end{bmatrix},\\
e_{c_1} &= T^+_{01} e_{c_0} = \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix},\\
e_{c_2} &= T^+_{11} e_{c_1} = \begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix},\\
e_{c_3} &= T^+_{11} e_{c_2} = \begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix},\\
e_{c_4} &= T^+_{01} e_{c_3} = \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix},\\
e_{c_5} &= T^+_{10} e_{c_4} = \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix},\\
e_{c_6} &= T^+_{11} e_{c_5} = \begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix},\\
e_{c_7} &= T^+_{01} e_{c_6} = \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix},\\
e_{c_8} &= T^+_{10} e_{c_7} = \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}.
\end{aligned}
\tag{10}
$$

The same information, written as a single carry product, is

$$
\begin{aligned}
e_{c_8} &= T^+_{10}T^+_{01}T^+_{11}T^+_{10}T^+_{01}T^+_{11}T^+_{11}T^+_{01}e_0\\
&= \begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}0&0\\1&1\end{bmatrix}\begin{bmatrix}1&0\\0&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}.
\end{aligned}
\tag{11}
$$

The rightmost factor corresponds to bit 0. Even in this 8-bit example, one visible addition has become an eight-factor matrix product whose factors are selected by the data bits. In a 64-bit word addition, the highest bit is controlled by a product of sixty-three such selected matrices.

Recovering the output vector from the carry states. Let $\pi_1^T = [0\ 1]$. Each output bit is

$$z_i = x_i \oplus q_i \oplus \pi_1^T e_{c_i}.$$

The carry states produced by the previous product are

$$e_{c_0} = \begin{bmatrix}1\\0\end{bmatrix},\quad e_{c_1} = \begin{bmatrix}1\\0\end{bmatrix},\quad e_{c_2} = e_{c_3} = e_{c_4} = e_{c_5} = e_{c_6} = e_{c_7} = e_{c_8} = \begin{bmatrix}0\\1\end{bmatrix}.$$

Equivalently, the input-carry vector used in the eight output bits is

$$\mathbf{c} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

Now the output word is recovered by an explicit vector XOR over $\mathbb{F}_2$:

$$\mathbf{z} = \mathbf{x} \oplus \mathbf{q} \oplus \mathbf{c} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}. \tag{12}$$

This last line is deliberately written as XOR-vector arithmetic. The non-XOR part has already been paid for by the carry-state matrix product; without the carry vector $\mathbf{c}$, the expression $\mathbf{x} \oplus \mathbf{q}$ would describe XOR, not modular addition. Therefore

i	x_i	q_i	c_i	$z_i = x_i \oplus q_i \oplus c_i$
0	0	1	0	1
1	1	1	0	0
2	1	1	1	1
3	0	1	1	0
4	1	0	1	0
5	1	1	1	1
6	0	1	1	0
7	1	0	1	0

and hence

$$\mathbf{z} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \iff z = 00100101_2 = \texttt{0x25}. \tag{13}$$

The source-level computation is simply

$$\texttt{0xb6} \boxplus_8 \big((\texttt{0x69} \lll 2) \oplus \texttt{0xca}\big) = \texttt{0xb6} \boxplus_8 \texttt{0x6f} = \texttt{0x25}.$$

The matrix-level computation exposes what the one-line expression hides: rotation is one visible permutation matrix, constant XOR is one affine matrix, and the addition is a selected product of singular and full-rank carry matrices.

Where the constant-dependent splitting appears in this example. Symbolically, the second addend bit is

$$q_i = y_{i-2 \bmod 8} \oplus k_i.$$

Thus the local addition atom is

$$T^{+}_{x_i,\, y_{i-2} \oplus k_i}.$$

For $k_i = 0$, the bit uses the ordinary y_{i-2}-indexed subfamily. For $k_i = 1$, the subfamily is flipped. The public constant therefore does not merely add decoration to the expression; it changes which identity or singular projection matrix is selected at that bit position. This is the hand-calculated version of the constant-operand splitting theorem above.

32.2.3 Level 1: state permutation matrix

Let $F : \mathbb{F}_2^n \to \mathbb{F}_2^n$ be a permutation and index the elements of $\mathbb{F}_2^n$ as $x^{(0)}, x^{(1)}, \ldots, x^{(2^n-1)}$. Define the state permutation matrix P_F by

$$(P_F)_{j,i} = 1 \quad \Longleftrightarrow \quad x^{(j)} = F\big(x^{(i)}\big). \tag{14}$$

This matrix is a 0/1 incidence matrix with exactly one nonzero entry in every row and every column. When it is used to represent the action on bit vectors, it may be read over $\mathbb{F}_2$. When it is used for counting differential pairs, traces and sums are taken over the integers. This distinction matters: counting in a DDT is not counting modulo two.

If

$$F = F_r \circ F_{r-1} \circ \cdots \circ F_1,$$

then

$$P_F = P_{F_r} P_{F_{r-1}} \cdots P_{F_1}. \tag{15}$$

This is the first shock of the matrix view: **function composition is matrix multiplication**. The source-level operations of an ARX round do not merely sit beside each other; after composition, they define a single permutation matrix on the whole state space.

For $n = 8$, the matrix P_F has size 256×256. For a 64-bit ARX-box layer, the analogous state matrix has $2^{64} \times 2^{64}$ positions, i.e. $2^{128} \approx 3.40 \times 10^{38}$ entries. For the full 128-bit LOPC block state, the permutation matrix has $2^{128} \times 2^{128}$ positions. The object is mathematically well-defined, but it is not human-printable. The 8-bit model is therefore not a reduced security claim; it is the last practical viewing window for the exact matrix object.

Example: fixed-addend addition. For an 8-bit fixed addend k, the map

$$F_k(x) = x \boxplus_8 k$$

is a permutation. The column indexed by x in P_{F_k} has its single 1 in the row indexed by $x \boxplus_8 k$. Even this simple operation produces a global permutation of the 256 input states. When rotations and XORs are composed around it, the round matrix becomes a product of such permutation matrices.

32.2.4 Level 2: differential and linear induced operators

Once P_F is accepted as the state object, the DDT and LAT are no longer separate empirical tables. They are induced operators obtained by viewing the same permutation through different bases.

XOR-translation matrices. For $\alpha \in \mathbb{F}_2^n$, define the XOR-translation matrix T_α by

$$(T_\alpha)_{x,y} = 1 \quad \Longleftrightarrow \quad x = y \oplus \alpha. \tag{16}$$

It is again a permutation matrix and $T_\alpha^{-1} = T_\alpha$.

DDT as an integer trace. The number of inputs satisfying

$$F(x \oplus \alpha) \oplus F(x) = \beta$$

is

$$\#\{x : F(x \oplus \alpha) \oplus F(x) = \beta\} = \mathrm{Tr}_{\mathbb{Z}}\big(T_\beta P_F T_\alpha P_F^{-1}\big), \tag{17}$$

where the trace is the ordinary integer trace of the 0/1 incidence matrix product.

Proof. A diagonal entry of $T_\beta P_F T_\alpha P_F^{-1}$ is one exactly when the corresponding state z is fixed by this composition. Write $z = F(x)$. Applying the product from right to left gives

$$z \xmapsto{P_F^{-1}} x \xmapsto{T_\alpha} x \oplus \alpha \xmapsto{P_F} F(x \oplus \alpha) \xmapsto{T_\beta} F(x \oplus \alpha) \oplus \beta.$$

The final value equals $z = F(x)$ if and only if $F(x \oplus \alpha) \oplus F(x) = \beta$. Summing the diagonal entries counts exactly those inputs. □

Thus the XOR-differential probability is

$$\mathrm{DP}_F^{\oplus}(\alpha \to \beta) = 2^{-n} \operatorname{Tr}_{\mathbb{Z}}\big(T_\beta P_F T_\alpha P_F^{-1}\big). \tag{18}$$

This is an exact DDT formula, but it is exact at the cost of a $2^n \times 2^n$ matrix object.

LAT as Hadamard conjugation. Let $H_n \in \{-1, 1\}^{2^n \times 2^n}$ be the Sylvester–Hadamard matrix

$$H_n[a, x] = (-1)^{a \cdot x}.$$

The raw Walsh/LAT entry is

$$W_F(a, b) = \sum_{x \in \mathbb{F}_2^n} (-1)^{a \cdot x \oplus b \cdot F(x)}.$$

With the above column convention,

$$W_F = H_n P_F H_n^T. \tag{19}$$

Since $H_n^T = H_n$, this is often written as $H_n P_F H_n$. The normalized correlation is

$$C_F(a, b) = 2^{-n} W_F(b, a), \tag{20}$$

up to the chosen row/column convention for input and output masks.

Proof. The (b, a)-entry of $H_n P_F H_n^T$ is

$$\sum_{y,x} (-1)^{b \cdot y} (P_F)_{y,x} (-1)^{a \cdot x} = \sum_x (-1)^{a \cdot x \oplus b \cdot F(x)}.$$

This is exactly the raw Walsh coefficient. □

The LAT is therefore the same state permutation matrix viewed through the Fourier basis of the XOR group.

32.2.5 Level 3: inner machinery – carry/borrow nested matrix products

The permutation matrix P_F does not appear from nowhere. For ARX/SRX primitives it is built from bit-local deterministic state updates. XOR and rotations are visible linear maps; addition and subtraction require hidden carry or borrow state.

XOR and rotations as ordinary matrices. Left and right rotations are $n \times n$ permutation matrices

$$(P_{\lll r})_{j,i} = 1 \iff j \equiv i + r \pmod{n}, \qquad (P_{\ggg r})_{j,i} = 1 \iff j \equiv i - r \pmod{n}.$$

Thus $X \lll r = P_{\lll r} X$ and $X \ggg r = P_{\ggg r} X$. XOR of two words is linear:

$$X \oplus Y = \begin{bmatrix} I_n & I_n \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}.$$

XOR by a fixed constant a is affine and can be homogenized as

$$\widetilde{X}(a) = \begin{bmatrix} I_n & a \\ 0^T & 1 \end{bmatrix}, \qquad \begin{bmatrix} x \oplus a \\ 1 \end{bmatrix} = \widetilde{X}(a) \begin{bmatrix} x \\ 1 \end{bmatrix}. \tag{21}$$

Over $\mathbb{F}_2$, $\widetilde{X}(a)^2 = I$.

Carry transition matrices for addition. For bit addition, the carry update is

$$c_{i+1} = \mathrm{Maj}(x_i, y_i, c_i) = x_i y_i \oplus x_i c_i \oplus y_i c_i.$$

Encode $c_i = 0$ as $e_0 = (1,0)^T$ and $c_i = 1$ as $e_1 = (0,1)^T$. For fixed (x_i, y_i), the carry update is linear in the one-hot carry vector:

$$e_{c_{i+1}} = T^+_{x_i y_i} e_{c_i},$$

where

$$T^+_{00} = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \quad T^+_{01} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad T^+_{10} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad T^+_{11} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}. \tag{22}$$

Starting from $c_0 = 0$,

$$e_{c_i} = T^+_{x_{i-1} y_{i-1}} T^+_{x_{i-2} y_{i-2}} \cdots T^+_{x_0 y_0} e_0. \tag{23}$$

The sum bit is

$$s_i = x_i \oplus y_i \oplus \pi_1^T e_{c_i}, \qquad \pi_1^T = \begin{bmatrix} 0 & 1 \end{bmatrix}.$$

For 8-bit addition this expands to the matrix wall

$$\begin{aligned}
s_0 &= x_0 \oplus y_0, \\
s_1 &= x_1 \oplus y_1 \oplus \pi_1^T T^+_{x_0 y_0} e_0, \\
s_2 &= x_2 \oplus y_2 \oplus \pi_1^T T^+_{x_1 y_1} T^+_{x_0 y_0} e_0, \\
s_3 &= x_3 \oplus y_3 \oplus \pi_1^T T^+_{x_2 y_2} T^+_{x_1 y_1} T^+_{x_0 y_0} e_0, \\
s_4 &= x_4 \oplus y_4 \oplus \pi_1^T T^+_{x_3 y_3} T^+_{x_2 y_2} T^+_{x_1 y_1} T^+_{x_0 y_0} e_0, \\
s_5 &= x_5 \oplus y_5 \oplus \pi_1^T T^+_{x_4 y_4} T^+_{x_3 y_3} T^+_{x_2 y_2} T^+_{x_1 y_1} T^+_{x_0 y_0} e_0, \\
s_6 &= x_6 \oplus y_6 \oplus \pi_1^T T^+_{x_5 y_5} T^+_{x_4 y_4} T^+_{x_3 y_3} T^+_{x_2 y_2} T^+_{x_1 y_1} T^+_{x_0 y_0} e_0, \\
s_7 &= x_7 \oplus y_7 \oplus \pi_1^T T^+_{x_6 y_6} T^+_{x_5 y_5} T^+_{x_4 y_4} T^+_{x_3 y_3} T^+_{x_2 y_2} T^+_{x_1 y_1} T^+_{x_0 y_0} e_0.
\end{aligned} \tag{24}$$

The visual content is the important part: the high bit of an 8-bit addition already contains a product of seven data-selected 2×2 carry matrices. In 64-bit width, the corresponding highest bit contains a product of 63 such factors.

Borrow transition matrices for subtraction. For subtraction $x \boxminus_n y$, the borrow update is

$$b_{i+1} = ((1 \oplus x_i)y_i) \oplus ((1 \oplus x_i \oplus y_i)b_i).$$

The one-hot borrow matrices are

$$T_{00}^- = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad T_{01}^- = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, \quad T_{10}^- = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \quad T_{11}^- = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{25}$$

Subtraction is therefore not a signed relabeling of addition. It has its own borrow-state transition family. Moreover, $x \boxminus_n k$ and $k \boxminus_n x$ are different fixed-operand objects because the borrow generation rule is not symmetric in its two operands.

Piecewise-affine exact matrix families. Fix an addend y and a carry prefix $m = (c_0, c_1, \ldots, c_{n-1})$ with $c_0 = 0$. Let

$$\Omega_m(y) = \{x \in \mathbb{F}_2^n : \text{ adding } x \text{ and } y \text{ produces carry prefix } m\}.$$

On this cell, the carry bits are constants and

$$x \boxplus_n y = x \oplus y \oplus m.$$

In homogeneous coordinates,

$$\forall x \in \Omega_m(y): \quad \begin{bmatrix} x \boxplus_n y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} I_n & y \oplus m \\ 0^T & 1 \end{bmatrix}}_{\widetilde{A}_{\boxplus}(y,m)} \begin{bmatrix} x \\ 1 \end{bmatrix}. \tag{26}$$

The same construction holds for subtraction using the borrow prefix and a matrix $\widetilde{A}_{\boxminus}(y, b)$. Thus a single source-level plus sign is an exact selector over carry-slice affine matrices.

Theorem 6 (Piecewise matrix exactness)**.** *The maps $x \mapsto x \boxplus_n y$ and $x \mapsto x \boxminus_n y$ are exact piecewise-affine binary matrix families over the partitions induced by their carry and borrow paths.*

Proof. On each partition cell, the hidden carry or borrow prefix is fixed. The visible output is therefore the input XORed with a fixed word, hence affine. The cells are disjoint and cover the domain because every input produces exactly one carry or borrow path. □

This is the second explosion: what looks like one arithmetic instruction in code becomes a selector over exponentially many exact affine matrices.

Boolean Jacobian: exact local linearization. For fixed y, define $F_y(x) = x \boxplus_n y$. The Boolean derivative of coordinate $(F_y)_i$ with respect to x_j is

$$\frac{\partial (F_y)_i}{\partial x_j}(x) = (F_y)_i(x) \oplus (F_y)_i(x \oplus e_j),$$

and the Jacobian $J_{F_y}(x)$ has entries

$$[J_{F_y}(x)]_{i,j} = \frac{\partial (F_y)_i}{\partial x_j}(x).$$

Lemma 15 (Jacobian structure of fixed-addend arithmetic)**.** *For every fixed y, the Boolean Jacobian of $x \mapsto x \boxplus_n y$ is lower triangular with all diagonal entries equal to one. Hence* $\operatorname{rank} J_{F_y}(x) = n$ *for every x. The same statement holds for $x \mapsto x \boxminus_n y$.*

Proof. The output bit $(F_y)_i = x_i \oplus y_i \oplus c_i$ depends only on $x_0, \ldots, x_i$, because c_i is produced by lower bits. Thus derivatives with respect to x_j for $j > i$ vanish. Since c_i is independent of x_i, the diagonal derivative is one. The Jacobian is unit lower triangular and therefore full rank. The subtraction case has the same prefix-dependence structure through borrows. □

This is not a differential-probability statement. It says that fixed-addend add/sub are locally full-rank in the Boolean derivative sense, while their global nonlinear behavior lives in how the Jacobian changes from point to point.

Lifted monomial matrix: exact global algebraic representation. Let $z = (x, y) \in \mathbb{F}_2^{2n}$. Let $\Phi_D(z)$ be the column vector of all square-free monomials in the bits of z of degree at most D, including the constant monomial. Every Boolean polynomial of degree at most D is a linear form in $\Phi_D(z)$.

For $n = 8$, the output bits of $x \boxplus_8 y$ have algebraic degree at most eight in the 16 visible variables. Hence there exists a binary matrix $B_{\boxplus}^{(8)}$ such that

$$x \boxplus_8 y = B_{\boxplus}^{(8)} \Phi_8(x, y). \tag{27}$$

Here

$$\dim \Phi_8 = \sum_{d=0}^{8} \binom{16}{d} = 39203.$$

Thus even the 8-bit two-variable lifted exact representation already uses an 8×39203 Boolean coefficient matrix. At 64-bit two-variable width, the corresponding degree-at-most-64 feature space has dimension

$$\sum_{d=0}^{64} \binom{128}{d},$$

which is already on the order of 2^{127}. The lifted representation is exact and global, but it pays for exactness by inflating the coordinate space to near the full Boolean-function space.

Theorem 7 (Three exact matrix views of ARX arithmetic)**.** *For modular addition and subtraction over $\mathbb{F}_2^n$, the following matrix views are exact:*

1. *carry/borrow-slice affine matrices in homogeneous coordinates;*
2. *Boolean Jacobians at a fixed point;*
3. *lifted monomial-coordinate matrices for the ANF representation.*

A single ordinary $n \times n$ binary matrix on the original visible coordinates cannot represent addition or subtraction globally in general.

Proof. The slice model follows by fixing the carry or borrow path. The Jacobian model follows from Boolean differentiation. The lifted model follows from the algebraic normal form of Boolean functions. The impossibility of a single visible-coordinate matrix follows from the monomial $x_i y_i$ in the carry update for addition, and the corresponding borrow monomials for subtraction. □

32.2.6 A fully explicit 8-bit ARX slice product

Let $k, sk \in \mathbb{F}_2^8$, put $\Delta = k \oplus sk$, and consider the toy ARX layer

$$E(x) = (((x \boxplus_8 \Delta) \oplus k) \ggg 2) \oplus ((k \boxplus_8 sk) \lll 6).$$

Fix a carry path m for $x \boxplus_8 \Delta$. On the carry cell $\Omega_m(\Delta)$,

$$\begin{bmatrix} E(x) \\ 1 \end{bmatrix} = \widetilde{E}_m \begin{bmatrix} x \\ 1 \end{bmatrix},$$

with

$$\widetilde{E}_m = \underbrace{\widetilde{X}((k \boxplus_8 sk) \lll 6)}_{\text{final constant xor}} \underbrace{\begin{bmatrix} P_{\ggg 2} & 0 \\ 0^T & 1 \end{bmatrix}}_{\text{rotation}} \underbrace{\widetilde{X}(k)}_{\text{key xor}} \underbrace{\widetilde{A}_{\boxplus}(\Delta, m)}_{\text{carry-slice addition}} . \tag{28}$$

This is the literal matrix multiplication form of the source expression. On each carry slice, the visible code has become a product of explicit 9×9 matrices.

Theorem 8 (Slice-wise invertibility of the 8-bit ARX product)**.** *For every admissible carry path m, the affine representative $\widetilde{E}_m$ is invertible as a homogeneous binary matrix.*

Proof. The matrices $\widetilde{X}(\cdot)$ are self-inverse, the rotation matrix is a permutation matrix, and $\widetilde{A}_{\boxplus}(\Delta, m)$ is an XOR-by-constant affine matrix on the fixed slice. The product of invertible matrices is invertible. □

The global map is not a single one of these products. It is the union of all slice products selected by the carry path generated by the input. In 8-bit width there are at most 2^7 carry prefixes because $c_0 = 0$; in 64-bit width the analogous worst-case count is 2^{63}. Multiple additions and subtractions multiply the number of compatible path combinations.

32.2.7 Random-walk extension: probability over the matrix family

The slice matrices are deterministic once the carry or borrow path is fixed. For cryptanalysis, the path is normally induced by an input distribution, a difference condition, a mask condition, or round material. We therefore attach a probability measure to the family of exact slice matrices.

For addition, define

$$G_i = x_i y_i, \qquad P_i = x_i \oplus y_i, \qquad K_i = (1 \oplus x_i)(1 \oplus y_i).$$

The carry update is

$$c_{i+1} = G_i \oplus P_i c_i.$$

Thus

$$\Pr[c_{i+1} = 1] = \Pr[G_i = 1] + \Pr[P_i = 1 \wedge c_i = 1].$$

If the current bit pair is independent of the previous carry state, this becomes

$$p_{i+1} = \lambda_i^+ + \lambda_i^0 p_i, \qquad p_i = \Pr[c_i = 1], \quad \lambda_i^+ = \Pr[G_i = 1], \quad \lambda_i^0 = \Pr[P_i = 1].$$

Under an unbiased independent baseline,

$$\lambda_i^+ = \frac{1}{4}, \qquad \lambda_i^0 = \frac{1}{2}, \qquad p_{i+1} = \frac{1}{4} + \frac{1}{2} p_i,$$

so

$$p_i = \frac{1}{2}(1 - 2^{-i}).$$

For dependent sources one must use the conditional expression $\Pr[P_i = 1 \wedge c_i = 1]$; the model does not silently assume Markov independence.

For subtraction, use

$$G_i^- = (1 \oplus x_i)y_i, \qquad P_i^- = 1 \oplus x_i \oplus y_i,$$

with

$$b_{i+1} = G_i^- \oplus P_i^- b_i.$$

An ARX/SRX layer with T arithmetic events is therefore a random product over a finite family of exact matrices:

$$\mathbf{M}_{\mathrm{ARX}}^{(T)} = \prod_{t=1}^{T} \widetilde{X}(a_t)\widetilde{R}_{r_t}\widetilde{A}_{\sigma_t}(y_t, h^{(t)}), \tag{29}$$

where $\sigma_t \in \{\boxplus, \boxminus\}$, and $h^{(t)}$ is the carry or borrow path selected at arithmetic event t. The randomness is not in the algebra of a selected matrix. The randomness is in *which* exact matrix product is selected.

32.2.8 Explosion table and visual diagnostics

The exact state-permutation object grows as follows.

Table 3: The 8-bit model is not a reduced security claim. It is the largest width at which the exact matrix tower can still be inspected directly.

Width n	State count 2^n	Permutation matrix size	Display status
4	16	16×16	printable
8	256	256×256	inspectable
16	65536	65536×65536	page-scale impossible
32	4.29×10^9	$2^{32} \times 2^{32}$	infeasible
64	1.84×10^{19}	$2^{64} \times 2^{64}$	ARX-box scale
128	3.40×10^{38}	$2^{128} \times 2^{128}$	full LOPC-state scale

For 8-bit instances, four visual diagnostics are particularly useful:

1. the 256×256 permutation-matrix heatmap;
2. the DDT heatmap induced by Eq. (17);
3. the LAT/Walsh heatmap induced by Eq. (19);
4. the Boolean-Jacobian family over all input points.

These figures are not security claims; they are visual diagnostics for the algebraic explosion.

32.2.9 Claim boundary of the matrix microscope

This subsection supplies an algebraic language for the later differential, linear, additive-differential, MILP, HULL, and game-based discussions. It does not prove a PRP theorem, a PRG theorem, or a full-construction bound. It explains why ARX/SRX composition is hard to compress: trail and hull analysis must operate inside products of massive permutation, transfer, and slice matrices whose inner factors are selected by carry, borrow, mask, or difference constraints.

End of the matrix microscope.

32.3 Additive Differential Refinement via the Niu–Sun–Hu ARX Matrix Model

The earlier manuscript kept an additive-differential view of ARX operations. This view is not a decorative side note: it is a different cryptanalytic coordinate from XOR differentials. The relevant modern reference for the two-rotation ARX form is Niu, Sun, and Hu's analysis of additive differential probability for

$$A_{d,e}(x,y) = ((x \boxplus_n y) \lll d) \oplus (y \lll e),$$

which appears in ARX components such as Alzette and Speck [NSH23]. Earlier work by Velichkov, Mouha, De Cannière, and Preneel gave an additive-differential algorithm for an ARX sequence, but the Niu–Sun–Hu paper is the reference that explicitly handles the two-rotation output form; their discussion also states that the older one-rotation formulation is not suitable for several constructions of this shape [VMCP11, NSH23]. The additive-differential behaviour of XOR itself is nontrivial and is part of the background line of this coordinate system [MKA+21].

Definition 48 (Additive differential coordinate)**.** Let $F : (\mathbb{Z}/2^n\mathbb{Z})^m \to (\mathbb{Z}/2^n\mathbb{Z})^k$ be a word map. For input increment Δ and output increment ∇, the additive differential probability is

$$\mathrm{ADP}_F(\Delta \to \nabla) = \Pr_x \left[F(x+\Delta) - F(x) = \nabla \pmod{2^n}\right].$$

Here the perturbation and the observation are taken in the additive residue group. This is not an XOR differential with renamed symbols: addition/subtraction of differences brings ordinary carry and borrow states into the perturbation law.

Definition 49 (Two-rotation ARX layer)**.** For $x, y \in \mathbb{F}_2^n$, define

$$A_{d,e}(x,y) = ((x \boxplus_n y) \lll d) \oplus (y \lll e), \qquad 0 \le d, e < n.$$

Let

$$f_{d,e}(u,v) = (u \lll d) \oplus (v \lll e).$$

The ARX layer can be written as $A_{d,e}(x,y) = f_{d,e}(x \boxplus_n y, y)$.

Lemma 16 (Reduction from ARX input differences to the two-rotation XOR map)**.** *Fix additive input increments (α', β) for (x,y). Put*

$$\alpha = \alpha' \boxplus_n \beta.$$

Then

$$\mathrm{ADP}_{A_{d,e}}\big((\alpha',\beta) \to \gamma\big) = \mathrm{ADP}_{f_{d,e}}\big((\alpha,\beta) \to \gamma\big).$$

Proof. Let $s = x \boxplus_n y$. Under additive perturbation,

$$x' = x \boxplus_n \alpha', \qquad y' = y \boxplus_n \beta.$$

The first branch entering $f_{d,e}$ changes from

$$s = x \boxplus_n y$$

to

$$s' = x' \boxplus_n y' = (x \boxplus_n \alpha') \boxplus_n (y \boxplus_n \beta) = (x \boxplus_n y) \boxplus_n (\alpha' \boxplus_n \beta) = s \boxplus_n \alpha.$$

The second branch changes from y to $y \boxplus_n \beta$. Therefore the additive transition of $A_{d,e}$ under (α', β) is exactly the additive transition of $f_{d,e}$ under (α, β). This is the change of variables used by the Niu–Sun–Hu theorem. □

Engineering interpretation. The modular addition has not disappeared. It has been moved into the input-difference transformation $\alpha = \alpha' \boxplus_n \beta$. After this change of variables, the remaining difficulty is the additive differential probability of a rotation-XOR expression $(u \lll d) \oplus (v \lll e)$, which is exactly where Niu–Sun–Hu introduce their partition and 8×8 matrices.

Definition 50 (Carry-boundary partition state)**.** For a prefix of length k, Niu–Sun–Hu partition $\mathbb{F}_2^k \times \mathbb{F}_2^k$ by the incoming and outgoing carry-boundary bits generated by the two additive perturbations and by the XOR consistency equation. In this manuscript we denote a typical cell by

$$\mathcal{S}^{(k)}_{u,v,s|a,b,c}(\alpha, \beta) \subseteq \mathbb{F}_2^k \times \mathbb{F}_2^k,$$

where (a, b, c) records the lower boundary state and (u, v, s) records the upper boundary state. The exact notation in Niu–Sun–Hu is written with carry symbols $\widehat{c}_a$ and boundary labels; the essential point for the LOPC paper is that each cell fixes the hidden carry-boundary information that the next prefix update needs.

Lemma 17 (One-bit matrix transfer)**.** *For every bit triple $(p, q, r) \in \mathbb{F}_2^3$ there is an 8×8 rational matrix $M_{p,q,r}$ such that the prefix vector*

$$V^{(k)} = (v_0^{(k)}, \ldots, v_7^{(k)})^T$$

of normalized cell masses satisfies

$$V^{(k+1)} = M_{\alpha_k, \beta_k, \gamma_k} V^{(k)}.$$

The eight matrices $M_{000}, \ldots, M_{111}$ are the local transition matrices printed in Niu–Sun–Hu's theorem organization.

Proof. For a fixed prefix cell, appending one more bit chooses one fresh pair $(x_k, y_k) \in \mathbb{F}_2^2$. The two perturbed additions determine the next carry-boundary bits, while the required output bit γ_k imposes a one-bit XOR consistency condition. Counting the four possible fresh pairs and normalizing by 2^2 gives one entry of the matrix that maps the old boundary state to the new boundary state. Repeating this for the eight possible old and new boundary labels gives the 8×8 matrix. Thus the matrix is not an empirical approximation; it is the exact one-bit transfer table of the partitioned additive-differential experiment. □

Theorem 9 (Niu–Sun–Hu matrix formula for the two-rotation ARX layer)**.** *Let $\alpha, \beta, \gamma \in \mathbb{F}_2^n$. For each index i, put*

$$\mu_i = M_{\alpha_{n-d+i \bmod n}, \beta_{n-e+i \bmod n}, \gamma_i}.$$

If $d \geq e$, define the three interval products

$$\Pi_0 = \prod_{i=0}^{e-1} \mu_i, \qquad \Pi_1 = \prod_{i=e}^{d-1} \mu_i, \qquad \Pi_2 = \prod_{i=d}^{n-1} \mu_i.$$

Then

$$\mathrm{ADP}_{f_{d,e}}\big((\alpha, \beta) \to \gamma\big) = \sum_{a,b \in \mathbb{F}_2} C^T_{a,b} \Pi_2 R_2 \Pi_1 R_1 \Pi_0 e_{4+2b+a}.$$

If $d \leq e$, the two cut intervals are exchanged, giving the analogous expression with R_1 and R_2 placed at the cut points specified by the Niu–Sun–Hu theorem. The boundary matrices are

$$R_1 = \sum_{c,h,v \in \mathbb{F}_2} e_{4c+h} e^T_{4c+2v+h}, \qquad R_2 = \sum_{w,z,u \in \mathbb{F}_2} e_{4z+2w} e^T_{4z+2w+u},$$

and

$$C_{a,b} = \sum_{s \in \mathbb{F}_2} e_{4s+2b+a}.$$

Consequently,

$$\mathrm{ADP}_{A_{d,e}}\big((\alpha', \beta) \to \gamma\big)$$

is obtained from the same expression after substituting $\alpha = \alpha' \boxplus_n \beta$.

Proof. The proof is a composition of the previous two ideas. First, the ARX layer is reduced to $f_{d,e}$ by the change of variables $\alpha = \alpha' \boxplus_n \beta$. Second, the rotations split the word indices into three intervals. When $d \geq e$, the intervals are $[0, e-1]$, $[e, d-1]$, and $[d, n-1]$. On each interval the local bit relation is represented by the corresponding μ_i. At the rotation cut points, the boundary state has to be rewired; these rewiring operations are exactly R_1 and R_2. Finally, $C_{a,b}$ sums over the terminal hidden boundary state while e_{4+2b+a} selects the initial boundary state. Multiplying the matrices and summing over (a, b) therefore counts exactly the favorable partition cells, normalized by the additive-differential probability space. □

Engineering interpretation. This theorem is the rigorous replacement for the older short paragraph. The reason the two-rotation ARX component is complex is not mystical: a rotated additive input coordinate does not line up with the same output coordinate, so the hidden carry-boundary state must be cut, transported, and reconnected by matrices. The proof is a finite-state transfer proof.

Lemma 18 (Local norm envelope of the Niu–Sun–Hu transfer matrices)**.** *For the eight printed matrices* $M_{000}, \ldots, M_{111}$*, direct singular-value evaluation gives*

$$\|M_{p,q,r}\|_2 = \frac{\sqrt{5}}{2} < \sqrt{2} \qquad ((p, q, r) \in \mathbb{F}_2^3).$$

Therefore any length-L product of local matrices satisfies

$$\left\| \prod_{i=1}^{L} M_i \right\|_2 \leq \prod_{i=1}^{L} \|M_i\|_2 \leq \left(\frac{\sqrt{5}}{2}\right)^L < (\sqrt{2})^L.$$

Proof. The first equality is obtained by evaluating the largest singular value of each of the eight rational 8×8 matrices printed by Niu–Sun–Hu. The product bound is the submultiplicativity of the spectral norm. This norm envelope is only a growth diagnostic for transfer products; the exact probability remains the scalar matrix expression of Theorem 9. □

Corollary 3 (Design-side sufficient bound under a verified per-layer ADP bound)**.** *Define*

$$p_{\max}(d, e, n) = \max_{\alpha', \beta, \gamma} \mathrm{ADP}_{A_{d,e}}\big((\alpha', \beta) \to \gamma\big).$$

If the selected rotation pair (d, e) *is verified by the Niu–Sun–Hu evaluator, or by an equivalent complete audit, to satisfy*

$$p_{\max}(d, e, n) \leq 2^{-n/2},$$

then an R-layer single-characteristic product estimate gives

$$p_{\max}(d, e, n)^R \leq 2^{-Rn/2}.$$

If one additionally applies a conservative R-term aggregation factor, then

$$\mathrm{ADP}^{(R)} \leq R \cdot 2^{-Rn/2} = 2^{-Rn/2 + \log_2 R}.$$

A sufficient condition for the right-hand side to be at most 2^{-n} *is*

$$-\frac{Rn}{2} + \log_2 R \le -n,$$

or equivalently

$$R \ge 2 + \frac{2\log_2 R}{n}.$$

For $n \ge 64$, $R = 3$ *satisfies this sufficient inequality.*

Proof. The first inequality is an explicit hypothesis: the per-layer maximum is not guessed but must be obtained from the exact matrix evaluator or from a complete equivalent audit. Assuming independent trail-style multiplication of the selected layer transitions gives the single-characteristic product $p_{\max}^{R}$. Replacing a single characteristic by at most R aggregated dominant contributions gives the displayed union-style bound. Solving $R2^{-Rn/2} \le 2^{-n}$ gives $-Rn/2 + \log_2 R \le -n$, hence $R \ge 2 + 2\log_2 R/n$. For $n = 64$, $R = 3$ gives $3 \ge 2 + 2\log_2 3/64$; the right-hand side decreases as n grows, so the same conclusion holds for every $n \ge 64$. □

Rotation-selection interpretation. The pair $(\lfloor n/3 \rfloor, \lfloor 2n/3 \rfloor)$ should be presented as a design target scored by the additive-differential matrix evaluator, not as a theorem directly claimed by Niu–Sun–Hu. In other words, the Niu–Sun–Hu framework supplies the exact scoring oracle for two-rotation ARX additive differentials; the design then uses balanced offsets to avoid concentrating the three transfer intervals into short aligned windows. A safe manuscript statement is

$$(d, e) \approx \left(\left\lfloor \frac{n}{3} \right\rfloor, \left\lfloor \frac{2n}{3} \right\rfloor \right)$$

as a balanced rotation-design heuristic evaluated under the matrix product objective.

Scope and non-claim. This subsection restores the original additive-differential refinement but removes the old overcompression. Niu–Sun–Hu prove an exact linear-time matrix method for the two-rotation ARX additive differential. The present manuscript uses that method as a component-level design and auditing tool. It does not claim that the full LOPC composition inherits a complete additive-differential proof automatically, and it does not identify additive differentials with XOR differentials.

33 Component A Security: NeoAlzette

33.1 Scope of the Component-A MILP Boxes

The boxes in this section are the reviewer-facing local MILP library used by the NeoAlzette component model. They encode single-characteristic differential and linear trails. Therefore, the reported quantities are MEDCP/MELCC-style component quantities, not endpoint differential hull probabilities or endpoint linear hull correlations. Endpoint HULL aggregation is a separate procedure and is only complete when the corresponding endpoint enumeration is complete.

The derivation of these boxes from the differential and linear oracles is deferred to Appendix A. The addition-differential boxes follow the Lipmaa–Moriai modular-addition condition [LM02]. The linear-addition boxes follow the Wallén carry-correlation view [Wal03a]. The fixed-public arithmetic call sites use fixed-addend oracles rather than averaged two-variable boxes, following the addend-dependence line of Miyano, Machado, and Azimi et al. [Miy98, Mac01, ARS+20]. The MEDCP/MELCC terminology follows the wide ARX-box security-evaluation style of Alzette [BBdS+20].

33.2 Binary and Bit-Vector MILP Boxes

All bit variables below are binary. Word-level equations are expanded by applying these local boxes to the bits selected by the accessors of Part II. In particular, bit reads are written as $\mathrm{ReadBit}(X, i)$, not as array access on a word object.

XOR. For $c = a \oplus b$, use

$$a + b - c \geq 0, \qquad a - b + c \geq 0, \qquad -a + b + c \geq 0, \qquad -a - b - c + 2 \geq 0.$$

AND. For $c = a \wedge b$, use

$$c \leq a, \qquad c \leq b, \qquad c \geq a + b - 1.$$

OR. For $c = a \vee b$, use

$$c \geq a, \qquad c \geq b, \qquad c \leq a + b.$$

NOT. For $c = \neg a$, use

$$c = 1 - a.$$

Copy and branch. If an input bit is copied to m destinations, use

$$b_1 = a, \quad b_2 = a, \quad \ldots, \quad b_m = a.$$

Rotation and bit permutation. For $Y = \mathrm{X} \lll^{\mathrm{r}}$ on n bits, use

$$\mathrm{ReadBit}(Y, i) = \mathrm{ReadBit}(X, (i - r) \bmod n) \qquad (0 \leq i < n).$$

For $Y = \mathrm{X} \ggg_{\mathrm{r}}$, use

$$\mathrm{ReadBit}(Y, i) = \mathrm{ReadBit}(X, (i + r) \bmod n) \qquad (0 \leq i < n).$$

More general fixed bit permutations are encoded by equalities of the same form.

Zero-fill shifts. For $Y = X \ll r$ under the zero-fill convention,

$$\mathrm{ReadBit}(Y, i) = \begin{cases} \mathrm{ReadBit}(X, i - r), & i \geq r, \\ 0, & i < r. \end{cases}$$

For $Y = X \gg r$, use

$$\mathrm{ReadBit}(Y, i) = \begin{cases} \mathrm{ReadBit}(X, i + r), & i + r < n, \\ 0, & i + r \geq n. \end{cases}$$

These shift boxes are distinct from rotation boxes.

33.3 Four Modular Addition/Subtraction MILP Boxes

The four arithmetic boxes used by the component model are

$$\mathrm{ADD}_n^{\Delta}, \qquad \mathrm{SUB}_n^{\Delta}, \qquad \mathrm{ADD}_n^{L}, \qquad \mathrm{SUB}_n^{L}.$$

The superscript Δ denotes an XOR-differential characteristic box, and the superscript L denotes a linear-characteristic box.

33.3.1 ADD-Δ: two-variable modular addition differential box

Let $Z = X \boxplus_n Y$ and let the XOR-difference triple be $(\alpha, \beta \mapsto \gamma)$. The least-significant bit condition is

$$\alpha_0 \oplus \beta_0 = \gamma_0.$$

For each $0 \le i < n-1$, introduce a binary weight bit p_i. The thirteen local inequalities are

$$\begin{cases} \beta_i - \gamma_i + p_i \ge 0, \\ \alpha_i - \beta_i + p_i \ge 0, \\ -\alpha_i + \gamma_i + p_i \ge 0, \\ \alpha_i + \beta_i + \gamma_i - p_i \ge 0, \\ -\alpha_i - \beta_i - \gamma_i - p_i \ge -3, \\ \alpha_{i+1} - \beta_i + \beta_{i+1} + \gamma_{i+1} + p_i \ge 0, \\ \beta_i + \alpha_{i+1} - \beta_{i+1} + \gamma_{i+1} + p_i \ge 0, \\ \beta_i - \alpha_{i+1} + \beta_{i+1} + \gamma_{i+1} + p_i \ge 0, \\ \alpha_i + \alpha_{i+1} + \beta_{i+1} - \gamma_{i+1} + p_i \ge 0, \\ \gamma_i - \alpha_{i+1} - \beta_{i+1} - \gamma_{i+1} + p_i \ge -2, \\ -\beta_i + \alpha_{i+1} - \beta_{i+1} - \gamma_{i+1} + p_i \ge -2, \\ -\beta_i - \alpha_{i+1} + \beta_{i+1} - \gamma_{i+1} + p_i \ge -2, \\ -\beta_i - \alpha_{i+1} - \beta_{i+1} + \gamma_{i+1} + p_i \ge -2. \end{cases}$$

The word-level differential characteristic weight is

$$W_{\boxplus}^{\Delta} = \sum_{i=0}^{n-2} p_i.$$

33.3.2 SUB-Δ: two-variable modular subtraction differential box

Subtraction is represented by the addition wrapper

$$Z = X \boxminus_n Y \quad \Longleftrightarrow \quad X = Z \boxplus_n Y.$$

Therefore

$$\mathrm{SUB}_n^{\Delta}(\alpha_X, \alpha_Y \mapsto \alpha_Z) = \mathrm{ADD}_n^{\Delta}(\alpha_Z, \alpha_Y \mapsto \alpha_X).$$

33.3.3 ADD-L: two-variable modular addition linear box

Let $Z = X \boxplus_n Y$. Let Γ be the output mask and let A, B be the two input masks. For each bit, use the state variables s_{i+1} and s_i. The local linear-addition box is

$$\begin{cases} s_{i+1} - \Gamma_i - A_i + B_i + s_i \ge 0, \\ s_{i+1} + \Gamma_i - A_i - B_i + s_i \ge 0, \\ s_{i+1} + \Gamma_i - A_i + B_i - s_i \ge 0, \\ -s_{i+1} + \Gamma_i + A_i + B_i + s_i \ge 0, \\ s_{i+1} + \Gamma_i + A_i - B_i - s_i \ge 0, \\ s_{i+1} - \Gamma_i + A_i - B_i + s_i \ge 0, \\ s_{i+1} - \Gamma_i + A_i + B_i - s_i \ge 0, \\ s_{i+1} + \Gamma_i + A_i + B_i + s_i \le 4. \end{cases}$$

The word-level linear-characteristic weight is

$$W_{\boxplus}^{L} = \sum_i s_{i+1}.$$

33.3.4 SUB-L: two-variable modular subtraction linear box

Use the wrapper

$$Z = X \boxminus_n Y \quad \Longleftrightarrow \quad X = Z \boxplus_n Y.$$

The mask relabeling is

$$\mathrm{SUB}_n^L(\Gamma_Z; A_X, A_Y) = \mathrm{ADD}_n^L(A_X; \Gamma_Z, A_Y).$$

This avoids duplicating the eight Wallén-style inequalities with a separate sign convention.

33.4 Fixed-Public Arithmetic Call Sites

The fixed-public sites $X \mapsto X \boxplus K$ and $X \mapsto X \boxminus K$ are not instantiated by the averaged two-variable boxes. They use exact fixed-public differential and linear oracles

$$\mathcal{O}^{\Delta}_{\pm K}, \qquad \mathcal{O}^{L}_{\pm K},$$

whose values depend on the concrete bit pattern of K. The oracle-to-MILP conversion is given in Appendix A.

33.5 Joint-Injection Oracle Hook

Let $\Phi_s = (X_s, Y_s)$ be a packed C/D injection map for orientation $s \in \{B \to A, A \to B\}$. The first component X_s is wired into the zero-weight XOR route, while the second component Y_s is wired into the non-constant operand of the following two-variable modular addition. The addition box itself remains the standard ADD-Δ or ADD-L box; the important modeling condition is that its second input is source-constrained by the same packed oracle as the XOR route.

Differential hook. For source difference Δ_s, the packed derivative is affine:

$$D_{\Delta_s}\Phi_s(z) = M_{s,\Delta_s} z \oplus c_{s,\Delta_s}.$$

Hence the pair of differences injected into the two downstream paths must obey

$$(\delta_X, \delta_Y) \in c_{s,\Delta_s} + \operatorname{im} M_{s,\Delta_s}, \qquad w^{\Delta}_{\mathrm{inj}} \geq \operatorname{rank}(M_{s,\Delta_s}).$$

If Δ_T is the target-branch difference before routed XOR, the following source-constrained addition sees the input-difference pair

$$(\Delta_T \oplus \delta_X, \delta_Y).$$

Thus a selected local differential characteristic is charged as

$$W^{\Delta}_{\mathrm{joint},s} = w^{\Delta}_{\mathrm{inj}} + W^{\Delta}_{\boxplus}(\Delta_T \oplus \delta_X, \delta_Y \mapsto \Delta_{T'}).$$

This is the formal reason the injection and the subsequent addition cannot be modeled as independent operations.

Linear hook. On the linear side, the Walsh coefficient of the packed oracle is

$$\widehat{\Phi}_s(\alpha, \beta) = \sum_x (-1)^{\alpha \cdot x \oplus \beta \cdot \Phi_s(x)}.$$

If λ is the mask on the first input of the following addition and η is the mask on the CD-derived addend input, then the packed oracle mask is

$$\beta = \mathrm{pack}(\lambda, \eta).$$

The live-mask condition and conservative weight hook are

$$\alpha \in \mathrm{WalshSupport}_{\Phi_s}(\beta), \qquad w^L_{\mathrm{inj}} \geq \frac{1}{2}\,\mathrm{rank}(S_{s,\beta}),$$

where $S_{s,\beta}$ is the alternating Hessian contraction of the quadratic injection under output mask β. The same λ and η also feed the standard addition-linear box, giving

$$W^L_{\mathrm{joint},s} = w^L_{\mathrm{inj}} + W^L_{\boxplus}(\gamma;\lambda,\eta),$$

up to deterministic phase terms from public constants. The full QFH, Hessian, rank, and oracle-to-MILP derivation is in Appendix A.

33.6 How These Boxes Assemble the NeoAlzette Model

Table 4: Component-A local box assembly.

NeoAlzette operation	**MILP box or oracle hook**
XOR / constant XOR	XOR / affine box
AND, OR, NOT	Boolean binary boxes
copy / branch	equality-copy constraints
rotation	permutation equality
zero-fill shift	shift equality with fixed zero coordinates
two-variable add/sub	ADD-Δ, SUB-Δ, ADD-L, SUB-L
fixed-public add/sub	$\mathcal{O}^{\Delta/L}_{\pm K}$
C/D injection	derivative-image/rank or Walsh-support/rank hook

Theorem 10 (Differential MILP feasible-assignment soundness)**.** *Every integral feasible assignment of the Component-A differential MILP boxes decodes to one single-characteristic* NeoAlzette *XOR-differential path under the modeled local relations. Its objective is the sum of the selected local differential characteristic weights.*

Proof. The Boolean boxes encode exact bit relations; the modular arithmetic boxes encode the two-variable add/sub relations; fixed-public sites are delegated to fixed-public oracles; and the injection hook enforces derivative-image support. The model wires these local relations in the NeoAlzette schedule order, so an integral assignment is a semantic single-characteristic path and its objective is the accumulated differential weight. □

Theorem 11 (Linear MILP feasible-assignment soundness)**.** *Every integral feasible assignment of the Component-A linear MILP boxes decodes to one single-characteristic* NeoAlzette *Walsh-linear path under the modeled local relations. Its objective is the sum of the selected local absolute-correlation weights. Endpoint linear hull signs are handled only in the endpoint-hull aggregation layer.*

Proof. Affine layers use transpose mask transport. The modular arithmetic boxes encode nonzero two-variable add/sub mask transitions, fixed-public sites use signed transfer oracles, and the injection hook enforces Walsh-support/rank compatibility. Hence an integral assignment is a valid single linear characteristic of the component model, and the objective is the accumulated MELCC-style characteristic weight. □

33.7 Result Status and MEDCP/MELCC Claim Boundary

A differential optimization over the component model reports MEDCP-style single-differential-characteristic evidence. A linear optimization reports MELCC-style

single-linear-characteristic evidence. A resource-limited incumbent is not a solver-proven optimum unless the solver status certifies optimality. Endpoint differential hull probability and endpoint linear hull correlation require separate endpoint HULL enumeration. Forest campaigns are search/coverage procedures and do not by themselves certify a complete endpoint hull.

33.8 NeoAlzette MILP Single-Characteristic Campaign Results

The previous subsections define the local MILP boxes and the MEDCP/MELCC claim boundary. This subsection records the current one-round solver campaigns for the NeoAlzette component. These campaigns are reported with two independent labels: a duration class and a certification status. A short fixed-budget run is called a time-bounded campaign. A frozen-model run measured in weeks or months is called a long-run strict campaign. Neither duration label proves optimality by itself; a result is certified only when the solver status or the enumeration status proves optimality or completeness.

Table 5: Campaign-duration and certification vocabulary for NeoAlzette MILP evidence.

Evidence label	Typical duration	Meaning
Diagnostic run	minutes to hours	Sanity-check or debugging run. It should not be used as a main security result.
Time-bounded campaign	hours to a few days	Fixed-budget search on a frozen model. It can report an incumbent and solver status, but not optimality unless the solver proves it.
Long-run strict campaign	weeks to months	Frozen-source, frozen-parameter, preserved-log solver campaign. This is stronger engineering evidence than a short run, but still reports an incumbent unless the solver proves optimality.
Certified optimum / complete enumeration	any duration	The solver or enumeration procedure proves the optimum or exhausts the endpoint space.

The current uploaded differential and linear runs are both 43200-second SCIP runs. They are therefore classified as time-bounded campaigns. The differential run was interrupted at the time limit with primal bound 80.431292345474, dual bound 0.30000763337679, and gap 26709.75%. The linear run was interrupted at the time limit with primal bound 16.6153859522286, dual bound 5.82243782154161, and gap 185.37%. Both results are reported as incumbents rather than certified optima.

Table 6: Current one-round single-characteristic MILP incumbents for the NeoAlzette component.

Mode	**Incumbent record**
Differential	Rounds: 1. Quantity: MEDCP-style single characteristic. Solver: SCIP. Budget: 43200 s. Incumbent weight: 80.4312923455. Value: $2^{-80.4312923455} \approx 6.13 \cdot 10^{-25}$. Status: time limit; incumbent only.
Linear	Rounds: 1. Quantity: MELCC-style single characteristic. Solver: SCIP. Budget: 43200 s. Incumbent weight: 16.6153859522. Value: $2^{-16.6153859522} \approx 9.96 \cdot 10^{-6}$. Status: time limit; incumbent only.

Table 7: Endpoint and consistency data for the uploaded one-round NeoAlzette incumbents.

Mode	Endpoint and consistency record
Differential	Input endpoint: $\Delta A_{in} = \texttt{0x3492db6a}$, $\Delta B_{in} = \texttt{0x00000000}$. Output endpoint: $\Delta A_{out} = \texttt{0x5f727fe6}$, $\Delta B_{out} = \texttt{0xbac57b00}$. Consistency checks: trace weight total equals objective; checkpoint trace preserved. Artifact log: `diff_r1_best_cmd_43200s_YYYYMMDD_HHMMSS.log`.
Linear	Input endpoint: $\Lambda A_{in} = \texttt{0x00000000}$, $\Lambda B_{in} = \texttt{0xbffee75d}$. Output endpoint: $\Lambda A_{out} = \texttt{0x10bca2ab}$, $\Lambda B_{out} = \texttt{0xed7ecbae}$. Consistency checks: oracle-valid; oracle weights consistent; trace matches objective; paper-usable characteristic. Artifact log: `linear_r1_best_cmd_43200s_YYYYMMDD_HHMMSS.log`.

These tables do not claim endpoint differential hull probability or endpoint linear hull correlation. They report current single-characteristic incumbents for the Component-A model. Complete MEDCP/MELCC certification would require solver-proven optimality for the corresponding model; complete endpoint-HULL claims would require endpoint enumeration and signed or unsigned aggregation as appropriate.

34 Component B Security: XCR CSPRNG

34.1 Analysis oracles for XCR before MILP/SAT modeling

These oracles are the mathematical interface for the next analysis stage. They are solver-ready, but they are not solver certificates. A future MILP, SAT, or SMT model should encode these oracle relations; it should not invent a different XCR round.

34.1.1 Structural facts for deployed zero-fill shadow operators

Lemma 19 (Complement duality of the shadow word operators)**.** *For the deployed zero-fill operators,*

$$a \boxminus_{64}^{\mathrm{sh,zf}} b = \neg_{64}\big((\neg_{64} a) \boxplus_{64}^{\mathrm{sh,zf}} b\big).$$

Proof. Expanding Definition 37, replacing a by $\neg_{64} a$ in the addition source turns the generate source into the subtraction borrow source and turns the direct parity into its complement. The two source updates and the two zero-fill projections are otherwise identical. Complementing the final addition-form output restores the direct parity word of the subtraction operator. □

Lemma 20 (Six-position locality of the deployed shadow operators)**.** *For either deployed shadow sign, the bit*

$$\mathrm{ReadBit}((a \boxplus_{64}^{\mathrm{sh,zf}} b), i) \quad \textit{or} \quad \mathrm{ReadBit}((a \boxminus_{64}^{\mathrm{sh,zf}} b), i)$$

depends only on

$$\mathrm{ReadBit}(a, i-t),\ \mathrm{ReadBit}(b, i-t) \qquad (0 \le t \le 5),$$

with out-of-range negative positions filled by the zero boundary.

Proof. The two source updates in Definition 37 can move a carry or borrow source at most three positions forward inside the bounded source word. The final output then projects the bounded source by offsets one and two. Therefore an output bit at coordinate i can see source material down to coordinate $i-5$, and no deeper bit can reach it. This gives the six-position window. The same argument applies to the borrow operator after replacing the carry source by the borrow source. □

Corollary 4 (Exact local-state sizes for shadow-operator oracles)**.** *The exact two-variable binary-coordinate oracle for either deployed shadow operator uses five previous input-pair positions, hence* $4^5 = 1024$ *histories. If one operand is fixed, only five variable bits remain, hence the fixed-operand oracle uses* $2^5 = 32$ *histories.*

Theorem 12 (Zero-fill fixed-operand shadow maps are triangular permutations)**.** *Fix one operand of either*

$$x \mapsto x \boxplus_{64}^{\mathrm{sh,zf}} K, \quad x \mapsto K \boxplus_{64}^{\mathrm{sh,zf}} x, \quad x \mapsto x \boxminus_{64}^{\mathrm{sh,zf}} K, \quad x \mapsto K \boxminus_{64}^{\mathrm{sh,zf}} x.$$

Each resulting unary zero-fill map is triangular with diagonal coefficient one and is therefore a permutation of $\mathbb{F}_2^{64}$*.*

Proof. By the zero-fill convention, no higher variable bit can influence a lower output coordinate. At coordinate i, the direct parity contributes the current variable bit $\mathrm{ReadBit}(x, i)$ with coefficient one. All source-projection terms entering the same coordinate are functions only of lower variable bits and fixed operand bits. The map is therefore unit triangular and can be inverted by reconstructing the variable word from low to high. □

Definition 51 (Shadow S-round component oracles)**.** For $\star \in \{+, -\}$, the 64-bit zero-fill shadow differential oracle is

$$\mathcal{O}_{\mathrm{sh}}^{\Delta,\star}(\alpha, \beta; \gamma) = (N, 2^{128}, w),$$

where N is the exact two-variable favorable numerator for

$$F_\star^{\mathrm{zf},64}(a \oplus \alpha, b \oplus \beta) \oplus F_\star^{\mathrm{zf},64}(a, b) = \gamma$$

and $w = -\log_2(N/2^{128})$ when $N \neq 0$. The signed linear-correlation oracle is

$$\mathcal{O}_{\mathrm{sh}}^{L,\star}(\lambda, \mu; \nu) = (W, 2^{128}, \sigma, w),$$

where W is the exact signed Walsh numerator, $\sigma = \mathrm{sign}(W)$ for $W \neq 0$, and $w = -\log_2(|W|/2^{128})$. The sign is part of the oracle output.

Lemma 21 (S-round component oracle exactness)**.** *The shadow oracles above are exact for one deployed XCR S-round component call, but only as component oracles.*

Proof. The deployed XCR S-round uses the Model-A zero-fill shadow maps at width 64. Shadow-Boxed Arithmetic supplies exact 1024-state two-variable transfers and 32-state fixed-side transfers for that branch. Hence the local numerator or signed numerator for one shadow operation is exact. The statement is component-local; it does not compose the four shared-lane operations into a full XCR probability. □

Definition 52 (Hybrid8 byte oracles)**.** Let S_A and S_B be the two byte maps implemented by `hybrid8_sa` and `hybrid8_sb`. For $s \in \{A, B\}$, define the byte differential table oracle

$$\mathcal{O}_H^{\Delta}(s, \alpha, \gamma) = \#\{x \in \mathbb{F}_2^8 : S_s(x \oplus \alpha) \oplus S_s(x) = \gamma\},$$

and the byte signed correlation oracle

$$\mathcal{O}_H^{L}(s, \lambda, \nu) = \sum_{x \in \mathbb{F}_2^8} (-1)^{\lambda \cdot x \oplus \nu \cdot S_s(x)}.$$

For $\mathrm{BigS}_{64}(u, o)$, byte j uses S_A if $(o + j) \bmod 2 = 0$ and S_B otherwise.

Lemma 22 (H-round byte oracle exactness)**.** *The H-round oracle is exact by exhaustive enumeration over the 8-bit byte domain.*

Proof. Each byte map is a total deterministic map $\mathbb{F}_2^8 \to \mathbb{F}_2^8$. Its DDT entry is a finite count over 256 inputs, and its LAT entry is a finite signed Walsh sum over the same domain. BigS_{64} applies these byte maps independently to the eight bytes according to the public offset rule; therefore the byte oracles exactly describe the nonlinear H-round components. □

Definition 53 (Affine wiring oracles)**.** The affine difference oracle $\mathcal{O}_{\mathrm{aff}}^{\Delta}$ enforces

$$\Delta(u \oplus v) = \Delta u \oplus \Delta v, \qquad \Delta(\mathrm{rotl}_r\, u) = \mathrm{rotl}_r(\Delta u).$$

The affine linear-mask oracle $\mathcal{O}_{\mathrm{aff}}^{L}$ enforces

$$\langle \beta, \mathrm{rotl}_r\, u \rangle = \langle \mathrm{rotr}_r\, \beta, u \rangle$$

and the usual dual propagation through XOR fan-out.

Lemma 23 (Affine oracle exactness)**.** *The affine wiring oracles are equality constraints, not approximations.*

Proof. XOR is addition in the vector space $\mathbb{F}_2^{64}$, and rotation is a coordinate permutation. Difference propagation follows by linearity. Mask propagation follows from the adjoint permutation identity for the inner product. □

Definition 54 (XCR step factor-graph oracle)**.** For a call index t, define the differential factor graph

$$\mathcal{G}_t^{\Delta} = (V_t^{\Delta}, F_t^{\Delta})$$

and the linear factor graph

$$\mathcal{G}_t^{L} = (V_t^{L}, F_t^{L})$$

by introducing variables for the state lanes, n, C_t, the four round constants, the rebound words, the nonlinear branch outputs, the final state lanes, the visible output, and the next counter. The factor set contains schedule factors, affine wiring factors, exactly one H/S branch factor, the corresponding Hybrid8 or Shadow component factors, final diffusion factors, output-extraction factors, and the counter-update factor.

Theorem 13 (XCR analysis-oracle soundness)**.** *If an assignment satisfies every factor emitted by $\mathcal{G}_t^{\Delta}$, then it is a valid one-step XOR-differential semantic trace of the implemented XCR step. If an assignment satisfies every factor emitted by $\mathcal{G}_t^{L}$, then it is a valid one-step linear-mask semantic trace of the implemented XCR step, with component signs retained for hull summation.*

Proof. The schedule factors enforce Definition 31. The affine factors enforce the exact XOR/rotation rebinding and final diffusion equations. The H-branch factors are exact byte DDT/LAT oracles; the S-branch factors are exact Model-A shadow component oracles. The output and counter factors enforce the same extraction and public counter update as $\mathcal{O}_{\mathrm{eval}}^{\mathrm{XCR}}$. Hence a satisfying assignment is precisely a semantic witness for one implementation step. The linear case differs only in using adjoint mask propagation and signed component sums; the sign is not discarded. □

Corollary 5 (Future SAT/SMT feasibility target)**.** *A future SAT or SMT model for XCR should bit-blast or theory-encode the factor relation of $\mathcal{G}_t^{\Delta}$ or $\mathcal{G}_t^{L}$. A satisfying assignment would prove feasibility of the encoded trace relation, not an optimality bound by itself.*

Corollary 6 (Future MILP trail-score target)**.** *A future MILP model may encode affine layers as exact linear equalities, H-rounds through byte DDT/LAT rows, S-rounds through shadow DP/LC oracle rows, and shared lanes through compatibility constraints. Its objective would be a component-trail score for the encoded factor graph. It would become a proof only when the solver status and the model scope certify the relevant optimum or complete enumeration.*

Remark 16 (Current boundary). This paper defines solver-ready XCR analysis oracles. It does not claim a completed XCR MILP, SAT, or SMT campaign, and it does not claim a full differential or linear hull bound for the stateful generator.

34.2 Exact differential-probability (DP) and linear-correlation (LC) transfer model for shadow operators

Definition 55 (Two-variable history)**.** Let $u_i = (a_i, b_i) \in \mathbb{F}_2^2$ and

$$H_i = (u_{i-5}, u_{i-4}, u_{i-3}, u_{i-2}, u_{i-1}) \in (\mathbb{F}_2^2)^5.$$

Let $f_\star(H_i, u_i)$ be the local output bit of the shadow operation, with $\star \in \{+, -\}$, and let

$$\mathrm{next}(H_i, u_i) = (u_{i-4}, u_{i-3}, u_{i-2}, u_{i-1}, u_i).$$

Definition 56 (Exact differential transfer)**.** For input-pair difference Δu_i, induced history difference ΔH_i, and output difference bit ∇_i, define

$$T_i^{\Delta}[H', H] = \#\left\{u \in \mathbb{F}_2^2 :\ H' = \mathrm{next}(H, u),\ f_\star(H, u) \oplus f_\star(H \oplus \Delta H_i, u \oplus \Delta u_i) = \nabla_i\right\}.$$

Theorem 14 (Zero-fill differential-probability numerator)**.** *Let e_0 be the all-zero history and let $\mathbf{1}$ sum all terminal histories. The exact two-variable numerator is*

$$N_{\Delta\to\nabla} = \mathbf{1}^T T_{n-1}^{\Delta} \cdots T_0^{\Delta} e_0,$$

and

$$\mathrm{DP}^{\mathsf{zf}}_{\star,n}(\Delta \to \nabla) = 2^{-2n} N_{\Delta\to\nabla}.$$

For fixed operand c, the same construction uses 32 states and

$$\mathrm{DP}^{\mathsf{fixed}}_{\star,n,c}(\Delta \to \nabla) = 2^{-n} \mathbf{1}^T T_{n-1}^{\Delta,c} \cdots T_0^{\Delta,c} e_0.$$

Proof. The state vector after coordinate i counts exactly the accepted prefixes ending in each history. Multiplication by T_i^{Δ} extends each prefix by every current input pair that produces the required local output difference. Induction over coordinates gives the numerator. The denominators are the sizes of the two-variable and fixed-operand sample spaces. □

Definition 57 (Exact signed linear-correlation transfer)**.** For masks $(\alpha_i, \beta_i, \gamma_i)$, set

$$T_i^{\mathrm{LC}}[H', H] = \sum_{\substack{u=(a_i,b_i)\in\mathbb{F}_2^2 \\ H'=\mathrm{next}(H,u)}} (-1)^{\alpha_i a_i \oplus \beta_i b_i \oplus \gamma_i f_\star(H,u)}.$$

The exact two-variable Boolean Walsh correlation is

$$C_{\alpha,\beta,\gamma} = 2^{-2n} \mathbf{1}^T T_{n-1}^{\mathrm{LC}} \cdots T_0^{\mathrm{LC}} e_0.$$

The fixed-operand form uses the 32-state signed transfer and denominator 2^n.

Lemma 24 (Signed sums are not unsigned probabilities)**.** *The linear-correlation transfer preserves signs. The absolute value is not taken inside the transfer.*

Proof. A Walsh coefficient is a signed numerator. Opposite phases cancel. Taking absolute values before summation changes the object into an upper bound and destroys hull cancellation. □

34.3 XCR factor-graph differential/linear model

Definition 58 (S-round factor graph)**.** For fixed public n, counter, and constants, one S-round has affine rebinding factors, public-constant factors, four shadow factors

$$yy_1 = yy \boxplus_{64}^{\mathrm{sh,zf}} x_0, \quad zz_1 = zz \boxminus_{64}^{\mathrm{sh,zf}} y_0, \quad ww_1 = ww \boxplus_{64}^{\mathrm{sh,zf}} z_0, \quad xx_1 = xx \boxminus_{64}^{\mathrm{sh,zf}} w_0,$$

final XOR/rotation diffusion factors, output-extraction factors, and a counter-schedule factor.

Lemma 25 (Component products are scores)**.** *The product of four marginal shadow component probabilities is a trail score unless shared-lane compatibility is explicitly proved or enumerated. It is not automatically the exact one-round XCR probability.*

Proof. The four shadow calls consume affine views of the same four state lanes. Locally valid component transitions can conflict on shared lane assignments. Exact one-round probability therefore requires a joint compatibility sum, not just multiplication of marginal probabilities. □

Theorem 15 (Component-guided trail soundness)**.** *If a candidate satisfies every affine rebinding equation, every public-constant equation, every selected H-round or S-round local transition, every final diffusion equation, and the output extraction equation, then it is a valid trail in the XCR factor graph. Without full shared-lane checking, its score is only a search heuristic.*

Proof. The factor graph is the conjunction of those relations. A satisfying assignment of all factors is exactly a satisfying assignment of the step relation. Omitting shared-lane factors can admit tuples that no common state assignment realizes. □

Theorem 16 (Solver-free mathematical evidence supplied for XCR)**.** *The XCR component analysis in this paper establishes the following solver-free mathematical facts.*

1. *The deployed shadow addition and shadow subtraction maps are finite Boolean maps with six-position locality and exact DP/LC transfer descriptions.*
2. *The H-round, S-round, affine rebinding layer, final XOR/rotation diffusion, output extraction, and counter update define a total deterministic step relation R_{XCR} on $(\mathbb{W}_{64})^4 \times \mathbb{Z}_{2^{64}}$.*
3. *Any proposed differential or linear trail that satisfies all factor-graph constraints is code-faithful to* `PermutationARX` *and* `StateIteration`.

These statements are mathematical component theorems, not MILP/SAT/SMT optimality certificates and not an unconditional PRG proof.

Proof. Item 1 follows from the coordinate locality lemma and from the transfer definitions: the transfer matrices enumerate all admissible coordinate extensions under the zero-fill boundary, so the resulting numerators are exact for the local shadow relation. Item 2 follows because every operation in `PermutationARX` is a total function on fixed-width words: XOR, rotation, constant injection, byte substitution, shadow addition/subtraction, and counter addition are all total maps on finite domains. Their composition is therefore a total deterministic relation. Item 3 is the factor-graph soundness theorem applied to the code-faithful factorization of that relation. None of these arguments minimizes a trail weight, proves absence of better trails, or enumerates the full generator state space; those stronger claims would require an additional solver or exhaustive enumeration certificate. □

Corollary 7 (What a future XCR solver would certify)**.** *A future MILP, SAT, or SMT model for XCR should be read as an implementation of the factor relation R_{XCR}. If such a solver reports an optimal or complete status, then the certificate would concern the encoded XCR search space. The present paper does not report such a certificate.*

34.4 XCR pseudorandom-generator (PRG) game boundary

Definition 59 (XCR pseudorandom-generator assumption)**.** For resource bound (t, q), define

$$\epsilon_{\mathrm{XCR}}(t, q) = \max_{A} \left| \Pr[A(O_1, \ldots, O_q) = 1] - \Pr[A(U_1, \ldots, U_q) = 1] \right|,$$

where O_i are 128-bit XCR production outputs under chosen public inputs and U_i are independent uniform 128-bit strings. This is an assumption for the full stateful generator.

Theorem 17 (Schedule replacement bound under the XCR assumption)**.** *Let* LOPC *use current XCR-derived round material, and let* LOPC* *replace the consumed XCR output streams by ideal independent 128-bit streams while keeping the folding equations. Any adversary using q XCR output calls has distinguishing advantage at most*

$$\epsilon_{\mathrm{XCR}}(t, q) + \binom{q}{2} 2^{-128}.$$

Proof. Replace the XCR output streams by ideal samples in hybrids. The key-state folding is deterministic post-processing, so it cannot increase distinguishing advantage. Repeated ideal 128-bit samples contribute the birthday term. □

Remark 17 (State-compromise boundary). The current generator is deterministic after initialization. It does not claim prediction resistance after full state compromise without fresh entropy.

35 Component C Security: Key Switch Layer

35.1 MixLinearTransform propagation

For a fixed generated key state, the four branches of `MixLinearTransform_Forward/Backward` are reversible. The XOR and complemented-XOR branches preserve XOR differences up to the affine constant part and preserve linear masks up to the corresponding phase. The rotation branches move difference and mask positions by the selected rotation amount. Thus the layer changes the geometry of active positions without creating a standalone nonlinear substitution table.

35.2 Keyed choice effect

The choice function is derived from XCR output words, for example by folding words such as $(a \oplus b \oplus c \oplus d)\&3$ in the implementation. For fixed round material the choice is deterministic; from the construction boundary it is key/nonce/state dependent through XCR.

35.3 RandomBitTweak propagation

RandomBitTweak is a fixed XOR constant injection once the round material is fixed. For XOR differentials, the fixed constant cancels. For linear correlations, the fixed constant contributes a sign/phase. Its composition role is to make the selected bit positions depend on generated round material.

35.4 Branchless implementation boundary

The implementation uses branchless mask selection for the keyed switching layer through `ConstantTimeEqualMask`. This avoids secret-dependent switch branches in this layer, but it is not claimed here as a complete constant-time proof for the entire implementation.

36 Composition-Level Security Discussion

36.1 Code-bound semantic object of the current LOPC round

Let $\mathbb{W}_{32} = \mathbb{Z}/2^{32}\mathbb{Z}$ and $\mathbb{W}_{64} = \mathbb{Z}/2^{64}\mathbb{Z}$. Let

$$\mathrm{pack}(u, v) = 2^{32}u + v, \qquad \mathrm{unpack}(X) = (X \gg 32,\ X \bmod 2^{32}).$$

The current LOPC state is a pair $(L_0, L_1) \in \mathbb{W}_{64}^2$, unpacked as

$$(w_0, w_1) = \mathrm{unpack}(L_0), \qquad (w_2, w_3) = \mathrm{unpack}(L_1).$$

Let $N : \mathbb{W}_{32}^2 \to \mathbb{W}_{32}^2$ denote the current NeoAlzette V6.5 second-schedule permutation, and let N^{-1} denote the implemented backward map.

Definition 60 (Current round-material schedule)**.** For a session seed s, a nonce/call value ν, a master key $K = (K_0, K_1) \in \mathbb{W}_{64}^2$, and round index i, the current C++ function `GenerateAndStoreKeyStates` uses two persistent XCR member instances. The public inputs to those instances are

$$n_L = \nu \oplus i, \qquad n_R = (\nu \oplus (i \ll 1)) \oplus (i \gg 1).$$

If the two calls return (a, b) and (c, d), then the stored round data are

$$\begin{aligned}
sk_i^{(0)} &= (K_0 + a) \oplus \mathrm{rotr}_{64}(c, i \bmod 64),\\
sk_i^{(1)} &= (K_1 - b) \oplus \mathrm{rotr}_{64}(d, (i+1) \bmod 64),\\
cf_i &= (a \oplus b \oplus c \oplus d) \bmod 4,\\
\rho_i^{(0)} &= \big((a \oplus b \oplus c \oplus d) \oplus \mathrm{rotl}_{64}(sk_i^{(0)}, 1) \oplus \mathrm{rotl}_{64}(sk_i^{(1)}, 3)\big) \bmod 64,\\
\rho_i^{(1)} &= \left\lfloor \frac{(a \oplus b \oplus c \oplus d) \oplus \mathrm{rotl}_{64}(sk_i^{(0)}, 1) \oplus \mathrm{rotl}_{64}(sk_i^{(1)}, 3)}{2^6} \right\rfloor \bmod 64.
\end{aligned}$$

The generator calls mutate the two XCR states; hence this schedule is stateful, not a pure function of (K, ν, i) unless the complete initialized XCR states are included in the domain.

Definition 61 (Current LOPC encryption round)**.** For stored key state $\kappa_i = (sk_i^{(0)}, sk_i^{(1)}, cf_i, \rho_i^{(0)}, \rho_i^{(1)})$, one encryption round is the following composition. First xor the two 64-bit lanes with the two subkey lanes. Then unpack the lanes, apply diagonal NeoAlzette calls

$$(w_0, w_2) \leftarrow N(w_0, w_2), \qquad (w_1, w_3) \leftarrow N(w_1, w_3),$$

repack them as $(\mathrm{pack}(w_0, w_1), \mathrm{pack}(w_2, w_3))$, and finally apply the keyed switching layer selected by cf_i. The four switch cases are xor by subkey, complement-xor by subkey, rotate-left by $\rho_i^{(1)}$, and rotate-right by $\rho_i^{(1)}$. The current implementation then applies a one-bit xor tweak selected by $\rho_i^{(0)}$ to one lane according to cf_i.

Theorem 18 (Code-level invertibility of the LOPC round)**.** *For every stored key state κ_i, the current LOPC decryption round is the inverse of the current LOPC encryption round.*

Proof. The subkey xor at the start of encryption is self-inverse and is applied in the opposite position in decryption. The unpack and pack operations are inverse bit reinterpretations. The two diagonal NeoAlzette calls are permutations because the current NeoAlzette source contains an explicit backward method reversing whitening, bridge updates, modular additions/subtractions, and the two injection calls in reverse order. The keyed switching layer is inverted case-by-case: xor is self-inverse, complement-xor is self-inverse when the same subkey is used, and the two rotation cases are inverted by the opposite rotation. The one-bit tweak is xor by a public unit word and is self-inverse. Therefore the composition of inverse steps in reverse order gives exactly the decryption round. □

Engineering interpretation. The theorem proves functional correctness: decrypting with the same regenerated key-state sequence returns the input block. It does not prove pseudorandomness. The security question is whether the round function, key-state generation, and their composition hide structure from an adversary; that is handled only as component evidence plus future composition analysis below.

Standalone ARX algebraic-analysis layer. The general matrix-analytic ARX complexity material, the explicit 8-bit matrix model, and the Niu–Sun–Hu additive-differential refinement are not XCR-specific and are not KeySwitchLayer-specific. They have therefore been collected as a standalone ARX algebraic security-analysis layer in Section 32. The composition section below only imports the resulting boundary: these matrix tools explain and audit ARX components, but they are not by themselves a complete LOPC security proof.

36.2 NeoAlzette component theory after absorption

The NeoAlzette component theory used by the formal-security section is now stated in Section 22. That section contains the state semantics, V6.5 second-schedule map, QBF/C-D injection theory, differential MILP model, linear MILP model, and HULL/Forest proof boundary. This security section does not redefine those objects. It uses the following reading rule.

Corollary 8 (Imported NeoAlzette evidence layer)**.** *Under the schedule, oracle, and MILP-box assumptions stated in Section 22 and Appendix A, a solver certificate for the NeoAlzette differential or linear MILP is a certificate for the encoded NeoAlzette component model. It is not a certificate for the full LOPC composition unless the XCR and Key Switch Layer assumptions are also discharged.*

Remark 18 (Why the duplicate text was collapsed)*.* The earlier draft repeated parts of the NeoAlzette proof layer in two places. The current version keeps the detailed mathematical object in the component chapter and keeps only the composition-boundary rule here.

36.3 XCR component proof boundary after the rebuilt XCR section

The Shadow-Boxed Arithmetic definitions, exact DP/LC transfer model, Hybrid8 layer, code-faithful XCR step, factor-graph warning, and PRG assumption are stated in Section 23. The formal-security section uses that material instead of redefining it. The boundary is unchanged: exact shadow arithmetic is component theory; full XCR pseudorandomness remains the explicit assumption of Section 34.4.

The algebraic matrix discussion that was previously adjacent to this paragraph is not an XCR-specific argument. It has therefore been moved to the standalone ARX algebraic analysis layer in Section 32.

36.4 Cryptographic Game Analysis of Encryption and Decryption Functions

XCR schedule replacement. The real-or-random replacement statement for XCR-derived round material is now stated in Section 34.4, after the XCR step, initialization, and factor-graph boundaries have been defined.

36.5 LOPC composition boundary

Theorem 19 (What the present evidence proves)**.** *The current manuscript proves or imports the following component facts.*

1. *The LOPC encryption and decryption procedures are mutual inverses under the same generated key-state schedule.*

2. *The NeoAlzette layer used by LOPC is bound to the V6.5 second-schedule C++ object and has dedicated differential/linear MILP/HULL analysis infrastructure.*

3. *The current NeoAlzette time-limited runs are valid resource-limited incumbents, not optimality certificates.*

4. *The XCR S-round shadow interactions are backed by exact zero-fill Shadow-Boxed Arithmetic component oracles.*

5. *The matrix-analytic ARX discussion is a real finite-state/carry-state explanation of nonlinearity and search complexity, not a stand-alone security proof.*

Proof. Item 1 is Theorem 18. Items 2 and 3 follow from the Component-A MILP box library in Section 33 and the mathematical oracle backend in Appendix A, together with the external solver-status artifacts and the stated run-status boundary. Item 4 follows from Theorem 4 and the component binding of XCR's S-round. Item 5 follows from Theorem 4. None of these items asserts a completed full-system PRP theorem. □

Theorem 20 (Composition statement under explicit assumptions)**.** *If the XCR schedule is replaced by ideal independent round material and if the resulting keyed round family is modeled as a pseudorandom permutation family after the stated number of rounds, then the usual nonce-respecting block-transform games reduce to those assumptions plus generic birthday terms. In the real construction, the additional gap is bounded only by the XCR PRG assumption and by future composition analysis of the NeoAlzette/key-schedule interaction.*

Proof. The idealized experiment supplies independent round material to the deterministic LOPC round function. Under the assumed PRP behavior of the idealized keyed round family, game hops against encryption/decryption or random-permutation oracles are standard. Returning from the idealized schedule to the real schedule costs the XCR replacement advantage of Theorem 17. The proof therefore reduces security to explicit assumptions; it does not eliminate them. □

Final boundary. NeoAlzette is the analysed ARX-box component. Shadow-Boxed Arithmetic is the exact component theory for the current XCR S-round. XCR is implementation-backed and component-theory-backed, but full PRG security remains assumption-backed. LOPC integrates these pieces, but total composition security remains future composition analysis.

Part VI
Software Implementation

37 Implementation Benchmark Methodology

This section records the benchmark method only. It does not repeat the security evaluation of Part V and does not promote empirical timings to cryptanalytic evidence. Performance numbers are platform-dependent and must be reported together with compiler, flags, CPU, input sizes, round count, reset policy, and timing statistics before being used as a comparative claim.

Platform. The historical harness used an Intel64 Family 6 Model 151 Stepping 2 processor at approximately 3.61 GHz, 65,277 MB RAM, a 64-bit Linux toolchain, and compiler optimizations enabled. Future tables should also report compiler version, exact flags, CPU governor, and commit hash.

Data generation. For each input size, an MT19937-64 generator seeded with 1 produces pseudorandom test data. Inputs are aligned to 128-bit blocks for LOPC.

Cipher parameters. LOPC is measured as a 128-bit block core with a 128-bit key and counter-indexed `number_once` values. The XCR instances are reset between independent trials according to the public-session rule. If external algorithms such as Ascon or ChaCha20 are measured in the same harness, their exact parameter set, nonce, associated-data handling, and authentication mode must be reported separately.

Message sizes and metrics. Message lengths double from 128 bits up to the selected maximum size. The benchmark should report wall-clock time, throughput, and preferably cycles/byte. Encryption and decryption must be measured separately.

Current status. The previous qualitative performance statements are treated as historical observations, not claims. A future version should replace them with a table containing repeated trials, median/mean, variance or confidence intervals, compiler flags, and source-artifact hash.

38 Global C++20 Implementation Boundary

The executable objects of this manuscript are the LOPC wrapper files `LittleOaldresPuzzle_Cryptic.cpp/.h`, the XCR files `XorConstantRotation.cpp/.h`, the round-constant generator `GenerateAndDisplay_XorConstantRotation_RoundConstant.py`, the NeoAlzette MILP/HULL artifact, and the Shadow-Boxed Arithmetic artifact used for the XCR S-round component theory.

39 Component A Implementation: NeoAlzette

The NeoAlzette implementation notes identify the executable artifact corresponding to the mathematical specification in Section 22. The component specification itself states the constants, mask maps, injection equations, routed addends, and forward/backward schedules; the C++ file is now an artifact for reproduction rather than a separate appendix.

- **Forward/backward entry points:** `NeoAlzetteSubstitutionBox::forward` and `NeoAlzetteSubstitutionBox::backward`.
- **Dynamic mask generators:** `generate_dynamic_diffusion_mask0` and `generate_dynamic_diffusion_mask1`.
- **Injection circuits:** `cd_injection_from_B` and `cd_injection_from_A`.

40 Component B Implementation: XCR

The XCR implementation boundary is defined by `PermutationARX`, `StateIteration`, `GenerateSubKey128`, the Hybrid8 lifted byte arithmetic functions, the BigS layer, and the shadow carry/borrow operations. The corresponding source files are included in the artifact package.

41 Component C Implementation: Key Switch Layer

The Key Switch implementation boundary is defined by `GenerateAndStoreKeyStates`, `MixLinearTransform_Forward`, `MixLinearTransform_Backward`, `ConstantTimeEqualMask`, and the RandomBitTweak lines in the encryption and decryption core functions.

42 Constants and Reproducibility

42.1 Round-constant generation for XCR

The normative XCR constant-stream rule is specified in Section 23.5. The implementation script fixes `mp.dps=100`, evaluates the V2 polynomial Weyl fractional sequence, and concatenates non-padded binary encodings of the scaled 128-bit fractional integers. Therefore the generated stream length is artifact-defined by the script, not by the formula 150×128, unless fixed-width padding is explicitly enabled. The first four XCR table words are manual public anchors; the first generated word maps to table entry $T[4]$ in the current artifact.

43 Repository, Run Tests, and Artifact Notes

The repository is treated as a research artifact rather than a standardized cryptographic library. It contains the LOPC core analyzed in this manuscript, the companion OPC artifact recorded in Appendix B, the XCR round-material generator, the constant-generation script, and the analysis materials referenced by the component sections. Readers should follow the repository README and build scripts associated with the artifact snapshot; raw repository links are intentionally not used as mathematical references.

Correct use of the current LOPC public-session API requires resetting the internal XCR state between independent top-level sessions, as the implementation does. For multi-block processing under one seed and key, the caller must assign distinct `number_once` values to successive 128-bit blocks. Authentication must be supplied by an external MAC or AEAD wrapper.

43.1 Artifact and Reproducibility Notes

The implementation artifacts associated with this manuscript should be treated as research artifacts rather than as a standardized cryptographic library. Reproducibility claims in this paper are limited to the stated source files, build scripts, MILP/HULL runners, and validation logs. The artifact is intended to let a reader inspect the exact C++ schedule, rerun the engineering tests, and distinguish executable evidence from assumptions.

Part VII
Hardware and Implementation Considerations

44 Global Cost Model

The LOPC reference structure uses a 128-bit data state, two 64-bit lanes, four 32-bit words for the diagonal NeoAlzette layer, two XCR generator instances during key-state generation, and a 300-entry XCR constant table.

45 Component A Notes: NeoAlzette

NeoAlzette uses 32-bit add/sub operations, rotations, XOR operations, and Boolean injection logic. The NeoAlzette ARX-box itself does not require a lookup-table S-box in the reference specification.

46 Component B Notes: XCR

XCR uses 64-bit state words, Hybrid8 byte transformations, Shadow carry/borrow finite-window arithmetic, public counter scheduling, and constant-table access. Hybrid8 may be implemented through logical formulae or byte-table style implementations depending on the target artifact; this manuscript does not claim a synthesized hardware design.

47 Component C Notes: Key Switch Layer

The Key Switch Layer uses XOR, NOT, rotations, branchless mask selection, and a single-bit tweak. It is cheap to invert because each selected branch is reversible and the bit tweak is self-inverse.

48 Limitations

No synthesized area, gate-equivalent count, or hardware throughput number is claimed unless a separate hardware artifact provides it. The present implementation discussion is therefore an engineering-cost and structure discussion, not a hardware benchmark.

Part VIII
Conclusion

49 Summary

The reorganized manuscript presents LOPC as a three-component construction: NeoAlzette, XCR, and the Key Switch Layer. The new structure keeps the original technical material while moving it into a Part-based layout that separates specification, design rationale, security evaluation, implementation, and artifact material.

50 Component Summaries

NeoAlzette currently has a specification, inverse, Quadratic Boolean Front Half rationale, and MILP/HULL-style analysis artifact. XCR currently has a reference implementation, Hybrid8/Shadow structure, Shadow-oracle support, and reduced diagnostics, but no full CSPRNG proof is claimed. The Key Switch Layer currently has a specification, inverse, branchless implementation, and affine/permutation propagation role.

51 Future Work

Future work includes complete endpoint HULL enumeration where feasible, full XCR Hybrid8–Shadow composition analysis, hardware synthesis, and explicit mode/AEAD/security-goal formalization if LOPC is promoted from a block/permutation-style core to a full encryption mode.

References

[AFK+08] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of latin dances: Analysis of salsa, chacha, and rumba. In *Fast Software Encryption 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 470–488. Springer, 2008. Full version: Cryptology ePrint Archive, Report 2007/472.

[ARS+20] S. A. Azimi, A. Ranea, M. Salmasizadeh, J. Mohajeri, M. R. Aref, and V. Rijmen. A bit-vector differential model for the modular addition by a constant. In *ASIACRYPT 2020*, volume 12491 of *Lecture Notes in Computer Science*. Springer, 2020.

[ARS+22] S. A. Azimi, A. Ranea, M. Salmasizadeh, J. Mohajeri, M. R. Aref, and V. Rijmen. A bit-vector differential model for the modular addition by a constant and its applications to differential and impossible-differential cryptanalysis. *Designs, Codes and Cryptography*, 90:1797–1855, 2022.

[B+08a] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer, 2008.

[B+08b] Daniel J Bernstein et al. The chacha family of stream ciphers. In *Workshop record of SASC*. Citeseer, 2008.

[BA20] E. Bagherzadeh and Z. Ahmadian. Milp-based automatic differential search for lea and hight block ciphers. IET Information Security, 2020.

[BBdS+20] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Alzette: A 64-bit arx-box (feat. crax and trax). In *Advances in Cryptology – CRYPTO 2020*, volume 12171 of *Lecture Notes in Computer Science*, pages 419–448. Springer, 2020. https://eprint.iacr.org/2019/1378.

[BBdS+21] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Amir Moradi, Léo Perrin, Alireza Rezaei Shahmirzadi, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Schwaemm and esch: Lightweight authenticated encryption and hashing using the Sparkle permutation family. Specification version 1.2, 2021.

[Ber05] Daniel J Bernstein. Salsa20 design. *Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. Chicago*, 2005.

[Ber08] Daniel J. Bernstein. The salsa20 family of stream ciphers. In *New Stream Cipher Designs: The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*, pages 84–97. Springer, 2008.

[BK98] Alex Biryukov and Eyal Kushilevitz. Improved cryptanalysis of rc5. In *EUROCRYPT 1998*, volume 1403 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 1998.

[BP17] Alex Biryukov and Leo Perrin. State of the art in lightweight symmetric cryptography. Cryptology ePrint Archive, Paper 2017/511, 2017. https://eprint.iacr.org/2017/511.

[BPV99] Johan Borst, Bart Preneel, and Joos Vandewalle. Linear cryptanalysis of rc5 and rc6. In *FSE 1999*, volume 1636 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1999.

[BV] Alex Biryukov and Vesselin Velichkov. Automatic search for differential trails in arx ciphers. Cryptology preprint.

[BVC16] Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. Automatic search for the best trails in arx: Application to block cipher speck. In *Fast Software Encryption 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 289–310. Springer, 2016.

[Car10a] Claude Carlet. Boolean functions for cryptography and error-correcting codes. In *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 257–397. Cambridge University Press, 2010.

[Car10b] Claude Carlet. Vectorial boolean functions for cryptography. In *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 398–469. Cambridge University Press, 2010.

[CCWZ22] Weilin Cai, Heng Chen, Ziheng Wang, and Xingjun Zhang. Implementation and optimization of chacha20 stream cipher on sunway taihulight supercomputer. *The Journal of Supercomputing*, 78(3):4199–4216, 2022.

[CRRY99] Scott Contini, Ronald L. Rivest, M. J. B. Robshaw, and Yiqun Lisa Yin. Improved analysis of some simplified variants of rc6. In *Fast Software Encryption 1999*, volume 1636 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1999.

[Dat] Data Security Technology. Zuc stream cipher algorithm, part 1: Specification of the zuc stream cipher algorithm. GM/T 0001.1 and related public specification material.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2: Lightweight authenticated encryption and hashing. *Journal of Cryptology*, 34(3):33, 2021.

[DPU+16] R. Dinu, L. Perrin, A. Udovenko, V. Velichkov, J. Großschädl, and A. Biryukov. Design strategies for arx with provable bounds: Sparx and lax. In *ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*. Springer, 2016.

[DRS15] S. M. Dehnavi, A. Mahmoodi Rishakani, and M. R. Mirzaee Shamsabad. A more explicit formula for linear probabilities of modular addition modulo a power of two. Cryptology preprint, 2015.

[DRS+16] S. M. Dehnavi, A. Mahmoodi Rishakani, M. R. Mirzaee Shamsabad, Hamidreza Maimani, and Einollah Pasha. Cryptographic properties of addition modulo 2^n. Preprint, 2016.

[EAY] M. ElSheikh, A. Abdelkhalek, and A. M. Youssef. On milp-based automatic search for differential trails through modular additions with application to bel-t. Cryptology preprint.

[ElS21] M. ElSheikh. *MILP-aided Cryptanalysis of Some Block Ciphers*. PhD thesis, Concordia University, 2021.

[Fei12] Du Fei. Research on safety of arx structures. Master's thesis, Xi'an University of Electronic Science and Technology, China, 2012.

[FFW23] Ting Fan, Wei Feng, and Yongzhuang Wei. Design and analysis of a large-state lightweight cryptographic s-box. *Journal of Xidian University*, 50(4):170–179, 2023.

[FWG+16] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. Milp-based automatic search algorithms for differential and linear trails for speck. In *Fast Software Encryption 2016*, volume 9783 of *Lecture Notes in Computer Science*, pages 268–288. Springer, 2016.

[GD10] Jie Guan and Lin Ding. Probability analysis and application of xor difference between $(x + k) \bmod 2^n$ and $x \oplus k$. In *IEEE Conference Proceedings*. IEEE, 2010.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM (JACM)*, 33(4):792–807, 1986.

[GM22] Nasratullah Ghafoori and Atsuko Miyaji. Differential cryptanalysis of salsa20 based on comprehensive analysis of pnbs. In *Information Security Practice and Experience – ISPEC 2022*, Lecture Notes in Computer Science, pages 520–536. Springer, 2022.

[HW19] M. Huang and L. Wang. Automatic search for the linear (hull) characteristics of arx ciphers: Applied to speck, sparx, chaskey and cham-64. Cryptology ePrint Archive, Report 2019/1319, 2019.

[HXW22] M. Huang, Z. Xu, and L. Wang. On the probability and automatic search of rotational-xor cryptanalysis on arx ciphers. *The Computer Journal*, 65(12):3062–3080, 2022.

[JR94] Burton S. Kaliski Jr. and M. J. B. Robshaw. Linear cryptanalysis using multiple approximations. In *CRYPTO 1994*, volume 839 of *Lecture Notes in Computer Science*, pages 26–39. Springer, 1994.

[JY95] Burton S. Kaliski Jr. and Yiqun Lisa Yin. On differential and linear cryptanalysis of the rc5 encryption algorithm. In *CRYPTO 1995*, volume 963 of *Lecture Notes in Computer Science*, pages 171–184. Springer, 1995.

[JYY+16] ZHUANG J, MA Y, ZHU S Y, LIN J Q, and JING J W. Q_value test: A new method on randomness statistical test. *China Journal of Cryptologic Research*, 3(2):192–201, 2016.

[KM96] Lars R. Knudsen and Willi Meier. Improved differential attacks on rc5. In *CRYPTO 1996*, volume 1109 of *Lecture Notes in Computer Science*, pages 216–228. Springer, 1996.

[KNP+15] Dmitry Khovratovich, Ivica Nikolic, Josef Pieprzyk, Przemyslaw Sokolowski, and Ron Steinfeld. Rotational cryptanalysis of ARX revisited. Cryptology ePrint Archive, Paper 2015/095, 2015.

[Leu15] Gaëtan Leurent. Differential and linear cryptanalysis of arx with partitioning: Application to feal and chaskey. HAL preprint, hal-01243166, 2015.

[LHS+11] Zhenhua Li, Xiaoying Huang, Fei Shen, Jihong Teng, and Kai Li. Analysis of addition modulo 2^n on boolean function. In *2011 Third International Conference on Multimedia Information Networking and Security*, pages 386–389. IEEE, 2011.

[LJC23] Guangqiu Lv, Chenhui Jin, and Ting Cui. A miqcp-based automatic search algorithm for differential-linear trails of arx ciphers(long paper). Cryptology ePrint Archive, Paper 2023/259, 2023. https://eprint.iacr.org/2023/259.

[LLJW21] Zhengbin Liu, Yongqiang Li, Lin Jiao, and Mingsheng Wang. A new method for searching optimal differential and linear trails in arx ciphers. *IEEE Transactions on Information Theory*, 2021. Also available as Cryptology ePrint Archive, Report 2019/1438.

[LM02] Helger Lipmaa and Shiho Moriai. Efficient algorithms for computing differential properties of addition. In *Fast Software Encryption 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2002.

[LNS+23] Y. Liu, Z. Niu, S. Sun, et al. Rotational differential-linear cryptanalysis revisited. *Journal of Cryptology*, 36(3), 2023.

[LP24] Gaëtan Leurent and Cédric Pernot. Design of a linear layer optimised for bitsliced 32-bit implementation. IACR Transactions on Symmetric Cryptology, 2024.

[LRH+21] Jun Liu, Vincent Rijmen, Yupu Hu, Jie Chen, and Baocang Wang. Warx: efficient white-box block cipher based on arx primitives and random mds matrix. *Science China Information Sciences*, 65(3):132302, 2021.

[LSL21] Y. Liu, S. Sun, and C. Li. Rotational cryptanalysis from a differential-linear perspective. In *EUROCRYPT 2021*, 2021.

[LWR16] Y. Liu, Q. Wang, and Vincent Rijmen. Automatic search of linear trails in arx with applications to speck and chaskey. In *ACNS 2016*, volume 9696 of *Lecture Notes in Computer Science*, pages 485–499. Springer, 2016.

[Mac01] A. W. Machado. Differential probability of modular addition with a constant operand. Cryptology ePrint Archive, Report 2001/052, 2001.

[Miy98] Hiroshi Miyano. Addend dependency of differential/linear probability of addition. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E81-A(1):106–110, 1998.

[MKA+21] Nicky Mouha, Nikolay Kolomeec, Danil Akhtiamov, Ivan Sutormin, Matvey Panferov, Kseniya Titova, Tatiana Bonich, Evgeniya Ishchukova, Natalia Tokareva, and Bulat Zhantulikov. Maximums of the additive differential probability of exclusive-or. *IACR Transactions on Symmetric Cryptology*, 2021(2):292–313, 2021.

[MN02] Atsuko Miyaji and Masao Nonaka. Cryptanalysis of the reduced-round rc6. In *Information and Communications Security – ICICS 2002*, volume 2513 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 2002.

[MPM15] Subhamoy Maitra, Goutam Paul, and Willi Meier. Salsa20 cryptanalysis: New moves and revisiting old styles. Cryptology ePrint Archive, Paper 2015/217, 2015. https://eprint.iacr.org/2015/217.

[Nat18] National Institute of Standards and Technology. Lightweight cryptography project. Project website, 2018. Accessed 2026-06-28.

[NBHW25] Zhongfeng Niu, Tim Beyne, Kai Hu, and Meiqin Wang. Cryptanalytic properties of mealy machines, 2025. Project reference manuscript supplied with the LOPC revision materials.

[NL18] Yoav Nir and Adam Langley. Chacha20 and poly1305 for ietf protocols. RFC 8439, Internet Engineering Task Force, 2018.

[NSH23] Zhongfeng Niu, Siwei Sun, and Lei Hu. On the additive differential probability of arx construction. *Journal of Surveillance, Security and Safety*, 4(4), 2023.

[NSL+22] Z. Niu, S. Sun, Y. Liu, et al. Rotational differential-linear distinguishers of arx ciphers with arbitrary output linear masks. In *CRYPTO 2022*, 2022.

[Nyb95] Kaisa Nyberg. Linear approximation of block ciphers. In *Advances in Cryptology — EUROCRYPT'94*, pages 439–444. Springer Berlin Heidelberg, 1995.

[RVP22] Adrián Ranea, Joachim Vandersmissen, and Bart Preneel. Implicit white-box implementations: White-boxing arx ciphers. In *Advances in Cryptology – CRYPTO 2022*, volume 13507 of *Lecture Notes in Computer Science*, pages 33–63. Springer, 2022.

[Saa11] Markku-Juhani O. Saarinen. Cryptographic analysis of all 4 x 4 - bit s-boxes. Cryptology ePrint Archive, Paper 2011/218, 2011.

[SC21] Lamis Sleem and Raphaël Couturier. Speck-R: An ultra light-weight cryptographic scheme for internet of things. *Multimedia Tools and Applications*, 80:17067–17091, 2021.

[SDS+22] Ronaldo Serrano, Ckristian Duran, Marco Sarmiento, Cong-Kha Pham, and Trong-Thuc Hoang. Chacha20-poly1305 authenticated encryption with additional data for transport layer security 1.3. *Cryptography*, 6(2):30, 2022.

[Sel98] Ali Aydın Selçuk. New results in linear cryptanalysis of rc5. In *Fast Software Encryption 1998*, volume 1372 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[SG13] Ernst Schulte-Geers. On ccz-equivalence of addition mod 2^n. *Designs, Codes and Cryptography*, 66:111–127, 2013.

[Sha49] Claude E Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[SHW+14] Siwei Sun, Lei Hu, P. Wang, K. Qiao, X. Ma, and L. Song. Towards finding the best characteristics of some bit-oriented block ciphers and automatic enumeration of differential and linear characteristics with predefined properties. In *ASIACRYPT 2014*, 2014.

[STK02] T. Shimoyama, M. Takenaka, and T. Koshiba. Multiple linear cryptanalysis of a reduced round rc6. In *FSE 2002*, 2002.

[TMC+25] Meltem S"onmez Turan, Kerry McKay, Donghoon Chang, Lawrence Bassham, Morris Dworkin, Jinkeon Kang, John Kelsey, Nicky Mouha, Andrew Regenscheid, and John Waller. Ascon-Based Lightweight Cryptography Standards for Constrained Devices. NIST Special Publication 800-232, National Institute of Standards and Technology, 2025.

[TSK+07] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, and Hiroki Nakashima. Differential cryptanalysis of salsa20/8. In *Workshop Record of SASC*, volume 28. Citeseer, 2007.

[TW18] R. Tse and W. Wong. The sm4 blockcipher algorithm and its modes of operations. Internet-Draft draft-ribose-cfrg-sm4-02, IETF, 2018.

[Twi26a] Twilight-Dream. LittleOaldresPuzzle_Cryptic: Reference implementation and reproducibility artifact. Project artifact and reference implementation, 2026. Implementation artifact used to reproduce the LOPC core described in this manuscript.

[Twi26b] Twilight-Dream. NeoAlzette ARX-box Specification Version 6.5 and MILP/HULL Analysis Artifact. https://github.com/Twilight-Dream-Of-Magic/NeoAlzette_ARX_CryptoAnalysis_MILP/blob/main/NeoAlzette_ARX_box_Specification_Version_6_5/iacrdoc.pdf, 2026. Specification and analysis artifact for the NeoAlzette V6.5 component.

[Twi26c] Twilight-Dream. A unified theory of shadow-boxed modular addition and subtraction: Raw binary cores, modulus selection, operand binding, prime fixed-operand closure, exact differential and correlation analysis, and the xcr interface. Technical report in preparation by the present author, 2026. In preparation; companion background for the shadow-boxed arithmetic operators used by the XCR discussion.

[Twi26d] Twilight-Dream-Of-Magic. A bit-vector linear correlation model for modular addition by a constant (bvcorr-flat). Technical manuscript, https://github.com/Twilight-Dream-Of-Magic/exact-fixed-addend-theory, 2026. Repository: https://github.com/Twilight-Dream-Of-Magic/exact-fixed-addend-theory.

[Twi26e] Twilight-Dream-Of-Magic. Exact fixed-addend differential and linear theory of modular addition with linear-time algorithms. Technical manuscript, https://github.com/Twilight-Dream-Of-Magic/exact-fixed-addend-theory, 2026. Repository: https://github.com/Twilight-Dream-Of-Magic/exact-fixed-addend-theory.

[Twi26f] Twilight-Dream-Of-Magic. linear-box-search: Search tools for binary circulant linear layers. GitHub repository, https://github.com/Twilight-Dream-Of-Magic/linear-box-search, 2026. Repository artifact for linear-layer search and decomposition experiments.

[VMCP11] Vesselin Velichkov, Nicky Mouha, Christophe De Cannière, and Bart Preneel. The additive differential probability of ARX. In *Fast Software Encryption 2011*, volume 6733 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 2011.

[Wal03a] Johan Wallén. Linear approximations of addition modulo 2^n. In *Fast Software Encryption 2003*, volume 2887 of *Lecture Notes in Computer Science*, pages 261–273. Springer, 2003.

[Wal03b] Johan Wallén. On the differential and linear properties of addition. Technical Report HUT-TCS-A84, Helsinki University of Technology, Laboratory for Theoretical Computer Science, 2003.

[XLW22] Zheng Xu, Yongqiang Li, and Mingsheng Wang. Security analysis of alzette. *Journal of Cryptologic Research*, 9(4):698–708, 2022.

[Ya17] Han Ya. Automatic method for searching impossible differentials and zero-correlation linear hulls of arx block ciphers. *Chinese Journal of Network and Information Security*, 3(7):58, 2017.

[YT24] Li Yu and Je Sen Teh. Revisiting boomerang attacks on lightweight ARX and AND-RX ciphers with applications to KATAN, SIMON and CHAM. Cryptology ePrint Archive, Paper 2024/2039, 2024.

Part IX
Appendices

A Imported NeoAlzette Oracle and MILP Backbone

A.1 Purpose and mathematical scope

This appendix gives the mathematical backbone behind the Component-A MILP boxes used for NeoAlzette. It is not a source-code listing and it is not an implementation trace. The implementation files instantiate the objects defined here; they do not replace the definitions. The artifact-to-object correspondence is postponed until Subsection A.20.

The immediate purpose is to justify the MILP boxes and oracle hooks used in Section 33. The central point is the fixed-public arithmetic boundary. In NeoAlzette, public constants enter the schedule through fixed-public additions or subtractions; those relations have a one-variable probability space and cannot be replaced by the averaged two-variable modular-addition box. The fixed-public differential and linear oracles are therefore stated first. The two-variable modular-addition oracles of Lipmaa–Moriai and Wallén are then stated as the second layer, used for the CD-derived variable-addend additions and the subtraction wrappers [LM02, Wal03a, SG13].

All words are read using the common bit-order and accessor convention of Part II. Unless explicitly stated otherwise, bit index 0 is the least significant bit and all sums over words are over the indicated finite sample space.

A.2 Notation and probability spaces

Let $n \in \{32, 64\}$ unless a general word size is stated. For a word map F, the XOR-differential probability and Walsh-linear correlation must be read with the correct sample space.

Two-variable differential addition. For

$$Z = X \boxplus_n Y,$$

the sampled variables are $(X, Y) \in (\mathbb{Z}/2^n\mathbb{Z})^2$. With XOR differences $(\alpha, \beta \to \gamma)$,

$$DP_{\boxplus_n}(\alpha, \beta \to \gamma) = \Pr_{X,Y}\left[(X \boxplus_n Y) \oplus ((X \oplus \alpha) \boxplus_n (Y \oplus \beta)) = \gamma\right].$$

Two-variable linear addition. For masks $(\Gamma; A, B)$,

$$C_{\boxplus_n}(\Gamma; A, B) = 2^{-2n} \sum_{X,Y} (-1)^{\Gamma\cdot(X\boxplus_n Y)\oplus A\cdot X\oplus B\cdot Y}.$$

The sign of a linear characteristic is a genuine object. MELCC uses the absolute magnitude of a single characteristic, but endpoint linear hull aggregation must keep the signed contributions.

Fixed-public differential arithmetic. For a fixed public constant K, set

$$F_{+K}(X) = X \boxplus_n K, \qquad F_{-C}(X) = X - C \pmod{2^n}.$$

The sampled variable is X only. The exact count is

$$N^{\Delta}_{\pm K}(u, v) = \#\{X \in \mathbb{Z}/2^n\mathbb{Z} : F_{\pm K}(X \oplus u) = F_{\pm K}(X) \oplus v\},$$

and the exact differential probability is

$$DP_{\pm K}(u \to v) = 2^{-n} N^{\Delta}_{\pm K}(u, v).$$

This is the addend-dependent probability space studied by Miyano and by later constant-operand recurrence work [Miy98, Mac01, ARS+20, ARS+22].

Fixed-public linear arithmetic. For masks (a, b),

$$C_{\pm K}(a, b) = 2^{-n} \sum_{X} (-1)^{a \cdot X \oplus b \cdot F_{\pm K}(X)}.$$

The numerator is signed. A zero numerator means the mask pair is outside the live support of a linear characteristic. This is not a probability count.

Remark 19 (Fixed-public arithmetic is not averaged addition). The map $X \mapsto X \boxplus K$ samples one variable and conditions every carry transition on the concrete bit pattern of K. The map $(X, Y) \mapsto X \boxplus Y$ samples two variables and averages over the second operand. Replacing the former by the latter changes the probability space and proves a statement about a different local relation.

A.3 Modeling questions Q1 and Q2

The oracle layer used by the NeoAlzette MILP model separates two local modeling questions. This distinction is independent of whether the local arithmetic relation is a two-variable addition/subtraction relation or a fixed-public add/sub relation, and it is also independent of whether the query is differential or linear.

Question-1: complete-transition evaluation. A Question-1 oracle is a point evaluator. All visible boundary data of the local transition are already fixed. The oracle evaluates whether this exact transition is feasible and, if it is feasible, returns the local bill paid by that transition. In the differential case the bill is the differential weight. In the linear case the bill also includes support, sign, and absolute-correlation weight. Thus Q1 is the local exact-evaluation layer used to attach a cost to a fully specified characteristic edge.

Question-2: free-slot completion. A Question-2 oracle is a local completion problem. Only part of the visible boundary is fixed, and one visible slot is left free. The oracle optimizes over the free slot and returns the locally best completion together with its weight and, in the linear case, the sign of the selected transition when the sign is part of the object. Q2 is useful for local CDDT/CLAT-style ranking, pruning, and completion. It is not a substitute for the global SCIP/MILP search, because the locally best completion may conflict with the other local relations in the full NeoAlzette schedule.

A.3.1 Two-variable modular addition and subtraction

For the two-variable addition relation

$$Z = X \boxplus_n Y,$$

all probabilities or Walsh sums are over the pair (X, Y). The differential point evaluation is the Lipmaa–Moriai two-variable modular-addition object, while the linear point evaluation uses the Wallén/Schulte–Geers carry-correlation object [LM02, Wal03a, SG13]. The subtraction relation $Z = X \boxminus_n Y$ is not assigned an independent oracle: it is reduced to addition by the relabeling used in the main Component-A MILP boxes.

Differential Q1 for two-variable addition. For a visible boundary $(\alpha, \beta \to \gamma)$, define

$$Q_1^{\mathrm{DP},+}(\alpha,\beta,\gamma) = \begin{cases} (\mathsf{false}, +\infty), & DP_{\boxplus_n}(\alpha,\beta\to\gamma) = 0, \\ (\mathsf{true}, -\log_2 DP_{\boxplus_n}(\alpha,\beta\to\gamma)), & DP_{\boxplus_n}(\alpha,\beta\to\gamma) > 0. \end{cases}$$

This is the complete-transition query for a fixed difference triple; it does not complete a missing word.

Differential Q2 for two-variable addition. A typical free-slot query fixes two visible difference words and optimizes over the third:

$$Q_2^{\mathrm{DP},+}(\alpha,\beta) = \arg\min_{\gamma\in\mathbb{F}_2^n} \big(-\log_2 DP_{\boxplus_n}(\alpha,\beta\to\gamma)\big),$$

with analogous forms $Q_2^{\mathrm{DP},+}(\alpha,\gamma)$ and $Q_2^{\mathrm{DP},+}(\beta,\gamma)$ for completing the other missing slot. The returned object is the completing word and the optimized local differential weight. Subtraction is reduced by

$$Z = X \boxminus_n Y \quad\Longleftrightarrow\quad X = Z \boxplus_n Y,$$

so

$$Q_1^{\mathrm{DP},-}(\alpha_X,\alpha_Y,\alpha_Z) = Q_1^{\mathrm{DP},+}(\alpha_Z,\alpha_Y,\alpha_X),$$

with the same relabeling applied inside Q2 completion queries.

Linear Q1 for two-variable addition. Let $(\Gamma; A, B)$ denote the output mask on Z and the two input masks on X, Y. The complete linear point query is

$$Q_1^{\mathrm{LC},+}(\Gamma,A,B) = \begin{cases} (\mathsf{false}, 0, +\infty), & C_{\boxplus_n}(\Gamma;A,B) = 0, \\ (\mathsf{true}, \operatorname{sgn} C_{\boxplus_n}(\Gamma;A,B), -\log_2 |C_{\boxplus_n}(\Gamma;A,B)|), & C_{\boxplus_n}(\Gamma;A,B) \neq 0. \end{cases}$$

The sign is retained because endpoint linear hull aggregation is signed, even when a single-characteristic MELCC-style weight uses the absolute magnitude.

Linear Q2 for two-variable addition. A typical free-slot query fixes the output mask and one input mask:

$$Q_2^{\mathrm{LC},+}(\Gamma,A) = \arg\min_{B\in\mathbb{F}_2^n} \big(-\log_2 |C_{\boxplus_n}(\Gamma;A,B)|\big),$$

or fixes the two input masks and completes the output mask:

$$Q_2^{\mathrm{LC},+}(A,B) = \arg\min_{\Gamma\in\mathbb{F}_2^n} \big(-\log_2 |C_{\boxplus_n}(\Gamma;A,B)|\big).$$

For subtraction, the mask relabeling is

$$Q_1^{\mathrm{LC},-}(\Gamma_Z;A_X,A_Y) = Q_1^{\mathrm{LC},+}(A_X;\Gamma_Z,A_Y),$$

with the analogous relabeling for Q2.

A.3.2 Fixed-public addend addition and subtraction

For the fixed-public relation

$$Y = X \boxplus_n K,$$

the public constant K is not sampled. It is a parameter of the oracle. The visible boundary is therefore a pair, not a triple. This is the first major modeling distinction: a two-variable Q1 oracle evaluates (α,β,γ) or $(\Gamma;A,B)$, whereas a fixed-public Q1 oracle evaluates (K,u,v) or (K,α,β). The second distinction is that fixed-public weights and signs are constant-dependent. Differential weights follow the Machado/Azimi fixed-addend zero-chain decomposition, and linear weights follow a signed two-state carry-transfer recurrence in the Miyano fixed-addend spirit [Miy98, Mac01, ARS+20, ARS+22, Twi26d].

Differential Q1 for fixed-public addition. For visible differences $(u \to v)$, define

$$Q_1^{\mathrm{DP},+K}(K,u,v) = \begin{cases} (\mathsf{false}, +\infty), & DP_K(u \to v) = 0, \\ (\mathsf{true}, -\log_2 DP_K(u \to v)), & DP_K(u \to v) > 0. \end{cases}$$

Here $DP_K(u \to v)$ is the exact fixed-addend differential probability over X only. Its computation is detailed in Subsection A.5; impossible cases include the forbidden state 001 and trigger events whose zero-chain numerator is zero.

Differential Q2 for fixed-public addition. A fixed-public completion query fixes one side of the pair and optimizes the other:

$$Q_2^{\mathrm{DP},+K}(K,u) = \arg\min_{v \in \mathbb{F}_2^n} W_K^{\mathrm{diff}}(u,v),$$

or

$$Q_2^{\mathrm{DP},+K}(K,v) = \arg\min_{u \in \mathbb{F}_2^n} W_K^{\mathrm{diff}}(u,v).$$

The output is the completing difference word and its optimized local weight. It is a local completion oracle; it does not assert compatibility with the rest of the NeoAlzette schedule.

Linear Q1 for fixed-public addition. For masks (α, β), the exact point query is

$$Q_1^{\mathrm{LC},+K}(K,\alpha,\beta) = \begin{cases} (\mathsf{false}, 0, +\infty), & \mathcal{N}_K(\alpha,\beta) = 0, \\ (\mathsf{true}, \operatorname{sgn}\mathcal{N}_K(\alpha,\beta), n - \log_2 |\mathcal{N}_K(\alpha,\beta)|), & \mathcal{N}_K(\alpha,\beta) \neq 0, \end{cases}$$

where

$$\mathcal{N}_K(\alpha,\beta) = 2^n C_K(\alpha,\beta)$$

is the exact signed numerator computed by the fixed-public two-state carry-transfer recurrence. This is the quantity linearized by the fixed-public linear exact MILP interface in Subsection A.7. It is not the two-variable Wallén automaton with one operand frozen after the fact.

Linear Q2 for fixed-public addition. The corresponding completion queries are

$$Q_2^{\mathrm{LC},+K}(K,\alpha) = \arg\min_{\beta \in \mathbb{F}_2^n} W_K^{\mathrm{lin}}(\alpha,\beta),$$

and

$$Q_2^{\mathrm{LC},+K}(K,\beta) = \arg\min_{\alpha \in \mathbb{F}_2^n} W_K^{\mathrm{lin}}(\alpha,\beta).$$

The completion returns the missing mask, the sign, and the optimized absolute-correlation weight when the selected transition has nonzero signed numerator.

Fixed-public subtraction. For a fixed-right subtraction site $Y = X \boxminus_n C$, the fixed-public reduction is

$$X \boxminus_n C = X \boxplus_n (-C \bmod 2^n).$$

Therefore

$$Q_1^{\mathrm{DP},-K}(C,u,v) = Q_1^{\mathrm{DP},+K}(-C \bmod 2^n, u, v),$$

and

$$Q_1^{\mathrm{LC},-K}(C,\alpha,\beta) = Q_1^{\mathrm{LC},+K}(-C \bmod 2^n, \alpha, \beta),$$

with the same reduction for Q2. Oriented borrow recurrences can be used as an equivalent audit route, but the paper's fixed-right subtraction sites may be priced through the two's-complement addition constant.

Case	Arithmetic relation	Q1: complete evaluation	Q2: free-slot completion
Two-variable differential	$Z = X \boxplus Y$	evaluate $(\alpha, \beta \to \gamma)$ and return support/weight	complete one missing word among α, β, γ
Two-variable linear	$Z = X \boxplus Y$	evaluate $(\Gamma; A, B)$ and return support/sign/weight	complete one missing mask among Γ, A, B
Fixed-public differential	$Y = X \boxplus K$	evaluate $(K, u \to v)$ and return support/weight	complete u or v with K fixed
Fixed-public linear	$Y = X \boxplus K$	evaluate (K, α, β) and return support/sign/weight	complete α or β with K fixed

Reader's interpretation. Q1 and Q2 are modeling levels rather than cryptographic primitives. Q1 is a point query: all local visible masks or differences are fixed, and the oracle returns feasibility plus weight and, for linear analysis, sign. Q2 is a local optimization query: one visible slot is left open and the oracle supplies the best local completion. The distinction matters most for fixed-public arithmetic. The two-variable laws average over two operands, while the fixed-public laws condition on the exact bit pattern of the public constant. Thus Q2 can be used for local ranking and pruning, but it cannot replace the global SCIP model that must satisfy all schedule constraints simultaneously.

A.4 Two-variable Q2 completion oracles: CDDT and CLAT

The two-variable Q2 layer has concrete exact solver instances. In the differential setting, a carry-dependent differential distribution table (CDDT) is an exact local optimizer for the missing output or input difference of the two-variable modular-addition relation. In the linear setting, a carry-dependent linear approximation table (CLAT) is an exact local optimizer for the missing mask side of the two-variable modular-addition relation. These objects are not merely heuristic scouts: for a fixed partial boundary they solve a local completion problem exactly, either by a closed-form bit-sliced formula or by a compressed dynamic-programming/table evaluator. In the 32-bit setting they are not stored as naive $2^{32} \times 2^{32}$ tables; they are algorithmic evaluators whose running time is linear or nearly linear in the word size. The CDDT/CLAT terminology is consistent with the carry-dependent ARX trail-search viewpoint used in automatic ARX analysis [LM02, SG13, LLJW21, FWG+16].

A.4.1 Differential Q2: CDDT-style completion

For two-variable addition $Z = X \boxplus_n Y$, the differential Q2 service used in the Forest scout stage fixes the two input differences (α, β) and completes the output difference:

$$\gamma^*(\alpha, \beta) \in \arg\min_{\gamma \in \mathbb{F}_2^n} W_+^{\mathrm{diff}}(\alpha, \beta, \gamma).$$

The returned local object is (γ^*, W^*) with

$$W^* = W_+^{\mathrm{diff}}(\alpha, \beta, \gamma^*).$$

This is a Q2 completion oracle, not a Q1 point-probability oracle. The Q2 layer constructs a maximizer of the two-variable differential probability, and only afterwards calls the Q1 oracle to obtain the bill paid by the completed triple. This separation is the same distinction used in the main model: Q1 evaluates a fully fixed boundary, while Q2 completes a free slot. The construction below is the Lipmaa–Moriai optimal-output-difference completion for ordinary two-variable modular addition [LM02].

Non-table nature of the CDDT completion. The name CDDT is used here in the carry-dependent ARX-search sense, not in the sense of a naively materialized $2^n \times 2^n$ table. For $n = 32$, no feasible implementation stores a full table indexed by (α, β). Instead, the completion is a closed-form bit-sliced evaluator. It uses only Boolean word operations, shifts, truncation masks, and an odd-run-parity kernel. Thus the local CDDT service is an exact algorithmic Q2 optimizer: for the fixed partial boundary (α, β) it returns a word γ^* whose Q1 weight is locally minimal.

Low bit and active-run structure. Let $\mathcal{M}_n = 2^n - 1$ and truncate all complemented and shifted words by intersecting with $\mathcal{M}_n$. The least significant bit is forced because the incoming carry difference at bit zero is zero:

$$\gamma_0^* = \alpha_0 \oplus \beta_0.$$

Put

$$r = \alpha \wedge 1, \qquad e = \overline{\alpha \oplus \beta} \wedge \overline{r}.$$

The active-start mask is

$$a_0 = e \wedge (e \ll 1) \wedge \big(\alpha \oplus (\alpha \ll 1)\big).$$

The mask a_0 marks the starts of the runs where the optimal output word differs from the naive affine relation. The required global operation is the run-parity map

$$p = \mathrm{AOPR}(a_0),$$

where AOPR marks every second bit in each maximal one-run, starting from the low end of that run. Formally, if $[l, r]$ is a maximal one-run in a word x, then

$$\mathrm{AOPR}(x)_i = 1 \iff l \le i \le r \text{ and } i - l + 1 \text{ is odd.}$$

Define

$$a_1 = (a_0 \vee (a_0 \gg 1)) \wedge \overline{r}, \qquad b = (a_1 \vee e) \ll 1.$$

Then the closed-form candidate is

$$\begin{aligned}\widetilde{\gamma} = &\big((\alpha \oplus p) \wedge a_1\big) \vee \big((\alpha \oplus \beta \oplus (\alpha \ll 1)) \wedge \overline{a_1} \wedge b\big) \\ &\vee \big(\alpha \wedge \overline{a_1} \wedge \overline{b}\big),\end{aligned}$$

and the final completion is

$$\gamma^* = (\widetilde{\gamma} \wedge \overline{1}) \vee \big((\alpha \oplus \beta) \wedge 1\big).$$

The local weight is then obtained by the two-variable differential Q1 evaluator:

$$W^* = W_+^{\mathrm{diff}}(\alpha, \beta, \gamma^*).$$

Log-depth run-parity kernel. A didactic definition of AOPR may scan the bits from low to high, toggling a one-bit parity inside each one-run. That presentation is useful for proof, but it is not the reason the construction is efficient. The word-level implementation uses parallel prefix/doubling operations. A related odd-run-start map can be built in $\Theta(\log n)$ word-operation stages by setting $x^{(0)} = x$ and

$$x^{(1)} = x \wedge (x \gg 1), \qquad x^{(j)} = x^{(j-1)} \wedge \big(x^{(j-1)} \gg 2^{j-1}\big),$$

with the associated output words initialized and updated by

$$y^{(0)} = 0, \qquad y^{(1)} = x \wedge \overline{x^{(1)}},$$

$$y^{(j)} = y^{(j-1)} \vee \big((y^{(j-1)} \gg 2^{j-1}) \wedge x^{(j-1)}\big).$$

The returned word after $\lceil \log_2 n \rceil$ stages identifies odd one-run information. The AOPR parity used in the optimal-γ formula is the low-end run-parity variant of the same run-structure computation. Consequently, the expensive global step in the Q2 completion is not a table lookup and not an exponential search; it is a word-parallel run-parity computation with logarithmic word-operation depth. For the fixed 32-bit instance used by the NeoAlzette analysis, this becomes a small constant number of machine-word operations.

Algorithm 18 Aopr: low-end odd-run parity mask

Require: word x and word size N
Ensure: $p = \mathrm{AOPR}(x)$
$x \leftarrow x \wedge (2^N - 1)$
$p \leftarrow 0$; $\rho \leftarrow 0$
for $i = 0$ to $N - 1$ **do**
  if $\mathrm{ReadBit}(x, i) = 1$ **then**
    $\rho \leftarrow \rho \oplus 1$
  else
    $\rho \leftarrow 0$
  end if
  if $\rho = 1$ **then**
    $p \leftarrow p \vee (1 \ll i)$
  end if
end for
return $p \wedge (2^N - 1)$

Algorithm 19 Add2DiffOptimalGamma: non-table two-variable differential Q2 completion

Require: input differences α, β and word size N
Ensure: one optimal output difference γ^*
$M \leftarrow 2^N - 1$
$\alpha \leftarrow \alpha \wedge M$; $\beta \leftarrow \beta \wedge M$
$r \leftarrow \alpha \wedge 1$
$e \leftarrow (M \oplus (\alpha \oplus \beta)) \wedge (M \oplus r)$
$a \leftarrow e \wedge ((e \ll 1) \wedge M) \wedge (\alpha \oplus ((\alpha \ll 1) \wedge M))$
$p \leftarrow \textsc{Aopr}(a, N)$
$a \leftarrow (a \vee (a \gg 1)) \wedge (M \oplus r)$
$b \leftarrow ((a \vee e) \ll 1) \wedge M$
$\gamma \leftarrow (\alpha \oplus p) \wedge a$
$\gamma \leftarrow \gamma \vee \big((\alpha \oplus \beta \oplus ((\alpha \ll 1) \wedge M)) \wedge (M \oplus a) \wedge b\big)$
$\gamma \leftarrow \gamma \vee \big(\alpha \wedge (M \oplus a) \wedge (M \oplus b)\big)$
$\gamma \leftarrow \gamma \wedge M$
$\gamma \leftarrow (\gamma \wedge (M \oplus 1)) \vee ((\alpha \oplus \beta) \wedge 1)$
return $\gamma \wedge M$

Algorithm 20 DiffTwoVarOptimalGamma: CDDT-style Q2 completion with Q1 bill

Require: input differences α, β and word size N
Ensure: optimal output difference γ^* and local weight W^*
$\gamma^* \leftarrow \textsc{Add2DiffOptimalGamma}(\alpha, \beta, N)$
$(\mathsf{possible}, W^*) \leftarrow Q_1^{\mathrm{DP},+}(\alpha, \beta, \gamma^*)$
return (γ^*, W^*)

The subtraction versions of this Q2 completion are obtained by the same addition/subtraction permutation used for Q1. For example, completing the missing word of $Z = X \boxminus Y$ is reduced to completing the corresponding word in $X = Z \boxplus Y$ and then relabeling the result back to the subtraction boundary. This CDDT service is therefore an exact local Q2 optimizer for two-variable add/sub boundaries. In a Forest campaign it is a scout and ordering tool; it never replaces the global model, because rotations, fixed-public constants, packed C/D injection, endpoint restrictions, and no-good cuts may invalidate the locally best completion.

A.4.2 Linear Q2: CLAT-style row and column completion

For the two-variable linear addition oracle, Q2 has two dual completion modes. A row completion fixes the output mask u and completes the input masks (v, w). A column completion fixes the input masks (v, w) and completes the output mask u. Both are exact local optimizers derived from the Schulte–Geers CCZ formulation of modular-addition correlations [SG13, Wal03a, LLJW21].

Row completion. Given an output mask u, the row-completion problem is

$$(v^*, w^*) \in \arg\min_{v,w} W_+^{\mathrm{lin}}(u; v, w).$$

Define $a_L(u)$ as the run-parity mask of u: inside every maximal run of ones of u, it keeps every second bit according to the shift convention of Part II. Equivalently,

$$a_L(u) = u \oplus (u \wedge L(u)) \oplus (u \wedge L(u) \wedge L^2(u)) \oplus \cdots,$$

where the product stops once the shift leaves the word. The optimal row completion is

$$v^* = a_L(u) \oplus L(a_L(u)), \qquad w^* = u,$$

with support mask $z^* = L(a_L(u))$ and local weight

$$W^* = \mathrm{wt}(z^*).$$

This construction selects a positive-sign representative of the optimal row.

Algorithm 21 LinearTwoVarRowCompletion: CLAT row Q2

Require: output mask u and word size n
Ensure: optimal input masks (v^*, w^*) and local weight W^*

$u \leftarrow u \bmod 2^n$; $a \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
  if ReadBit$(u, i) = 0$ **then**
    continue
  end if
  $j \leftarrow i$
  while $j < n$ and ReadBit$(u, j) = 1$ **do**
    $j \leftarrow j + 1$
  end while
  if $(j - i) \bmod 2 = 1$ **then**
    $a \leftarrow a \vee (1 \ll i)$
  end if
end for
$z \leftarrow L(a)$ ▷ use the Part II shift convention
$v^* \leftarrow a \oplus z$; $w^* \leftarrow u$
$W^* \leftarrow \mathrm{wt}(z)$
return (v^*, w^*, W^*)

Column completion. Given input masks (v, w), the column-completion problem is

$$u^* \in \arg\min_u W_+^{\mathrm{lin}}(u; v, w).$$

Let

$$o = v \vee w, \qquad h = v \wedge w, \qquad z = \overline{o}.$$

If $o = 0$, the zero mask is the trivial completion. Otherwise let $p = \mathrm{msb}(o)$. If the highest active position of o is not shared by both input masks, i.e. $h_p = 0$, the column is zero and the completion is reported with a sentinel weight. In the reduced width $N = p + 1$, define the rule-2 start mask

$$S = (o \wedge \overline{o \gg 1} \wedge \overline{h}) \wedge \overline{2^p}.$$

Let R denote the shift toward the next higher bit. The seed for the zero-run closure is

$$a_{\mathrm{seed}} = R(v \oplus w) \vee (S \ll 1).$$

Let $\mathrm{Close}_z(a)$ be the least mask containing a and closed under propagation through zero positions:

$$z_i = 1,\ a_i = 1 \quad \Longrightarrow \quad a_{i+1} = 1.$$

Then

$$m(v, w) = \mathrm{Close}_z(a_{\mathrm{seed}}), \qquad a^* = a_L(h) \vee m(v, w),$$

and the optimal output mask is

$$u^* = (v \oplus w) \oplus a^* \oplus L(a^*), \qquad W^* = \mathrm{wt}(L(a^*)).$$

Algorithm 22 LinearTwoVarColumnCompletion: CLAT column Q2

Require: input masks v, w and word size n
Ensure: optimal output mask u^* and local weight W^*

$v \leftarrow v \bmod 2^n$; $w \leftarrow w \bmod 2^n$; $o \leftarrow v \vee w$
if $o = 0$ **then**
  return $(0, 0)$
end if
$p \leftarrow \mathrm{msb}(o)$
if $\mathrm{ReadBit}(v, p) = 0$ or $\mathrm{ReadBit}(w, p) = 0$ **then**
  return $(0, n)$ ▷ zero column / sentinel weight
end if
$N \leftarrow p + 1$; reduce v, w, o to N bits
$h \leftarrow v \wedge w$; $z \leftarrow \overline{o} \wedge (2^N - 1)$
$S \leftarrow o \wedge \overline{o \gg 1} \wedge \overline{h}$; clear bit p of S
$a \leftarrow ((v \oplus w) \ll 1) \vee ((S \ll 1) \bmod 2^N)$
$r \leftarrow z$; $d \leftarrow 1$
while $d < N$ **do**
  $a \leftarrow a \vee (((a \wedge r) \ll d) \bmod 2^N)$
  $r \leftarrow r \wedge (r \gg d)$
  $d \leftarrow 2d$
end while
$a_L \leftarrow \mathrm{RunParityMask}(h, N)$
$a^* \leftarrow a_L \vee a$
$u^* \leftarrow (v \oplus w) \oplus a^* \oplus L(a^*)$
$W^* \leftarrow \mathrm{wt}(L(a^*))$
return $(u^* \bmod 2^n, W^*)$

These CDDT/CLAT Q2 completions are exact local optimizers. In the Forest campaign they are used as scouts: they can certify that a local partial boundary cannot beat the current global incumbent, provide a good completion order, or seed a candidate branch. They still do not replace the global SCIP/MILP model, because a locally optimal completion can conflict with rotations, injection constraints, fixed-public arithmetic sites, or endpoint restrictions elsewhere in the schedule.

A.5 Fixed-public add/sub differential Q1 oracle

This subsection states the fixed-public differential point oracle used at the public-constant add/sub sites of NeoAlzette. This oracle is a first-class object in the paper, not an optional shortcut. In the two-variable modular-addition experiment, the probability is averaged over both operands (X, Y). In the fixed-public experiment, the addend K is a deterministic public bit-string and the probability space contains only the variable word X. Consequently, the carry/borrow chains are controlled by the specific bit pattern of K, and the transition probability is a function of K. This is exactly why the fixed-public relation cannot be replaced by the averaged Lipmaa–Moriai two-variable add-box. The addend-dependency viewpoint goes back to Miyano, and exact constant-operand differential recurrences are treated by Machado; later bit-vector models such as Azimi et al. further motivate a dedicated fixed-addend modeling boundary [Miy98, Mac01, ARS+20, ARS+22].

A.5.1 Problem definition

Let n be the word size. In the implemented NeoAlzette fixed-public subtraction sites, $n = 32$, but the oracle is stated for general n. For a fixed public constant $K \in \mathbb{F}_2^n$, define

$$F_{+K} : \mathbb{F}_2^n \to \mathbb{F}_2^n, \qquad F_{+K}(X) = X \boxplus_n K.$$

For input XOR difference u on X and output XOR difference v on $F_{+K}(X)$, the exact differential probability is

$$\mathrm{DP}_K(u \to v) = 2^{-n} \#\{X \in \mathbb{F}_2^n : F_{+K}(X) \oplus F_{+K}(X \oplus u) = v\}.$$

The fixed-addend differential point oracle is

$$Q_1^{\mathrm{DP},+K}(K, u, v) = \begin{cases} (\mathsf{false}, +\infty), & \mathrm{DP}_K(u \to v) = 0, \\ (\mathsf{true}, -\log_2 \mathrm{DP}_K(u \to v)), & \mathrm{DP}_K(u \to v) > 0. \end{cases}$$

The subtraction version $Q_1^{\mathrm{DP},-K}$ is defined analogously for $F_{-C}(X) = X - C \pmod{2^n}$, using the borrow recurrence or, equivalently for fixed-right subtraction, the two's-complement addition wrapper.

A.5.2 Carry-difference identity

Let c_i and c'_i be the carry bits entering bit position i in the two additions $X \boxplus_n K$ and $(X \oplus u) \boxplus_n K$, respectively. Let $c_0 = c'_0 = 0$, and set $d_i = c_i \oplus c'_i$. The output bits are

$$Y_i = X_i \oplus K_i \oplus c_i, \qquad Y'_i = (X_i \oplus u_i) \oplus K_i \oplus c'_i.$$

XORing the two output equations gives

$$v_i = Y_i \oplus Y'_i = u_i \oplus c_i \oplus c'_i = u_i \oplus d_i.$$

Hence the visible transition forces

$$d_i = u_i \oplus v_i \qquad (0 \le i < n).$$

At the low boundary $d_0 = 0$, because $c_0 = c'_0 = 0$. Therefore

$$u_0 = v_0$$

is a necessary condition for a nonzero fixed-public differential transition. The later oracle formula is simply a way of counting how many values of X realize the forced carry-difference word $d = u \oplus v$.

A.5.3 Machado conditional recurrence

Let the virtual boundary bits be $u_{-1} = v_{-1} = K_{-1} = 0$, and define

$$e_i = u_i \oplus v_i, \qquad S_i = (u_{i-1}, v_{i-1}, e_i) \in \{0,1\}^3.$$

We also use the integer index

$$S_i = 4u_{i-1} + 2v_{i-1} + e_i \in \{0, \ldots, 7\}.$$

The state $S_i = 001$ is forbidden. Equivalently, a prefix with no lower carry-difference support cannot create the demanded next carry-difference bit under the fixed-public carry logic.

For valid prefixes, let δ_i be the conditional probability that the unprimed carry into position i is one, conditioned on all lower output-difference conditions being satisfied. Let ϕ_i be the conditional probability that the bit-i condition is satisfied, again conditioned on a valid lower prefix. Then

$$\mathrm{DP}_K(u \to v) = \prod_{i=0}^{n-1} \phi_i.$$

The recurrence is table-driven by S_i and K_{i-1}:

$$\delta_i = \begin{cases} (K_{i-1} + \delta_{i-1})/2, & S_i = 000, \\ 0, & S_i = 001, \\ K_{i-1}, & S_i \in \{010, 100, 110\}, \\ \delta_{i-1}, & S_i \in \{011, 101\}, \\ 1/2, & S_i = 111, \end{cases}$$

with $\delta_0 = 0$, and

$$\phi_i = \begin{cases} 1, & S_i = 000, \\ 0, & S_i = 001, \\ 1/2, & S_i \in \{010, 011, 100, 101\}, \\ 1 - (K_{i-1} + \delta_{i-1} - 2K_{i-1}\delta_{i-1}), & S_i = 110, \\ K_{i-1} + \delta_{i-1} - 2K_{i-1}\delta_{i-1}, & S_i = 111. \end{cases}$$

This recurrence is the conditional-probability form of the fixed-addend differential oracle. It is already enough to compute the exact transition probability, but the zero-chain form below is more convenient for exact weights and MILP interfaces.

A.5.4 Zero-chain closed form

Define the base contribution of the states with constant factor 1/2:

$$W_{\mathrm{base}}(u, v) = \sum_{i=1}^{n-1} \mathbf{1}[u_{i-1} \oplus v_{i-1} = 1].$$

The remaining nontrivial local states are the trigger states $S_i \in \{110, 111\}$, where the local factor depends on a backward run of lower 000 states and on the corresponding public constant bits.

For a trigger at position i, define its exact backward zero-chain length by

$$\lambda_i = 1 + \max\{L \geq 0 : S_{i-1} = S_{i-2} = \cdots = S_{i-L} = 000\},$$

with the chain stopping at the low boundary. Let

$$q_i = e_i \oplus K_{i-1}.$$

The constant block value is

$$B_{i,\lambda} = K_{i-\lambda-1} + \sum_{j=i-\lambda}^{i-2} 2^{j-(i-\lambda)} K_j,$$

where $K_j = 0$ for $j < 0$. It is the integer block of public constant bits immediately below the trigger, with the boundary bit included according to the fixed-addend recurrence convention. The trigger numerator is

$$p_{i,\lambda,q} = \begin{cases} B_{i,\lambda}, & q = 1, \\ 2^{\lambda-1} - B_{i,\lambda}, & q = 0. \end{cases}$$

If $p_{i,\lambda_i,q_i} = 0$, then the transition is impossible. Otherwise the trigger contributes

$$W_{\text{trig}}(i) = (\lambda_i - 1) - \log_2 p_{i,\lambda_i,q_i}.$$

Thus the exact total differential weight for the fixed-addend map is

$$\boxed{W_K^{\Delta}(u, v) = W_{\text{base}}(u, v) + \sum_{\substack{1 \leq i < n \\ S_i \in \{110,111\}}} \big((\lambda_i - 1) - \log_2 p_{i,\lambda_i,q_i}\big).}$$

If any trigger has zero numerator, we set $W_K^{\Delta}(u, v) = +\infty$.

A.5.5 Exact count equivalence

Let T be the set of trigger positions. The exact number of valid words X can be recovered from the closed form:

$$\#\{X : F_K(X) \oplus F_K(X \oplus u) = v\} = 2^{n - W_{\text{base}} - \sum_{i \in T}(\lambda_i - 1)} \prod_{i \in T} p_{i,\lambda_i,q_i}.$$

Consequently

$$\text{DP}_K(u \to v) = 2^{-W_{\text{base}}} \prod_{i \in T} \frac{p_{i,\lambda_i,q_i}}{2^{\lambda_i - 1}}.$$

The same oracle can therefore be evaluated either by summing the real-valued zero-chain weights or by computing the exact integer count and taking $n - \log_2 N$ at the end. The integer route is an audit route for the zero-chain route.

Algorithm 23 StateIndex

Require: differences u, v, bit index i, width n
Ensure: state index $S_i \in \{0, \ldots, 7\}$
1: $a \leftarrow 0$ if $i = 0$, otherwise $a \leftarrow \text{ReadBit}(u, i - 1)$
2: $b \leftarrow 0$ if $i = 0$, otherwise $b \leftarrow \text{ReadBit}(v, i - 1)$
3: $e \leftarrow \text{ReadBit}(u, i) \oplus \text{ReadBit}(v, i)$
4: **return** $4a + 2b + e$

Algorithm 24 ConstantBlockValue

Require: constant K, trigger position i, chain length λ
Ensure: integer $B_{i,\lambda}$

$B \leftarrow 0$
for $r = 0$ to $\lambda - 2$ **do**
    $B \leftarrow B + 2^r \,\mathrm{ReadBit}(K, i - \lambda + r)$
end for
$B \leftarrow B + \mathrm{ReadBit}(K, i - \lambda - 1)$
return B

Algorithm 25 FixedAddendDiffOracle: zero-chain evaluator

Require: public constant K, input difference u, output difference v, width n
Ensure: $(\mathsf{possible}, W)$, where $W = -\log_2 DP$ or $+\infty$

$W \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    $S \leftarrow \textsc{StateIndex}(u, v, i, n)$
    if $S = 1$ **then**
        return $(\mathsf{false}, +\infty)$
    end if
    if $i \geq 1$ and $\lfloor S/2 \rfloor \in \{1, 2\}$ **then**
        $W \leftarrow W + 1$
    end if
    if $i \geq 1$ and $\lfloor S/2 \rfloor = 3$ **then**
        $\lambda \leftarrow 1$; $j \leftarrow i - 1$
        while $j \geq 0$ and $\textsc{StateIndex}(u, v, j, n) = 0$ **do**
            $\lambda \leftarrow \lambda + 1$; $j \leftarrow j - 1$
        end while
        $q \leftarrow \mathrm{ReadBit}(u, i) \oplus \mathrm{ReadBit}(v, i) \oplus \mathrm{ReadBit}(K, i - 1)$
        $B \leftarrow \textsc{ConstantBlockValue}(K, i, \lambda)$
        if $q = 1$ **then**
            $p \leftarrow B$
        else
            $p \leftarrow 2^{\lambda - 1} - B$
        end if
        if $p = 0$ **then**
            return $(\mathsf{false}, +\infty)$
        end if
        $W \leftarrow W + (\lambda - 1) - \log_2 p$
    end if
end for
return (true, W)

Algorithm 26 FixedAddendExactCount: integer audit evaluator

Require: public constant K, input difference u, output difference v, width n
Ensure: exact integer count $N = \#\{X : F_K(X) \oplus F_K(X \oplus u) = v\}$

$d \leftarrow u \oplus v$
if $\mathrm{ReadBit}(d, 0) = 1$ **then**
    return 0
end if
$(A, B) \leftarrow (1, 0)$
$i \leftarrow 0$
while $i + 1 < n$ **do**

```
    π ← (ReadBit(d, i), ReadBit(d, i + 1))
    if π = (1, 1) then
        ℓ ← 1
        while i + ℓ < n and ReadBit(d, i + ℓ) = 1 do
            ℓ ← ℓ + 1
        end while
        i ← i + ℓ − 1
    else if π = (0, 0) then
        r ← i
        while r < n and ReadBit(d, r) = 0 do
            r ← r + 1
        end while
        z_end ← r − 1
        while i < z_end do
            k ← ReadBit(K, i); ℓ ← 1
            while i + ℓ < z_end and ReadBit(K, i + ℓ) = k do
                ℓ ← ℓ + 1
            end while
            U ← Σ_{t=0}^{ℓ−1} ReadBit(u, i + t)2^t; M ← 2^ℓ
            if U = 0 then
                if k = 0 then
                    (A, B) ← (MA + (M − 1)B, B)
                else
                    (A, B) ← (A, (M − 1)A + MB)
                end if
            else
                r_0 ← min{t : ReadBit(U, t) = 1}; ρ ← M − 2^{ℓ−r_0}
                if k = 0 then
                    (A, B) ← (MA + ρB, 0)
                else
                    (A, B) ← (0, ρA + MB)
                end if
            end if
            if A = 0 and B = 0 then
                return 0
            end if
            i ← i + ℓ
        end while
    else if π = (1, 0) then
        if ReadBit(K, i) = 0 then
            (A, B) ← (A + B, 0)
        else
            (A, B) ← (0, A + B)
        end if
        i ← i + 1
    else
        if ReadBit(u, i) = 0 then
            return 0
        end if
        if ReadBit(K, i) = 0 then
            (A, B) ← (B, B)
        else
```

```
            (A, B) ← (A, A)
        end if
        i ← i + 1
    end if
end while
return 2(A + B)
```

The count evaluator returns the same oracle in exact integer form. If

$$N = \textsc{FixedAddendExactCount}(K, u, v, n),$$

then, for $N > 0$,

$$\mathrm{DP}_K(u \to v) = N/2^n, \qquad W_K(u \to v) = n - \log_2 N.$$

A.5.6 Fixed-public subtraction

For $F_{-C}(X) = X - C \pmod{2^n}$, the same point-evaluation framework applies with oriented borrow states. The visible identity remains

$$v_i = u_i \oplus b_i \oplus b'_i,$$

but the next-state update is the subtractor borrow recurrence. Equivalently, fixed-right subtraction may be evaluated through the addition wrapper

$$X - C \pmod{2^n} = X + (-C \bmod 2^n) \pmod{2^n}.$$

Thus the MILP layer treats a fixed-public subtraction as a fixed-public relation with its own exact oracle verdict and exact local weight; it is not replaced by an averaged two-variable subtraction box.

A.6 Fixed-public add/sub linear Q1 oracle

The linear analysis of a modular addition or subtraction with one variable operand and one fixed public operand is fundamentally different from the two-variable linear-addition case of Wallen and Schulte-Geers. In the two-variable case, the Walsh sum is taken over both operands (X, Y). In the fixed-public case, the addend K is a deterministic bit-string, the sum is taken over the single variable word X, and the carry or borrow chain is driven by the actual bit pattern of K. Consequently, the correlation is not a universal function of the masks alone. It is a function of the masks and of the fixed constant K, so the local relation needs its own signed two-state carry-transfer oracle [Miy98, Wal03a, SG13, Twi26d].

A.6.1 Problem definition and signed correlation

Let n be the word size. In the implemented NeoAlzette fixed-public arithmetic sites, $n = 32$, but the oracle is stated for general n. For a fixed public constant $K \in \mathbb{F}_2^n$, define

$$F_{+K} : \mathbb{F}_2^n \to \mathbb{F}_2^n, \qquad F_{+K}(X) = X \boxplus_n K.$$

Let $\alpha, \beta \in \mathbb{F}_2^n$ be the input mask on X and the output mask on $Y = F_{+K}(X)$. The exact signed Walsh correlation is

$$C_K(\alpha, \beta) = 2^{-n} \sum_{X \in \mathbb{F}_2^n} (-1)^{\alpha \cdot X \oplus \beta \cdot F_{+K}(X)}.$$

The fixed-addend linear point oracle is

$$Q_1^{\mathrm{LC},+K}(K,\alpha,\beta)=\begin{cases}(\mathsf{false},0,+\infty), & C_K(\alpha,\beta)=0,\\ (\mathsf{true},\operatorname{sgn}(C_K(\alpha,\beta)),-\log_2|C_K(\alpha,\beta)|), & C_K(\alpha,\beta)\neq 0.\end{cases}$$

The subtraction oracle $Q_1^{\mathrm{LC},-K}$ is defined analogously for $F_{-C}(X)=X-C \pmod{2^n}$, either with the oriented borrow recurrence or, for fixed-right subtraction, through the two's-complement addition wrapper $X-C\equiv X+(-C \bmod 2^n) \pmod{2^n}$.

A.6.2 Carry-state derivation and phase separation

Let c_i be the carry entering bit position i in $X\boxplus_n K$. Thus $c_0=0$, and

$$Y_i=X_i\oplus K_i\oplus c_i,\qquad c_{i+1}=\mathrm{Maj}(X_i,K_i,c_i),$$

where

$$\mathrm{Maj}(a,b,c)=(a\wedge b)\vee(a\wedge c)\vee(b\wedge c).$$

Substituting Y_i into the Walsh phase gives

$$\begin{aligned}\alpha\cdot X\oplus\beta\cdot Y&=\bigoplus_{i=0}^{n-1}\alpha_iX_i\oplus\bigoplus_{i=0}^{n-1}\beta_i(X_i\oplus K_i\oplus c_i)\\&=\bigoplus_{i=0}^{n-1}(\alpha_i\oplus\beta_i)X_i\oplus\bigoplus_{i=0}^{n-1}\beta_iK_i\oplus\bigoplus_{i=0}^{n-1}\beta_ic_i.\end{aligned}$$

Define the local parameters

$$t_i=\alpha_i\oplus\beta_i,\qquad b_i=\beta_i.$$

For an incoming carry state $s=c_i\in\{0,1\}$ and outgoing carry state $r=c_{i+1}\in\{0,1\}$, the normalized one-bit transfer is

$$M_i(s,r)=\frac{1}{2}\sum_{\substack{x\in\{0,1\}\\ \mathrm{Maj}(x,K_i,s)=r}}(-1)^{t_ix\oplus b_iK_i\oplus b_is}.$$

Let $R_i=(R_i^{(0)},R_i^{(1)})$ be the row vector of signed prefix averages conditioned on the carry entering bit i. Then

$$R_0=(1,0),\qquad R_{i+1}=R_iM_i,\qquad C_K(\alpha,\beta)=R_n^{(0)}+R_n^{(1)}.$$

This is the signed carry-state analogue of the fixed-public differential carry-pair recurrence: the hidden state is the carry, but the accumulated value is a signed Walsh numerator rather than a nonnegative count.

A.6.3 Integer-scaled transfer matrix

For exact integer arithmetic, multiply every one-bit transfer by 2. Put

$$S_i=2^iR_i=(P_i,Q_i),\qquad S_0=(1,0),\qquad S_{i+1}=S_iA_i,\qquad A_i=2M_i.$$

The final signed numerator is

$$\mathcal{N}_K(\alpha,\beta)=P_n+Q_n=2^nC_K(\alpha,\beta),$$

so

$$C_K(\alpha,\beta) = 2^{-n}\mathcal{N}_K(\alpha,\beta).$$

The numerator is an exact signed integer. If $\mathcal{N}_K(\alpha,\beta) = 0$, the mask pair is outside the live support of the fixed-public linear relation. Otherwise the absolute-correlation weight is

$$W_K^L(\alpha,\beta) = n - \log_2 |\mathcal{N}_K(\alpha,\beta)|.$$

This real-valued weight is not generally an integer because $|\mathcal{N}_K|$ need not be a power of two.

A.6.4 Explicit integer transfer table

Write the integer transfer matrix as

$$A_i = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}, \qquad (P_{i+1}, Q_{i+1}) = (P_i, Q_i)A_i.$$

The table below is derived by enumerating the two possible values of X_i under the full-adder carry constraint and by attaching the signed phase $(-1)^{t_i X_i \oplus b_i K_i \oplus b_i c_i}$.

Case $K_i = 0$.

(t_i, b_i)	P_{i+1}	Q_{i+1}	A_i
$(0,0)$	$2P_i + Q_i$	Q_i	$\begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix}$
$(0,1)$	$2P_i - Q_i$	$-Q_i$	$\begin{pmatrix} 2 & 0 \\ -1 & -1 \end{pmatrix}$
$(1,0)$	Q_i	$-Q_i$	$\begin{pmatrix} 0 & 0 \\ 1 & -1 \end{pmatrix}$
$(1,1)$	$-Q_i$	Q_i	$\begin{pmatrix} 0 & 0 \\ -1 & 1 \end{pmatrix}$

Case $K_i = 1$.

(t_i, b_i)	P_{i+1}	Q_{i+1}	A_i
$(0,0)$	P_i	$P_i + 2Q_i$	$\begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix}$
$(0,1)$	$-P_i$	$-P_i + 2Q_i$	$\begin{pmatrix} -1 & -1 \\ 0 & 2 \end{pmatrix}$
$(1,0)$	P_i	$-P_i$	$\begin{pmatrix} 1 & -1 \\ 0 & 0 \end{pmatrix}$
$(1,1)$	$-P_i$	P_i	$\begin{pmatrix} -1 & 1 \\ 0 & 0 \end{pmatrix}$

The matrix entries are written in row-vector convention. In particular, each displayed matrix matches the displayed update formula $(P_i, Q_i)A_i$. This convention prevents accidental transposition of the transfer table.

A.6.5 Triangular necessary condition

The fixed-public map $X \mapsto X \boxplus_n K$ is triangular in the bit order: Y_j depends only on $X_0, \ldots, X_j$ and contains X_j with coefficient one. Therefore a nonzero correlation requires the highest active input-mask bit and highest active output-mask bit to coincide. Let

$$h(m) = \max\{i : m_i = 1\}, \qquad h(0) = -1.$$

Then

$$C_K(\alpha,\beta)\neq 0 \quad\Longrightarrow\quad h(\alpha)=h(\beta).$$

This condition is necessary, not sufficient. It is an exact early rejection test for the point oracle: if the highest active bit positions differ, the oracle returns the zero-correlation result without running the transfer recurrence.

A.6.6 Run-event exact numerator

The sequential recurrence $S_{i+1}=S_iA_i$ is exact and already linear-time in the word size. The implementation-oriented oracle also admits an exact run-event evaluator by grouping maximal runs of equal public constant bits. This is an optimization of the same signed numerator, not a different correlation definition.

Factor the public constant phase out of the sum:

$$(-1)^{\beta\cdot K}.$$

Let $t=\alpha\oplus\beta$. With the constant phase removed, the base kernels for $b_i=0$ are

$$B_{0,0}=\begin{pmatrix}2&0\\1&1\end{pmatrix},\quad B_{0,1}=\begin{pmatrix}0&0\\1&-1\end{pmatrix},\quad B_{1,0}=\begin{pmatrix}1&1\\0&2\end{pmatrix},\quad B_{1,1}=\begin{pmatrix}1&-1\\0&0\end{pmatrix}.$$

The subscript (r,τ) means $K_i=r$ and $t_i=\tau$. Inside a maximal run of constant K-bit r, the base matrices for $t_i=0$ and $t_i=1$ can be grouped by closed-form powers. For a row vector $U=(u_0,u_1)$, the useful closed forms are

$$UB_{0,0}^k=((u_0+u_1)2^k-u_1,\ u_1),$$

$$UB_{0,1}^k=\begin{cases}(u_1,-u_1), & k \text{ odd},\\(-u_1,u_1), & k \text{ even},\end{cases}$$

$$UB_{1,0}^k=(u_0,\ (u_0+u_1)2^k-u_0),$$

$$UB_{1,1}^k=(u_0,-u_0)\qquad(k\geq 1).$$

When $\beta_i=1$, the local constant/carry phase is represented by a sign-flip matrix

$$D=\operatorname{diag}(1,-1)$$

inserted before the corresponding base step. The run-event evaluator scans maximal runs of equal K-bits, applies closed-form powers on subruns with $\beta_i=0$, handles $\beta_i=1$ positions by inserting D, and finally multiplies by the public sign factor $(-1)^{\beta\cdot K}$. For the full exact evaluator the denominator exponent is n, and the final numerator is $\mathcal{N}=P+Q$.

Algorithm 27 FixedAddendLinearNumerator: sequential exact signed transfer

Require: public constant K, input mask α, output mask β, width n
Ensure: signed numerator $\mathcal{N}$, correlation C, and weight W, or zero-correlation result

truncate K,α,β to n bits
if $h(\alpha)\neq h(\beta)$ **then**
    return $(0,0,+\infty)$
end if
$(P,Q)\leftarrow(1,0)$
for $i=0$ to $n-1$ **do**
    $k\leftarrow\mathrm{ReadBit}(K,i)$, $t\leftarrow\mathrm{ReadBit}(\alpha,i)\oplus\mathrm{ReadBit}(\beta,i)$, $b\leftarrow\mathrm{ReadBit}(\beta,i)$
    if $k=0$ **then**
        if $(t,b)=(0,0)$ **then**

```
            (P, Q) ← (2P + Q, Q)
        else if (t, b) = (0, 1) then
            (P, Q) ← (2P − Q, −Q)
        else if (t, b) = (1, 0) then
            (P, Q) ← (Q, −Q)
        else
            (P, Q) ← (−Q, Q)
        end if
    else
        if (t, b) = (0, 0) then
            (P, Q) ← (P, P + 2Q)
        else if (t, b) = (0, 1) then
            (P, Q) ← (−P, −P + 2Q)
        else if (t, b) = (1, 0) then
            (P, Q) ← (P, −P)
        else
            (P, Q) ← (−P, P)
        end if
    end if
end for
N ← P + Q
if N = 0 then
    return (0, 0, +∞)
end if
C ← N/2^n; W ← n − log_2 |N|
return (N, C, W)
```

Algorithm 28 FixedAddendRunEventNumerator: grouped exact signed transfer

Require: public constant K, input mask α, output mask β, width n
Ensure: signed numerator $\mathcal{N}$ and denominator exponent σ

```
truncate K, α, β to n bits
publicSign ← 1 if β · K = 0, otherwise −1
t ← α ⊕ β
(u_0, u_1) ← (1, 0); σ ← 0
for each maximal run I = [s, s + ℓ − 1] of equal K-bits with value r do
    β_I ← the ℓ-bit slice of β on I
    t_I ← the ℓ-bit slice of t on I
    j ← 0
    while j < ℓ do
        Let p be the first index p ≥ j with ReadBit(β_I, p) = 1, or p = ℓ if none exists
        L ← p − j
        if L > 0 then
            k_1 ← wt((t_I ≫ j) mod 2^L); k_0 ← L − k_1
            apply the closed-form power of B_{r,0}^{k_0} to (u_0, u_1)
            apply the closed-form power of B_{r,1}^{k_1} to (u_0, u_1)
            σ ← σ + L
        end if
        if p = ℓ then
            break
        end if
        (u_0, u_1) ← (u_0, −u_1)                          ▷ insert D = diag(1, −1)
        τ ← ReadBit(t_I, p); apply one step of B_{r,τ} to (u_0, u_1)
```

$\sigma \leftarrow \sigma + 1$; $j \leftarrow p + 1$
    end while
end for
$\mathcal{N} \leftarrow u_0 + u_1$
if publicSign $= -1$ **then**
    $\mathcal{N} \leftarrow -\mathcal{N}$
end if
return $(\mathcal{N}, \sigma)$

Algorithm 29 FixedAddendLinearOracle

Require: public constant K, masks α, β, width n
Ensure: $(\mathsf{possible}, \mathrm{sign}, W, |C|)$
$(\mathcal{N}, C, W) \leftarrow \textsc{FixedAddendLinearNumerator}(K, \alpha, \beta, n)$
if $\mathcal{N} = 0$ **then**
    return $(\mathsf{false}, 0, +\infty, 0)$
end if
sign $\leftarrow 1$ if $\mathcal{N} > 0$, otherwise -1
$|C| \leftarrow |\mathcal{N}|/2^n$; $W \leftarrow n - \log_2 |\mathcal{N}|$
return $(\mathsf{true}, \mathrm{sign}, W, |C|)$

A.6.7 MILP meaning of the oracle output

The exact numerator oracle is the mathematical source of the fixed-public linear MILP interface. A live transition must satisfy $\mathcal{N} \neq 0$. The local linear characteristic stores its sign $\mathrm{sgn}(\mathcal{N})$, while the MELCC-style objective uses the absolute weight $n - \log_2 |\mathcal{N}|$. If the model exposes P_i, Q_i as integer state variables and selects one of the transfer rows with binary variables, the bilinear selector-products are linearized by standard McCormick envelopes. The exact-log part is then handled by the epigraph cuts of Subsection A.7. Alternatively, a precomputed live-row table can attach the exact signed numerator and exact log-weight to each live mask pair.

Chinese explanation. This fixed-public linear oracle computes a signed integer numerator rather than a probability count. The constant K is not sampled; it drives the carry transition bit by bit. The row vector (P, Q) stores signed prefix sums for carry-in states 0 and 1. At the end, $\mathcal{N} = P + Q$ and $C = \mathcal{N}/2^n$. If $\mathcal{N} = 0$, the mask pair is dead. Otherwise the exact weight is $n - \log_2 |\mathcal{N}|$. This is the fixed-addend linear object needed by the NeoAlzette fixed-public arithmetic sites; it cannot be replaced by the averaged two-variable Wallen/Schulte-Geers oracle.

A.7 Fixed-public MILP interfaces and exact-log epigraph

This subsection turns the point-evaluation oracles of Subsections A.5 and A.6 into MILP objects. The most important case for NeoAlzette is the fixed-public arithmetic site: one operand is a state word and the other operand is a public component constant. The model below is therefore not the two-variable addition box with a zero difference on the second input. It is the exact linearization of the Machado–Azimi zero-chain decomposition of the fixed-addend differential oracle. The public bit pattern of the constant is part of the local relation.

A.7.1 Differential fixed-public exact MILP: problem setup

Let

$$F_{+K}(X) = X \boxplus_n K,$$

where $K \in \mathbb{F}_2^n$ is public and fixed. Let $u, v \in \mathbb{F}_2^n$ be the visible input and output XOR differences. The bit convention is the same as in Subsection A.5: virtual boundary bits satisfy

$$u_{-1} = v_{-1} = K_{-1} = 0,$$

all probabilities are over the single variable word X, and the fixed addend is not sampled.

For each coordinate define

$$e_i = u_i \oplus v_i, \qquad S_i = (u_{i-1}, v_{i-1}, e_i) \in \{0,1\}^3.$$

We identify this state with the integer

$$r_i = 4u_{i-1} + 2v_{i-1} + e_i \in \{0, \dots, 7\}.$$

Thus $S_i = 000$ corresponds to $r_i = 0$, and $S_i = 001$ corresponds to $r_i = 1$. By the fixed-public differential oracle, state 001 is impossible. For feasible transitions the exact weight is

$$W_K^{\Delta}(u, v) = W_{\text{base}} + \sum_{i \in \mathcal{T}} \Big((\lambda_i - 1) - \log_2 p_{i,\lambda_i,q_i} \Big), \tag{A.18}$$

where

$$W_{\text{base}} = \sum_{i=1}^{n-1} \mathbf{1}[S_i \in \{010, 011, 100, 101\}], \qquad \mathcal{T} = \{i : 1 \le i < n, S_i \in \{110, 111\}\}.$$

For a trigger position i, the zero-chain length is

$$\lambda_i = 1 + \max\{L \ge 0 : S_{i-1} = S_{i-2} = \cdots = S_{i-L} = 000\}, \tag{A.19}$$

where the chain stops at the low-word boundary. Let

$$q_i = e_i \oplus K_{i-1}.$$

For every candidate chain length $\lambda \in \{1, \dots, i+1\}$, define the public constant block value

$$B_{i,\lambda} = K_{i-\lambda-1} + \sum_{j=i-\lambda}^{i-2} 2^{j-(i-\lambda)} K_j, \tag{A.20}$$

with $K_j = 0$ for $j < 0$. The trigger numerator is

$$p_{i,\lambda,q} = \begin{cases} B_{i,\lambda}, & q = 1, \\ 2^{\lambda-1} - B_{i,\lambda}, & q = 0. \end{cases} \tag{A.21}$$

If $p_{i,\lambda,q} = 0$, that selector branch is absent from the MILP model and the corresponding trigger configuration is infeasible.

A.7.2 State encoding variables

For each bit position $0 \le i < n$, introduce a binary variable e_i and one-hot binary state variables

$$s_{i,r} \in \{0,1\}, \qquad r \in \{0,\dots,7\}.$$

The one-hot condition is

$$\sum_{r=0}^{7} s_{i,r} = 1. \tag{A.22}$$

The XOR relation $e_i = u_i \oplus v_i$ is encoded by the exact four-inequality XOR hull

$$\begin{aligned} e_i &\le u_i + v_i, \\ e_i &\ge u_i - v_i, \\ e_i &\ge v_i - u_i, \\ e_i &\le 2 - u_i - v_i. \end{aligned} \tag{A.23}$$

Let $\bar{u}_{i-1} = u_{i-1}$ for $i > 0$, and $\bar{u}_{-1} = 0$. Define $\bar{v}$ analogously. The one-hot state variables are bound to $(\bar{u}_{i-1}, \bar{v}_{i-1}, e_i)$ by

$$\begin{aligned} \sum_{r:\lfloor r/4 \rfloor = 1} s_{i,r} &= \bar{u}_{i-1}, \\ \sum_{r:(\lfloor r/2 \rfloor \bmod 2)=1} s_{i,r} &= \bar{v}_{i-1}, \\ \sum_{r:r \bmod 2=1} s_{i,r} &= e_i. \end{aligned} \tag{A.24}$$

The forbidden state is removed by

$$s_{i,1} = 0 \qquad (0 \le i < n). \tag{A.25}$$

At $i = 0$, (A.24)–(A.25) force $e_0 = 0$, hence $u_0 = v_0$, matching the low-boundary condition of the oracle.

A.7.3 Zero-chain selector variables

For each trigger position $1 \le i < n$, each candidate chain length $1 \le \lambda \le i + 1$, and each $q \in \{0,1\}$, precompute $p_{i,\lambda,q}$ from (A.20)–(A.21). If $p_{i,\lambda,q} > 0$, create a binary selector

$$z_{i,\lambda,q} \in \{0,1\}.$$

If $p_{i,\lambda,q} = 0$, no selector is created. This convention is part of the exact infeasibility mechanism: a zero numerator cannot be selected.

The trigger equation states that a trigger state chooses exactly one valid zero-chain branch, while a non-trigger chooses none:

$$\sum_{\lambda=1}^{i+1} \sum_{\substack{q\in\{0,1\} \\ p_{i,\lambda,q}>0}} z_{i,\lambda,q} = s_{i,6} + s_{i,7} \qquad (1 \le i < n). \tag{A.26}$$

A.7.4 Exact chain-length constraints

The selected λ must be the exact length of the consecutive 000-chain immediately below the trigger. This is encoded by two families of inequalities.

First, if $z_{i,\lambda,q} = 1$, the previous $\lambda - 1$ states are all 000:

$$z_{i,\lambda,q} \le s_{i-k,0}, \qquad 1 \le k \le \lambda - 1. \tag{A.27}$$

Second, if the chain does not reach the low boundary, then the state immediately before the chain is not 000:

$$z_{i,\lambda,q} \le 1 - s_{i-\lambda,0}, \qquad \text{if } i - \lambda \ge 0. \tag{A.28}$$

If $i - \lambda < 0$, no stopping constraint is needed because the zero-chain reaches the virtual boundary. Equations (A.27)–(A.28) prevent the solver from selecting either a shorter chain or a longer chain than the oracle would compute.

A.7.5 Constant-dependent q-branch binding

The branch bit is

$$q = e_i \oplus K_{i-1}.$$

Since K_{i-1} is public, the selector partition can be bound linearly to the trigger state. If $K_{i-1} = 0$, state 110 has $q = 0$ and state 111 has $q = 1$:

$$\begin{aligned} \sum_{\lambda : p_{i,\lambda,0} > 0} z_{i,\lambda,0} &= s_{i,6}, \\ \sum_{\lambda : p_{i,\lambda,1} > 0} z_{i,\lambda,1} &= s_{i,7}. \end{aligned} \tag{A.29}$$

If $K_{i-1} = 1$, the roles are swapped:

$$\begin{aligned} \sum_{\lambda : p_{i,\lambda,0} > 0} z_{i,\lambda,0} &= s_{i,7}, \\ \sum_{\lambda : p_{i,\lambda,1} > 0} z_{i,\lambda,1} &= s_{i,6}. \end{aligned} \tag{A.30}$$

If a required q-class has no positive-numerator selector, the corresponding sum is the empty sum zero, and (A.29) or (A.30) forces the trigger state variable to zero. This is the MILP counterpart of the oracle rule $p = 0 \Rightarrow \mathrm{DP} = 0$.

A.7.6 Objective: exact real differential weight

The objective is the zero-chain weight (A.18), expressed as a linear objective over binary variables. The base contribution is

$$W_{\text{base}}^{\text{MILP}} = \sum_{i=1}^{n-1} \sum_{r \in \{2,3,4,5\}} s_{i,r}. \tag{A.31}$$

For each created selector, attach the exact real coefficient

$$c_{i,\lambda,q} = (\lambda - 1) - \log_2 p_{i,\lambda,q}. \tag{A.32}$$

The trigger contribution is

$$W_{\text{trig}}^{\text{MILP}} = \sum_{i=1}^{n-1} \sum_{\lambda,q : p_{i,\lambda,q} > 0} c_{i,\lambda,q} z_{i,\lambda,q}. \tag{A.33}$$

The total local objective is

$$\boxed{W_{+K}^{\Delta,\text{MILP}}(u,v) = W_{\text{base}}^{\text{MILP}} + W_{\text{trig}}^{\text{MILP}}.} \tag{A.34}$$

All non-integer terms are public real constants known before the solve. The solver does not approximate the oracle probability; it chooses a state/chain selector pattern, and the objective accumulates the exact real coefficient corresponding to that pattern.

A.7.7 Correctness of the fixed-public differential MILP

Proposition 2 (Exactness of the fixed-public differential MILP)**.** *For fixed K, u, v, the MILP system (A.22)–(A.30) is feasible if and only if $\mathrm{DP}_{+K}(u \to v) > 0$. For every feasible integral assignment, the objective value (A.34) equals $-\log_2 \mathrm{DP}_{+K}(u \to v)$.*

Proof. The one-hot and binding constraints (A.22)–(A.24) make $s_{i,r}$ the exact encoding of $S_i = (u_{i-1}, v_{i-1}, e_i)$. Constraint (A.25) is exactly the forbidden-state check $S_i \neq 001$. If $S_i \notin \{110, 111\}$, (A.26) selects no zero-chain branch. If $S_i \in \{110, 111\}$, (A.26) selects exactly one branch. Constraints (A.27)–(A.28) force the selected λ to be precisely the maximal backward 000-chain length. Constraints (A.29)–(A.30) force $q = e_i \oplus K_{i-1}$, and the absence of a selector with $p_{i,\lambda,q} = 0$ enforces the zero-probability trigger cases. Thus feasible assignments are in one-to-one correspondence with the zero-chain choices made by the fixed-public differential oracle. The base terms (A.31) add one unit for the half-probability states, and the selector terms (A.32)–(A.33) add exactly the trigger weights of (A.18). Therefore the total objective is the exact fixed-public differential weight. □

For fixed-public subtraction sites $F_{-C}(X) = X - C \pmod{2^n}$, the same construction is used with the oriented borrow-state version of Subsection A.5, or with the fixed-right two's-complement addition wrapper $X - C \equiv X + (-C) \pmod{2^n}$. In either case the site remains a fixed-public one-variable relation; it is not replaced by a two-variable averaged subtraction box.

Algorithm 30 BuildFixedPublicDiffExactMILP

Require: public constant K, binary difference variables u, v, width n
Ensure: MILP constraints and objective for the exact fixed-public differential box

$\mathsf{Obj} \leftarrow 0$
for $i = 0$ to $n-1$ **do**
  create binary variable e_i
  add the four XOR-hull inequalities for $e_i = u_i \oplus v_i$
  for $r = 0$ to 7 **do**
    create binary state variable $s_{i,r}$
  end for
  add $\sum_{r=0}^{7} s_{i,r} = 1$
  set $u_{\mathrm{prev}} \leftarrow 0$ if $i = 0$, else u_{i-1}
  set $v_{\mathrm{prev}} \leftarrow 0$ if $i = 0$, else v_{i-1}
  bind the one-hot state bits to $(u_{\mathrm{prev}}, v_{\mathrm{prev}}, e_i)$
  add $s_{i,1} = 0$
end for
for $i = 1$ to $n-1$ **do**
  for $r \in \{2, 3, 4, 5\}$ **do**
    $\mathsf{Obj} \leftarrow \mathsf{Obj} + s_{i,r}$
  end for
end for
for $i = 1$ to $n-1$ **do**
  initialize selector set $Z_i \leftarrow \emptyset$
  for $\lambda = 1$ to $i+1$ **do**
    compute $B_{i,\lambda}$ by (A.20)
    $M \leftarrow 2^{\lambda-1}$; $p_{i,\lambda,0} \leftarrow M - B_{i,\lambda}$; $p_{i,\lambda,1} \leftarrow B_{i,\lambda}$
    for $q \in \{0, 1\}$ **do**
      if $p_{i,\lambda,q} > 0$ **then**
        create binary selector $z_{i,\lambda,q}$ and add it to Z_i
        $\mathsf{Obj} \leftarrow \mathsf{Obj} + ((\lambda - 1) - \log_2 p_{i,\lambda,q}) z_{i,\lambda,q}$
      end if
    end for
  end for
  add $\sum_{z \in Z_i} z = s_{i,6} + s_{i,7}$
  for each created selector $z_{i,\lambda,q} \in Z_i$ **do**
    for $k = 1$ to $\lambda - 1$ **do**
      add $z_{i,\lambda,q} \le s_{i-k,0}$
    end for
    if $i - \lambda \ge 0$ **then**
      add $z_{i,\lambda,q} \le 1 - s_{i-\lambda,0}$
    end if
  end for
  if $\mathrm{ReadBit}(K, i-1) = 0$ **then**
    add $\sum_{\lambda : p_{i,\lambda,0} > 0} z_{i,\lambda,0} = s_{i,6}$
    add $\sum_{\lambda : p_{i,\lambda,1} > 0} z_{i,\lambda,1} = s_{i,7}$
  else
    add $\sum_{\lambda : p_{i,\lambda,0} > 0} z_{i,\lambda,0} = s_{i,7}$
    add $\sum_{\lambda : p_{i,\lambda,1} > 0} z_{i,\lambda,1} = s_{i,6}$
  end if
end for
return the constraint set and Obj

A.7.8 Fixed-public linear exact MILP: signed numerator and exact-log weight

The MILP model for the linear analysis of the fixed-addend map

$$Y = X \boxplus_n K$$

is an exact linearization of the signed two-state carry-transfer recurrence in Subsection A.6. It is not the Wallén/Schulte–Geers two-variable linear box with a frozen second mask. In the fixed-public case the Walsh sum is over the single variable word X, while the public bit string K chooses the carry-transfer matrix at every bit position. The model therefore exposes the signed numerator recurrence, linearizes the products of binary mask selectors with the signed prefix state, enforces a nonzero numerator through a sign branch, binds an absolute numerator, and prices the local correlation by an exact-log epigraph.

Problem setup. Let $K \in \mathbb{F}_2^n$ be the fixed public addend and let $\alpha, \beta \in \mathbb{F}_2^n$ be the input and output masks. The exact signed correlation is

$$C_K(\alpha, \beta) = 2^{-n} \sum_{X \in \mathbb{F}_2^n} (-1)^{\alpha \cdot X \oplus \beta \cdot (X \boxplus_n K)} = \frac{\mathcal{N}}{2^n},$$

where $\mathcal{N} \in \mathbb{Z}$ is the signed numerator and $|\mathcal{N}| \le 2^n$. With

$$t_i = \alpha_i \oplus \beta_i, \qquad b_i = \beta_i,$$

the scaled signed prefix row

$$(P_i, Q_i) = 2^i (R_i^{(0)}, R_i^{(1)})$$

evolves from

$$P_0 = 1, \qquad Q_0 = 0$$

by an integer transfer

$$(P_{i+1}, Q_{i+1}) = (P_i, Q_i) A_i,$$

where A_i is selected by (K_i, t_i, b_i). The final numerator is

$$\mathcal{N} = P_n + Q_n.$$

A live fixed-public linear transition is one with $\mathcal{N} \neq 0$, and its local MELCC-style characteristic weight is

$$W_{+K}^{L}(\alpha, \beta) = n - \log_2 |\mathcal{N}|.$$

Explicit integer transfer matrices. For a fixed bit position, put $k = K_i$, $t = \alpha_i \oplus \beta_i$, and $b = \beta_i$. The matrix $A(k, t, b)$ is defined by the following exact table.

For $k = 0$:

(t, b)	(P_{i+1}, Q_{i+1})	$A(0, t, b)$
$(0, 0)$	$(2P_i + Q_i,\ Q_i)$	$\begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}$
$(0, 1)$	$(2P_i - Q_i,\ -Q_i)$	$\begin{bmatrix} 2 & 0 \\ -1 & -1 \end{bmatrix}$
$(1, 0)$	$(Q_i,\ -Q_i)$	$\begin{bmatrix} 0 & 0 \\ 1 & -1 \end{bmatrix}$
$(1, 1)$	$(-Q_i,\ Q_i)$	$\begin{bmatrix} 0 & 0 \\ -1 & 1 \end{bmatrix}$

For $k = 1$:

(t, b)	(P_{i+1}, Q_{i+1})	$A(1, t, b)$
$(0, 0)$	$(P_i,\ P_i + 2Q_i)$	$\begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$
$(0, 1)$	$(-P_i,\ -P_i + 2Q_i)$	$\begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}$
$(1, 0)$	$(P_i,\ -P_i)$	$\begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}$
$(1, 1)$	$(-P_i,\ P_i)$	$\begin{bmatrix} -1 & -1 \\ 0 & 2 \end{bmatrix}$

These matrices are integral because they are the integer-scaled versions of the one-bit signed Walsh transfer. They enumerate the two choices of the current input bit under the full-adder carry constraint and include the phase term from $\beta_i K_i$ and $\beta_i c_i$.

Highest-active-bit condition. Since the output bit Y_j of $X \boxplus_n K$ depends only on $X_0, \ldots, X_j$ and contains X_j with coefficient one, a nonzero fixed-public correlation requires

$$h(\alpha) = h(\beta), \qquad h(m) = \max\{i : m_i = 1\}, \quad h(0) = -1.$$

The MILP encodes this by suffix-OR variables. For a mask m, introduce $\rho_i(m)$ satisfying

$$\rho_{n-1}(m) = m_{n-1},$$

$$\rho_i(m) \geq m_i, \qquad \rho_i(m) \geq \rho_{i+1}(m), \qquad \rho_i(m) \leq m_i + \rho_{i+1}(m) \quad (0 \leq i < n-1).$$

Then enforce

$$\rho_i(\alpha) = \rho_i(\beta) \qquad (0 \leq i < n).$$

This rejects mask pairs with different top active coordinates before the signed transfer is priced.

Mask selectors and signed prefix state. For each position i, introduce four one-hot selector variables

$$z_{i,00}, z_{i,01}, z_{i,10}, z_{i,11} \in \{0, 1\}, \qquad \sum_{\tau \in \{00,01,10,11\}} z_{i,\tau} = 1.$$

If $\tau = (a, b)$, bind the selector to the actual masks by

$$\alpha_i = z_{i,10} + z_{i,11}, \qquad \beta_i = z_{i,01} + z_{i,11}.$$

The signed prefix variables satisfy

$$P_i, Q_i \in [-2^i, 2^i] \qquad (0 \leq i \leq n),$$

with $P_0 = 1$ and $Q_0 = 0$. They may be continuous in the LP relaxation; for an integral selector assignment, the transfer equations force them to integer signed prefix sums.

McCormick split variables. The selected transfer matrix depends on $z_{i,\tau}$, and the recurrence contains products $z_{i,\tau} P_i$ and $z_{i,\tau} Q_i$. For each selector τ and each state coordinate $d \in \{0, 1\}$, introduce split variables

$$Y_{i,\tau}^{(0)} = z_{i,\tau} P_i, \qquad Y_{i,\tau}^{(1)} = z_{i,\tau} Q_i.$$

Let $L_i = -2^i$ and $U_i = 2^i$. For $Y = zS$, with $z \in \{0,1\}$ and $S \in [L_i, U_i]$, use the exact binary McCormick envelope

$$\begin{aligned} Y &\geq L_i z, & Y &\leq U_i z, \\ Y &\geq S - U_i(1-z), & Y &\leq S - L_i(1-z). \end{aligned}$$

For integral z, these inequalities force $Y = 0$ when $z = 0$ and $Y = S$ when $z = 1$. The recovery equalities

$$P_i = \sum_{\tau} Y_{i,\tau}^{(0)}, \qquad Q_i = \sum_{\tau} Y_{i,\tau}^{(1)}$$

strengthen the model and keep the split state tied to the selected row.

Transfer constraints. For a selector $\tau = (a, b)$, define

$$t(\tau) = a \oplus b, \qquad b(\tau) = b,$$

and let

$$A_{K_i,\tau} = A(K_i, t(\tau), b(\tau)).$$

Write the entries of $A_{K_i,\tau}$ as $A_{K_i,\tau}(d, r)$, where d is the current carry-state coordinate and r is the next carry-state coordinate. The next signed state is enforced by

$$P_{i+1} = \sum_{\tau} \sum_{d \in \{0,1\}} A_{K_i,\tau}(d, 0) Y_{i,\tau}^{(d)},$$

$$Q_{i+1} = \sum_{\tau} \sum_{d \in \{0,1\}} A_{K_i,\tau}(d, 1) Y_{i,\tau}^{(d)}.$$

This is exactly the row-vector multiplication $(P_{i+1}, Q_{i+1}) = (P_i, Q_i)A_i$ with a selected local transfer matrix.

Final numerator, sign branch, and absolute numerator. Introduce a signed numerator variable

$$\mathcal{N} \in [-2^n, 2^n]$$

and enforce

$$\mathcal{N} = P_n + Q_n.$$

To require a nonzero live transition, or more generally $|\mathcal{N}| \geq T$, introduce a binary sign selector η and put

$$T = \max(1, T_{\text{req}}), \qquad M_T = 2^n + T.$$

Use

$$\mathcal{N} + M_T \eta \geq T, \qquad \mathcal{N} + M_T \eta \leq M_T - T.$$

If $\eta = 0$, then $T \leq \mathcal{N} \leq 2^n$. If $\eta = 1$, then $-2^n \leq \mathcal{N} \leq -T$. Thus the model enforces the live-support condition and records the sign branch.

Now introduce an integer variable

$$A \in [1, 2^n]$$

for the absolute numerator. With

$$M_A = 2^{n+1} + 2,$$

use

$$\begin{aligned} A - \mathcal{N} - M_A \eta &\leq 0, & A - \mathcal{N} + M_A \eta &\geq 0, \\ A + \mathcal{N} + M_A \eta &\leq M_A, & A + \mathcal{N} - M_A \eta &\geq -M_A. \end{aligned}$$

If $\eta = 0$, these force $A = \mathcal{N}$; if $\eta = 1$, they force $A = -\mathcal{N}$.

Exact-log epigraph. The exact local weight is

$$W = n - \log_2 A.$$

The function

$$f(A) = n - \log_2 A$$

is convex on $A > 0$. A continuous epigraph variable $W \in [0, n]$ is constrained by supporting hyperplanes. For every integer tangent point $a \in [1, 2^n]$, the tangent inequality is

$$W \geq f(a) + f'(a)(A - a), \qquad f'(a) = -\frac{1}{a \ln 2}.$$

Equivalently,

$$W \geq -\frac{A}{a \ln 2} + n - \log_2 a + \frac{1}{\ln 2}.$$

The model initially inserts tangents at $a = 1$ and $a = 2^n$. During separation, if an LP solution (A^*, W^*) violates $W \geq n - \log_2 A$, the separator clamps A^* to $[1, 2^n]$, rounds it to a nearby integer a_0, and adds the tangents at $a_0 - 1$, a_0, and $a_0 + 1$ when they are in range and violated. At an accepted integral solution, the check routine verifies the exact nonlinear inequality $W \geq n - \log_2 A$. Since W is minimized in the characteristic objective, it is driven to equality at optimum within solver tolerance.

Correctness statement.

Proposition 3 (Exactness of the fixed-public linear MILP interface)**.** *For an integral assignment of the mask bits, the selector, McCormick, and transfer constraints compute exactly the signed numerator $\mathcal{N} = 2^n C_K(\alpha, \beta)$. The sign and absolute-value constraints enforce $A = |\mathcal{N}|$ and $A \geq 1$ for live transitions. The epigraph constraint prices the exact weight $n - \log_2 A$ up to the separation/checking tolerance of the nonlinear cut handler. Therefore the model is an exact MILP interface for the fixed-public linear point oracle $Q_1^{\mathrm{LC},+K}$; it is not the two-variable Wallén automaton applied with a frozen second operand.*

Proof. For each bit i, exactly one selector $z_{i,\tau}$ is one and it is bound to (α_i, β_i). The McCormick envelopes are exact for products of a binary selector and a bounded signed state variable in every integral assignment. Hence the split variables equal either zero or the selected (P_i, Q_i). The transfer constraints are therefore exactly the integer row-vector update by the matrix determined by $(K_i, \alpha_i \oplus \beta_i, \beta_i)$. Induction over i gives the same (P_n, Q_n) and the same signed numerator as the fixed-public linear oracle. The big-M constraints separate the positive and negative live numerator cases and bind the absolute value. The tangent inequalities and exact separation/check routine enforce the convex weight function $n - \log_2 A$. This proves the stated correspondence. □

Algorithm 31 BuildFixedAddendLinearNumeratorMILP

Require: public constant K, input mask variables α, output mask variables β, width n, requested threshold T_{req}
Ensure: MILP constraints for the signed numerator $\mathcal{N}$, sign selector η, and live threshold T

```
T ← 1 if T_req = 0, otherwise T ← min(T_req, 2^n)
if T > 2^n then
    add contradiction and return
end if
for m ∈ {α, β} do
    create suffix-OR variables ρ_i(m) for 0 ≤ i < n
    add ρ_{n−1}(m) = m_{n−1}
```

    for $i = n-2$ down to 0 **do**
        add $\rho_i(m) \geq m_i$, $\rho_i(m) \geq \rho_{i+1}(m)$, and $\rho_i(m) \leq m_i + \rho_{i+1}(m)$
    end for
end for
for $i = 0$ to $n-1$ **do**
    add $\rho_i(\alpha) = \rho_i(\beta)$
end for
fix $P_0 = 1$ and $Q_0 = 0$
for $i = 1$ to n **do**
    create continuous $P_i, Q_i \in [-2^i, 2^i]$
end for
for $i = 0$ to $n-1$ **do**
    create binary selectors $z_{i,00}, z_{i,01}, z_{i,10}, z_{i,11}$ and add $\sum_\tau z_{i,\tau} = 1$
    bind $\alpha_i = z_{i,10} + z_{i,11}$ and $\beta_i = z_{i,01} + z_{i,11}$
    set $L = -2^i$, $U = 2^i$
    for $\tau \in \{00, 01, 10, 11\}$ **do**
        for $d \in \{0, 1\}$ **do**
            create split variable $Y_{i,\tau}^{(d)} \in [L, U]$
            let $S = P_i$ if $d = 0$, and $S = Q_i$ if $d = 1$
            add $Y \geq Lz$, $Y \leq Uz$, $Y \geq S - U(1-z)$, $Y \leq S - L(1-z)$
        end for
    end for
    add $P_i = \sum_\tau Y_{i,\tau}^{(0)}$ and $Q_i = \sum_\tau Y_{i,\tau}^{(1)}$
    $k \leftarrow \mathrm{ReadBit}(K, i)$
    add the transfer equalities for P_{i+1} and Q_{i+1} using $A_{k,\tau}$
end for
create $\mathcal{N} \in [-2^n, 2^n]$ and add $\mathcal{N} = P_n + Q_n$
create binary sign selector η; set $M_T = 2^n + T$
add $\mathcal{N} + M_T\eta \geq T$ and $\mathcal{N} + M_T\eta \leq M_T - T$
return the constraints and variables $(\mathcal{N}, \eta, T)$

Algorithm 32 AttachExactLogWeightMILPObjective

Require: signed numerator $\mathcal{N}$, sign selector η, width n
Ensure: absolute numerator A and exact-log weight variable W

create integer $A \in [1, 2^n]$ and set $M_A = 2^{n+1} + 2$
add $A - \mathcal{N} - M_A\eta \leq 0$ and $A - \mathcal{N} + M_A\eta \geq 0$
add $A + \mathcal{N} + M_A\eta \leq M_A$ and $A + \mathcal{N} - M_A\eta \geq -M_A$
create continuous $W \in [0, n]$ and give W objective coefficient 1
add the tangent at $a = 1$: $W \geq n - A/\ln 2 + 1/\ln 2$
add the tangent at $a = 2^n$: $W \geq 1/\ln 2 - A/(2^n \ln 2)$
register the exact-log separation and checking routine for $W \geq n - \log_2 A$
return (A, W)

Algorithm 33 SeparateExactLogTangent

Require: current LP solution (A^*, W^*), width n

$\bar{A} \leftarrow \max(1, \min(A^*, 2^n))$
$W_{\mathrm{req}} \leftarrow n - \log_2 \bar{A}$
if $W^* \geq W_{\mathrm{req}} - 10^{-8}$ **then**
    return no cut
end if
$a_0 \leftarrow \mathrm{round}(\bar{A})$ clamped to $[1, 2^n]$
for $a \in \{a_0 - 1, a_0, a_0 + 1\}$ **do**

```
    if 1 ≤ a ≤ 2^n then
        add W ≥ (−1/(a ln 2))A + n − log_2 a + 1/ ln 2
    end if
end for
```

Chinese explanation. The fixed-public linear MILP is an exact translation of the signed numerator oracle. The state variables P_i, Q_i are signed prefix sums for the two carry states. The local mask pair selects one integer transfer matrix at each bit. McCormick envelopes linearize the selected multiplication with the signed prefix state. The final numerator $\mathcal{N} = P_n + Q_n$ may be positive or negative, so a sign selector binds the absolute numerator $A = |\mathcal{N}|$. The exact local weight is the real value $n - \log_2 A$, enforced by a convex tangent-cut epigraph. Thus the model prices the real fixed-public linear correlation weight without replacing it by a two-variable average or a proxy carry count.

A.8 Two-variable modular addition differential oracle

The two-variable XOR-differential oracle follows the Lipmaa–Moriai modular-addition formula [LM02]. For

$$Z = X \boxplus_n Y$$

and differences $(\alpha, \beta \to \gamma)$, define

$$t_i = \alpha_i \oplus \beta_i \oplus \gamma_i, \qquad \alpha_{-1} = \beta_{-1} = \gamma_{-1} = 0.$$

The transition is feasible exactly when

$$\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1} \quad \Longrightarrow \quad t_i = \alpha_{i-1}$$

for every $0 \le i < n$. If feasible, the single-characteristic weight is

$$W_{\boxplus}^{\Delta} = \sum_{i=0}^{n-2} \mathbf{1}[(\alpha_i, \beta_i, \gamma_i) \notin \{000, 111\}],$$

and

$$DP_{\boxplus_n}(\alpha, \beta \to \gamma) = 2^{-W_{\boxplus}^{\Delta}}.$$

Algorithm 34 EvaluateAddDiffOracle$(\alpha, \beta, \gamma, n)$

```
if α_0 ⊕ β_0 ≠ γ_0 then
    return (false, +∞)
end if
W ← 0
for i = 0 to n − 2 do
    if α_i = β_i = γ_i then
        if α_{i+1} ⊕ β_{i+1} ⊕ γ_{i+1} ≠ α_i then
            return (false, +∞)
        end if
    else
        W ← W + 1
    end if
end for
return (true, W)
```

A.9 Two-variable modular addition linear oracle

For

$$Z = X \boxplus_n Y$$

with output mask Γ and input masks A, B, the signed correlation is

$$C_{\boxplus_n}(\Gamma; A, B) = 2^{-2n} \sum_{X,Y} (-1)^{\Gamma \cdot (X \boxplus_n Y) \oplus A \cdot X \oplus B \cdot Y}.$$

The oracle follows Wallén's carry-correlation classification and the related CCZ/carry-state formulations [Wal03a, SG13]. In the MILP-friendly view, a binary state/weight sequence q_i encodes the local nonzero-correlation transition. The most significant boundary is fixed according to the Wallén recurrence, while the lower boundary is kept free in the implemented model so that valid perfect approximations are not removed.

The single-characteristic linear weight is the sum of the live transition bits:

$$W^L_{\boxplus} = \sum_i q_i, \qquad |C_{\boxplus_n}| = 2^{-W^L_{\boxplus}}$$

for a live transition. The sign is computed by the oracle and must be retained by any endpoint linear-hull aggregation.

A.10 Modular subtraction wrappers

Both two-variable subtraction boxes are wrappers around addition:

$$Z = X \boxminus_n Y \quad \Longleftrightarrow \quad X = Z \boxplus_n Y.$$

Consequently,

$$SUB^{\Delta}_n(\alpha_X, \alpha_Y \to \alpha_Z) = ADD^{\Delta}_n(\alpha_Z, \alpha_Y \to \alpha_X),$$

and

$$SUB^{L}_n(\Gamma_Z; A_X, A_Y) = ADD^{L}_n(A_X; \Gamma_Z, A_Y).$$

This wrapper convention prevents an independent subtraction truth table from drifting away from the addition oracle.

A.11 Boolean and bit-vector MILP box derivations

All Boolean boxes are exact convex-hull encodings over binary variables. For each box below, the listed inequalities cut out precisely the convex hull of the satisfying truth-table rows.

XOR. For $c = a \oplus b$,

$$a + b - c \geq 0, \quad a - b + c \geq 0, \quad -a + b + c \geq 0, \quad -a - b - c + 2 \geq 0.$$

AND. For $c = a \wedge b$,

$$c \leq a, \quad c \leq b, \quad c \geq a + b - 1.$$

OR. For $c = a \vee b$,

$$c \geq a, \quad c \geq b, \quad c \leq a + b.$$

NOT. For $c = \neg a$,

$$c = 1 - a.$$

Copy and fixed bit permutations. A copied bit is modeled by equality constraints. Rotations and zero-fill shifts use the accessor convention of Part II:

$$\mathrm{ReadBit}(Y, i) = \mathrm{ReadBit}(X, (i - r) \bmod n)$$

for left rotation, and fixed-zero coordinates for zero-fill shifts.

A.12 ADD-Δ MILP derivation

The thirteen-inequality ADD-Δ box is the MILP local form of the Lipmaa–Moriai feasibility/weight condition; it is the same ARX-MILP style used in automatic trail search for Speck-like additions [FWG+16, LLJW21].

Let p_i be the local transition-weight bit for the adjacent transition $i \to i+1$. The first five inequalities force p_i to be the indicator of a non-equal local triple:

$$\begin{aligned}
\beta_i - \gamma_i + p_i &\geq 0,\\
\alpha_i - \beta_i + p_i &\geq 0,\\
-\alpha_i + \gamma_i + p_i &\geq 0,\\
\alpha_i + \beta_i + \gamma_i - p_i &\geq 0,\\
-\alpha_i - \beta_i - \gamma_i - p_i &\geq -3.
\end{aligned}$$

Thus $p_i = 0$ exactly when $(\alpha_i, \beta_i, \gamma_i) \in \{000, 111\}$, and $p_i = 1$ otherwise. The remaining eight inequalities enforce the next-bit Lipmaa–Moriai condition whenever $p_i = 0$:

$$\begin{aligned}
\alpha_{i+1} - \beta_i + \beta_{i+1} + \gamma_{i+1} + p_i &\geq 0,\\
\beta_i + \alpha_{i+1} - \beta_{i+1} + \gamma_{i+1} + p_i &\geq 0,\\
\beta_i - \alpha_{i+1} + \beta_{i+1} + \gamma_{i+1} + p_i &\geq 0,\\
\alpha_i + \alpha_{i+1} + \beta_{i+1} - \gamma_{i+1} + p_i &\geq 0,\\
\gamma_i - \alpha_{i+1} - \beta_{i+1} - \gamma_{i+1} + p_i &\geq -2,\\
-\beta_i + \alpha_{i+1} - \beta_{i+1} - \gamma_{i+1} + p_i &\geq -2,\\
-\beta_i - \alpha_{i+1} + \beta_{i+1} - \gamma_{i+1} + p_i &\geq -2,\\
-\beta_i - \alpha_{i+1} - \beta_{i+1} + \gamma_{i+1} + p_i &\geq -2.
\end{aligned}$$

Together with the least significant condition $\alpha_0 \oplus \beta_0 = \gamma_0$, the objective contribution is

$$W_{\boxplus}^{\Delta} = \sum_{i=0}^{n-2} p_i.$$

A.13 ADD-L MILP derivation

The ADD-L box encodes the nonzero-correlation transition polytope associated with the Wallén carry-state view [Wal03a]. For every bit, introduce adjacent transition variables s_i, s_{i+1} and masks (Γ_i, A_i, B_i). The local inequalities are

$$\begin{aligned}
s_{i+1} - \Gamma_i - A_i + B_i + s_i &\geq 0,\\
s_{i+1} + \Gamma_i - A_i - B_i + s_i &\geq 0,\\
s_{i+1} + \Gamma_i - A_i + B_i - s_i &\geq 0,\\
-s_{i+1} + \Gamma_i + A_i + B_i + s_i &\geq 0,\\
s_{i+1} + \Gamma_i + A_i - B_i - s_i &\geq 0,\\
s_{i+1} - \Gamma_i + A_i - B_i + s_i &\geq 0,\\
s_{i+1} - \Gamma_i + A_i + B_i - s_i &\geq 0,\\
s_{i+1} + \Gamma_i + A_i + B_i + s_i &\leq 4.
\end{aligned}$$

The model uses one binary weight/state bit per mask coordinate, fixes the top boundary required by the recurrence, and keeps the lower boundary free. This free lower-boundary state is intentional: fixing it to zero would remove valid approximations, including perfect one-bit all-one addition masks. The objective sums the live transition bits according to the convention used by the assembled model.

A.14 Joint C/D injection oracle

The NeoAlzette C/D injection is a joint map. For each direction $s \in \{B \to A, A \to B\}$, define

$$\Phi_s : \mathbb{F}_2^{32} \to \mathbb{F}_2^{64}, \qquad \Phi_s(x) = \big(\Phi_s^{\mathrm{xor}}(x), \Phi_s^{\mathrm{add}}(x)\big).$$

The first half is consumed by routed XOR injection, and the second half is consumed by the CD-derived modular-addition operand. These two halves share the same source word and the same quadratic front half. Modeling them as independent variables describes a different component.

For an input difference Δ, the derivative of a degree-two packed injection is affine:

$$D_\Delta \Phi_s(x) = \Phi_s(x) \oplus \Phi_s(x \oplus \Delta) = M_{\Delta,s} x \oplus c_{\Delta,s}.$$

Therefore an output difference Γ is supported exactly when

$$\Gamma \in c_{\Delta,s} + \operatorname{im} M_{\Delta,s},$$

and a conservative single-characteristic differential charge is

$$w_{\mathrm{inj},s}^{\Delta} \geq \operatorname{rank} M_{\Delta,s}.$$

On the linear side,

$$\widehat{\Phi_s}(\alpha, \beta) = \sum_x (-1)^{\alpha \cdot x \oplus \beta \cdot \Phi_s(x)}.$$

The scalar phase $x \mapsto \beta \cdot \Phi_s(x)$ has an alternating Hessian contraction $S_{\beta,s}$. The live input masks form an affine support coset, and the characteristic weight hook is

$$w_{\mathrm{inj},s}^{L} \geq \frac{1}{2} \operatorname{rank} S_{\beta,s}.$$

A.15 QFH / Hessian / rank derivation

Let $\mu(u, v) = u \circ v$ denote coordinatewise multiplication over $\mathbb{F}_2$. For $\varepsilon \in \mathbb{F}_2$, define the unified NAND/NOR carrier

$$\Pi_\varepsilon(u, v) = 1 \oplus \mu(u, v) \oplus \varepsilon(u \oplus v).$$

Thus $\Pi_0(u, v) = \neg(u \wedge v)$ and $\Pi_1(u, v) = \neg(u \vee v)$. For affine views $U(x) = Lx \oplus a$, $Z(x) = Mx \oplus b$, and $\ell(x) = Nx \oplus c$, define

$$Q_\varepsilon[U, Z; \ell](x) = \ell(x) \oplus \Pi_\varepsilon(U(x), Z(x)).$$

Lemma 26 (Second-derivative skeleton)**.** *For $Q = Q_\varepsilon[U, Z; \ell]$,*

$$H_Q(u, v) = Q(0) \oplus Q(u) \oplus Q(v) \oplus Q(u \oplus v) = (Lu) \circ (Mv) \oplus (Lv) \circ (Mu).$$

Proof. All affine decorations cancel under a second Boolean derivative. Expanding the coordinate product $(Lx \oplus a)(Mx \oplus b)$ at $0, u, v, u \oplus v$ and XORing the four values cancels constant and linear terms. The remaining bilinear terms are $(Lu)(Mv)$ and $(Lv)(Mu)$ coordinatewise. The NAND/NOR polarity term $\varepsilon(U \oplus Z)$ is affine and also cancels. □

The packed C/D injection is assembled from these carriers, linear diffusion-mask views, cross-coupling products, anti-coupling products, shifts, rotations, and output XOR wiring. Consequently its differential behavior is controlled by derivative image ranks, while its Walsh-linear behavior is controlled by Hessian-contraction ranks. The two rank objects arise from the same quadratic tensor but they are not interchangeable.

A.16 Top-level NeoAlzette MILP model assembly

The assembled single-characteristic model is the ordered factor graph of the NeoAlzette schedule:

$$
\begin{aligned}
&B \leftarrow B \boxminus RC[1],\\
&\Phi_{B\to A}(B),\\
&A \leftarrow A \oplus \Phi^{\mathrm{xor}}_{B\to A}(B),\\
&A \leftarrow A \boxplus \Phi^{\mathrm{add}}_{B\to A}(B),\\
&B \leftarrow B \oplus \mathrm{A} \lll^{22} \oplus RC[4],\\
&A \leftarrow A \oplus \mathrm{B} \lll^{13},\\
&A \leftarrow A \boxminus RC[6],\\
&\Phi_{A\to B}(A),\\
&B \leftarrow B \oplus \Phi^{\mathrm{xor}}_{A\to B}(A),\\
&B \leftarrow B \boxplus \Phi^{\mathrm{add}}_{A\to B}(A),\\
&A \leftarrow A \oplus \mathrm{B} \lll^{5} \oplus RC[9],\\
&B \leftarrow B \oplus \mathrm{A} \lll^{25},\\
&A \leftarrow A \oplus RC[10], \qquad B \leftarrow B \oplus RC[11].
\end{aligned}
$$

Affine operations use equality/XOR/permutation boxes. Two-variable additions use ADD-Δ or ADD-L. Fixed-public subtractions use the fixed-public oracles above. Injection sites use the packed derivative-image and Walsh-support hooks.

Algorithm 35 AssembleNeoAlzetteSingleCharacteristicMILP

Require: schedule relation, mode $T \in \{\Delta, L\}$

for each affine, XOR, rotation, or zero-fill shift relation **do**
    Add the corresponding exact Boolean/equality box
end for
for each two-variable modular add/sub relation **do**
    Add ADD-Δ/SUB-Δ if $T = \Delta$, or ADD-L/SUB-L if $T = L$
end for
for each fixed-public add/sub relation **do**
    Add the exact fixed-public oracle interface and its weight variable
end for
for each packed C/D injection relation **do**
    Add derivative-image/rank constraints if $T = \Delta$
    Add Walsh-support/Hessian-rank constraints if $T = L$
end for
Minimize the accumulated single-characteristic weight

A.17 MEDCP/MELCC search semantics

A differential characteristic fixes every internal difference and every local transition. Its probability is the product of local probabilities, and its weight is the sum of local weights. A MILP minimization over these weights reports MEDCP-style single-characteristic evidence. A linear characteristic fixes every internal mask and every local signed transition; its absolute-correlation weight is the sum of local weights and gives MELCC-style evidence.

A diagnostic run is a short solver check used to test the model. A time-bounded campaign is a fixed-budget run measured in hours or days. A long-run strict campaign is a frozen-model campaign measured in weeks or months with fixed solver version, fixed parameters, and preserved logs. None of those labels implies optimality. A result is certified only when the solver status or a complete enumeration proves optimality or completeness.

A.18 Endpoint HULL / Forest semantics

Endpoint HULL enumeration and Forest campaigns are distinct from single-characteristic MEDCP/MELCC search. A single characteristic fixes one complete internal trace. A differential endpoint hull aggregates all differential characteristics with the same external endpoint,

$$P_{\mathrm{hull}}(\Delta_{\mathrm{in}}, \Delta_{\mathrm{out}}) = \sum_{\chi \in H(\Delta_{\mathrm{in}}, \Delta_{\mathrm{out}})} 2^{-W(\chi)}.$$

A linear endpoint hull aggregates signed characteristic correlations,

$$C_{\mathrm{hull}}(\Lambda_{\mathrm{in}}, \Lambda_{\mathrm{out}}) = \sum_{\chi \in H(\Lambda_{\mathrm{in}}, \Lambda_{\mathrm{out}})} \sigma(\chi) 2^{-W(\chi)}.$$

Differential aggregation is a nonnegative sum. Linear aggregation is a signed sum and therefore cannot be replaced by path counting or by an unsigned sum of weights.

A.18.1 Forest, trunk, branch, fruit, and seed

A Forest campaign is a layered search over many local endpoints or endpoint sectors. It treats one round of the ARX-box as a directed transition relation on 64-bit difference or mask states. The terminology is:

$$\text{trunk} = s_i, \qquad \text{branch} = \tau_i, \qquad \text{fruit} = \mathrm{out}(\tau_i), \qquad \text{seed} = s_{i+1}.$$

For differential search, a trunk is a fixed input difference state

$$s_i = (\Delta A_i, \Delta B_i) \in \mathbb{W}_{32}^2.$$

For linear search, it is a fixed input mask state

$$s_i = (\Lambda A_i, \Lambda B_i) \in \mathbb{W}_{32}^2.$$

A branch τ_i is not a single internal operation. It is one complete single-characteristic MILP trace from the fixed trunk to a new output state. The output state is the fruit, and if it survives pruning it becomes the seed for a new trunk in the next layer. A path in a Forest is

$$P = (s_0, \tau_0, s_1, \tau_1, \ldots, s_t), \qquad \mathrm{in}(\tau_i) = s_i, \quad s_{i+1} = \mathrm{out}(\tau_i),$$

with cumulative weight

$$W(P) = \sum_i W(\tau_i).$$

A.18.2 CDDT/CLAT scouts in the Forest

The local scout stage uses the Q2 completion oracles of Subsection A.4. In the differential case, the CDDT-style scout completes the missing word of a two-variable add/sub boundary and obtains a local best possible weight. In the linear case, the CLAT-style row or column completion does the same for missing masks. These scouts are exact for the isolated two-variable local relation. They are used to compute lower bounds, branch orderings, and candidate completions before the more expensive global SCIP model is solved. They are not merely heuristic tables, but they are also not global proofs: the full NeoAlzette schedule may invalidate a locally optimal completion because of rotations, packed injection, fixed-public constants, endpoint restrictions, or no-good cuts.

A.18.3 Pruning rules

A branch or trunk can be discarded for several mathematically distinct reasons.

1. **Local CDDT/CLAT scout miss.** The local Q2 lower bound already implies

$$W_{\text{prefix}} + LB_{\text{remaining}} \geq W_{\text{best}}.$$

   The candidate cannot improve the current incumbent and is not sent to the full model.

2. **SCIP infeasibility.** The fixed trunk has no feasible complete characteristic under the current endpoint and no-good constraints.

3. **Zero-state continuation.** The fruit is the all-zero state. This state is not expanded as a useful nonzero differential or mask frontier.

4. **Cycle detection.** The fruit has already appeared in the current path history, so expanding it would loop.

5. **Frontier dominance.** The same fruit state has already been reached with cumulative weight no larger than the new one.

6. **Resource limit.** The campaign stops due to a time, memory, node, or solution cap. This produces a bounded campaign, not a complete endpoint proof.

A bounded endpoint enumeration is a partial sum stopped by a resource limit. A complete endpoint enumeration reaches infeasibility under the endpoint and the accumulated semantic no-good constraints. A Forest campaign is therefore a coverage/search mechanism. It is not automatically an exact endpoint HULL computation, and duplicate semantic characteristics across workers must be de-duplicated before numerical aggregation.

Algorithm 36 RunForestCampaign: layered endpoint/HULL exploration

Require: root source state s_0, total budget T, local model M
Ensure: explored frontier evidence and best found paths

Initialize priority queue $Q \leftarrow \{s_0\}$
$W_{\text{best}} \leftarrow +\infty$; $\mathcal{E} \leftarrow \varnothing$
while Q is nonempty and budget T remains **do**
    Pop a trunk s according to the scheduling priority
    if CDDT/CLAT local scout proves $W_{\text{prefix}} + LB \geq W_{\text{best}}$ **then**
        Record local-scout pruning and continue
    end if
    Solve the single-characteristic SCIP/MILP model with fixed input trunk s
    if the model is infeasible **then**
        Record dead trunk and continue
    end if
    if no incumbent characteristic is decoded **then**
        Record solver failure or resource stop and continue
    end if
    Decode the full branch trace τ and set $s' \leftarrow \text{out}(\tau)$
    if $s' = 0$ **then**
        Prune zero-state continuation and continue
    end if
    if s' is in the current path history **then**
        Prune cycle and continue

21: **end if**
22: **if** s' is reached elsewhere with cumulative weight no larger **then**
23: Prune by frontier dominance and continue
24: **end if**
25: Push s' into Q with updated cumulative weight
26: Add s' to explored frontier set $\mathcal{E}$
27: Optionally run endpoint enumeration for the fixed pair (s, s')
28: **end while**
29: Report explored frontier, termination reasons, and any endpoint partial sums

A.19 Mathematical pseudocode summary

Algorithm 37 EvaluateLocalNeoAlzetteOracle

Require: local relation R, visible differences or masks, public constants when present, mode $T \in \{\Delta, L\}$
1: **if** R is Boolean or affine **then**
2: Return the exact truth-table/equality box
3: **else if** R is two-variable modular addition or subtraction **then**
4: Return the corresponding ADD/SUB oracle and MILP box
5: **else if** R is fixed-public addition or subtraction **then**
6: Return the exact fixed-public count or numerator and exact weight interface
7: **else if** R is a packed C/D injection **then**
8: Return derivative-image/rank constraints if $T = \Delta$
9: Return Walsh-support/Hessian-rank constraints if $T = L$
10: **end if**
Ensure: local feasibility relation and local weight contribution

Algorithm 38 EnumerateEndpointHull

Require: endpoint E, search limits L, local model M
1: $H \leftarrow \varnothing$
2: **loop**
3: Solve the single-characteristic model under endpoint E and no-good constraints for H
4: **if** the model is infeasible **then**
5: **return** $(H, \text{complete})$
6: **end if**
7: **if** a resource limit is reached **then**
8: **return** $(H, \text{bounded})$
9: **end if**
10: Decode the characteristic χ and add it to H
11: Add the semantic no-good constraint blocking χ
12: **end loop**

A.20 Artifact traceability table

This table is a reproduction index only. It is not used as a mathematical proof. All probability spaces, recurrences, MILP boxes, and search semantics are stated before this point. Long artifact paths are printed as full-width wrapped entries so that file names remain readable in the PDF.

Artifact traceability entries

Two-variable differential addition oracle.
Artifact path: `arx_analysis_operators/differential_probability/weight_evaluation.hpp`
Symbol / route: `xdp_add_lm2001_n`
Role: evaluator for Subsection A.8.

Two-variable differential Q2 optimal-gamma completion.
Artifact path: `arx_analysis_operators/differential_probability/optimal_gamma.hpp`
Symbol / route: `find_optimal_gamma` and `find_optimal_gamma_with_weight`
Role: implementation of the non-table Lipmaa–Moriai optimal-γ Q2 completion in Subsection A.4.1.

Fixed-public differential exact count.
Artifact path: `arx_analysis_operators/differential_probability/constant_weight_evaluation.hpp`
Symbol / route: `diff_addconst_exact_count_n`
Role: evaluator for Subsection A.5.

Fixed-public differential exact weight.
Artifact path: same as the fixed-public differential exact-count artifact above.
Symbol / route: `diff_addconst_exact_weight_n`
Role: exact-weight evaluator for the fixed-public differential oracle.

Two-variable linear addition oracle.
Artifact path: `arx_analysis_operators/linear_correlation/weight_evaluation_ccz.hpp`
Symbol / route: CCZ/carry-state evaluator.
Role: evaluator for Subsection A.9.

Fixed-public linear numerator.
Artifact path: `arx_analysis_operators/linear_correlation/constant_weight_evaluation.hpp`
Symbol / route: signed two-state transfer route.
Role: evaluator for Subsection A.6.

BvCorr-FLAT run/event specialization.
Artifact path: `arx_analysis_operators/linear_correlation/constant_weight_evaluation_flat.hpp`
Symbol / route: exact/windowed flat evaluator.
Role: structural acceleration of the fixed-public linear oracle.

Differential NeoAlzette MILP box emitter.
Artifact path: `differential/model/neoalzette_scip_operator_analysis_milp_constraint.hpp`
Symbol / route: row emitter.
Role: implements Subsections A.12 and A.7.

Differential NeoAlzette local oracle.
Artifact path: `differential/model/neoalzette_scip_operator_analysis_oracle.hpp`
Symbol / route: local oracle evaluator.
Role: differential support and weight checking.

Linear NeoAlzette MILP box emitter.
Artifact path: `linear/model/neoalzette_scip_operator_analysis_milp_constraint.hpp`
Symbol / route: row emitter and exact-log interface.
Role: implements Subsections A.13 and A.7.

Linear NeoAlzette local oracle.
Artifact path: `linear/model/neoalzette_scip_operator_analysis_oracle.hpp`
Symbol / route: signed local oracle evaluator.
Role: linear support and weight checking.

Round assembly.
Artifact paths: `differential/model/neoalzette_scip_search_round_function.hpp`
and `linear/model/neoalzette_scip_search_round_function.hpp`
Symbol / route: schedule model builder.
Role: implementation route for Subsection A.16.

Endpoint enumeration.
Artifact paths: `differential/hull_multiple_thread_runner.cpp`
and `linear/hull_multiple_thread_runner.cpp`
Symbol / route: endpoint/Forest runner.
Role: implementation route for Subsection A.18.

B Companion OPC Block-Cipher Artifact in the Repository

The repository contains two major algorithm families: LOPC and OPC. LOPC is the counter-indexed 128-bit core specified and analyzed in this paper. OPC is a companion OaldresPuzzle_Cryptic block-cipher artifact maintained in the same repository. This appendix records that repository boundary so that readers do not confuse repository coverage with the security claims of the LOPC paper.

OPC is not analyzed by the Component-A/B/C security arguments above. It does not inherit LOPC's MEDCP/MELCC component evidence, the XCR round-material boundary, or the LOPC counter-indexed API claims. The OPC artifact was designed with large-key and post-quantum-motivation concerns in mind, but this appendix does not provide a quantum-security proof, a brute-force-resistance theorem, or a full cryptanalysis of OPC. Such claims require a separate specification and security evaluation.

C Historical BPHT Note and Current Status

Older drafts described a Binary Pseudo-Hadamard Transform layer. The current C++ core described in Algorithms 4 and 5 instead uses the keyed switching layer above after the diagonal NeoAlzette calls. Any future reintroduction of BPHT should be treated as a new round function and should rerun the NeoAlzette/LOPC differential and linear analysis pipeline.

D Data Tables and Benchmark Tables

The tables below are artifact-boundary and benchmark-planning tables. They are not security evidence and should not be read as proofs of AEAD, PRP, PRG, MEDCP, MELCC, or endpoint-HULL properties.

Table 9: Authentication and mode boundary for selected artifacts.

Artifact or primitive	Built-in authentication	External MAC/AEAD compatible	Reference
Ascon AEAD	Yes	Not applicable	[TMC+25]
ChaCha20 with Poly1305	With Poly1305 construction	Yes	[NL18]
LOPC core	No	Yes, via an external MAC/AEAD wrapper	this paper
OPC companion artifact	Not claimed here	Out of scope here	Appendix B

Table 10: Internal artifact roles in the LOPC repository.

Artifact	Role in this paper	Security status in this paper
LOPC	Main 128-bit counter-indexed block core	Specified; component evidence only
NeoAlzette	Component-A ARX-box	MILP/Oracle backbone; MEDCP/MELCC boundary
XCR	Round-material generator	Component oracles; full PRG not claimed
Key Switch Layer	Reversible switching/tweak layer	Invertibility and affine/permutation propagation
OPC	Companion block-cipher artifact	Repository boundary only

E MILP Campaign Artifact Summaries

This appendix summarizes the solver-log artifacts that support the one-round NeoAlzette incumbent table in Section 33.8. The complete logs are included in the artifact bundle as `diff_r1_best_cmd_43200s_YYYYMMDD_HHMMSS.log` and `linear_r1_best_cmd_43200s_YYYYMMDD_HHMMSS.log`. They are not endpoint-HULL certificates and they are not optimality certificates; they are preserved solver-campaign artifacts for the reported time-bounded incumbents.

Table 11: Solver-log summary for the current NeoAlzette one-round campaigns.

Mode	**Log summary**
Differential	Log file: `diff_r1_best_cmd_43200s_YYYYMMDD_HHMMSS.log`. Solver status: time limit reached; 43200 s. Main incumbent: weight 80.4312923455; $2^{-80.4312923455}$. Preserved evidence: SCIP bounds, endpoint differences, checkpoint trace, local weight trace, result/weight-trace JSON references.
Linear	Log file: `linear_r1_best_cmd_43200s_YYYYMMDD_HHMMSS.log`. Solver status: time limit reached; 43200 s. Main incumbent: weight 16.6153859522; $2^{-16.6153859522}$. Preserved evidence: SCIP bounds, endpoint masks, oracle validity flags, exact evaluated total, local weight trace, result/weight-trace JSON references.

F Implementation Code Listings

This appendix is generated from the minted-based source file `appendix_E_code_listings_source.tex`. The included pages print the Python/C++ implementation listings as implementation artifacts. The normative mathematical specification remains in Parts III–V, and the NeoAlzette oracle/MILP derivations remain in Appendix A. The PDF build for this listing requires `-shell-escape` because minted uses Pygments.

1 Implementation Code Listings

This appendix prints the implementation files that define the round-constant generator, the XCR component, and the LOPC wrapper. The listings are included with the `minted` package so that the paper records the executable engineering object together with the mathematical model. These listings are implementation artifacts. The normative mathematical specification is in Parts III–V, and the main paper Appendix A gives the mathematical oracle/MILP backbone.

For compile stability, long source files are split into consecutive minted chunks. The chunks are contiguous and cover the full source file; no code lines are intentionally omitted.

1.1 GenerateAndDisplay_XorConstantRotation_RoundConstant.py

This file is split into 2 contiguous minted chunk(s).

Lines 1–120.

```
import numpy as np
from mpmath import mp

# Set the desired decimal precision
# Precision (must stay 100)
mp.dps = 100

def plastic_number():
    """
    Compute the Plastic number (real root of x^3 = x + 1)
    with dps digits of precision.
    Returns the high-precision value as an mp.mpf.
    """
    # Use the closed-form algebraic expression
    num = ( (9 - mp.sqrt(69))**(1/mp.mpf(3)) +
            (9 + mp.sqrt(69))**(1/mp.mpf(3)) )
    den = (2**(1/mp.mpf(3))) * (3**(2/mp.mpf(3)))
    return num / den

# Constants (high precision)
e = mp.e
pi = mp.pi
phi = (1 + mp.sqrt(5)) / 2
sqrt_2 = mp.sqrt(2)
sqrt_3 = mp.sqrt(3)
gamma = mp.euler
delta = mp.mpf("4.669201609102990671853203820466201617258185577475768632745651343004134313430041341334330211313473137386897440239348013 8")
rho = plastic_number()

print(str(e) + '\n')
print(str(pi) + '\n')
print(str(phi) + '\n')
print(str(sqrt_2) + '\n')
print(str(sqrt_3) + '\n')
print(str(gamma) + '\n')
print(str(delta) + '\n')
print(str(rho) + '\n')

def frac(x):
    return x - mp.floor(x)

# V2 polynomial Weyl with all high-precision constants
def f(n):
    n = mp.mpf(n)
    x = mp.mpf("0")
    x = frac(x + e      *  n      )  # n^1
    x = frac(x + pi     *  n**2   )  # n^2
    x = frac(x + phi    *  n**3   )  # n^3
    x = frac(x + sqrt_2 *  n**4   )  # n^4
    x = frac(x + sqrt_3 *  n**5   )  # n^5
    x = frac(x + gamma  *  n**6   )  # n^6
    x = frac(x + delta  *  n**7   )  # n^7
    x = frac(x + rho    *  n**8   )  # n^8
    return x

# Old Version with all high-precision constants
'''
e = mp.e
pi = mp.pi
phi = (1 + mp.sqrt(5)) / 2
sqrt_2 = mp.sqrt(2)
sqrt_3 = mp.sqrt(3)
gamma = mp.euler  # Eulerâ€™Mascheroni constant
delta = mp.mpf('4.669201609102990671853203820466')  # Feigenbaum constant delta
rho = mp.cbrt((9 + mp.sqrt(69)) / 18) + mp.cbrt((9 - mp.sqrt(69)) / 18) # Plastic number

print(str(e) + '\n')
print(str(pi) + '\n')
print(str(phi) + '\n')
print(str(sqrt_2) + '\n')
print(str(sqrt_3) + '\n')
print(str(gamma) + '\n')
print(str(delta) + '\n')
print(str(rho) + '\n')

def f(x):
    x = mp.mpf(x)
    term1 = (e ** x - mp.cos(pi * x))
    term2 = (phi * x ** 2 - phi * x - 1)
    term3 = (x * sqrt_2 - mp.floor(x * sqrt_2))
    term4 = (x * sqrt_3 - mp.floor(x * sqrt_3))
    term5 = mp.log(1 + x)
    term6 = (x * delta - mp.floor(x * delta))
    term7 = (x * rho - mp.floor(x * rho))

    return term1 * term2 * term3 * term4 * term5 * term6 * term7
'''

assert mp.dps == 100, "mp.dps MUST be 100 decimal (N * 10^{-100}) for bit-exact reproduction"

def print_console():
    round = 1

    binary_string = ""

    for index in range(150):
        # Calculate the result for a given input value
        result = f(round)

        print("Round: ", index)

        # Print the decimal result
        print("Decimal number:", result)

        # Convert the fractional part to binary and print it
        fractional_part = result - mp.floor(result)
        binary_fractional_part = format(int(fractional_part * 2**128), 'b')  # Using 128 bits of precision for the binary representation
        # print("Binary representation of fractional part:", binary_fractional_part)

        # Convert the fractional part to hexadecimal and print it
        hexadecimal_fractional_part = format(int(fractional_part * 2**128), 'x')
        print("Hexadecimal representation of fractional part:", hexadecimal_fractional_part)

        # Print the integer part
        integer_part = int(result)
        print("Integer part:", integer_part)

        round += 1
        binary_string += (binary_fractional_part)
```

Lines 121–133.

```
    print("Binary String: ", binary_string)
    # Convert the binary string to an integer
    integer_value = int(binary_string, 2)

    # Convert the integer to a hexadecimal string
    hexadecimal_string = format(integer_value, 'x')

    print("Hexadecimal representation:", hexadecimal_string)

    grouped_hexadecimal_string = ','.join([hexadecimal_string[i:i+16] for i in range(0, len(hexadecimal_string), 16)])
    print("Grouped Hexadecimal representation:", grouped_hexadecimal_string)

print_console()
```

1.2 XorConstantRotation.h

This file is split into 3 contiguous minted chunk(s).

Lines 1–120.

```
#pragma once

#include <array>
#include <cstddef>
#include <cstdint>
#include <bit>  // std::rotl

namespace TwilightDreamOfMagical::CustomSecurity::CSPRNG
{
  // ---------------------------------------------------------------------------
  // XCR (Xor-Constant-Rotation)
  // ---------------------------------------------------------------------------
  // Experimental / analysis-friendly stateful generator core.
  //
  // Design goals:
  // - Experimental: this is NOT claimed to be a proven CSPRNG.
  // - Hybrid-analysis friendly: remove native add/sub from the core permutation and keep nonlinear operations explicit.
  // - Engineering friendly: integer-only, branch-free core, no heavy runtime math.
  // - Constant discipline: runtime mixing constants come only from
  //   `XCR_ROUND_CONSTANTS[]` (plus the fixed default seed constant used by ctor).
  //
  // State model:
  // - Internal state is 256-bit: (w, x, y, z), each lane is uint64_t.
  // - A public schedule variable `counter` is also maintained.
  //
  // External inputs:
  // - `seed` enters only through ctor / Seed(), then is absorbed by StateInitialize().
  // - `number_once` is the per-call external input of the permutation path.
  //
  // Important API semantics (current implementation):
  // - `GenerateSubKey128(number_once)`
  //      Stateful production call. It mutates (w,x,y,z), returns {x ^ y, z ^ w},
  //      and DOES advance `counter += COUNTER_STEP` via StateIteration().
  // - `operator()(number_once)`
  //      Same public behavior as GenerateSubKey128() in the current implementation:
  //      one permutation, one output extraction, one explicit counter advance.
  // - `StateIteration(number_once)`
  //      Private helper path used internally, and also the common stepping backend
  //      for the current public production calls.
  //
  // Therefore:
  // - Public production calls evolve through BOTH:
  //      (1) internal state mutation,
  //      (2) explicit public-counter progression.
  // - The warm-up path also uses the same stepping rule, but after warm-up
  //   `counter` is reset to the fixed public start value.
  //
  // Permutation-level Hybrid8-Shadow constraint:
  // - Native modular add/sub operations are intentionally removed from the core step.
  // - Public period-4 H-round uses alternating Hybrid8 byte substitutions.
  // - The remaining S-rounds use bounded shadow carry/borrow word operations.
  // - Everything else in the round is XOR / ROTL / public table lookup.
  //
  // Seed separation rule:
  // - The seed is consumed only during StateInitialize().
  // - Production calls do not directly re-inject the seed.
  // ---------------------------------------------------------------------------

  class XorConstantRotation
  {

  public:
    /*
      Seed = 20260221
      Found in tries = 493
      Prime (hex) = 0xadb136136669d153
      HW total = 32
      HW low32/high32 = 16 / 16
      max_run low/high = 3 / 4
    */
    static constexpr std::uint64_t XCR_CSPRNG_DEFAULT_INITIALIZE_CONSTANT = 0xADB136136669D153;
    // ROUND_CONSTANT.size() == 300
    // gcd(COUNTER_STEP, ROUND_CONSTANT_SIZE) = 1, popcount(COUNTER_STEP) = 32
    static constexpr std::uint64_t COUNTER_STEP = 0xC8522A96E53AF749;

    struct GeneratedSubKey128
    {
      uint64_t a, b;

      GeneratedSubKey128() = default;
      GeneratedSubKey128(std::uint64_t left, std::uint64_t right)
        : a(left), b(right) {}

      std::uint64_t& operator[](std::size_t index)
      {
        return (index == 0) ? a : b;
      }

      const std::uint64_t& operator[](std::size_t index) const
      {
        return (index == 0) ? a : b;
      }

      GeneratedSubKey128 operator^(const GeneratedSubKey128& other) const
      {
        return { a ^ other.a, b ^ other.b };
      }

      GeneratedSubKey128& operator^=(const GeneratedSubKey128& other)
      {
        a ^= other.a;
        b ^= other.b;
        return *this;
      }

      bool operator==(const GeneratedSubKey128& other) const = default;

      std::uint64_t GetBit(std::size_t bit_index) const
      {
        if (bit_index < 64)
        {
          return static_cast<std::uint64_t>((a >> bit_index) & 1ULL);
        }
        return static_cast<std::uint64_t>((b >> (bit_index - 64)) & 1ULL);
      }
    };

  private:
    void                PermutationARX( const std::uint64_t number_once );
    GeneratedSubKey128 StateIteration( std::uint64_t number_once );
```

Lines 121–240.

```
  void         StateInitialize();

public:

  XorConstantRotation()
  : w(XCR_CSPRNG_DEFAULT_INITIALIZE_CONSTANT), x(0), y(0), z(0), counter(COUNTER_STEP)
  {
    StateInitialize();
  }

  XorConstantRotation(std::uint64_t seed)
    : w(seed), x(0), y(0), z(0), counter(COUNTER_STEP)
  {
    StateInitialize();
  }

  void Seed(std::uint64_t seed)
  {

    // Reset lanes for explicit key path
    x = 0;
    y = 0;
    z = 0;
    // key slot
    w = seed;
    // Reset to fixed public start value for production phase
    counter = COUNTER_STEP;
    StateInitialize();
  }

  GeneratedSubKey128 operator()( std::uint64_t number_once );
  GeneratedSubKey128 GenerateSubKey128( std::uint64_t number_once );

  // 256-bit state
  std::uint64_t w = 0;
  std::uint64_t x = 0;
  std::uint64_t y = 0;
  std::uint64_t z = 0;

  // public counter
  std::uint64_t counter = 0;

  // NOTE:
  // - First 4 constants are "manually mixed anchors"
  // - The rest are generated by your high-order continuous function discretization.
  // - This code never introduces any other constants.
  // GenerateAndDisplay_XorConstantRotation_RoundConstant.py generate this
  static constexpr std::array<std::uint64_t, 300> XCR_ROUND_CONSTANTS =
  {
    // [0..3] manual anchors: Fibonacci-bit concat, pi/phi-style, etc.
    0x01B70C8E97AD5F98ULL, 0x243F6A8885A308D3ULL, 0x9E3779B97F4A7C15ULL, 0xB7E151628AED2A6AULL,

    // [4..299] generated stream (truncated here in this snippet).
    //x ∈ [1, 150]
    // V2 polynomial Weyl with all high-precision constants
    // f(x) = frac( frac( frac( frac( frac( frac( frac(0 + e·x)
    //                + π·x²) + φ·x³) + √2·x⁴) + √3·x⁵) + γ·x⁶) + δ·x⁷) + ρ·x⁸ )
    //
    // where frac(t) = t - ⌊t⌋   x ∈ ℕ (e.g., [1, 150])
    0xc7fed6d75df59ae9ULL, 0x20b23e1962e49836ULL, 0xe91e4940fe913bb5ULL, 0x97de4bff5dfc30baULL,
    0xea49951290ea2540ULL, 0xb3383d9dab30ae94ULL, 0x5621f313742832d0ULL, 0x995116b66478bba4ULL,
    0x01044133 8f759db8ULL, 0x536398d10308029dULL, 0x2894ed6378c5fb69ULL, 0xad85fa88c56d7b09ULL,
    0xb86c2492a5407da3ULL, 0x1fd735150a7fc7f8ULL, 0x02c080 1e8d2ad5c4ULL, 0x345b92dda0378c7bULL,
    0xc858e896facbd0ecULL, 0x0d7e968ff47082a5ULL, 0x4dc82f0a88eaba49ULL, 0xdead923252d0b51eULL,
    0x02236f27027eec7bULL, 0x8bfd5a2cd7d0dfe1ULL, 0xb2d91055e1680ec7ULL, 0x447f69d7bbb2de96ULL,
    0xf24675a6731d39b1ULL, 0x74e6924e74f3f522ULL, 0x273d55b4f2f77942ULL, 0x4ec79e443fe532aeULL,
    0x29884015291e47cdULL, 0xf535ca62c5946508ULL, 0x194d875b934005 4eULL, 0x7dff309c505e61d3ULL,
    0x33f089308e7d76f6ULL, 0x66d8aca59455423dULL, 0x5443a4614fb8b79aULL, 0x8860a0331815a37fULL,
    0x94490b5c348488e3ULL, 0x2aca9822e5516650ULL, 0x733050d3dd13b717ULL, 0xf333ffb03119bcfbULL,
    0x8994c64cc9ce655bULL, 0xd1c78bb7fbf62cbcULL, 0x9ce1b356d7fd08eaULL, 0x529095a30fb494bcULL,
    0x8944133827f7750fULL, 0x8e244af81eea7291ULL, 0x8f678cfcb2d509dULL, 0x9b3c82a2ec6bb9a6ULL,
    0x7cec3e3bddc61b85ULL, 0x2ea6563b009e077eULL, 0x12a53272075261 9ULL, 0xc4cb57d77bef7619ULL,
    0xe7c7edf59a172e38ULL, 0x356b6d4bbb9e2252ULL, 0x606a34fcaf87622bULL, 0x8614cb5cdabd7b2eULL,
    0x6601482c230d6a6bULL, 0x7c5670b353a5474aULL, 0xdd632cfec4c04a7eULL, 0x120f463deafa672bULL,
    0x5c7ad57132be6d99ULL, 0x66d84709d4c2c125ULL, 0xdb7713d125404702ULL, 0xda87b7f935c26251ULL,
    0x0283a7e70abef3fcULL, 0x00a2056879c6d403ULL, 0xb581bda3f9472b77ULL, 0x546b6f54c7e6d880ULL,
    0xb95d731a19e3a5e5ULL, 0xfc7b4fb032470fd8ULL, 0x20058bcd42aa267fULL, 0x788b5830d262515aULL,
    0xb0d85d6c080678f74ULL, 0xea8ed898d6ec9081ULL, 0x67d60e41bb8008c2ULL, 0x6773f799e997275ULL,
    0xd620fc33328703f4ULL, 0xdb5d284693d0c501ULL, 0xccd1485ffee19ddcULL, 0xba423bf2cfcbf84dULL,
    0x10038a54845bc465ULL, 0x72f9bcb93f278861ULL, 0xcf5610e8d98eea9fULL, 0xa8c611ff76712442ULL,
    0x8f5107f7835a2743ULL, 0xe582e5371d41f3a6ULL, 0x2a6a4181 7ad64a1fULL, 0xe1284d87d0461dd8ULL,
    0x5cc60669d6133c09ULL, 0x5a36fec5cd11724eULL, 0x645ed91ceffb0fe1ULL, 0x29480de78f4a1f66ULL,
    0x0ad6643fcc72994bULL, 0x5ef91e9d64c388efULL, 0x0934f85bf11c3a26ULL, 0x924252b668e465e4ULL,
    0xf2c4e7935494be68ULL, 0xf5303d0fdb1cef2aULL, 0x6284e1f5efede4a2ULL, 0x5d51abe2cc906eb3ULL,
    0x07d0a125283799baULL, 0xf1857876b708c71dULL, 0xca7edcca5a57f378ULL, 0x1606487fb80ba32bULL,
    0xc880e89d05a25051ULL, 0x9f11b85e1b76708dULL, 0x20e033151e08c2c9ULL, 0x080446531ed145a5ULL,
    0x89d051c6ab320d8aULL, 0x09cf59743a58763fULL, 0xafd20c8650719fc6ULL, 0x8c9827c610851d36ULL,
    0xe6823b3ee36b03ecULL, 0x9b88e73002cfaf2fULL, 0x3259d6e322745dc5ULL, 0xd438035d0061e914ULL,
    0x7ec7b283d4e5cf83ULL, 0x3bc07bbe3d5de00eULL, 0x03b76ca9748b0829ULL, 0x996d690fc620872bULL,
    0x84fd544e588c6dc5ULL, 0x62f56a662af4c27aULL, 0xecbf3492b48777beULL, 0xeb022ee345562417ULL,
    0xd64f319faf3877bfULL, 0x10f6afc24765674eULL, 0x462fb541297e4e47ULL, 0x5830e8fb6290ddc5ULL,
    0x72177a6ca01d4160ULL, 0xc9ec7db4c500d928ULL, 0xf07669f3bd4bdbf1ULL, 0x10bca5be25e4c519ULL,
    0xea9957226b48cabeULL, 0x32bdec964180aa7cULL, 0x73060ec748740ddfULL, 0x2f3740a4b808f6f3ULL,
    0x5b5f24c00870223cULL, 0x37834e526eb85528ULL, 0x342b94c1d614e690ULL, 0x0c8f744af72273a8ULL,
    0x25b1fe35877a6681ULL, 0x96ca44044 8a0dd4aULL, 0x83c0f58d0d052779ULL, 0x978ace71c34fcb6bULL,
    0xe85605ca4aceb32cULL, 0x2e4ed59fb92b0bcfULL, 0xba1f41b35b1eea12ULL, 0xb44b1352b69058cdULL,
    0xba3ceca3e27d32f9ULL, 0x5a7441c6528131e9ULL, 0xedeb8055c017c2d7ULL, 0x147efc6cb7dac993ULL,
    0x3690bf3b60cfbf1fULL, 0xfeda458d3b67d4deULL, 0xd9bcb6b2fb353ff0ULL, 0x906fe8490f900801ULL,
    0x4b71ed710366b9f0ULL, 0xc969ef0a4e9c1107ULL, 0x90572c255c96834fULL, 0x5f3eb2edbe4ac0b3ULL,
    0x634f3a1b98d9933dULL, 0x6f0f361ee182539eULL, 0xebea4c97f7c64d8aULL, 0x1cd3159ca76b6d85ULL,
    0xcda3a82a5d66a4a5ULL, 0x85339303bb54b830ULL, 0x349f8b78600a084cULL, 0x1c3f55fe2af85f36ULL,
    0x02b1d6d131c2ca0dULL, 0x8657320787dcff9ULL, 0x275a0b26c80789 06ULL, 0xcc2fd8020350f07ULL,
    0xe475319860cbee73ULL, 0xc9c6e475780bba5cULL, 0xf7ddb43214f431b2ULL, 0xa60fd32705228cb7ULL,
    0xc2d119cc67a24dbaULL, 0xa590929d94e6373eULL, 0x6410210895867 2a8ULL, 0xff38b0e1252eedacULL,
    0x551540277911dfc8ULL, 0x7e82148109cce0efULL, 0xd0209d9df9eaaed b44ULL, 0x4284bc2794b543a8ULL,
    0xd6f053a12f2ad5beULL, 0x666157855db0fd29ULL, 0x2c8364d46812dd0fULL, 0x3a2bf7697fe971c9ULL,
    0xb217b28f07654afdULL, 0x0964d6a25e0c0674ULL, 0x6b78d335c4b26354ULL, 0xb3a0d240490834bbULL,
    0x7cbdf7e25cf5e2e8ULL, 0x66096b450186248dULL, 0x3f957dc892bab775ULL, 0x204be476cb0e6204ULL,
    0x2e93270c352877a0ULL, 0xcf3c54fcf7e25340ULL, 0xed466b3a6aa99784ULL, 0x687ee87389d3ce44ULL,
    0xfce153fc6f3f0a12ULL, 0x538554fa09d45b61ULL, 0x0eb22bdbcbd2e657ULL, 0x99eef7b2dd6af18ULL,
    0x9820d3f84a539881ULL, 0xf1ca001532e7c261ULL, 0x7fed339969bd2c3cULL, 0xbcf5b28dc2c218c9ULL,
    0xcfb439da5c0aa91aULL, 0x07c8c143feb901ceULL, 0xc9bfebacc4962fe9ULL, 0xa578a3e9fbb8b2c5ULL,
    0x0b70dd4a8d1ff46cULL, 0xd50d0ca3e67516a6ULL, 0x1b0d9ab302075e3aULL, 0xcc65e117fd8e4c5bULL,
    0xf02b8f19187dfd52ULL, 0x14180b374125aa9fULL, 0x953c0c06d6a98a04ULL, 0x4057b8bb6c200930ULL,
    0xdc2f04b2766 1c6f1ULL, 0xa1df060a256bb02f7ULL, 0x010cf06e893f27a5ULL, 0xd1a080d1450084 55ULL,
    0x229ff6b6e7640df1ULL, 0xd24b66100d9479f5ULL, 0xceb0b1f750e7ceb0ULL, 0xe2e6dbece5714183ULL,
    0xf67045ae46918afdULL, 0x7a872dea312a9fd8ULL, 0x94297bdd53ab20f8ULL, 0x4dae2ef810bbe342ULL,
    0xc1054ff43ad21fc4ULL, 0x26ebf6433551b4b2ULL, 0x11e3a876d6a1351fULL, 0x295d26f5462909acULL,
    0x402c466e92a38ea1ULL, 0x1cc8e5c9da28663 2ULL, 0xe43fd84c98af9eaaULL, 0x70c12a97ae6d8a4eULL,
    0x7b68298191a178b8ULL, 0xfe27c41b52815151ULL, 0xb7cb37f9a5786ee3ULL, 0x8b7ec6586bfaf9e4ULL,
```

Lines 241–258.

```
    0x16ea247a1b8eff1eULL, 0x7a08eee72ce4dc99ULL, 0xf8563c4f27161a69ULL, 0x67776334a28721a1ULL,
    0x143997e86de1f9b6ULL, 0x1c00f33ca5d83e43ULL, 0x0ecbf99e2f8da830ULL, 0xbcb6a5b4641902aeULL,
    0x5add72af71ec10daULL, 0x3cb671fb55910acbULL, 0xb556cbe0c235c663ULL, 0xbf7582d30c7b709bULL,
    0xe6419e49d98970dbULL, 0x94c02a4e57e78417ULL, 0x8a90e51246212fc6ULL, 0x08e74b091203eba1ULL,
    0xbc8e2b48be85990dULL, 0x60984e2a9011efeaULL, 0x7f13660985641607ULL, 0xa70628a5966dbe6aULL,
    0x11668a2501213 76fULL, 0x98e2fb9ba6830b7dULL, 0xfbcb881bd8e04362ULL, 0x818b615d377d79e8ULL,
    0xd363eaf614a598aaULL, 0xb98ef05d4f0fa7a0ULL, 0xa1e43deb0e801ee6ULL, 0x09e89d231ce3c300ULL,
    0x32b97dcc525c6c1cULL, 0xe783951df5595388ULL, 0xd19bb2b7ec16093aULL, 0x66c4177f7c1d79bfULL,
    0x7ac7eef4150fb82cULL, 0xad5e164ab4c0d474ULL, 0x85b10215b554d1fbULL, 0xe075a3032fa906ddULL,
```

```
        0xa80eb743bc799201ULL, 0x0f01944c84b85959ULL, 0x757529145922cd25ULL, 0x19e0eadc8379ba3cULL,
        0x1e72937f49943a41ULL, 0x136c3db980bb2ddeULL, 0x2900ffffebe647ddULL, 0xc292f1aed24bb838ULL,
        0x6dc406a8d9a846a8ULL, 0x9d4d0d3846ae9fcbULL, 0xb1b5b679ec677686ULL, 0x2da508234e18b51bULL,
        0xb67732180c341acaULL, 0x4ffb9be0b5ee605aULL, 0x15d5f9e38f2044cbULL, 0x352c030d11302197ULL
    };

    static constexpr std::size_t ROUND_CONSTANT_SIZE = XCR_ROUND_CONSTANTS.size();
  };
} // namespace TwilightDreamOfMagical::CustomSecurity::CSPRNG
```

1.3 `XorConstantRotation.cpp`

This file is split into 14 contiguous minted chunk(s).

Lines 1–90.

```cpp
#include "XorConstantRotation.h"
#include <numeric>

namespace TwilightDreamOfMagical::CustomSecurity::CSPRNG
{
  /*
    import random

    SEED = 20260221
    rng = random.Random(SEED)

    # -------- Miller-Rabin (deterministic for 64-bit) --------
    def is_prime(n: int) -> bool:
      if n < 2:
        return False

      small_primes = [2,3,5,7,11,13,17,19,23,29,31,37]
      for p in small_primes:
        if n % p == 0:
          return n == p

      d = n - 1
      s = 0
      while (d & 1) == 0:
        d >>= 1
        s += 1

      bases = [2, 325, 9375, 28178, 450775, 9780504, 1795265022]
      for a in bases:
        if a % n == 0:
          return True
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
          continue
        for _ in range(s - 1):
          x = (x * x) % n
          if x == n - 1:
            break
        else:
          return False
      return True


    def rotl32(x: int, r: int) -> int:
      r &= 31
      return ((x << r) | (x >> (32 - r))) & 0xFFFFFFFF


    def reverse32(x: int) -> int:
      # bit-reversal of 32-bit
      x &= 0xFFFFFFFF
      x = ((x & 0x55555555) << 1) | ((x >> 1) & 0x55555555)
      x = ((x & 0x33333333) << 2) | ((x >> 2) & 0x33333333)
      x = ((x & 0x0F0F0F0F) << 4) | ((x >> 4) & 0x0F0F0F0F)
      x = ((x & 0x00FF00FF) << 8) | ((x >> 8) & 0x00FF00FF)
      x = ((x & 0x0000FFFF) << 16) | ((x >> 16) & 0x0000FFFF)
      return x & 0xFFFFFFFF


    def popcount32(x: int) -> int:
      return (x & 0xFFFFFFFF).bit_count()


    def hamming_distance32(a: int, b: int) -> int:
      return ((a ^ b) & 0xFFFFFFFF).bit_count()


    def max_run_length32(x: int) -> int:
      # longest run of equal bits in 32-bit word
      x &= 0xFFFFFFFF
      max_run = 1
      cur_run = 1
      prev = x & 1
      for i in range(1, 32):
        bit = (x >> i) & 1
        if bit == prev:
          cur_run += 1
        else:
          if cur_run > max_run:
            max_run = cur_run
          cur_run = 1
          prev = bit
      if cur_run > max_run:
        max_run = cur_run
      return max_run


    def nibble_uniformity32(x: int, nib_min=1, nib_max=3, max_dist=6) -> bool:
      # 8 nibbles; ideal=2 ones per nibble
      dist = 0
```

Lines 91–180.

```cpp
      for i in range(8):
        nib = (x >> (4*i)) & 0xF
        pc = nib.bit_count()
        if pc < nib_min or pc > nib_max:
          return False
        dist += abs(pc - 2)
      return dist <= max_dist


    def byte_uniformity32(x: int, byte_min=2, byte_max=6, max_dist=4) -> bool:
      # 4 bytes; ideal=4 ones per byte
      dist = 0
      for i in range(4):
        b = (x >> (8*i)) & 0xFF
        pc = b.bit_count()
        if pc < byte_min or pc > byte_max:
          return False
        dist += abs(pc - 4)
      return dist <= max_dist


    def sliding_window_checks32(x: int,
                  max_dev8=3,   # 8-bit window ideal=4, allow [1..7] default
                  max_dev16=5   # 16-bit window ideal=8, allow [3..13] default
                  ) -> bool:
      x &= 0xFFFFFFFF

      # 8-bit windows (25 windows)
      for i in range(0, 32 - 8 + 1):
```

```
    w = (x >> i) & 0xFF
    pc = w.bit_count()
    if abs(pc - 4) > max_dev8:
      return False

  # 16-bit windows (17 windows)
  for i in range(0, 32 - 16 + 1):
    w = (x >> i) & 0xFFFF
    pc = w.bit_count()
    if abs(pc - 8) > max_dev16:
      return False

  return True


def anti_self_similarity32(x: int,
                min_rot_hd=10,
                min_rev_hd=12) -> bool:
  # forbid too-close rotations (structure repeats)
  for r in (1, 3, 5, 7, 11, 13, 17):
    if hamming_distance32(x, rotl32(x, r)) < min_rot_hd:
      return False

  # forbid too-close bit-reversal similarity
  if hamming_distance32(x, reverse32(x)) < min_rev_hd:
    return False

  return True


def small_mod_bias_filter(n: int) -> bool:
  # cheap extra: avoid tiny congruence patterns that show up suspiciously often
  # (this does NOT prove RK security; it's just a cheap "no obvious stink" filter)
  bad_mods = [3, 5, 7, 11, 13, 17, 19]
  for m in bad_mods:
    r = n % m
    # avoid r=1 or r=m-1 simultaneously across many mods? too strict if stacked,
    # but we can still avoid a few obvious residues.
    if r == 0:
      return False
  return True


def strict_uniform_half32(x: int) -> bool:
  # you can tighten these numbers further, but default already brutal
  if popcount32(x) != 16:
    return False
  if max_run_length32(x) > 7:
    return False
  if not nibble_uniformity32(x, nib_min=1, nib_max=3, max_dist=6):
    return False
  if not byte_uniformity32(x, byte_min=2, byte_max=6, max_dist=4):
    return False
  if not sliding_window_checks32(x, max_dev8=3, max_dev16=5):
    return False
  if not anti_self_similarity32(x, min_rot_hd=10, min_rev_hd=12):
    return False
  return True


def random_candidate_exact_16_16() -> int:
```

Lines 181–270.

```
  # generate exactly: low32 has 16 ones, high32 has 16 ones (no bit fixing!)
  low_positions  = rng.sample(range(0, 32), 16)
  high_positions = rng.sample(range(32, 64), 16)
  n = 0
  for pos in low_positions:
    n |= 1 << pos
  for pos in high_positions:
    n |= 1 << pos
  return n


def find_extreme_constant64(max_tries: int = 200_000_000):
  for tries in range(1, max_tries + 1):
    n = random_candidate_exact_16_16()

    # prime implies odd (>2), so even candidates are rejected (not modified)
    if (n & 1) == 0:
      continue

    # small mod filter first (cheap), then primality (expensive)
    if not small_mod_bias_filter(n):
      continue
    if not is_prime(n):
      continue

    low32  = n & 0xFFFFFFFF
    high32 = (n >> 32) & 0xFFFFFFFF

    if not strict_uniform_half32(low32):
      continue
    if not strict_uniform_half32(high32):
      continue

    return n, tries

  raise RuntimeError("Not found within max_tries")


if __name__ == "__main__":
  value, tries = find_extreme_constant64()
  low32 = value & 0xFFFFFFFF
  high32 = (value >> 32) & 0xFFFFFFFF

  print("Seed =", SEED)
  print("Found in tries =", tries)
  print("Prime (hex) =", hex(value))
  print("HW total =", value.bit_count())
  print("HW low32/high32 =", low32.bit_count(), "/", high32.bit_count())
  print("max_run low/high =", max_run_length32(low32), "/", max_run_length32(high32))
*/

/*

from math import gcd


# ============================================================
#  Deterministic Miller-Rabin primality test for n < 2^64
# ============================================================

SMALL_PRIMES = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37)
MR_BASES_64 = (2, 325, 9375, 28178, 450775, 9780504, 1795265022)


def is_prime64(n: int) -> bool:
  """
  Deterministic Miller-Rabin primality test for unsigned 64-bit integers.
  Valid for 0 <= n < 2^64.
```

```
    """
    if n < 2:
      return False

    if n >= (1 << 64):
      raise ValueError("is_prime64 only supports n < 2^64")

    for p in SMALL_PRIMES:
      if n % p == 0:
        return n == p

    d = n - 1
    s = 0
    while (d & 1) == 0:
      d >>= 1
      s += 1

    for a in MR_BASES_64:
      a %= n
      if a == 0:
        continue
```

Lines 271–360.

```
      x = pow(a, d, n)
      if x == 1 or x == n - 1:
        continue

      for _ in range(s - 1):
        x = (x * x) % n
        if x == n - 1:
          break
      else:
        return False

    return True


# ============================================================
#  Bit utility functions
# ============================================================

def popcount32(x: int) -> int:
    return (x & 0xFFFFFFFF).bit_count()


def rotl32(x: int, r: int) -> int:
    x &= 0xFFFFFFFF
    return ((x << r) | (x >> (32 - r))) & 0xFFFFFFFF


def bit_reverse32(x: int) -> int:
    x &= 0xFFFFFFFF
    y = 0

    for _ in range(32):
        y = (y << 1) | (x & 1)
        x >>= 1

    return y


def hamming32(a: int, b: int) -> int:
    return ((a ^ b) & 0xFFFFFFFF).bit_count()


def max_run_length(x: int, bits: int) -> int:
    """
    Return the maximum consecutive 0/1 run length in a fixed-width bit string.
    """
    s = f"{x & ((1 << bits) - 1):0{bits}b}"

    best = 1
    cur = 1

    for a, b in zip(s, s[1:]):
        if a == b:
            cur += 1
            best = max(best, cur)
        else:
            cur = 1

    return best


def has_run_of_ones(x: int, run_len: int) -> bool:
    """
    Fast test: whether x contains run_len consecutive one-bits.
    """
    y = x

    for i in range(1, run_len):
        y &= x >> 1
        if y == 0:
            return False

    return y != 0


def max_run_leq(x: int, bits: int, limit: int) -> bool:
    """
    Fast predicate: maximum consecutive 0/1 run length <= limit.
    """
    mask = (1 << bits) - 1
    x &= mask

    # Forbid limit+1 consecutive ones.
    if has_run_of_ones(x, limit + 1):
        return False

    # Forbid limit+1 consecutive zeros.
    inv = (~x) & mask
    if has_run_of_ones(inv, limit + 1):
        return False
```

Lines 361–450.

```

    return True


def nibble_popcounts32(x: int) -> list[int]:
    """
    Big-endian nibble popcounts of a 32-bit word.
    """
    return [
        ((x >> (4 * (7 - i))) & 0xF).bit_count()
        for i in range(8)
```

```
  ]


def byte_popcounts(x: int, byte_count: int) -> list[int]:
  """
  Big-endian byte popcounts.
  """
  return [
    ((x >> (8 * (byte_count - 1 - i))) & 0xFF).bit_count()
    for i in range(byte_count)
  ]


def nibble_uniformity32(
  x: int,
  nib_min: int = 1,
  nib_max: int = 3,
  max_dist: int = 6,
) -> bool:
  pcs = nibble_popcounts32(x)

  return (
    all(nib_min <= v <= nib_max for v in pcs)
    and max(pcs) - min(pcs) <= max_dist
  )


def byte_uniformity32(
  x: int,
  byte_min: int = 2,
  byte_max: int = 6,
  max_dist: int = 4,
) -> bool:
  pcs = byte_popcounts(x, 4)

  return (
    all(byte_min <= v <= byte_max for v in pcs)
    and max(pcs) - min(pcs) <= max_dist
  )


def sliding_window_popcount_range(x: int, bits: int, win: int) -> tuple[int, int]:
  """
  Linear sliding-window popcount range over a fixed-width bit string.
  """
  x &= (1 << bits) - 1
  wmask = (1 << win) - 1

  values = [
    ((x >> shift) & wmask).bit_count()
    for shift in range(bits - win, -1, -1)
  ]

  return min(values), max(values)


def sliding_window_checks32(
  x: int,
  max_dev8: int = 3,
  max_dev16: int = 5,
) -> bool:
  """
  Check whether all 8-bit and 16-bit linear windows stay near half density.
  """
  mn8, mx8 = sliding_window_popcount_range(x, 32, 8)
  mn16, mx16 = sliding_window_popcount_range(x, 32, 16)

  return (
    4 - max_dev8 <= mn8 <= mx8 <= 4 + max_dev8
    and 8 - max_dev16 <= mn16 <= mx16 <= 8 + max_dev16
  )


def anti_self_similarity32(
  x: int,
  min_rot_hd: int = 10,
  min_rev_hd: int = 12,
) -> bool:
  """
```

Lines 451–540.

```
  Reject words that are too similar to their rotations or bit-reversal.
  """
  min_rot_hd_actual = min(
    hamming32(x, rotl32(x, r))
    for r in range(1, 32)
  )

  rev_hd_actual = hamming32(x, bit_reverse32(x))

  return (
    min_rot_hd_actual >= min_rot_hd
    and rev_hd_actual >= min_rev_hd
  )


# ============================================================
#  Asymmetric 64-bit bit-shape predicate
# ============================================================

def asymmetric_uniform_halves64(x: int) -> bool:
  """
  Global-balanced but locally asymmetric 64-bit shape predicate.

  The whole word has Hamming weight 32, but the two 32-bit halves
  are deliberately shifted from 16:16 to 13:19.
  """
  high = x >> 32
  low = x & 0xFFFFFFFF

  return (
    # Global balance, local imbalance.
    x.bit_count() == 32
    and popcount32(high) == 13
    and popcount32(low) == 19

    # Bounded local runs.
    and max_run_length(high, 32) <= 4
    and max_run_length(low, 32) <= 4

    # High half: sparse but uniform.
    and nibble_uniformity32(high, nib_min=1, nib_max=3, max_dist=6)
    and byte_uniformity32(high, byte_min=2, byte_max=6, max_dist=4)
    and sliding_window_checks32(high, max_dev8=3, max_dev16=5)

    # Low half: denser, but still controlled.
    and sliding_window_checks32(low, max_dev8=3, max_dev16=5)
    and anti_self_similarity32(low, min_rot_hd=10, min_rev_hd=12)

    # Cross-half parity asymmetry.
    and (high & 1) == 0
```

```
    and (low & 1) == 1
  )


# ============================================================
#  Number-theoretic derivation
# ============================================================

def prime_factors(n: int) -> list[int]:
  """
  Return distinct prime factors of n.
  """
  factors = []

  if n % 2 == 0:
    factors.append(2)
    while n % 2 == 0:
      n //= 2

  d = 3
  while d * d <= n:
    if n % d == 0:
      factors.append(d)
      while n % d == 0:
        n //= d
    d += 2

  if n > 1:
    factors.append(n)

  return factors


def previous_prime(n: int) -> int:
  """
  Return the largest prime strictly smaller than n.
  """
  x = n - 1

  if x == 2:
```

Lines 541–630.

```
    return 2

  if x % 2 == 0:
    x -= 1

  while x >= 2:
    if is_prime64(x):
      return x
    x -= 2

  raise RuntimeError("no previous prime found")


def derive_sigma_rho(table_size: int) -> tuple[int, int, int]:
  """
  Derive sigma and rho from the table size.

  For table_size = 300:
    pmax  = largest_prime_factor(300) = 5
    sigma = previous_prime(300 / (2 * 5)) = previous_prime(30) = 29
    rho   = sigma^2 mod 300 = 29^2 mod 300 = 241
  """
  pmax = max(prime_factors(table_size))
  sigma = previous_prime(table_size // (2 * pmax))
  rho = (sigma * sigma) % table_size

  return pmax, sigma, rho


def derive_small_factor(table_size: int, sigma: int) -> tuple[int, int]:
  """
  Derive the small prime factor P.

  Rule:
    P is the largest 21-bit prime satisfying:
      P mod table_size = sigma
      popcount(P) = 11
  """
  bitlen = 21
  target_weight = (bitlen + 1) // 2

  lo = 1 << (bitlen - 1)
  hi = (1 << bitlen) - 1

  p = hi - ((hi - sigma) % table_size)

  steps = 0

  while p >= lo:
    steps += 1

    # Cheap filters first, primality last.
    if p.bit_count() == target_weight and is_prime64(p):
      return p, steps

    p -= table_size

  raise RuntimeError("small factor P not found")


# ============================================================
#  Large factor and final kappa filters
# ============================================================

def cheap_large_factor_filter(q: int) -> bool:
  """
  Cheap pre-primality filters for Q.
  """
  return (
    q.bit_length() == 43
    and q.bit_count() == 18
    and max_run_leq(q, 43, 5)
    and byte_popcounts(q, 6) == [2, 3, 3, 4, 3, 3]
  )


def kappa_filter(kappa: int, table_size: int, rho: int) -> bool:
  """
  Final acceptance filter for the 64-bit KappaCounter step.
  """
  high = kappa >> 32
  low = kappa & 0xFFFFFFFF

  return (
    kappa.bit_length() == 64
    and kappa.bit_count() == 32
    and high.bit_count() == 13
    and low.bit_count() == 19
    and (high & 1) == 0
```

```
    and (low & 1) == 1
```

Lines 631–720.

```
    and max_run_length(kappa, 64) == 4
    and kappa % table_size == rho
    and gcd(kappa, table_size) == 1
    and byte_popcounts(kappa, 8) == [3, 3, 3, 4, 5, 4, 7, 3]
    and asymmetric_uniform_halves64(kappa)
  )


def derive_large_factor(
  table_size: int,
  sigma: int,
  rho: int,
  p: int,
) -> tuple[int, int, int]:
  """
  Derive the large prime factor Q.

  The search anchor is derived from P, rho, and the target 64-bit scale:

    shift  = 64 - 2 * bit_length(P)
    anchor = floor(P * 2^shift * rho / table_size)

  Starting from the first q >= anchor with q mod table_size = sigma,
  scan q += table_size until the first Q passing the cheap filters,
  primality test, and final KappaCounter acceptance filter.
  """
  shift = 64 - 2 * p.bit_length()
  anchor = (p * (1 << shift) * rho) // table_size

  q = anchor + ((sigma - anchor) % table_size)

  steps = 0

  while True:
    steps += 1

    # Cheap filters first.
    if cheap_large_factor_filter(q):
      # Expensive primality test last.
      if is_prime64(q):
        kappa = p * q

        if kappa_filter(kappa, table_size, rho):
          return q, anchor, steps

    q += table_size


def derive_kappa_counter() -> dict[str, int]:
  """
  Canonical public derivation procedure for KappaCounter.
  """
  table_size = 300

  pmax, sigma, rho = derive_sigma_rho(table_size)

  p, p_steps = derive_small_factor(table_size, sigma)
  q, anchor, q_steps = derive_large_factor(table_size, sigma, rho, p)

  kappa = p * q

  return {
    "table_size": table_size,
    "pmax": pmax,
    "sigma": sigma,
    "rho": rho,
    "P": p,
    "Q": q,
    "anchor": anchor,
    "kappa": kappa,
    "P_search_steps": p_steps,
    "Q_search_steps": q_steps,
  }


# ============================================================
# Diagnostic printer
# ============================================================

def print_diagnostics(result: dict[str, int]) -> None:
  table_size = result["table_size"]
  pmax = result["pmax"]
  sigma = result["sigma"]
  rho = result["rho"]
  p = result["P"]
  q = result["Q"]
  anchor = result["anchor"]
  kappa = result["kappa"]

  high = kappa >> 32
```

Lines 721–810.

```
  low = kappa & 0xFFFFFFFF

  print(f"table_size = {table_size}")
  print(f"pmax       = {pmax}")
  print(f"sigma      = {sigma}")
  print(f"rho        = {rho}")
  print()

  print(f"P      = 0x{p:X}")
  print(f"Q      = 0x{q:X}")
  print(f"anchor   = 0x{anchor:X}")
  print(f"kappa    = 0x{kappa:X}")
  print()

  print(f"P is prime = {is_prime64(p)}")
  print(f"Q is prime = {is_prime64(q)}")
  print(f"P * Q == kappa = {p * q == kappa}")
  print()

  print(f"P % 300     = {p % table_size}")
  print(f"Q % 300     = {q % table_size}")
  print(f"sigma^2 % 300 = {(sigma * sigma) % table_size}")
  print(f"kappa % 300   = {kappa % table_size}")
  print(f"gcd(kappa, 300) = {gcd(kappa, table_size)}")
  print()

  print(f"kappa bit_length = {kappa.bit_length()}")
  print(f"kappa popcount   = {kappa.bit_count()}")
  print(f"kappa max_run  = {max_run_length(kappa, 64)}")
  print(f"kappa byte popcounts = {byte_popcounts(kappa, 8)}")
  print()
```

```
    print(f"high32 = 0x{high:08X}")
    print(f"high32 popcount = {high.bit_count()}")
    print(f"high32 max_run  = {max_run_length(high, 32)}")
    print(f"high32 even     = {(high & 1) == 0}")
    print(f"high32 nibble popcounts = {nibble_popcounts32(high)}")
    print(f"high32 byte popcounts   = {byte_popcounts(high, 4)}")
    print()

    print(f"low32 = 0x{low:08X}")
    print(f"low32 popcount = {low.bit_count()}")
    print(f"low32 max_run  = {max_run_length(low, 32)}")
    print(f"low32 odd      = {(low & 1) == 1}")
    print(f"low32 nibble popcounts = {nibble_popcounts32(low)}")
    print(f"low32 byte popcounts   = {byte_popcounts(low, 4)}")
    print()

    min_rot_hd_low = min(hamming32(low, rotl32(low, r)) for r in range(1, 32))
    rev_hd_low = hamming32(low, bit_reverse32(low))

    print(f"low32 min rotation HD = {min_rot_hd_low}")
    print(f"low32 reversal HD     = {rev_hd_low}")
    print()

    print(f"asymmetric_uniform_halves64(kappa) = {asymmetric_uniform_halves64(kappa)}")
    print()

    print(f"P-search steps = {result['P_search_steps']}")
    print(f"Q-search steps = {result['Q_search_steps']}")


  if __name__ == "__main__":
    result = derive_kappa_counter()

    assert result["sigma"] == 29
    assert result["rho"] == 241
    assert result["P"] == 0x1F9289
    assert result["Q"] == 0x658459A0EC1
    assert result["kappa"] == 0xC8522A96E53AF749

    print_diagnostics(result)

  */

  // ------------------------------------------------------------------------
  // Hybrid8 byte substitutions and Shadow carry/borrow operations
  // ------------------------------------------------------------------------
  // H-round uses a 256-bit BigS layer built from alternating byte S-boxes:
  //   SA = BTM -> Borrow -> BTM -> Carry
  //   SB = Carry -> BTM -> Borrow -> BTM
  // S-round uses bounded stage-2 shadow carry/borrow word operations.
  // Both families are table-free in this implementation.
  // ------------------------------------------------------------------------

  namespace
  {
    [[nodiscard]] constexpr std::uint8_t byte8( std::uint64_t value, std::size_t index ) noexcept
    {
      return static_cast<std::uint8_t>( ( value >> ( index * 8U ) ) & 0xFFULL );
```

Lines 811–900.

```
    }

    [[nodiscard]] constexpr std::uint64_t with_byte8( std::uint64_t value, std::size_t index, std::uint8_t byte ) noexcept
    {
      const std::uint64_t shift = static_cast<std::uint64_t>( index * 8U );
      const std::uint64_t mask  = 0xFFULL << shift;
      return ( value & ~mask ) | ( static_cast<std::uint64_t>( byte ) << shift );
    }

    [[nodiscard]] constexpr std::uint8_t lifted_carry_add8( std::uint8_t a, std::uint8_t k ) noexcept
    {
      const std::uint8_t p  = static_cast<std::uint8_t>( a ^ k );
      const std::uint8_t g  = static_cast<std::uint8_t>( a & k );
      const std::uint8_t n1 = static_cast<std::uint8_t>( p >> 1 );
      const std::uint8_t n2 = static_cast<std::uint8_t>( ( p >> 2 ) ^ ( p >> 3 ) );

      const std::uint8_t w1 = g;
      const std::uint8_t w2 = static_cast<std::uint8_t>( w1 & n1 );
      const std::uint8_t w4 = static_cast<std::uint8_t>( w2 & n2 );

      return static_cast<std::uint8_t>( p ^ ( w1 << 1 ) ^ ( w2 << 2 ) ^ ( w4 << 4 ) );
    }

    [[nodiscard]] constexpr std::uint8_t lifted_borrow_sub8( std::uint8_t a, std::uint8_t k ) noexcept
    {
      const std::uint8_t p  = static_cast<std::uint8_t>( a ^ k );
      const std::uint8_t h  = static_cast<std::uint8_t>( ( ~a ) & k );
      const std::uint8_t r  = static_cast<std::uint8_t>( ~p );
      const std::uint8_t n1 = static_cast<std::uint8_t>( r >> 1 );
      const std::uint8_t n2 = static_cast<std::uint8_t>( ( r >> 2 ) ^ ( r >> 3 ) );

      const std::uint8_t w1 = h;
      const std::uint8_t w2 = static_cast<std::uint8_t>( w1 & n1 );
      const std::uint8_t w4 = static_cast<std::uint8_t>( w2 & n2 );

      return static_cast<std::uint8_t>( p ^ ( w1 << 1 ) ^ ( w2 << 2 ) ^ ( w4 << 4 ) );
    }

    [[nodiscard]] constexpr std::uint8_t ltam_l8( std::uint8_t a, std::uint8_t k ) noexcept
    {
      return static_cast<std::uint8_t>( a ^ ( ( a << 1 ) & k ) ^ ( ( a << 2 ) & k ) ^ ( ( a << 4 ) & k ) );
    }

    [[nodiscard]] constexpr std::uint8_t ltam_r8( std::uint8_t a, std::uint8_t k ) noexcept
    {
      return static_cast<std::uint8_t>( a ^ ( ( a >> 1 ) & k ) ^ ( ( a >> 2 ) & k ) ^ ( ( a >> 4 ) & k ) );
    }

    [[nodiscard]] constexpr std::uint8_t btm8( std::uint8_t a, std::uint8_t k ) noexcept
    {
      return ltam_r8( ltam_l8( a, k ), k );
    }

    [[nodiscard]] constexpr std::uint8_t hybrid8_sa( std::uint8_t x ) noexcept
    {
      x = btm8( x, 0x2b );
      x = lifted_borrow_sub8( x, 0x3f );
      x = btm8( x, 0x2b );
      x = lifted_carry_add8( x, 0x3f );
      return x;
    }

    [[nodiscard]] constexpr std::uint8_t hybrid8_sb( std::uint8_t x ) noexcept
    {
      x = lifted_carry_add8( x, 0x3f );
      x = btm8( x, 0x2b );
      x = lifted_borrow_sub8( x, 0x3f );
      x = btm8( x, 0x2b );
      return x;
    }
```

```
  [[nodiscard]] constexpr std::uint64_t bigs64( std::uint64_t value, std::size_t byte_offset ) noexcept
  {
    std::uint64_t result = 0;

    for ( std::size_t i = 0; i < 8; ++i )
    {
      const std::uint8_t in  = byte8( value, i );
      const std::uint8_t out = ( ( byte_offset + i ) & 1U ) == 0U ? hybrid8_sa( in ) : hybrid8_sb( in );
      result = with_byte8( result, i, out );
    }

    return result;
  }

  [[nodiscard]] constexpr std::uint64_t shadow_carry64( std::uint64_t a, std::uint64_t b ) noexcept
  {
    const std::uint64_t p = a ^ b;

    std::uint64_t c = a & b;
```

Lines 901–990.

```
    std::uint64_t q = p;

    c ^= q & ( c << 1 );
    q &= q << 1;
    c ^= q & ( c << 2 );

    return p ^ ( c << 1 ) ^ ( c << 2 );
  }

  [[nodiscard]] constexpr std::uint64_t shadow_borrow64( std::uint64_t a, std::uint64_t b ) noexcept
  {
    const std::uint64_t p = a ^ b;

    std::uint64_t c = ( ~a ) & b;
    std::uint64_t q = ~p;

    c ^= q & ( c << 1 );
    q &= q << 1;
    c ^= q & ( c << 2 );

    return p ^ ( c << 1 ) ^ ( c << 2 );
  }
} // namespace

// Core permutation step.
//
// External input rebinding:
// - `number_once` is injected only through XOR / ROTL-based rebinding of the
//   current state lanes into temporary working words (ww, xx, yy, zz).
//
// Constant access:
// - Four public table indices are derived from `number_once`, `counter`, and
//   simple combinations of them.
// - No hidden runtime constants are introduced beyond XCR_ROUND_CONSTANTS[].
//
// Hybrid8-Shadow nonlinear replacement:
// - The previous native modular add/sub step is intentionally removed.
// - The public counter selects a constant-time public round type:
//      H-round: counter % 4 == 0
//          yy = BigS64(yy ^ x,  0)
//          zz = BigS64(zz ^ y,  8)
//          ww = BigS64(ww ^ z, 16)
//          xx = BigS64(xx ^ w, 24)
//      S-round: otherwise
//          yy = shadow_carry64(yy, x)
//          zz = shadow_borrow64(zz, y)
//          ww = shadow_carry64(ww, z)
//          xx = shadow_borrow64(xx, w)
// - H-round has byte-table DDT/LAT oracle semantics.
// - S-round is intended to be modeled by finite-window DP/LC shadow oracles.
//
// Everything after that is XOR / ROTL diffusion.
// The state is updated in place.

void XorConstantRotation::PermutationARX( const std::uint64_t number_once )
{
  // Rebind lanes with number_once injection (XOR/ROTL-only).
  // This keeps the per-call input explicit and modelable.
  std::uint64_t ww = x ^ number_once;
  std::uint64_t xx = y ^ std::rotl( number_once, 9 );
  std::uint64_t yy = z ^ std::rotl( number_once, 27 );
  std::uint64_t zz = w ^ std::rotl( number_once, 43 );

  // Table-driven constants (public indices).
  // Using different index "views" of n to avoid trivial repetition.
  const std::uint64_t RC0 = XCR_ROUND_CONSTANTS[ ( number_once ) % ROUND_CONSTANT_SIZE ];
  const std::uint64_t RC1 = XCR_ROUND_CONSTANTS[ ( counter ) % ROUND_CONSTANT_SIZE ];
  const std::uint64_t RC2 = XCR_ROUND_CONSTANTS[ ( number_once + counter ) % ROUND_CONSTANT_SIZE ];
  const std::uint64_t RC3 = XCR_ROUND_CONSTANTS[ ( ( number_once ^ std::rotl( number_once ^counter, 3 ) ) ) % ROUND_CONSTANT_SIZE ];

  // Pre-mix (XOR-only) -> feeds the 4 carries below.
  z = xx ^ RC0;
  w = yy ^ RC1;
  x = zz ^ RC2;
  y = ww ^ RC3;

  // --- Periodic Hybrid8-Shadow nonlinear layer ---
  // H-round: exact byte-table oracle path, public period 4.
  // S-round: shadow carry/borrow finite-window oracle path.
  if ( ( counter & 3ULL ) == 0ULL )
  {
    yy = bigs64( yy ^ x, 0 );
    zz = bigs64( zz ^ y, 8 );
    ww = bigs64( ww ^ z, 16 );
    xx = bigs64( xx ^ w, 24 );
  }
  else
  {
    yy = shadow_carry64( yy, x );
    zz = shadow_borrow64( zz, y );
```

Lines 991–1080.

```
    ww = shadow_carry64( ww, z );
    xx = shadow_borrow64( xx, w );
  }

  // Mix
  // Rotation/XOR-only diffusion layer (no more add/sub).
  x = x ^ std::rotl( xx, 7 ) ^ ( std::rotl( yy, 19 ) ^ zz );
  y = y ^ std::rotl( yy, 11 ) ^ ( std::rotl( zz, 23 ) ^ ww );
  z = z ^ std::rotl( zz, 17 ) ^ ( std::rotl( ww, 29 ) ^ xx );
  w = w ^ std::rotl( ww, 13 ) ^ ( std::rotl( xx, 31 ) ^ yy );
}

// ------------------------------------------------------------------------
// Private helper iteration
```

```
// ------------------------------------------------------------------------
// Role:
// - This is an internal helper path used by initialization / warm-up style code.
// - It accepts the same external per-call input `number_once` as the public
//   production path, but unlike the public API it also advances the public
//   schedule variable `counter`.
//
// Input separation:
// - `number_once` is the only explicit per-call external input here.
// - The seed is NOT directly injected here; seed influence exists only through
//   the already-initialized internal state (w, x, y, z).
//
// Counter semantics:
// - `counter` is public / trackable state used for round-constant indexing.
// - This helper advances `counter += COUNTER_STEP` after one permutation.
// - Because gcd(COUNTER_STEP, ROUND_CONSTANT_SIZE) = 1, the counter walk covers
//   the constant table in a full-period modular schedule.
//
// Output:
// - Returns a 128-bit view of the updated state as
//     { x ^ y, z ^ w }.
//
// Modeling note:
// - Native modular add/sub operations are not used inside PermutationARX().
// - Nonlinearity comes from the public-period Hybrid8 H-round and the
//   finite-window shadow carry/borrow S-round.
// ------------------------------------------------------------------------

XorConstantRotation::GeneratedSubKey128 XorConstantRotation::StateIteration( std::uint64_t number_once )
{
  static_assert( ( COUNTER_STEP % ROUND_CONSTANT_SIZE ) != 0, "counter step must be nonzero mod ROUND_CONSTANT size" );
  static_assert( std::gcd( COUNTER_STEP, ROUND_CONSTANT_SIZE ) == 1, "counter step must be coprime with ROUND_CONSTANT size" );

  this->PermutationARX( number_once );

  GeneratedSubKey128 out = { x ^ y, z ^ w };

  // Public counter (uniform add; kept for period expansion & external modeling)
  counter += COUNTER_STEP;

  return out;
}

XorConstantRotation::GeneratedSubKey128 XorConstantRotation::operator()( std::uint64_t number_once )
{
  // Public production path now shares the same stepping semantics:
  // one permutation + one explicit counter advance.
  return this->StateIteration( number_once );
}

XorConstantRotation::GeneratedSubKey128 XorConstantRotation::GenerateSubKey128( std::uint64_t number_once )
{
  // Same public semantics as operator().
  return this->StateIteration( number_once );
}

// ------------------------------------------------------------------------
// State initialization / seeded warm-up
// ------------------------------------------------------------------------
// Purpose:
// - Absorb the ctor / Seed() input into the 256-bit internal state.
// - Avoid trivial early structure such as all-zero side lanes.
// - Build a reproducible seeded starting state before normal production calls.
//
// Seed path:
// - The incoming seed is first stored in `w`.
// - x, y, z are reset explicitly.
// - If the input seed is zero, the code forces `w != 0` in a branch-free way
//   before continuing, so initialization never starts from the fully-zero root.
//
// Warm-up structure:
// - Initialization uses a short self-contained expansion driven by the internal
//   helper path StateIteration().
// - During this phase, the evolving local variable `random` is used as the
//   `number_once` source, so warm-up does not depend on any external nonce flow.
// - A small GGM-like two-branch expansion is used to derive two 64-bit leaves,
```

Lines 1081–1170.

```
//   then those leaves are folded back into the state.
//
// Counter handling:
// - During warm-up, StateIteration() advances `counter`, so the helper path
//   walks the public round-constant schedule while diffusing the seed.
// - After warm-up, `counter` is reset to the fixed public start value
//     counter = COUNTER_STEP;
//   so the externally visible production phase begins from a clean,
//   reproducible schedule state.
//
// Final state intent:
// - After initialization, the object is in a seeded, nontrivial, reproducible
//   state ready for public GenerateSubKey128()/operator() calls.
// - Subsequent public production calls mutate the internal state AND
//   explicitly advance `counter` in the current implementation.
// ------------------------------------------------------------------------
void XorConstantRotation::StateInitialize()
{
  // Ensure w != 0 without branch:
  const std::uint64_t nonzero_flag = ( w | ( 0ULL - w ) ) >> 63;
  const std::uint64_t is_zero = nonzero_flag ^ 1ULL;
  w = w + is_zero;

  // The state w is seed
  std::uint64_t random = 0;
  std::uint64_t backup_seed = w;

  auto ggm64_rounds = [ & ]( std::uint64_t seed64 ) -> std::uint64_t {
    constexpr std::uint64_t WARMUP = 0x5741524D5550ULL;   // "WARMUP"
    std::uint64_t       out64 = seed64;

    // 2 rounds, each round produces 64 bits
    for ( int round = 0; round < 2; ++round )
    {
      std::uint64_t next64 = 0;

      for ( int bit_index = 0; bit_index < 64; ++bit_index )
      {
        const std::uint64_t in = ( WARMUP << 48 ) ^ ( out64 << 32 ) ^ ( out64 << 16 ) ^ ( round << 8 ) ^ bit_index;

        const GeneratedSubKey128 out128 = StateIteration( in );

        const std::uint64_t bit = out128.GetBit( 127 );

        next64 = ( next64 << 1 ) | bit;
      }

      out64 = next64;
    }

    return out64;
  };
```

```
    const std::uint64_t left_input = backup_seed ^ XCR_ROUND_CONSTANTS[ ROUND_CONSTANT_SIZE - 1 ];
    const std::uint64_t right_input = backup_seed ^ XCR_ROUND_CONSTANTS[ ROUND_CONSTANT_SIZE - 2 ];
    std::uint64_t lx, rx, ly, ry, lz, rz;

    // LEFT LEAF NODES
    const std::uint64_t left_leaf = ggm64_rounds( left_input );
    // COPY
    lx = x;
    ly = y;
    lz = z;

    // MUST RESET TO GGM TREE ROOT!!!
    x = 0;
    y = 0;
    z = 0;
    w = backup_seed;
    // Reset to fixed public start value for production phase
    counter = COUNTER_STEP;

    // RIGHT LEAF NODES
    const std::uint64_t right_leaf = ggm64_rounds( right_input );
    // COPY
    rx = x;
    ry = y;
    rz = z;

    // MUST RESET TO GGM TREE ROOT!!!
    x = 0;
    y = 0;
    z = 0;
    w = backup_seed;
    // Reset to fixed public start value for production phase
    counter = COUNTER_STEP;

    // Set Random Node
    x = lx ^ rx;
    y = ly ^ ry;
```

Lines 1171–1177.

```
    z = lz ^ rz;
    random = left_leaf ^ right_leaf;

    // Minimal fold-back (key whitening)
    w = backup_seed ^ random;
  }
} // namespace TwilightDreamOfMagical::CustomSecurity::CSPRNG
```

1.4 LittleOaldresPuzzle_Cryptic.h

This file is split into 3 contiguous minted chunk(s).

Lines 1–120.

```
/*
 * Copyright (C) 2023-2050 Twilight-Dream
 *
 * 本文件是 Algorithm_OaldresPuzzleCryptic 的一部分。
 *
 * Algorithm_OaldresPuzzleCryptic 是自由软件：你可以再分发之和/或依照由自由软件基金会发布的 GNU 通用公共许可证修改之，无论是版本 3 许可证，还是（按你的决定）任何以后版都可以。
 *
 * 发布 Algorithm_OaldresPuzzleCryptic 是希望它能有用，但是并无保障；甚至连可销售和符合某个特定的目的都不保证。请参看 GNU 通用公共许可证，了解详情。
 * 你应该随程序获得一份 GNU 通用公共许可证的复本。如果没有，请看 <https://www.gnu.org/licenses/>。
 */

/*
 * Copyright (C) 2023-2050 Twilight-Dream
 *
 * This file is part of Algorithm_OaldresPuzzleCryptic.
 *
 * Algorithm_OaldresPuzzleCryptic is free software: you may redistribute it and/or modify it under the GNU General Public License as published by the Free Software Foundation,
↪  either under the Version 3 license, or (at your discretion) any later version.
 *
 * TDOM-EncryptOrDecryptFile-Reborn is released in the hope that it will be useful, but there are no guarantees; not even that it will be marketable and fit a particular
↪  purpose. Please see the GNU General Public License for details.
 * You should get a copy of the GNU General Public License with your program. If not, see <https://www.gnu.org/licenses/>.
 */

#ifndef ALGORITHM_OALDRESPUZZLE_CRYPTIC_LITTLEOALDRESPUZZLE_CRYPTIC_HPP
#define ALGORITHM_OALDRESPUZZLE_CRYPTIC_LITTLEOALDRESPUZZLE_CRYPTIC_HPP

#include <vector>
#include <utility>
#include <random>
#include "XorConstantRotation.h"

#if _DEBUG
#include <chrono>
#endif

namespace TwilightDreamOfMagical::CustomSecurity
{
  //SymmetricEncryptionDecryption
  namespace SED::StreamCipher
  {
    using Key128   = std::pair<std::uint64_t, std::uint64_t>;   // 128-bit key
    using Block128 = std::pair<std::uint64_t, std::uint64_t>;   // 128-bit block

    class LittleOaldresPuzzle_Cryptic
    {

    public:
      using XorConstantRotation = CSPRNG::XorConstantRotation;

      LittleOaldresPuzzle_Cryptic(const std::uint64_t seed, std::uint64_t rounds)
        :
        seed(seed),
        prng(seed),
        prng_second(~seed ^ std::rotl(seed, 32)),
        rounds(rounds),
        KeyStates(std::vector<KeyState>(rounds, KeyState()))
      {
      }

      LittleOaldresPuzzle_Cryptic(const std::uint64_t seed)
        :
        seed(seed),
        prng(seed),
        prng_second(~seed ^ std::rotl(seed, 32)),
        rounds(4),
        KeyStates(std::vector<KeyState>(rounds, KeyState()))
      {
      }

      LittleOaldresPuzzle_Cryptic()
        :
        seed(1),
        prng(seed),
        prng_second(~seed ^ std::rotl(seed, 32)),
        rounds(4),
        KeyStates(std::vector<KeyState>(rounds, KeyState()))
      {
      }

      Block128 SingleRoundEncryption(const Block128 data, const Key128 key, const std::uint64_t number_once)
      {
        Block128 result = EncryptionCoreFunction(data, key, number_once);
        ResetPRNG();
        return result;
      }

      Block128 SingleRoundDecryption(const Block128 data, const Key128 key, const std::uint64_t number_once)
      {
        Block128 result = DecryptionCoreFunction(data, key, number_once);
        ResetPRNG();
        return result;
      }

      void MultipleRoundsEncryption(const std::vector<Block128>& data_array, std::vector<Key128>& keys, std::vector<Block128>& result_data_array)
      {
        // Ensure result_data_array is of the same size as data_array
        if(data_array.empty())
          return;
        else if (result_data_array.size() < data_array.size())
          result_data_array.resize(data_array.size());

        #if _DEBUG

        auto start = std::chrono::high_resolution_clock::now();

        #endif

        // Encryption
        for (size_t i = 0; i < data_array.size(); ++i)
        {
          result_data_array[i] = EncryptionCoreFunction(data_array[i], keys[i % keys.size()], i);
        }

        #if _DEBUG

        auto end = std::chrono::high_resolution_clock::now();
        encryptionTime = std::chrono::duration_cast<std::chrono::nanoseconds>( end - start );

        #endif

        // Reset the PRNG state for the next encryption or decryption (Must be call this function)
```

Lines 121–240.

```
    ResetPRNG();
  }

  void MultipleRoundsDecryption(const std::vector<Block128>& data_array, std::vector<Key128>& keys, std::vector<Block128>& result_data_array)
  {
    // Ensure result_data_array is of the same size as data_array
    if(data_array.empty())
      return;
    else if (result_data_array.size() < data_array.size())
      result_data_array.resize(data_array.size());

    #if _DEBUG

    auto start = std::chrono::high_resolution_clock::now();

    #endif

    // Decryption
    for (size_t i = 0; i < data_array.size(); ++i)
    {
      result_data_array[i] = DecryptionCoreFunction(data_array[i], keys[i % keys.size()], i);
    }

    #if _DEBUG

    auto end = std::chrono::high_resolution_clock::now();
    encryptionTime = std::chrono::duration_cast<std::chrono::nanoseconds>( end - start );

    #endif

    // Reset the PRNG state for the next encryption or decryption (Must be call this function)
    ResetPRNG();
  }

  std::vector<Block128> GenerateSubkey_WithUseEncryption(const Key128 key, std::uint64_t loop_count)
  {
    Key128 subkey {0,0};
    Key128 buffer {0,0};
    std::vector<Key128> subkeys(loop_count, {0,0});

    std::mt19937_64 cpp_prng(key.first ^ key.second ^ loop_count);
    Block128 number_once {0,0};

    //NumberOnce/CounterMode
    for(std::uint64_t counter = 0; counter < loop_count; ++counter)
    {
      number_once = {cpp_prng(), cpp_prng()};
      buffer = EncryptionCoreFunction(number_once, key, counter);
      subkey.first  ^= buffer.first;
      subkey.second ^= buffer.second;
      subkeys[counter] = subkey;
    }

    // Reset the PRNG state for the next encryption or decryption (Must be call this function)
    ResetPRNG();

    return subkeys;
  }

  std::vector<Block128> GenerateSubkey_WithUseDecryption(const Key128 key, std::uint64_t loop_count)
  {
    Key128 subkey {0,0};
    Key128 buffer {0,0};
    std::vector<Key128> subkeys(loop_count, {0,0});

    std::mt19937_64 cpp_prng(key.first ^ key.second ^ loop_count);
    Block128 number_once {0,0};

    //NumberOnce/CounterMode
    for(std::uint64_t counter = 0; counter < loop_count; ++counter)
    {
      number_once = {cpp_prng(), cpp_prng()};
      buffer = DecryptionCoreFunction(number_once, key, counter);
      subkey.first  ^= buffer.first;
      subkey.second ^= buffer.second;
      subkeys[counter] = subkey;
    }

    // Reset the PRNG state for the next encryption or decryption (Must be call this function)
    ResetPRNG();

    return subkeys;
  }

  // Reset both internal XCR instances back to their deterministic seeded states.
  //
  // Why this exists:
  // - Encryption / decryption / subkey generation consume persistent member PRNGs.
  // - Without reset, successive top-level calls on the same
  //   `LittleDaldresPuzzle_Cryptic` object would continue from the previously
  //   mutated internal XCR states.
  //
  // Reset policy:
  // - `prng`        is reset with the original constructor seed.
  // - `prng_second` is reset with the paired derived seed: ~seed ^ std::rotl(seed, 32)
  //
  // Result:
  // - Each top-level encryption / decryption session starts from the same
  //   reproducible pair of seeded XCR states.
  // - Inside one session, however, round material is still generated from
  //   continuously evolving member states.
  void ResetPRNG()
  {
    prng.Seed(seed);
    prng_second.Seed(~seed ^ std::rotl(seed, 32));
  }

  #if _DEBUG

  std::chrono::nanoseconds encryptionTime;
  std::chrono::nanoseconds decryptionTime;

  #endif

private:
  std::uint64_t seed = 0;
  // Left / Right domain PRNG instances (member variables, no locals in GenerateAndStoreKeyStates)
  XorConstantRotation prng;
  XorConstantRotation prng_second;
  std::uint64_t rounds = 4;
```

Lines 241–265.

```
  struct KeyState
  {
    Key128 subkey{0,0};
    std::uint64_t choice_function = 0;
    std::uint64_t bit_rotation_amount_a = 0;
```

```
        std::uint64_t bit_rotation_amount_b = 0;
        //std::uint32_t round_constant_index = 0;
      };

      std::vector<KeyState> KeyStates;

      void GenerateAndStoreKeyStates(const Key128 key_128bit, const std::uint64_t number_once);

      void MixLinearTransform_Forward(uint64_t& lane0, uint64_t& lane1, const KeyState& current_key_state);
      void MixLinearTransform_Backward(uint64_t& lane0, uint64_t& lane1, const KeyState& current_key_state);

      Block128 EncryptionCoreFunction(const Block128 data, const Key128 key_128bit, const std::uint64_t round);
      Block128 DecryptionCoreFunction(const Block128 data, const Key128 key_128bit, const std::uint64_t round);
    };
  }

} // TwilightDreamOfMagical

#endif //ALGORITHM_OALDRESPUZZLE_CRYPTIC_LITTLEOALDRESPUZZLE_CRYPTIC_HPP
```

1.5 LittleOaldresPuzzle_Cryptic.cpp

This file is split into 10 contiguous minted chunk(s).

Lines 1–90.

```
#include "LittleOaldresPuzzle_Cryptic.h"

/*

  Linear layer box attribute:

  find_linear_box.exe --bits 32 --efficient-implementation --quality-threshold-branch-number 12 --max-xor 6 --seed 4 --need-found-result 2 --no-progress
  M(rotl)_hex = 0xd05a0889  M(rotl)^{-1}_hex = 0x5fc08ef4
  M(rotr)_hex = 0x2220b417  M(rotr)^{-1}_hex = 0x5ee207f4
  minimum weight found (pair) = 12
  rotl: diff=12 lin=12 combined=12
  rotr: diff=12 lin=12 combined=12
  Operations(rotl): start_bit=0 steps=6
  v0 = (1 << 0)  [0x00000001]
  v1 = v0 ^ rotl(v0,2)  [0x00000005]
  v2 = v0 ^ rotl(v1,17)  [0x0000a0001]
  v3 = v0 ^ rotl(v2,4)  [0x00a00011]
  v4 = v3 ^ rotl(v3,24)  [0x11a0a011]
  v5 = v2 ^ rotl(v4,7)  [0xd05a0889]
  Operations(rotr): start_bit=0 steps=6
  v0 = (1 << 0)  [0x00000001]
  v1 = v0 ^ rotr(v0,2)  [0x40000001]
  v2 = v0 ^ rotr(v1,17)  [0x0000a001]
  v3 = v0 ^ rotr(v2,4)  [0x10000a01]
  v4 = v3 ^ rotr(v3,24)  [0x100a0b11]
  v5 = v2 ^ rotr(v4,7)  [0x2220b417]

  M(rotl)_hex = 0x29082a87  M(rotl)^{-1}_hex = 0x7868ab73
  M(rotr)_hex = 0xc2a82129  M(rotr)^{-1}_hex = 0x9daa2c3d
  minimum weight found (pair) = 12
  rotl: diff=12 lin=12 combined=12
  rotr: diff=12 lin=12 combined=12
  Operations(rotl): start_bit=0 steps=6
  v0 = (1 << 0)  [0x00000001]
  v1 = v0 ^ rotl(v0,2)  [0x00000005]
  v2 = v1 ^ rotl(v0,24)  [0x01000005]
  v3 = v2 ^ rotl(v1,4)  [0x01000055]
  v4 = v2 ^ rotl(v3,27)  [0xa9080007]
  v5 = v4 ^ rotl(v3,7)  [0x29082a87]
  Operations(rotr): start_bit=0 steps=6
  v0 = (1 << 0)  [0x00000001]
  v1 = v0 ^ rotr(v0,2)  [0x40000001]
  v2 = v1 ^ rotr(v0,24)  [0x40000101]
  v3 = v2 ^ rotr(v1,4)  [0x54000101]
  v4 = v2 ^ rotr(v3,27)  [0xc000212b]
  v5 = v4 ^ rotr(v3,7)  [0xc2a82129]

  DONE.
  Tested candidates = 5736
  Accepted candidates (printed) = 2
  Accepted candidates (total)   = 2
  Elapsed seconds = 1.35393
  Random ISD iterations executed (Prange screen) = 8896
  Random ISD iterations executed (Quality gate)  = 32768
  Random ISD iterations executed (Final confirm) = 16384
  quality_threshold_branch_number(B) = 12
  Quality gate = ON  (quality_trials=4096 exhaustive_input_weight_max=2 full_unit_scan=yes)

  --- Best candidate (post-search confirmation) ---
  M(rotl)_hex = 0x29082a87  M(rotl)^{-1}_hex = 0x7868ab73
  M(rotr)_hex = 0xc2a82129  M(rotr)^{-1}_hex = 0x9daa2c3d
  minimum weight found = 12


  --- Search-phase upper bounds (before final confirmation) ---
  Best(rotl) differential branch upper bound = 12
  Best(rotl) linear branch upper bound       = 12
  Best(rotl) combined branch upper bound     = 12
  Best(rotr) differential branch upper bound = 12
  Best(rotr) linear branch upper bound       = 12
  Best(rotr) combined branch upper bound     = 12
  Best(pair) combined branch upper bound     = 12

  --- Post-search confirmed upper bounds ---
  Confirmed(rotl) differential upper bound   = 12
  Confirmed(rotl) linear upper bound         = 12
  Confirmed(rotl) combined upper bound       = 12
  Confirmed(rotr) differential upper bound   = 12
  Confirmed(rotr) linear upper bound         = 12
  Confirmed(rotr) combined upper bound       = 12
  Confirmed(pair) combined upper bound       = 12

  Threshold check (confirmed combined >= 12) = PASS

  Quality: ACCEPTED (safe to forward to heuristic decomposer)
  Next step: run linear_box_heuristic_decomposer --verify <hex> for strict validation (do this for BOTH matrices).

*/

namespace TwilightDreamOfMagical::CustomSecurity
```

Lines 91–180.

```
{
  //SymmetricEncryptionDecryption
  namespace SED::StreamCipher
  {
    /*
      Implementation of Custom Data Encrypting Worker and Decrypting Worker
      自定义加密和解密数据工作器的实现

      OaldresPuzzle-Cryptic (Type 1)
      隐秘的奥尔德雷斯之谜 (类型 1)
    */

    //NeoAlzette is like the Alzette ARX-box of Sparkle algorithms, but not, just similar in structure.
    //NeoAlzette 就像 Sparkle 算法的 Alzette ARX-box，但又不是，只是结构相似而已。
    //https://eprint.iacr.org/2019/1378.pdf

    /**
     * NeoAlzette V6.5 Second Schedule - ARX-box / ARX S-box implementation
     *
     * This class is written in the older NeoAlzetteSubstitutionBox style:
     * - forward/backward are instance methods;
     * - the whole ARX-box is kept in one class body;
     * - C++20 std::rotl/std::rotr and inline constexpr constants are used directly;
     * - the V6.5 second schedule is preserved exactly from NeoAlzetteCore.
     *
     * Important design note:
     * The first V6.5 schedule is intentionally abandoned here.  The second schedule
     * moves the injection windows so the solver cannot cheaply kill the injection
     * input branch around the nonlinear layer and reactivate it only after the
```

```
 * injection layer has been bypassed.
 */
class NeoAlzetteSubstitutionBox
{
public:
  constexpr void forward( std::uint32_t& a, std::uint32_t& b ) const noexcept
  {
    const auto& RC = ROUND_CONSTANTS;

    std::uint32_t A = a;
    std::uint32_t B = b;

    std::uint32_t CD0{};
    std::uint32_t CD1{};
    std::uint32_t CD2{};
    std::uint32_t CD3{};

    // ======================================================================
    // Subround 0 : Constant subtract -> B-to-A injection -> ARX add -> bridge
    // 第 0 子轮: 常量模减 -> B 到 A 的非线性注入 -> ARX 模加 -> 交叉桥接
    // ======================================================================

    // Step 0.1: fixed-public constant subtraction on B
    // 步骤 0.1: 对 B 执行固定公开常量模减。
    //
    // This is hardcore.
    // Constant addition/subtraction inside an ARX-style trail is still costly to model precisely.
    // Existing differential treatments are possible, but practical low-complexity and broadly reusable
    // linear/correlation-oriented models are still awkward for this kind of construction.
    B -= RC[ 1 ];

    // Step 0.2: B-to-A nonlinear injection, before the first cross-branch bridge.
    // 步骤 0.2: B -> A 非线性注入，放在第一组交叉桥接之前。
    //
    // Scheduling point:
    // This is the V6.5 second schedule.  The injection consumes the already
    // constant-subtracted B state, and its C/D derivatives feed the following
    // modular addition.  This prevents the old zero-injection window where the
    // solver could keep the injection source inactive and reactivate it later.
    {
      const auto [ C0, D0 ] = cd_injection_from_B( B );

      CD0 = ( C0 << 2 ) ^ ( D0 >> 2 );
      CD1 = ( C0 >> 5 ) ^ ( D0 << 5 );

      A ^= std::rotl( B, 24 )
        ^  std::rotl( C0, 16 )
        ^  std::rotl( B, 8 );
    }

    // Step 0.3: CD-driven modular addition into A
    // 步骤 0.3: 由注入层派生的 CD0/CD1 驱动 A 侧模加。
    A += ( std::rotl( CD0, 31 ) ^ std::rotl( CD1, 17 ) ^ RC[ 0 ] );

    // Step 0.4: cross-branch bridge, line 0
    // 步骤 0.4: 交叉桥接第 0 行，把 A 的状态压回 B。
    B ^= std::rotl( A, CROSS_XOR_ROT_R0 ) ^ RC[ 4 ];

    // Step 0.5: cross-branch bridge, line 1
    // 步骤 0.5: 交叉桥接第 1 行，使后续 A-to-B 注入消耗桥接后的 A。
    //
```

Lines 181–270.

```
    // IMPORTANT:
    // This bridge must happen before the A-to-B nonlinear injection.
    A ^= std::rotl( B, CROSS_XOR_ROT_R1 );

    // ======================================================================
    // Subround 1 : Constant subtract -> A-to-B injection -> ARX add -> output bridge
    // 第 1 子轮: 常量模减 -> A 到 B 的非线性注入 -> ARX 模加 -> 输出桥接
    // ======================================================================

    // Step 1.1: fixed-public constant subtraction on A
    // 步骤 1.1: 对 A 执行固定公开常量模减。
    //
    // This is hardcore.
    // Constant addition/subtraction inside an ARX-style trail is still costly to model precisely.
    // Existing differential treatments are possible, but practical low-complexity and broadly reusable
    // linear/correlation-oriented models are still awkward for this kind of construction.
    A -= RC[ 6 ];

    // Step 1.2: A-to-B nonlinear injection
    // 步骤 1.2: A -> B 非线性注入。
    //
    // The A branch has already passed through both bridge lines, so the injector
    // no longer sees the same clean window that the abandoned first V6.5 schedule exposed.
    {
      const auto [ C1, D1 ] = cd_injection_from_A( A );

      CD2 = ( C1 >> 3 ) ^ ( D1 << 3 );
      CD3 = ( C1 << 1 ) ^ ( D1 >> 1 );

      B ^= std::rotr( A, 24 )
        ^  std::rotr( D1, 16 )
        ^  std::rotr( A, 8 );
    }

    // Step 1.3: CD-driven modular addition into B
    // 步骤 1.3: 由注入层派生的 CD2/CD3 驱动 B 侧模加。
    B += ( CD2 ^ CD3 ^ RC[ 5 ] );

    // Step 1.4: final asymmetric XOR/ROT bridge
    // 步骤 1.4: 最终非对称 XOR/ROT 桥接。
    A ^= std::rotl( B, 5 ) ^ RC[ 9 ];
    B ^= std::rotl( A, 25 );

    // Final light whitening (reversible)
    // 最终轻量白化（可逆）。
    A ^= RC[ 10 ];
    B ^= RC[ 11 ];

    a = A;
    b = B;
  }

  constexpr void backward( std::uint32_t& a, std::uint32_t& b ) const noexcept
  {
    const auto& RC = ROUND_CONSTANTS;

    std::uint32_t A = a;
    std::uint32_t B = b;

    // ======================================================================
    // Undo final whitening
    // 撤销最终轻量白化
    // ======================================================================
    B ^= RC[ 11 ];
    A ^= RC[ 10 ];

    // ======================================================================
    // Undo Subround 1
```

```
    // 撤销第 1 子轮
    // ================================================================

    // Reverse Step 1.4.2: undo final B update
    // 逆步骤 1.4.2: 撤销最终 B 侧桥接。
    B ^= std::rotl( A, 25 );

    // Reverse Step 1.4.1: undo final A update
    // 逆步骤 1.4.1: 撤销最终 A 侧桥接。
    //
    // In forward direction this update used the post-addition B state,
    // so in backward direction it must be undone before recovering B.
    A ^= std::rotl( B, 5 ) ^ RC[ 9 ];

    // Reverse Step 1.3 + 1.2: undo B modular addition, then undo A-to-B injection.
    // 逆步骤 1.3 + 1.2: 先撤销 B 侧模加，再撤销 A -> B 注入。
    {
      const auto [ C1, D1 ] = cd_injection_from_A( A );

      const std::uint32_t CD2 = ( C1 >> 3 ) ^ ( D1 << 3 );
      const std::uint32_t CD3 = ( C1 << 1 ) ^ ( D1 >> 1 );
```

Lines 271–360.

```
      B -= ( CD2 ^ CD3 ^ RC[ 5 ] );

      B ^= std::rotr( A, 24 )
        ^  std::rotr( D1, 16 )
        ^  std::rotr( A, 8 );
    }

    // Reverse Step 1.1: undo fixed-public constant subtraction on A
    // 逆步骤 1.1: 撤销 A 侧固定公开常量模减。
    //
    // This is hardcore.
    // Constant addition/subtraction inside an ARX-style trail is still costly to model precisely.
    // Existing differential treatments are possible, but practical low-complexity and broadly reusable
    // linear/correlation-oriented models are still awkward for this kind of construction.
    A += RC[ 6 ];

    // ================================================================
    // Undo Subround 0
    // 撤销第 0 子轮
    // ================================================================

    // Reverse Step 0.5: undo cross-branch bridge, line 1
    // 逆步骤 0.5: 撤销交叉桥接第 1 行。
    A ^= std::rotl( B, CROSS_XOR_ROT_R1 );

    // Reverse Step 0.4: undo cross-branch bridge, line 0
    // 逆步骤 0.4: 撤销交叉桥接第 0 行。
    //
    // This restores the B state consumed by the original B-to-A injection.
    B ^= std::rotl( A, CROSS_XOR_ROT_R0 ) ^ RC[ 4 ];

    // Reverse Step 0.3 + 0.2: undo A modular addition, then undo B-to-A injection.
    // 逆步骤 0.3 + 0.2: 先撤销 A 侧模加，再撤销 B -> A 注入。
    {
      const auto [ C0, D0 ] = cd_injection_from_B( B );

      const std::uint32_t CD0 = ( C0 << 2 ) ^ ( D0 >> 2 );
      const std::uint32_t CD1 = ( C0 >> 5 ) ^ ( D0 << 5 );

      A -= ( std::rotl( CD0, 31 )
        ^  std::rotl( CD1, 17 )
        ^  RC[ 0 ] );

      A ^= std::rotl( B, 24 )
        ^  std::rotl( C0, 16 )
        ^  std::rotl( B, 8 );
    }

    // Reverse Step 0.1: undo fixed-public constant subtraction on B
    // 逆步骤 0.1: 撤销 B 侧固定公开常量模减。
    //
    // This is hardcore.
    // Constant addition/subtraction inside an ARX-style trail is still costly to model precisely.
    // Existing differential treatments are possible, but practical low-complexity and broadly reusable
    // linear/correlation-oriented models are still awkward for this kind of construction.
    B += RC[ 1 ];

    a = A;
    b = B;
  }

private:
  // ==== NeoAlzette ARX-box constants / NeoAlzette ARX-box 常量 ====
  static inline constexpr auto ROUND_CONSTANTS = std::to_array<std::uint32_t>
  ({
    // 1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181 (Fibonacci numbers)
    // Concatenation of Fibonacci numbers : 1235813213455891442333776109871597258441 81
    // Hexadecimal : 16b2c40bc117176a0f9a2598a1563aca6d5
    0x16B2C40B, 0xC117176A, 0x0F9A259B, 0xA1563ACA,

    /*
      Mathematical Constants - Millions of Digits
      http://www.numberworld.org/constants.html
    */

    // π Pi (3.243f6a8885a308d313198a2e03707344)
    0x243F6A88, 0x85A308D3, 0x13198A2E, 0x03707344,
    // φ Golden ratio (1.9e3779b97f4a7c15f39cc0605cedc834)
    0x9E3779B9, 0x7F4A7C15, 0xF39CC060, 0x5CEDC834,
    // e Natural Constant (2.b7e151628aed2a6abf7158809cf4f3c7)
    0xB7E15162, 0x8AED2A6A, 0xBF715880, 0x9CF4F3C7
  });

  // ================================================================
  // NeoAlzette Cross-branch XOR/ROT bridge constants
  //
  // Ordered bridge assignment:
  //   R0 = 22, R1 = 13
  //
  // Structural rule:
```

Lines 361–450.

```
  //   gcd(((R0 + R1) mod 32), 32) == 1
  //
  // Since 32 = 2^5, this is equivalent to requiring the bridge sum to be odd.
  // For the current pair:
  //   (22 + 13) mod 32 = 3, gcd(3, 32) = 1.
  //
  // Bottom line:
  // do not "clean up", "simplify", or "make symmetric" these constants unless
  // the full differential / linear / trace tooling is rerun and checked again.
  // ================================================================
  static constexpr int CROSS_XOR_ROT_R0 = 22;
```

```
static constexpr int CROSS_XOR_ROT_R1 = 13;
static constexpr int CROSS_XOR_ROT_SUM = ( ( CROSS_XOR_ROT_R0 + CROSS_XOR_ROT_R1 ) & 31 );
static_assert( ( CROSS_XOR_ROT_SUM & 1 ) == 1, "CROSS_XOR_ROT_R0 + CROSS_XOR_ROT_R1 must be odd (coprime with 32) to avoid large rotation fixed-point subspaces." );

// ============================================================================
// Dynamic diffusion masks (rotation XOR family)
//
// These are the V6.5 second-version linear diffusion layers carried over
// from NeoAlzetteCore.  The selected pair was screened by the linear-box-search
// tooling with confirmed differential/linear combined branch upper bound 12.
// ============================================================================
static constexpr std::uint32_t generate_dynamic_diffusion_mask0( std::uint32_t x ) noexcept
{
  const std::uint32_t v0 = x;
  const std::uint32_t v1 = v0 ^ std::rotl( v0, 2 );
  const std::uint32_t v2 = v0 ^ std::rotl( v1, 17 );
  const std::uint32_t v3 = v0 ^ std::rotl( v2, 4 );
  const std::uint32_t v4 = v3 ^ std::rotl( v3, 24 );
  return v2 ^ std::rotl( v4, 7 );
}

static constexpr std::uint32_t generate_dynamic_diffusion_mask1( std::uint32_t x ) noexcept
{
  const std::uint32_t v0 = x;
  const std::uint32_t v1 = v0 ^ std::rotr( v0, 2 );
  const std::uint32_t v2 = v0 ^ std::rotr( v1, 17 );
  const std::uint32_t v3 = v0 ^ std::rotr( v2, 4 );
  const std::uint32_t v4 = v3 ^ std::rotr( v3, 24 );
  return v2 ^ std::rotr( v4, 7 );
}

// ============================================================================
// Precomputed constant views for the injection layer
//
// C++20 note:
// These are real inline constexpr data members, not accessor functions.
// For the two mask constants we use consteval lambdas, because class-scope
// data-member initializers cannot call a static member function before the
// class definition is complete.  The generated constants are still compile-time
// values and do not introduce runtime storage or runtime initialization.
// ============================================================================
static inline constexpr std::uint32_t RC7_R24  = std::rotr( ROUND_CONSTANTS[ 7 ], 24 );
static inline constexpr std::uint32_t RC8_R24  = std::rotr( ROUND_CONSTANTS[ 8 ], 24 );
static inline constexpr std::uint32_t RC13_R24 = std::rotr( ROUND_CONSTANTS[ 13 ], 24 );
static inline constexpr std::uint32_t RC2_L8   = std::rotl( ROUND_CONSTANTS[ 2 ], 8 );
static inline constexpr std::uint32_t RC3_L8   = std::rotl( ROUND_CONSTANTS[ 3 ], 8 );
static inline constexpr std::uint32_t RC12_L8  = std::rotl( ROUND_CONSTANTS[ 12 ], 8 );

static inline constexpr std::uint32_t MASK0_RC7 = []() consteval
{
  const std::uint32_t v0 = ROUND_CONSTANTS[ 7 ];
  const std::uint32_t v1 = v0 ^ std::rotl( v0, 2 );
  const std::uint32_t v2 = v0 ^ std::rotl( v1, 17 );
  const std::uint32_t v3 = v0 ^ std::rotl( v2, 4 );
  const std::uint32_t v4 = v3 ^ std::rotl( v3, 24 );
  return v2 ^ std::rotl( v4, 7 );
}();

static inline constexpr std::uint32_t MASK1_RC2 = []() consteval
{
  const std::uint32_t v0 = ROUND_CONSTANTS[ 2 ];
  const std::uint32_t v1 = v0 ^ std::rotr( v0, 2 );
  const std::uint32_t v2 = v0 ^ std::rotr( v1, 17 );
  const std::uint32_t v3 = v0 ^ std::rotr( v2, 4 );
  const std::uint32_t v4 = v3 ^ std::rotr( v3, 24 );
  return v2 ^ std::rotr( v4, 7 );
}();

// ============================================================================
// Cross-branch injection (value domain with constants)
//
// Design rationale:
// - add a second nonlinearity source beyond the carry/borrow effects of the main ARX path;
// - keep the injector lightweight;
// - preserve reversibility at the round level via cross-branch XOR-style injection,
//   so the local function itself does not need to be invertible.
// ============================================================================

// Feistel-like nonlinear branch injection: B -> A
```

Lines 451–540.

```
// Local nonlinear mixing function from B into A (PRF-like role, not a formal PRF claim)
[[nodiscard]] static constexpr std::pair<std::uint32_t, std::uint32_t> cd_injection_from_B( std::uint32_t B ) noexcept
{
  const std::uint32_t companion0 = std::rotr( B, 24 );

  const std::uint32_t mask = generate_dynamic_diffusion_mask0( B );
  const std::uint32_t companion_mask = std::rotr( mask, 24 ) ^ MASK0_RC7;
  const std::uint32_t mask_r1 = std::rotr( mask, 5 );

  const std::uint32_t x0 = companion0 ^ mask;
  const std::uint32_t x1 = B ^ mask;
  const std::uint32_t view = companion0 ^ companion_mask;
  const std::uint32_t bridge_state = std::rotr( B, 19 ) ^ ( B << 9 );

  const std::uint32_t q_state_na = RC7_R24 ^ ( ~( B & mask ) );
  const std::uint32_t q_comp_no  = companion0 ^ B ^ RC8_R24 ^ ( ~( companion0 | mask_r1 ) );
  const std::uint32_t q_bridge   = bridge_state ^ B ^ RC13_R24 ^ ( ~( bridge_state & companion_mask ) );
  const std::uint32_t q_shared   = q_state_na ^ q_comp_no;

  const std::uint32_t cross_q = ( B ^ mask_r1 ) & std::rotr( mask ^ companion_mask, 7 );
  const std::uint32_t anti_q  = ( ( x1 >> 3 ) ^ ( view >> 5 ) ^ mask_r1 ) & ( B ^ std::rotr( x0, 11 ) );

  const std::uint32_t c = q_shared ^ std::rotr( q_comp_no, 5 ) ^ std::rotr( q_comp_no, 11 ) ^ anti_q;
  const std::uint32_t d = q_shared ^ std::rotr( q_state_na, 5 ) ^ std::rotr( q_bridge, 13 ) ^ cross_q ^ anti_q;
  return { c, d };
}

// Feistel-like nonlinear branch injection: A -> B
// Local nonlinear mixing function from A into B (PRF-like role, not a formal PRF claim)
[[nodiscard]] static constexpr std::pair<std::uint32_t, std::uint32_t> cd_injection_from_A( std::uint32_t A ) noexcept
{
  const std::uint32_t companion0 = std::rotl( A, 8 );

  const std::uint32_t mask = generate_dynamic_diffusion_mask1( A );
  const std::uint32_t companion_mask = std::rotl( mask, 8 ) ^ MASK1_RC2;
  const std::uint32_t mask_r1 = std::rotr( mask, 5 );

  const std::uint32_t x0 = companion0 ^ mask;
  const std::uint32_t x1 = A ^ mask;
  const std::uint32_t view = companion0 ^ companion_mask;
  const std::uint32_t bridge_state = std::rotl( A, 19 ) ^ ( A >> 9 );

  const std::uint32_t q_state_no = RC2_L8 ^ ( ~( A | mask ) );
  const std::uint32_t q_comp_na  = companion0 ^ A ^ RC3_L8 ^ ( ~( companion0 & mask_r1 ) );
  const std::uint32_t q_bridge   = bridge_state ^ A ^ RC12_L8 ^ ( ~( bridge_state | companion_mask ) );
  const std::uint32_t q_shared   = q_state_no ^ q_comp_na;

  const std::uint32_t cross_q = ( A ^ mask_r1 ) & std::rotl( mask ^ companion_mask, 13 );
  const std::uint32_t anti_q  = ( ( x1 << 3 ) ^ ( view << 5 ) ^ mask_r1 ) | ( A ^ std::rotl( x0, 11 ) );
```

```
    const std::uint32_t c = q_shared ^ std::rotl( q_comp_na, 5 ) ^ std::rotl( q_comp_na, 11 ) ^ anti_q;
    const std::uint32_t d = q_shared ^ std::rotl( q_state_no, 5 ) ^ std::rotl( q_bridge, 13 ) ^ cross_q ^ anti_q;
    return { c, d };
  }
};

// ------------------------------------------------------------------------
// Generate and cache per-round key states
// ------------------------------------------------------------------------
// This function derives round-dependent key material from two persistent
// XorConstantRotation member instances: `prng` and `prng_second`.
//
// Important usage semantics:
// - No local XCR instances are created here.
// - Both generators are stateful member objects and are consumed continuously
//   across rounds inside this function.
// - Therefore, round-to-round variation comes from BOTH:
//      (1) the explicit per-round inputs derived from `number_once` and `round`,
//      (2) the continuously evolving internal states of the two XCR instances.
//
// Current XCR API behavior matters here:
// - `GenerateSubKey128(input)` mutates the XCR internal state, returns
//   a 128-bit view {x ^ y, z ^ w}, and explicitly advances that XCR's
//   public `counter`.
// - So the main source of progression here is BOTH:
//      (1) state evolution,
//      (2) explicit counter stepping on every public call.
//
// Construction overview:
// - Left generator input : input_left  = number_once ^ round
// - Right generator input: input_right = (number_once ^ (round << 1)) ^ (round >> 1)
// - Each generator emits 128 bits, producing four 64-bit words total: a,b,c,d.
// - These words are folded into:
//      * round subkey (first / second lane),
//      * 2-bit choice_function,
//      * two rotation amounts.
//
// No extra ad-hoc round constants are injected here;
// round only affects position / routing.
void LittleOaldresPuzzle_Cryptic::GenerateAndStoreKeyStates(const Key128 key_128bit, const std::uint64_t number_once)
```

Lines 541–630.

```
{
  // 注意：这里不构造任何 XorConstantRotation 实例，只使用成员 prng / prng_second
  for (std::uint64_t round = 0; round < rounds; ++round)
  {
    KeyState& key_state = KeyStates[round];

    const std::uint64_t input_left  = number_once ^ round;
    const std::uint64_t input_right = (number_once ^ (round << 1)) ^ (round >> 1);

    // 两个成员实例各吐 128-bit：合计 4×64
    const auto out_left  = prng.GenerateSubKey128(input_left);
    const auto out_right = prng_second.GenerateSubKey128(input_right);

    const std::uint64_t a = out_left.a;
    const std::uint64_t b = out_left.b;
    const std::uint64_t c = out_right.a;
    const std::uint64_t d = out_right.b;

    // ------------------------------------------------------------------
    // ARX-shaped subkey folding
    //
    // IMPORTANT:
    // We intentionally use the form
    //   (key +/- xcr_word) ^ rotated_xcr_word
    // and NOT
    //   key +/- (xcr_word ^ rotated_xcr_word).
    //
    // Reason:
    // - The chosen form keeps the modular add/sub core isolated first,
    //   then applies a rotated-XOR outer perturbation.
    // - This preserves a cleaner modeling boundary for differential / linear /
    //   dependency-bit / state-machine style analysis.
    // - The rejected alternative would push the mixed XCR value directly into the
    //   carry/borrow chain, entangling XCR-side structure with the modular core
    //   and forcing a broader rework of the subkey model.
    //
    // In short:
    //   chosen   : ARX core first, XOR shell later
    //   rejected : XCR mixture first, carry entanglement later
    // ------------------------------------------------------------------
    // round 参与“位置”，不引入额外常量
    // 生成 128-bit subkey (Key128 的 first/second)
    key_state.subkey.first  = (key_128bit.first  + a) ^ std::rotr(c, static_cast<int>(round & 63ULL));
    key_state.subkey.second = (key_128bit.second - b) ^ std::rotr(d, static_cast<int>((round + 1) & 63ULL));

    // choice: 只要 2-bit
    key_state.choice_function = (a ^ b ^ c ^ d) & 3ULL;

    // rotation amounts: 从同一轮输出切片 (6+6)
    const std::uint64_t rot_pool =
      (a ^ b) ^ (c ^ d) ^ ^
      std::rotl(key_state.subkey.first, 1) ^
      std::rotl(key_state.subkey.second, 3);

    key_state.bit_rotation_amount_a = ( rot_pool        ) & 63ULL;  // bits 0..5
    key_state.bit_rotation_amount_b = ((rot_pool >> 6 ) ) & 63ULL;  // bits 6..11
  }
}

static inline uint64_t pack64( uint32_t hi, uint32_t lo )
{
  return ( uint64_t( hi ) << 32 ) | uint64_t( lo );
}

static inline void unpack64( uint64_t v, uint32_t& hi, uint32_t& lo )
{
  hi = uint32_t( v >> 32 );
  lo = uint32_t( v );
}

// Return 0xFFFFFFFFFFFFFFFF iff x == y, else 0x0. Constant-time, branchless.
inline uint64_t ConstantTimeEqualMask( uint64_t x, uint64_t y )
{
  uint64_t q = x ^ y;
  q |= ( uint64_t )0 - q;   // q | (-q)
  q >>= 63;         // 0 if equal, 1 otherwise
  return q - 1;       // 0xFFFFFFFFFFFFFFFF if equal, 0x0 otherwise
}

//Mix Linear Transform Layer (Forward)
inline void LittleOaldresPuzzle_Cryptic::MixLinearTransform_Forward
(
  uint64_t& lane0, uint64_t& lane1, const KeyState& current_key_state
)
{
  /*
    switch ( current_key_state.choice_function & 3ULL )
    {
    case 0:
```

```
    lane0 ^= current_key_state.subkey.first;
```

Lines 631–720.

```
      lane1 ^= current_key_state.subkey.second;
      break;
    case 1:
      lane0 = (~lane0) ^ current_key_state.subkey.first;
      lane1 = (~lane1) ^ current_key_state.subkey.second;
      break;
    case 2:
      lane0 = std::rotl( lane0, current_key_state.bit_rotation_amount_b );
      lane1 = std::rotl( lane1, current_key_state.bit_rotation_amount_b );
      break;
    case 3:
      lane0 = std::rotr( lane0, current_key_state.bit_rotation_amount_b );
      lane1 = std::rotr( lane1, current_key_state.bit_rotation_amount_b );
      break;
    }
  */

  const uint64_t& choice_function = current_key_state.choice_function;
  const uint64_t& subkey_first = current_key_state.subkey.first;
  const uint64_t& subkey_second = current_key_state.subkey.second;

  const uint64_t lane0_case0 = (lane0) ^ subkey_first;
  const uint64_t lane1_case0 = (lane1) ^ subkey_second;

  const uint64_t lane0_case1 = (~lane0) ^ subkey_first;
  const uint64_t lane1_case1 = (~lane1) ^ subkey_second;

  const uint64_t lane0_case2 = std::rotl(lane0, current_key_state.bit_rotation_amount_b & 63ULL);
  const uint64_t lane1_case2 = std::rotl(lane1, current_key_state.bit_rotation_amount_b & 63ULL);

  const uint64_t lane0_case3 = std::rotr(lane0, current_key_state.bit_rotation_amount_b & 63ULL);
  const uint64_t lane1_case3 = std::rotr(lane1, current_key_state.bit_rotation_amount_b & 63ULL);

  const uint64_t m0 = ConstantTimeEqualMask(choice_function & 3ULL, 0ULL);
  const uint64_t m1 = ConstantTimeEqualMask(choice_function & 3ULL, 1ULL);
  const uint64_t m2 = ConstantTimeEqualMask(choice_function & 3ULL, 2ULL);
  const uint64_t m3 = ConstantTimeEqualMask(choice_function & 3ULL, 3ULL);

  lane0 = (lane0_case0 & m0) | (lane0_case1 & m1) | (lane0_case2 & m2) | (lane0_case3 & m3);
  lane1 = (lane1_case0 & m0) | (lane1_case1 & m1) | (lane1_case2 & m2) | (lane1_case3 & m3);
}

// Mix Linear Transform Layer (Backward)
inline void LittleOaldresPuzzle_Cryptic::MixLinearTransform_Backward
(
  uint64_t& lane0, uint64_t& lane1, const KeyState& current_key_state
)
{
  /*
    switch ( current_key_state.choice_function & 3ULL )
    {
    case 0:
      lane0 ^= current_key_state.subkey.first;
      lane1 ^= current_key_state.subkey.second;
      break;
    case 1:
      lane0 = (~lane0) ^ current_key_state.subkey.first;
      lane1 = (~lane1) ^ current_key_state.subkey.second;
      break;
    case 2:
      lane0 = std::rotr( lane0, current_key_state.bit_rotation_amount_b );
      lane1 = std::rotr( lane1, current_key_state.bit_rotation_amount_b );
      break;
    case 3:
      lane0 = std::rotl( lane0, current_key_state.bit_rotation_amount_b );
      lane1 = std::rotl( lane1, current_key_state.bit_rotation_amount_b );
      break;
    }
  */

  const uint64_t& choice_function = current_key_state.choice_function;
  const uint64_t& subkey_first = current_key_state.subkey.first;
  const uint64_t& subkey_second = current_key_state.subkey.second;

  const uint64_t lane0_case0 = (lane0) ^ subkey_first;
  const uint64_t lane1_case0 = (lane1) ^ subkey_second;

  const uint64_t lane0_case1 = (~lane0) ^ subkey_first;
  const uint64_t lane1_case1 = (~lane1) ^ subkey_second;

  const uint64_t lane0_case2 = std::rotr(lane0, current_key_state.bit_rotation_amount_b & 63ULL);
  const uint64_t lane1_case2 = std::rotr(lane1, current_key_state.bit_rotation_amount_b & 63ULL);

  const uint64_t lane0_case3 = std::rotl(lane0, current_key_state.bit_rotation_amount_b & 63ULL);
  const uint64_t lane1_case3 = std::rotl(lane1, current_key_state.bit_rotation_amount_b & 63ULL);

  const uint64_t m0 = ConstantTimeEqualMask(choice_function & 3ULL, 0ULL);
  const uint64_t m1 = ConstantTimeEqualMask(choice_function & 3ULL, 1ULL);
  const uint64_t m2 = ConstantTimeEqualMask(choice_function & 3ULL, 2ULL);
  const uint64_t m3 = ConstantTimeEqualMask(choice_function & 3ULL, 3ULL);
```

Lines 721–810.

```
  lane0 = (lane0_case0 & m0) | (lane0_case1 & m1) | (lane0_case2 & m2) | (lane0_case3 & m3);
  lane1 = (lane1_case0 & m0) | (lane1_case1 & m1) | (lane1_case2 & m2) | (lane1_case3 & m3);
}

Block128 LittleOaldresPuzzle_Cryptic::EncryptionCoreFunction( const Block128 data, const Key128 key, const std::uint64_t number_once )
{
  // 生成并缓存密钥状态（保持实现不变）
  GenerateAndStoreKeyStates( key, number_once );

  NeoAlzetteSubstitutionBox SubstitutionBox;

  // 128-bit 状态按两条 64-bit 车道存放
  uint64_t lane0 = data.first;   // (w0 || w1)
  uint64_t lane1 = data.second;  // (w2 || w3)

  // 拆成 4×32（注意：hi 在前、lo 在后）
  uint32_t w0, w1, w2, w3;

  for ( size_t round = 0; round < rounds; ++round )
  {
    const KeyState& current_key_state = KeyStates[ round ];

    // Add Round Key
    lane0 ^= current_key_state.subkey.first;
    lane1 ^= current_key_state.subkey.second;

    unpack64( lane0, w0, w1 );
    unpack64( lane1, w2, w3 );

    /*
      NeoAlzette ARX Layer (Forward)
```

```
      ——采用 “对角配对”：(w0,w2) 与 (w1,w3)，跨车道混合
      */
      SubstitutionBox.forward( w0, w2 );
      SubstitutionBox.forward( w1, w3 );

      // 重新打包回两条 64-bit 车道
      lane0 = pack64( w0, w1 );
      lane1 = pack64( w2, w3 );

      /* Keyed Switching Layer - MixLinearTransform (Forward) */
      MixLinearTransform_Forward(lane0, lane1, current_key_state);

      /* Keyed Switching Layer - Random Bit Tweak (Nonlinear)(Forward) */
      lane0 ^= ( uint64_t( 1 ) << current_key_state.bit_rotation_amount_a );
      lane1 ^= ( uint64_t( 1 ) << ( 63 - current_key_state.bit_rotation_amount_a ) );
    }

    return Block128 { lane0, lane1 };
  }

  Block128 LittleOaldresPuzzle_Cryptic::DecryptionCoreFunction( const Block128 data, const Key128 key, const std::uint64_t number_once )
  {
    // 生成并缓存密钥状态（保持实现不变）
    GenerateAndStoreKeyStates( key, number_once );

    NeoAlzetteSubstitutionBox SubstitutionBox;

    uint64_t lane0 = data.first;
    uint64_t lane1 = data.second;

    // NeoAlzette ARX Layer (Backward)
    uint32_t w0, w1, w2, w3;

    for ( size_t round = rounds; round > 0; --round )
    {
      const KeyState& current_key_state = KeyStates[ round - 1 ];

      /* Keyed Switching Layer^{-1} - Random Bit Tweak (Nonlinear)(Backward) */
      lane0 ^= ( uint64_t( 1 ) << current_key_state.bit_rotation_amount_a );
      lane1 ^= ( uint64_t( 1 ) << ( 63 - current_key_state.bit_rotation_amount_a ) );

      /* Keyed Switching Layer^{-1} - MixLinearTransform (Backward) */
      MixLinearTransform_Backward(lane0, lane1, current_key_state);

      unpack64( lane0, w0, w1 );
      unpack64( lane1, w2, w3 );

      SubstitutionBox.backward( w1, w3 );
      SubstitutionBox.backward( w0, w2 );

      lane0 = pack64( w0, w1 );
      lane1 = pack64( w2, w3 );

      // Subtract Round key
      lane0 ^= current_key_state.subkey.first;
      lane1 ^= current_key_state.subkey.second;
    }
```

Lines 811–815.

```
    return Block128 { lane0, lane1 };
  }
} // TwilightDreamOfMagical

}
```