

Whitepaper

Canary

Funded discrete-log rungs on Solana as an on-chain quantum signal, with the markets and vaults that settle on it.

v0 · October 2026

Contents

Abstract

1. Introduction
 2. Background: what it costs to break a curve
 3. The ladder
 4. Parameter generation
 5. The claim protocol
 6. The canary_rungs program
 7. Bounty vaults
 8. Rung-event markets
 9. The canary_markets program
 10. Q-Vault
 11. The canary_vault program
 12. Security design
 13. Off-chain infrastructure
 14. Integration
 15. Economics
 16. Upgrade authority and deployment
 17. Roadmap
-

References

- Appendix A. Account layouts
 - Appendix B. Instruction reference
 - Appendix C. Error codes
 - Appendix D. Rung parameters
 - Appendix E. Status feed and app configuration
 - Appendix F. Glossary
-

Abstract

Canary is a protocol on Solana that turns progress against elliptic-curve cryptography into an account a program can read. It funds discrete-logarithm puzzles, called rungs, on six curves chosen so that a quantum computer reaches them before a classical one and in a known order: three prime-order curves of the secp256k1 family over 160-, 192- and 224-bit fields; a classical control over secp256k1's own 256-bit field with a 160-bit subgroup; and nothing-up-my-sleeve points on secp256k1 and Ed25519 themselves. Every rung's points are derived from a public seed by hash-to-curve under a published transcript, so nobody knows their discrete logarithm.

A solver who finds a rung's discrete logarithm claims it with a Schnorr proof whose challenge commits to the rung, the payout address and a job nonce. The proof never reveals the logarithm, and nobody who observes it can redirect the payout. On-chain verification is staged across transactions for the rungs whose field arithmetic Solana has no syscall for, and done in one transaction with native curve syscalls for the terminal rungs. When verification completes, the rung's `RungState` account changes from `Open` to `Solved`, recording the time the claim began, and the rung's bounty pays the recipient in the same instruction.

Three things are built on that one bit. Bounty vaults, funded by sponsors and irrevocable until 2040, pay solvers and give every capable lab a reason to disclose. Rung-event markets, binary YES and NO tokens fully collateralized in USDC, resolve permissionlessly by reading the rung, and their prices across expiries form a market-implied date curve for quantum capability. Q-Vaults hold funds under an Ed25519 key with a rate-limited instant budget and a delay, and a Falcon-512 key that can always spend and veto; when the vault's trigger rung reads `Solved`, the Ed25519 path is rejected permanently.

The protocol's token, CANARY, has a fixed supply launched in full to the public; the protocol's share of every settlement fee buys it and burns it, and it never touches the trigger, collateral or resolution.

This paper specifies the ladder and its parameter generation, the claim protocol and its verification, the three programs with their accounts and instructions, the market and fee mathematics, the token, the vault's key and authorization scheme, the security design, the off-chain infrastructure and the deployment model.

1. Introduction

1.1 A dated risk with no price

In 2026, estimates of the quantum resources needed to break elliptic-curve keys started to come with dates attached. Google Quantum AI estimates that solving the discrete logarithm on a 256-bit curve needs fewer than 1,200 logical qubits and 90 million Toffoli gates, or fewer than 500,000 physical qubits running for minutes on superconducting hardware [6][7]. Caltech and Oratomic estimate 10,000 to 26,000 neutral-atom qubits, with a P-256 run taking days [8]. Project Eleven places Q-Day possibly as early as 2030 [4]. Allianz counts about 6 million BTC, some 30 % of supply and about \$400 billion, behind public keys that are already exposed on chain [3], and Project Eleven puts the value secured by elliptic-curve cryptography across digital assets above \$2.5 trillion [5].

The risk has a size and, increasingly, a range of dates. It has no price. There is no market in which a holder can buy protection against it, no instrument whose value moves with the probability of a cryptographically relevant quantum computer (CRQC) arriving by a given year, and no market-implied date to anchor planning. Experts surveyed in the literature put the chance of a CRQC within ten years at 28 % to 49 % [3]; nobody can trade that range.

1.2 Signals a program can't read

The evidence that does exist arrives in forms that software cannot act on:

Form	Example	Why a program can't act on it
Papers and resource estimates	Babbush et al. (2026) [7], Cain et al. (2026) [8]	Estimates, not events; they move expectations but trigger nothing
Judged prizes	Project Eleven's Q-Day Prize, won with a 15-bit key in April 2026 [5]	Judged off chain; toy sizes are classically trivial
Zero-knowledge proofs of cryptanalysis	Google's 2026 proof, whose first version had a soundness bug found by Trail of Bits [7][11]	A proof system's soundness becomes the trigger's soundness
Unclaimed classical puzzles	Bitcoin puzzles #135–#160, ~916 BTC across 78 keys [14][15]	Range-restricted keys measure classical progress only; Shor gains nothing from a range restriction [10]
A 256-bit canary	BitMEX Research's proposal for a NUMS address on Bitcoin [1]	Fires at the moment theft is possible, leaving no lead time

A protocol that wants to react automatically — freeze a bridge, migrate a treasury, pay out a hedge — needs something objective to read. A committee's vote is not objective, an oracle's report adds a party to trust, and a 256-bit canary is objective but arrives too late to be useful.

1.3 What Canary provides

Canary provides an on-chain fact with three properties:

1. **Objective.** A rung reads `Solved` if and only if its program verified a proof of knowledge of the rung's discrete logarithm. No party decides it.
2. **Ordered.** The rungs are chosen so that a quantum computer reaches them in a known order, from Q-160 at 1,466 logical qubits to the terminal rungs at about 2,330, and so that the lower rungs fall before 256-bit keys are in danger.
3. **Funded.** Each rung carries a bounty that pays its solver, so anyone who can solve a rung has a reason to claim it rather than keep the result quiet.

On top of that fact, Canary builds the two instruments the problem lacks: rung-event markets that price the risk at fixed dates, and Q-Vaults that act on it automatically.

1.4 Design goals

Goal	How Canary meets it
No oracle, no committee	Markets and vaults read <code>RungState</code> directly; resolution is a permissionless instruction
Nobody knows the answer	Rung points come from hash-to-curve on public seeds with a published, reproducible transcript
A claim can't be stolen	The proof's challenge commits to the recipient; the discrete logarithm is never revealed
The signal carries its cause	A classical control rung costs the same classically as the lowest quantum rung but needs the qubits of a full 256-bit break
Markets can't be gamed by brute force	Per-rung open interest is capped at no more than 1 % of the rung's low-end classical break cost
Collateral holds when it has to pay	USDC only; crypto-native collateral would fall with the event it insures
Every wallet can sign	Every instruction fits a legacy 1,232-byte transaction
Credible neutrality	The rungs program becomes immutable after audit; the CANARY token is never collateral, a resolution input or a vote on rungs and vaults

1.5 Structure of this paper

Section 2 sets out the cost models behind the ladder. Section 3 defines the ladder and how a fall is read; Section 4 specifies how rung parameters are generated. Sections 5 to 7 cover the claim protocol, the rungs program and the bounty vaults. Sections 8 and 9 cover rung-event markets and their program; Sections 10 and 11 cover Q-Vault and its program. Section 12 describes the security design, Section 13 the off-chain infrastructure, Section 14 integration, Section 15 the economics, Section 16 upgrade authority and deployment, and Section 17 the roadmap. The appendices give account layouts, the instruction reference, error codes, the rung parameters, the status feed schema and a glossary.

2. Background: what it costs to break a curve

The ladder's design rests on how the cost of the elliptic-curve discrete logarithm problem (ECDLP) grows with key size, classically and on a quantum computer. This section collects the models the rest of the paper uses.

2.1 The problem

Let E be an elliptic curve over a prime field $\mathbb{F}_p$ and G a point of prime order n . Given Q in the subgroup generated by G , the ECDLP asks for the integer $k \in [0, n)$ with $Q = k \cdot G$. Every elliptic-curve signature scheme in use on public blockchains, ECDSA on secp256k1 and EdDSA on Ed25519 among them, is secure only while this problem is infeasible at the sizes they use: a public key is $k \cdot G$ for a secret k .

Two parameters matter for cost:

- **n** , the size of the subgroup, which sets the classical cost;
- **the field size** (the bit length of p), which sets the size of the arithmetic a quantum circuit must perform, and therefore most of its qubit count.

For most curves these are the same number of bits. Canary's classical control rung, C-160, deliberately separates them.

2.2 Classical cost: Pollard's rho

The best known classical algorithms for a generic prime-order subgroup are Pollard's rho and its parallel variant by van Oorschot and Wiener, and Pollard's kangaroo for discrete logarithms known to lie in an interval. A random walk through the group finds a collision after about $\sqrt{(\pi n)/2}$ steps. The negation map, which identifies P with $-P$ at no cost, divides the effective group size by 2. On curves with j -invariant 0, of the form $y^2 = x^3 + b$, an efficiently computable automorphism of order 3 divides it by a further 3. The expected step counts are therefore:

generic curve with negation map: $\sqrt{(\pi n)/4}$ steps

$j = 0$ curve ($y^2 = x^3 + b$): $\sqrt{(\pi n)/12}$ steps

For the ladder's sizes:

Rung	Subgroup n	Curve	Expected steps	$\log_2$
Q-160	$\sim 2^{160}$ (161-bit)	$j = 0$	$2^{80} \times \sqrt{(\pi/12)}$	79.0
Q-192	$\sim 2^{192}$	$j = 0$	$2^{96} \times \sqrt{(\pi/12)}$	95.0
Q-224	$\sim 2^{224}$ (225-bit)	$j = 0$	$2^{112} \times \sqrt{(\pi/12)}$	111.0
C-160	$\sim 2^{160}$	generic	$2^{80} \times \sqrt{(\pi/4)}$	79.8
T-256k	$\sim 2^{256}$	$j = 0$	$2^{128} \times \sqrt{(\pi/12)}$	127.0
T-255e	$\sim 2^{252}$	generic	$2^{126} \times \sqrt{(\pi/4)}$	125.8

Each step is one elliptic-curve point addition plus bookkeeping. Parallel collision search distributes the walk across any number of processors with near-linear speedup, so the cost is a budget in group operations, not a wall-clock limit.

2.3 The classical frontier

The public record of what has been solved fixes how far classical computation reaches today:

Event	Size	Effort
Academic ECDLP record, 2009	112-bit prime field	A PlayStation 3 cluster [12]
Academic ECDLP record	114-bit, Barreto–Naehrig curve	2,000 cores for about six months [13]
Bitcoin puzzle #130	Key in a 2^{130} interval	About $2^{65.5}$ kangaroo steps, on GPUs [14][15]
Bitcoin puzzle #135	Key in a 2^{135} interval	About 2^{68} steps; still open, 13.5 BTC bounty, public key exposed since 2019 [14][15]

The frontier sits around 2^{65} to 2^{68} group operations. Q-160, at about 2^{79} , is 2^{11} times beyond puzzle #135.

Cost of brute force at the low rung. A modern GPU running a tuned kangaroo or rho kernel performs about 8×10^9 group operations per second (RCKangaroo on an RTX 4090 [15]). At an assumed rental price of \$0.30 per GPU-hour:

$$\begin{aligned}
 &2^{79} \approx 6.0 \times 10^{23} \text{ operations} \\
 &\div 8 \times 10^9 \text{ per GPU-second} = 7.6 \times 10^{13} \text{ GPU-seconds} = 2.1 \times 10^{10} \text{ GPU-hours} \\
 &\times \$0.30 = \text{about } \$6 \text{ billion}
 \end{aligned}$$

Allowing for better kernels, cheaper hardware and the spread of rental prices, this paper uses \$1 billion to \$10 billion as the classical cost of Q-160 and C-160. The low end of that range sets the market caps of Section 8.6.

2.4 Quantum cost: Shor's algorithm

Shor's algorithm solves the ECDLP in polynomial time on a fault-tolerant quantum computer. Its cost is dominated by reversible modular arithmetic over the curve's base field. Roetteler, Naehrig, Svore and Lauter give explicit circuit counts for a curve over an n -bit prime field [9]:

$$\begin{aligned}
 \text{logical qubits} &= 9n + 2\lceil \log_2 n \rceil + 10 \\
 \text{Toffoli gates} &= 448 n^3 \log_2 n + 4090 n^3
 \end{aligned}$$

Field bits n	Logical qubits	Toffoli gates	Scaled from Google 2026 (est.)
160	1,466	3.0×10^{10}	~750 qubits, ~21M Toffoli
192	1,754	5.3×10^{10}	~900 qubits, ~37M Toffoli
224	2,042	8.5×10^{10}	~1,050 qubits, ~59M Toffoli
256	2,330	1.3×10^{11}	$\leq 1,200$ qubits, $\leq 90\text{M}$ Toffoli [7]

Google Quantum AI's 2026 estimate for 256-bit ECDLP is fewer than 1,200 logical qubits and 90 million Toffoli gates [7]; Google did not publish per-size figures, so the last column scales its 256-bit figure by Roetteler's ratios. Every quantum resource figure in this paper is a logical count; physical qubit counts depend on the error-correction scheme and hardware, and the references give them where they exist [7][8].

Two properties of these formulas shape the ladder:

- 1. Qubits are dominated by the field term $9n$.** The subgroup size enters only through the length of the exponent register. A curve over a 256-bit field with a 160-bit subgroup needs nearly the qubits of a full 256-bit break, while costing classically what a 160-bit curve costs. This is the basis of the classical control rung.
- 2. Quantum cost grows polynomially in n , classical cost exponentially.** From 160 to 256 bits, logical qubits grow by a factor of 1.59 and Toffoli count by about 4.3, while classical cost grows by 2^{48} .

2.5 Why the ladder can't start lower

The second property sets a floor. A curve small enough to fall to a quantum computer long before 256-bit keys is also small enough to fall to classical computation: at 120 bits the classical cost is about 2^{59} , within reach of a modest GPU budget. Below about 150 bits, classical solvers would collect the bounty and fire the signal without any quantum progress at all.

The lowest trustless rung therefore sits where classical cost leaves the reach of any plausible budget, at 160 bits, and the ladder's lead time is the distance in logical qubits from there to 256 bits: from 1,466 to 2,330, a machine with 63 % of the qubits of a secp256k1 breaker. That is strictly more warning than a 256-bit canary gives, which is none.

2.6 Where 256-bit keys sit on Solana

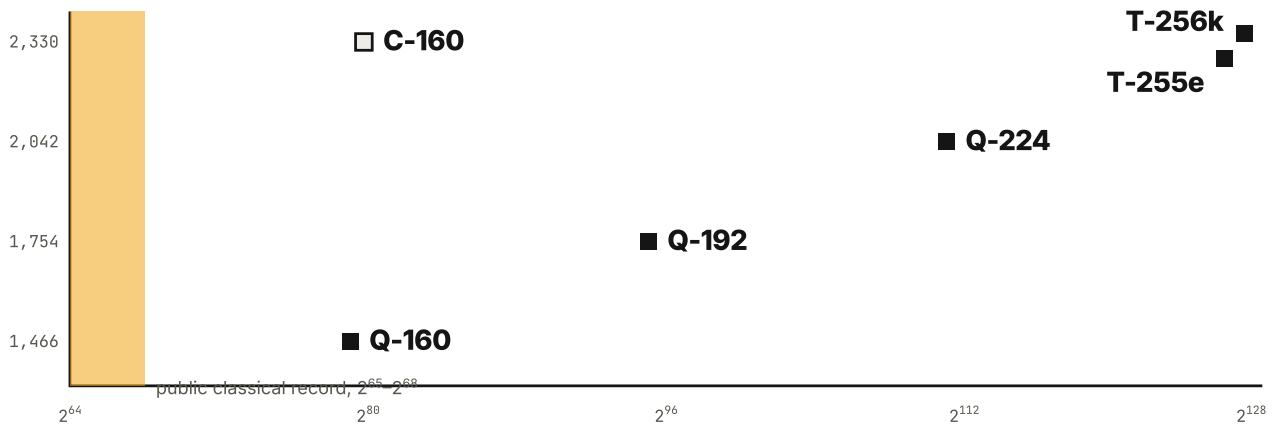
On Solana, every address that is not a program-derived address is an Ed25519 public key [24]. A program-derived address (PDA) is a point that lies off the curve by construction and has no private key at all [16]. Funds held by a PDA can only move under the rules of the program that derives it. A program can therefore hold funds that no Ed25519 key can move and require a different signature scheme to release them. Section 10 builds Q-Vault on this property.

3. The ladder

3.1 Six rungs in three families

A rung is a curve, a generator G of a prime-order subgroup, and a target point Q whose discrete logarithm k with $Q = k \cdot G$ nobody knows. Canary's ladder has six:

Rung	Family	Curve	Field bits	Subgroup bits	Classical steps	Logical qubits
Q-160	Quantum	secp160k1: $y^2 = x^3 + 7$ over a 160-bit prime, prime order	160	161	$\sim 2^{79}$	1,466
Q-192	Quantum	secp192k1: $y^2 = x^3 + 3$ over a 192-bit prime, prime order	192	192	$\sim 2^{95}$	1,754
Q-224	Quantum	secp224k1: $y^2 = x^3 + 5$ over a 224-bit prime, prime order	224	225	$\sim 2^{111}$	2,042
C-160	Classical control	A generic curve over secp256k1's 256-bit field with a ~ 160 -bit prime factor in its order	256	~ 160	$\sim 2^{79} - 2^{80}$	$\sim 2,330$
T-256k	Terminal	secp256k1, NUMS target point	256	256	$\sim 2^{127}$	2,330
T-255e	Terminal	Ed25519, NUMS target point in the prime-order subgroup	255	253	$\sim 2^{126}$	$\sim 2,330$



The ladder by classical cost (across) and logical qubits (up). The Q-rungs step together in both; C-160 shares Q-160's classical cost and the terminal rungs' qubit count.

3.2 Quantum rungs

The three Q-rungs are the SEC 2 curves secp160k1 ($y^2 = x^3 + 7$), secp192k1 ($y^2 = x^3 + 3$) and secp224k1 ($y^2 = x^3 + 5$) [28]: prime-order curves with j -invariant 0, the same family as secp256k1, over 160-, 192- and 224-bit prime fields. A laboratory that builds arithmetic for secp256k1 builds it for these first. Their subgroups have 161, 192 and 225 bits. Canary uses the standard curve equations but not the standard base points: on every rung both G and Q are derived by hash-to-curve (Section 4), so no published generator with a known relation to anything else is involved. Dallaire-Demers, Doyle and Foo publish a deterministic suite of the same family stepping from 256 bits down to 6 [10], which places these sizes in the context of a public ladder of reference instances.

Because quantum cost is set by field size, each Q-rung is cheaper for a quantum computer than secp256k1: Q-160 needs 63 % of the logical qubits and 23 % of the Toffoli gates of a 256-bit break.

3.3 The classical control

C-160 is a generic curve over secp256k1's 256-bit base field whose order has a prime factor r of about 160 bits. G and Q are cofactor-cleared into the order- r subgroup. Classically, only the subgroup matters: C-160 costs about 2^{79} – 2^{80} steps, the same as Q-160 within a factor of 2. A quantum attack, however, must perform arithmetic over the 256-bit field, so its qubit count is that of a full secp256k1 break, about 2,330.

C-160 is the instrument that separates classical from quantum progress at the bottom of the ladder. Both Q-160 and C-160 cost the same to a classical attacker; only a quantum attacker finds Q-160 meaningfully cheaper. Its curve needs a certified point count (Schoof–Elkies–Atkin) over the 256-bit field and a certified ~ 160 -bit prime factor; C-160 joins the parameter suite, and the quantum-signal market opens, when those certificates are published.

3.4 Terminal rungs

T-256k is a pair of nothing-up-my-sleeve points on secp256k1, the curve of Bitcoin and Ethereum keys, the counterpart of the canary BitMEX Research proposes [1]. T-255e is a pair of nothing-up-my-sleeve points in the 253-bit prime-order subgroup of Ed25519, the curve of every Solana address. A fall of either means keys on that curve can be broken.

3.5 Reading a fall

Markets settle on the fact that a rung fell. The ladder is arranged so that the pattern of falls also carries its cause:

Observation	Reading	A 256-bit-keys-at-risk event?
Q-160 falls, C-160 stands	Quantum, or an attack specific to small fields	Yes: the strong quantum signal
Q-160 and C-160 both fall	Classical money or a classical algorithm; both cost $\sim 2^{79}$ classically	Says nothing about qubits
Q-192 or Q-224 falls	Only a CRQC or a classical ECDLP breakthrough fits: 2^{95} and more is beyond any budget	Yes; a classical break endangers 256-bit keys as surely
T-256k or T-255e falls	Theft capability exists on that curve	Yes, terminal

Attribution matters only at the bottom, where enough money could buy a fall. Canary gives that case its own contract, the quantum-signal market, which resolves YES when Q-160 falls while C-160 still reads Open (Section 8.4).

3.6 The lead time the ladder buys

The ladder's lowest trustless rung is pinned by classical feasibility (Section 2.5), so its lead time is measured in qubits: the distance between a machine that can break Q-160, with 1,466 logical qubits, and one that can break secp256k1, with 2,330. Across that distance the markets carry information continuously: YES prices at each expiry move as capability estimates move, long before any rung falls, and the curve they draw is a market-implied date for each threshold.

3.7 Bounties and the decision to claim

A rung below 256 bits guards nothing but its own bounty. A solver who can break Q-160 gains nothing by keeping the result secret and gains the bounty, priority and recognition by claiming. A claim has no victim and is legal, can be pseudonymous and can name any recipient. The bounty for Q-160 and C-160 is set alike, so a classical solver gains nothing by claiming one to look like the other. If a rung's discrete logarithm is ever published anywhere, anyone can claim the rung with it, so the signal does not depend on the original solver choosing to act.

4. Parameter generation

A rung is only a canary if nobody knows its answer. Each rung's two points, the base point G and the target point Q , are therefore derived from a public seed by a deterministic procedure that anyone can rerun. This section specifies that procedure, the checks applied to its output, the transcript that records it, and how the transcript reaches the program.

4.1 Requirements

Requirement	How the generator meets it
Nobody knows $\log_G Q$	G and Q are independent hash-to-curve outputs; no party ever chooses a scalar
Anyone can check it	The procedure is deterministic: the same seed reproduces the same points, transcript and hash bit for bit
The curves are recognisable	The Q-rungs and T-256k use SEC 2 curve equations; T-255e uses Ed25519
The program verifies the published rung	The program's constants and the rung's <code>transcript_hash</code> both come from the same transcript
No hidden freedom	The only input is the seed text; the counters the procedure needed are recorded

4.2 The parameter suite

Rung	Curve	Equation	Field bits	Subgroup bits	Cofactor
Q-160	secp160k1	$y^2 = x^3 + 7$	160	161	1
Q-192	secp192k1	$y^2 = x^3 + 3$	192	192	1
Q-224	secp224k1	$y^2 = x^3 + 5$	224	225	1
C-160	joins when its point count is certified (Section 4.9)		256	~160	
T-256k	secp256k1	$y^2 = x^3 + 7$	256	256	1
T-255e	Ed25519	$-x^2 + y^2 = 1 + d \cdot x^2 y^2$, $d = -121665/121666$	255	253	8

The curve equations, field primes and group orders are those of SEC 2 [28][36] and RFC 8032 [32]; Appendix D gives them in full. The standard base points are not used. On every rung, both G and Q are produced by the procedure below, so the rung does not depend on how any standard's generator was chosen.

4.3 Hash to curve

For a rung id p , a role (the byte `G` or `Q`) and a counter c , the generator hashes:

$$H(p, \text{role}, c) = \text{SHA-256}(\text{"canary.nums.v1"} \parallel \text{len}(\text{seed}) \parallel \text{seed} \parallel p \parallel \text{role} \parallel c)$$

where $\text{len}(\text{seed})$ is the seed's UTF-8 length as a 32-bit big-endian integer, seed is its UTF-8 bytes, p is one byte and c is a 32-bit big-endian integer. The counter starts at 0 and increments until the candidate is accepted.

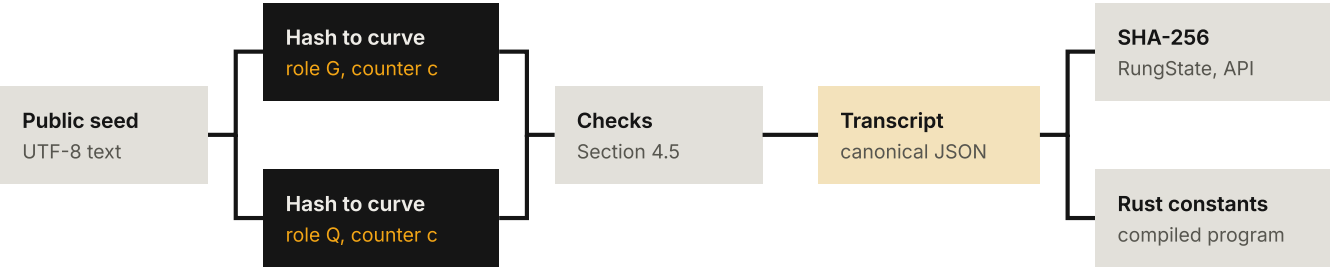
Short Weierstrass curves (the Q-rungs and T-256k).

1. $x \leftarrow H \bmod p$.
2. If $x^3 + b$ is not a square mod p , increment c and repeat.
3. $y \leftarrow$ the smaller of the two square roots, computed with Tonelli–Shanks.
4. The point is (x, y) . Since every curve here has cofactor 1, every point other than the identity generates the full group.

5. For Q, the candidate must differ from G.

Twisted Edwards (T-255e).

- 1. $y \leftarrow H \bmod p$.
- 2. Solve for x; if no x exists, increment c and repeat. Take the even root.
- 3. Multiply the point by the cofactor 8, which moves it into the prime-order subgroup.
- 4. Reject the identity, require $n \cdot P = O$, and for Q require $Q \neq G$.



From a public seed to the constants a rung verifies against.

Try-and-increment is not constant-time, which is irrelevant here: every input is public. Its output is uniform enough for the purpose, and what matters is that it is a fixed function of the seed.

4.4 Why nobody knows the answer

G and Q are two outputs of SHA-256 mapped to the curve. Knowing k with $Q = k \cdot G$ for such a pair is exactly the discrete-logarithm problem the rung poses; no choice of seed changes that, because for every seed the relation between the two outputs is as unknown as it is for any other. The only freedom anyone had is the seed text, and grinding over seeds produces other uniformly random pairs, never a pair with a known logarithm. This is the nothing-up-my-sleeve property BitMEX Research proposes for a Bitcoin canary [1] and that the Brace-for-Impact suite applies to a ladder of curves [10].

Separate roles in the hash keep G and Q independent: neither is derived from the other, and Q is rejected if it equals G.

4.5 Checks

Every entry in the suite passes these checks before its transcript is written:

Check	Method
p is prime	BPSW probable-prime test (<code>sympy.isprime</code>)
n is prime	BPSW probable-prime test
The group order is $h \cdot n$	n -torsion is non-trivial, and $h \cdot n$ is the unique multiple of n inside the Hasse interval $p + 1 \pm 2\sqrt{p}$
Not anomalous	The trace of Frobenius $t = p + 1 - h \cdot n$ is not 1
Large embedding degree	$p^k \bmod n \neq 1$ for every $1 \leq k \leq B$, with B chosen at generation (128 or more for the published suite)
Non-singular curve	$4a^3 + 27b^2 \neq 0 \bmod p$ (Edwards: $a \neq d$, both non-zero)
Points in the subgroup	$n \cdot G = O$ and $n \cdot Q = O$, $G \neq O$, $Q \neq O$
Independent points	$G \neq Q$
Edwards cofactor cleared	Points are multiplied by 8 before acceptance

BPSW has no known counterexample but is a probable-prime test, not a certificate. The SEC 2 and RFC 8032 primes are long-published values; the deployment's primality certificates for each p and n are listed with the parameter suite (`RUNG_PRIME_CERTIFICATES` in `/v1/programs`).

4.6 The transcript

Each rung's transcript is its manifest entry, every field except `transcriptSha256` itself, serialized as canonical JSON: keys sorted, separators `,` and `:` with no spaces, ASCII only, no trailing newline. The entry records:

Field	Content
<code>schemaVersion</code> , <code>network</code>	<code>canary.parameters.v1</code> , <code>canonical</code>
<code>id</code> , <code>label</code> , <code>parameterSet</code> , <code>model</code>	The rung id and name, the curve and its form
<code>pHex</code> , <code>nHex</code> , <code>aHex</code> , <code>bHex</code> (or <code>dHex</code>), <code>cofactor</code>	The curve constants, as 32-byte hexadecimal
<code>gHex</code> , <code>qHex</code> , <code>gMinusQHex</code>	The two points and the precomputed $G - Q$ used by the verifier
<code>numsCounters</code>	The counter at which G and Q were accepted
<code>publicSeed</code> , <code>hashToCurve</code>	The seed and the procedure's name
<code>fieldBits</code> , <code>subgroupBits</code> , <code>scalarBits</code>	Sizes the verifier depends on
<code>checks</code>	The result of every check in Section 4.5, with its method

```
transcriptSha256 = SHA-256( canonical JSON of the entry without transcriptSha256 )
```

The hash names the transcript file, appears in the manifest, is written into the rung's `RungState.transcript_hash` at initialization, enters every claim challenge (Section 5.3), and the API serves the exact transcript bytes at `GET /v1/transcripts/{hash}`. Anyone can fetch the transcript, hash it, and compare the result with the 32 bytes stored on chain.

4.7 From transcript to program

The generator also writes the curve constants as Rust tables that the rungs program compiles in. For a Weierstrass rung they are p , n , a , b , G , Q , $G - Q$, the transcript hash, the scalar bit length and the cofactor; for the Edwards rung, G , Q , n and the transcript hash. There is no instruction that changes them.

Three things tie the compiled program to the published transcript:

1. `initialize_rung` writes the compiled transcript hash into `RungState`, and refuses a rung id for which no parameters are compiled.
2. `begin_claim` and the terminal claim instructions require the compiled transcript hash to equal the one in `RungState`.
3. The verifier's setup recomputes $G + (-Q)$ from the compiled points and requires it to equal the compiled $G - Q$, and requires the scalar bit length to equal the bit length of n .

The indexer checks every rung it reads against the published manifest and its program-derived address before it reports the rung, and the API reports the deployment as waiting until every rung on chain carries the published transcript hash.

4.8 Reproducing the suite

The generator is a single Python 3.13 program with one dependency, SymPy 1.14.0. From the backend source:

```
python canary-gen/generate.py --production --suite sec2 --seed "<published seed>" --embedding-bound <B>
```

It writes `generated/canonical.json`, its SHA-256, one transcript file per rung named by its hash, and the Rust tables. A reproduction matches when the manifest hash and every transcript hash equal the published ones; a single differing byte changes them.

Reproduction is deliberately cheap: it needs no special hardware and uses a general-purpose computer-algebra library rather than any Canary code beyond the generator itself. Independent reproductions are a launch condition (Section 17.2).

4.9 C-160

C-160 needs a curve over secp256k1's 256-bit field whose group order has a prime factor of about 160 bits. Finding one means counting points on random curves over a 256-bit field (Schoof–Elkies–Atkin) until the order has such a factor, then certifying the count and the factor. The rung's points are then cleared by the cofactor into the 160-bit subgroup, and the staged verifier runs over 160-bit scalars in 256-bit field arithmetic.

Until a certified curve is published, the parameter suite carries no C-160 entry, and `initialize_rung` for rung id 3 returns `ParameterUnavailable`. The quantum-signal market, which reads C-160 as its control, opens with it.

5. The claim protocol

A claim proves knowledge of a rung's discrete logarithm k , where $Q = k \cdot G$, without revealing k , and binds the payout to one recipient. The proof is a Schnorr proof of knowledge made non-interactive with a domain-separated hash. This section specifies the proof, its encodings, the challenge, and how each family of rungs verifies it on Solana.

5.1 The proof

The solver holds k . To claim for a recipient A :

1. choose w uniformly in $[1, n)$ and compute $R = w \cdot G$
2. $e = H_{\text{claim}}(\text{program}, \rho, A, \text{nonce}, R, G, Q, \text{transcript}) \bmod n$
3. $s = w + e \cdot k \bmod n$
4. publish (R, s)

The program accepts when

$$s \cdot G = R + e \cdot Q, \text{ with } 0 < s < n \text{ and } R \text{ a valid point}$$

which holds for an honest proof because $s \cdot G = w \cdot G + e \cdot k \cdot G = R + e \cdot Q$. Only R and s go on chain; k does not, and w is discarded after use. Because k never leaves the solver, a claim leaves no artefact anyone else can reuse: the next rung is untouched, and Q 's logarithm remains known only to whoever solved it.

5.2 Why a valid proof implies knowledge

Schnorr proofs are special-sound. Given two accepting proofs (R, s) and (R, s') with the same R and different challenges $e \neq e'$, anyone can compute

$$k = (s - s') \cdot (e - e')^{-1} \bmod n$$

In the random-oracle model this means that anyone able to produce accepting proofs with noticeable probability can be turned into an algorithm that outputs k [33][34]. A rung that reads Solved therefore records that someone was able to compute its discrete logarithm.

The proof is also zero-knowledge: (R, s) can be simulated without k by choosing s and e first and setting $R = s \cdot G - e \cdot Q$, so a published proof reveals nothing about k beyond the fact that its author knew it.

5.3 The challenge

The challenge binds the proof to everything that must not change after the solver signs:

$e = \text{SHA-256}(\text{"canary.claim.v1"} \parallel \text{program_id} \parallel \rho \parallel \text{recipient} \parallel \text{nonce} \parallel R \parallel G \parallel Q \parallel \text{transcript_hash}),$ read as a big-endian integer and reduced mod n

Field	Bytes (Weierstrass)	Bytes (Edwards)	What it binds
Domain tag <code>canary.claim.v1</code>	15	15	The purpose of the hash
Program id	32	32	One deployment of the rungs program
Rung id ρ	1	1	One rung

Field	Bytes (Weierstrass)	Bytes (Edwards)	What it binds
Recipient	32	32	The owner of the account that receives the bounty
Nonce	8, little-endian	8, little-endian	One claim attempt
R	64	32	The commitment
G	64	32	The rung's base point
Q	64	32	The rung's target point
Transcript hash	32	32	The exact published parameters
Total	312	216	

Weierstrass points are encoded as $x \parallel y$, each coordinate 32 bytes big-endian (the smaller fields are left-padded). Edwards points use the 32-byte compressed encoding of RFC 8032 [32]. The scalar s is 32 bytes big-endian on Weierstrass rungs and 32 bytes little-endian on T-255e, matching each curve's convention. Reduction mod n is computed bitwise on chain, so no big-integer library is involved.

Because the recipient is inside the challenge, a proof cannot be redirected: changing A changes e , and producing the matching s for the new e requires k . Because the program id and transcript hash are inside, a proof made for one deployment or one parameter set is invalid for every other.

5.4 Proof hash and the claim job

Staged claims keep their progress in a `VerifyJob` account. Its address commits to the proof itself:

```
proof_hash = SHA-256( "canary.proof.v1" || p || recipient || nonce || R || s )
job = PDA( "job", p, recipient, nonce, proof_hash )
```

A different proof maps to a different job, so a party who sees a solver's first transaction cannot create a job at that address with a modified proof. A copied valid proof maps to the same job and the same recipient, so copying it gains nothing. The proof hash is recorded in `RungState` when the rung falls.

The nonce is a random 64-bit value chosen by the client. A solver who needs to retry, for example after a job expires, uses a new nonce and so a new job; old jobs never block new ones.

5.5 Staged verification on the quantum rungs

The Q-rungs' fields have no Solana syscall, so the rungs program computes $s \cdot G - e \cdot Q$ itself and compares it with R . A full check takes more compute than one transaction allows, so it runs as a sequence of transactions over a stored accumulator.

Field arithmetic. Every field element is eight little-endian 32-bit limbs in Montgomery form with $R = 2^{256}$, whatever the field size. One representation serves all three curves, so the same audited code verifies Q-160, Q-192 and Q-224.

Point arithmetic. The accumulator is a point in Jacobian coordinates $(X : Y : Z)$, representing $(X/Z^2, Y/Z^3)$. Each bit doubles the accumulator and, depending on the bits, adds an affine point with mixed addition. No step needs a field inversion.

Shamir's trick. Instead of computing $s \cdot G$ and $e \cdot Q$ separately, the engine scans the bits of s and e together, most significant first:

bit of s	bit of e	Added after doubling
0	0	nothing
1	0	G
0	1	$-Q$

bit of s	bit of e	Added after doubling
1	1	G - Q (precomputed in the parameters)

After the last bit the accumulator holds $s \cdot G - e \cdot Q$. The final comparison with $R = (x, y)$ is projective, $X = x \cdot Z^2$ and $Y = y \cdot Z^3$, so again no inversion is needed. The accumulator starts at the point at infinity ($Z = 0$).

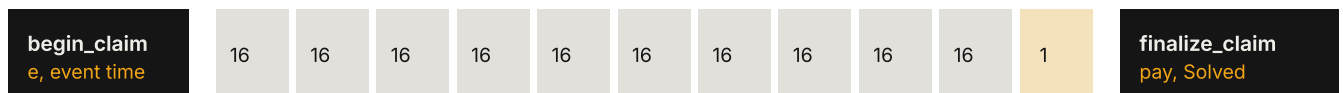
Instructions.

1. `begin_claim(p, recipient, nonce, R, s)` checks that the rung is Open, that R lies on the curve with coordinates below p, that $0 < s < n$ and that the transcript matches; it computes e, records `begin_ts` and `begin_slot` from the clock, and creates the job with the accumulator at infinity.
2. `step_claim(max_bits)` advances the accumulator by up to 16 bits. It is permissionless: anyone can send it, and it only advances the stored computation. After the last bit, the job becomes Verified if the accumulator equals R, and Rejected otherwise.
3. `finalize_claim()` requires a Verified job, pays the bounty and sets the rung Solved.

steps = $\lceil \text{scalar_bits} \div 16 \rceil$ transactions = steps + 2

Rung	Scalar bits	Steps	Transactions
Q-160	161	11 (the last is 1 bit)	13
Q-192	192	12	14
Q-224	225	15 (the last is 1 bit)	17

step_claim × 11, bits per step



begin_slot ≤ begin_slot + 300

The thirteen transactions of a Q-160 claim, inside its claim window.

Sixteen bits per step is the measured fit: a 64-bit step exceeded the 1.4M compute-unit limit, and the heaviest 16-bit step, a scalar whose bits force an addition at every position, used 721,585 compute units. Measured Q-224 costs on a local validator:

Transaction	Compute units	Bytes
<code>begin_claim</code>	174,953–176,453	455
<code>step_claim</code> , 16 bits	469,585–714,170	285
<code>step_claim</code> , final bit and comparison	177,548–180,435	285
<code>finalize_claim</code>	22,334	448

Each prepared transaction stays at or below 1,082 bytes, which leaves 150 bytes of the 1,232-byte limit for instructions a wallet may add.

5.6 Verification on the terminal rungs

T-256k uses `sol_secp256k1_recover`, the syscall that recovers an ECDSA public key. Recovery computes $r^{-1} \cdot (s \cdot P - z \cdot G_{\text{std}})$, where P is the point whose x-coordinate is r and whose parity is given by the recovery id. Setting the message hash $z = 0$, $r = x(P) \bmod n$ and $s = r \cdot t \bmod n$ turns it into a scalar multiplication:

$$\text{recover}(z = 0, r = x(P), s = r \cdot t, v) = r^{-1} \cdot (r \cdot t) \cdot P = t \cdot P$$

The recovery id is the parity of $y(P)$ plus 2 if $x(P) \geq n$. The program performs two such multiplications, $s \cdot G$ and $e \cdot Q$, rejects any case where r , t or s is zero, and checks $e \cdot Q + R = s \cdot G$ projectively, in one transaction. Both G and Q are the rung's own hash-derived points; the standard generator G_{std} cancels out because $z = 0$. Measured cost: 344,884 compute units, 553 bytes.

T-255e uses `sol_curve_multiscalar_mul` on Ed25519. The program rejects the identity encoding for R , $s = 0$ and $s \geq n$, then computes one two-term multi-scalar multiplication and compares the compressed result with R 's bytes:

$$s \cdot G + (n - e) \cdot Q = s \cdot G - e \cdot Q \stackrel{?}{=} R$$

Measured cost: 51,949 compute units, 520 bytes.

Both terminal instructions are single transactions and both are permissionless: the transaction can be sent by anyone, because the payout goes to the recipient the proof names.

5.7 Event time and the claim window

Family	<code>event_ts</code> , <code>event_slot</code>	Recorded by
Q-rungs	The clock at <code>begin_claim</code>	The job, then copied to <code>RungState</code> at finalize
T-256k, T-255e	The clock at the claim transaction	<code>RungState</code> directly

Recording the begin time means that a claim begun before a market's expiry counts for that market even if its verification finishes after (Section 8.3). It cannot be backdated: the program reads the clock; the solver supplies no time.

A staged claim must finish within the claim window: `step_claim` and `finalize_claim` require

$$\text{Clock.slot} \leq \text{begin_slot} + 300$$

which is about two minutes at Solana's nominal 400 ms slot time. At finalize the program records `finalize_slot`, and every reader of a Solved rung re-checks that $\text{event_slot} \leq \text{finalize_slot} \leq \text{event_slot} + 300$. A job that missed its window can be closed by anyone; `close_job` returns the job's rent to the account that paid for it, and the solver begins again with a new nonce.

5.8 Payout

The payout instruction (`finalize_claim`, `claim_secp256k1` or `claim_ed25519`) transfers the entire balance of the rung's bounty vault, at that moment, to a USDC token account owned by the recipient, and sets the rung Solved in the same instruction. The account need not be the recipient's associated token account; the browser plan creates the associated account in the finalize transaction if it does not yet exist. Once the rung reads Solved, every claim instruction fails, so a rung pays exactly once.

Claims remain possible after the 2040-01-01 sunset. A claim and a sponsor's reclaim (Section 7.6) then draw on the same vault, and whichever executes first is served first.

5.9 Making a claim

The solver needs k as a 32-byte big-endian integer, a funded Solana wallet and a recipient. Two clients build the proof locally:

Client	Where k lives	Transactions
The web app's claim page	In the page's memory only; never sent to any server or written to storage	Prepared together for one wallet approval
The claim CLI	Read from a file or standard input, never from a command-line argument	Sent in order, with one extra transaction that creates the recipient's token account if needed

Both draw w and the nonce from a cryptographically secure random source, check that the rung on chain is Open and carries the published transcript hash, and refuse to build a proof for parameters that do not match. If a sequence is interrupted, the CLI resumes the existing job from its stored cursor, and the browser starts a new job with a new nonce.

6. The canary_rungs program

`canary_rungs` holds the ladder. It owns one `RungState` account per rung, the rung's bounty vault and receipt mint, and the claim verifiers of Section 5. It is the only program whose state other programs need to trust, and it is kept small for that reason: nine instructions, no administrator after initialization, and constants compiled in.

6.1 Accounts and addresses

Every account the program owns sits at a program-derived address (PDA), so any reader can derive it from the rung id alone:

Account	Seeds	Size	Purpose
<code>RungState</code>	"rung", ρ	271 bytes	The public state of rung ρ
Bounty authority	"bounty", ρ	no data	Signs bounty transfers and mints receipts
Bounty vault	"collateral", ρ	SPL token account	Holds the rung's USDC; its authority is the bounty authority
Receipt mint	"receipt", ρ	Token-2022 mint	NonTransferable sponsor receipts, 6 decimals
<code>SponsorPosition</code>	"sponsor", ρ , sponsor	75 bytes	One sponsor's totals on rung ρ
<code>VerifyJob</code>	"job", ρ , recipient, nonce, proof_hash	360 bytes	One staged claim in progress

Sizes include Anchor's 8-byte discriminator, `SHA-256("account:<Name>")` truncated to 8 bytes. Account data is Borsh-encoded after it. Rung ids are 0 to 5 for Q-160, Q-192, Q-224, C-160, T-256k and T-255e.

6.2 RungState

`RungState` is the interface the rest of Canary, and any integrator, reads. Its fields in order:

Field	Type	Meaning
<code>version</code>	u8	Layout version, 2
<code>rung_id</code>	u8	ρ
<code>family</code>	u8	0 quantum, 1 classical control, 2 terminal
<code>status</code>	u8	0 Open, 1 Solved
<code>bump</code> , <code>bounty_bump</code> , <code>receipt_bump</code>	$u8 \times 3$	PDA bumps
<code>collateral_mint</code>	Pubkey	USDC
<code>collateral_vault</code>	Pubkey	The bounty vault
<code>receipt_mint</code>	Pubkey	The receipt mint
<code>transcript_hash</code>	[u8; 32]	SHA-256 of the published parameter transcript
<code>event_ts</code>	i64	Unix time of the claim's begin (staged) or of the claim (terminal); 0 while Open
<code>event_slot</code>	u64	The matching slot
<code>recipient</code>	Pubkey	Who was paid
<code>proof_hash</code>	[u8; 32]	The hash of the accepted proof
<code>total_funded</code>	u64	Cumulative sponsor deposits
<code>total_paid</code>	u64	The bounty paid on the fall
<code>total_reclaimed</code>	u64	Cumulative reclaims after the sunset

Field	Type	Meaning
<code>sponsor_count</code> , <code>receipt_count</code>	$u32 \times 2$	Distinct sponsors
<code>initialized_slot</code>	$u64$	When the rung was created
<code>finalize_slot</code>	$u64$	The slot of the payout

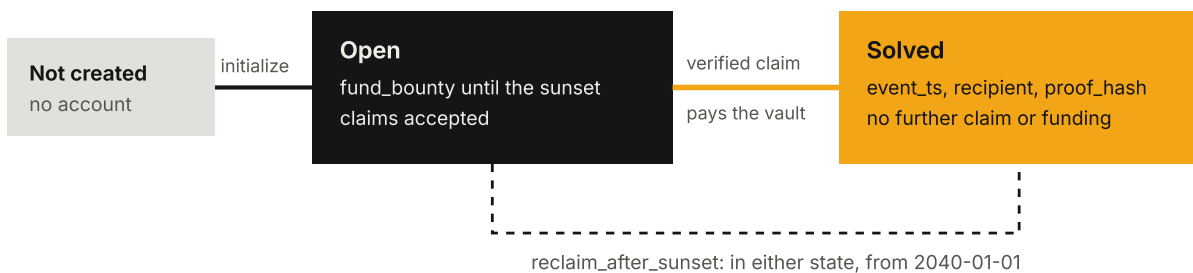
Appendix A gives byte offsets. A reader that only needs the signal reads four things: that the account is the rung's PDA under the rungs program, its discriminator and version, `status`, and, when Solved, `event_ts` with the claim-window check of Section 5.7.

6.3 Instructions

Instruction	Caller	Effect
<code>initialize_rung(p)</code>	The program's upgrade authority, once per rung	Creates <code>RungState</code> (Open), the bounty vault and the receipt mint
<code>fund_bounty(p, amount)</code>	Any sponsor	Moves USDC into the bounty vault and mints receipts 1:1
<code>begin_claim(p, recipient, nonce, R, s)</code>	Anyone	Starts a staged claim on a Q-rung
<code>step_claim(max_bits)</code>	Anyone	Advances a staged claim by up to 16 bits
<code>finalize_claim()</code>	Anyone	Pays a verified staged claim and sets the rung Solved
<code>close_job()</code>	Anyone	Closes a finished or expired job and refunds its rent to its payer
<code>claim_secp256k1(p, recipient, nonce, R, s)</code>	Anyone	Verifies and pays a T-256k claim in one transaction
<code>claim_ed25519(p, recipient, nonce, R, s)</code>	Anyone	Verifies and pays a T-255e claim in one transaction
<code>reclaim_after_sunset(p, receipts)</code>	A receipt holder	After 2040-01-01, burns receipts for their share of the vault

Each instruction checks the rung's family: `begin_claim` accepts the staged rungs only, `claim_secp256k1` only T-256k and `claim_ed25519` only T-255e, and a mismatch fails with `WrongClaimMethod`. Appendix B lists every account and check; Appendix C lists the 22 error codes.

6.4 The life of a rung



A rung moves from Open to Solved once; reclaim depends only on the clock.

The transition is one-way. Every claim path requires Open, and the internal `solve` step requires Open again before it writes, so a Solved rung can never be paid twice or reopened. `fund_bounty` requires Open and a time before the sunset. `reclaim_after_sunset` depends only on the clock and the holder's receipts.

6.5 The life of a claim job

Status	Value	Reached by	Next
Verifying	0	<code>begin_claim</code>	<code>step_claim</code> until the last bit
Verified	1	The last <code>step_claim</code> , when $s \cdot G - e \cdot Q = R$	<code>finalize_claim</code>
Finalized	2	<code>finalize_claim</code>	<code>close_job</code>
Rejected	3	The last <code>step_claim</code> , when the check fails	<code>close_job</code>

`close_job` succeeds when the job is Finalized or Rejected, when the rung is already Solved by any claim, or when the claim window has passed. It refunds the job's rent to the account that paid for it, recorded in the job as `payer`, and to no one else.

Several jobs can be in flight on the same rung at once, for different recipients or nonces. The first to finalize sets the rung Solved; the others then fail at their next step and can be closed.

6.6 Initialization

`initialize_rung(p)` is the only privileged instruction, and it can succeed once per rung:

- Parameters must be compiled for `p`; otherwise `ParameterUnavailable`.
- The collateral mint must be the deployment's configured USDC mint, with 6 decimals.
- The signer must be the current upgrade authority of the rungs program, read from its ProgramData account.
- The receipt mint's address must be unused. The program allocates it, assigns it to Token-2022, initializes the NonTransferable extension, and then initializes the mint with 6 decimals, the bounty authority as mint authority and no freeze authority.
- The program writes `RungState`: Open, the compiled transcript hash and the current slot.

There is no instruction that changes a rung's parameters, collateral, vault or receipt mint after this. Once the upgrade authority is revoked (Section 16), even initialization of a new rung id is impossible.

6.7 Events

Event	Emitted by	Fields
<code>RungInitialized</code>	<code>initialize_rung</code>	<code>rung_id</code> , <code>collateral_mint</code> , <code>transcript_hash</code>
<code>BountyFunded</code>	<code>fund_bounty</code>	<code>rung_id</code> , <code>sponsor</code> , <code>amount</code> , <code>total_funded</code>
<code>ClaimBegun</code>	<code>begin_claim</code>	<code>job</code> , <code>rung_id</code> , <code>recipient</code> , <code>nonce</code> , <code>begin_ts</code> , <code>begin_slot</code> , <code>proof_hash</code>
<code>ClaimAdvanced</code>	<code>step_claim</code>	<code>job</code> , <code>bit_cursor</code> , <code>status</code>
<code>RungSolved</code>	the three payout instructions	<code>rung_id</code> , <code>event_ts</code> , <code>event_slot</code> , <code>recipient</code> , <code>proof_hash</code> , <code>amount</code> , <code>job</code> (when staged)
<code>SponsorReclaimed</code>	<code>reclaim_after_sunset</code>	<code>rung_id</code> , <code>sponsor</code> , <code>receipt_amount</code> , <code>amount</code>

`RungSolved` is 98 bytes for a terminal claim and 130 bytes for a staged one, after its 8-byte discriminator `SHA-256("event:RungSolved")[0..8]`. Events are a convenience for indexers; the account is the source of truth, and the indexer reads accounts.

6.8 Build profiles

The program is compiled in exactly one of two profiles:

Profile	Program id and collateral	Parameters
<code>production</code>	Taken from the deployment's configuration at build time	The published canonical suite only
<code>localnet</code>	Fixed local ids, a local test mint	A local suite; optionally test rungs with known logarithms for automated tests

A production build fails if test rungs are enabled or if its parameter sources contain test material, so a deployed program cannot contain a rung with a known answer. The workspace builds with overflow checks on in release, fat link-time optimization and one codegen unit, on Anchor 0.31.1.

6.9 Measured costs

Instruction	Compute units	Transaction bytes
<code>initialize_rung</code>	41,031–48,517	550
<code>fund_bounty</code>	58,550–64,550	623
<code>begin_claim</code> (Q-224)	174,953–176,453	455
<code>step_claim</code> (16 bits)	469,585–714,170	285
<code>finalize_claim</code>	22,334	448
<code>claim_secp256k1</code>	344,884	553
<code>claim_ed25519</code>	51,949	520
<code>reclaim_after_sunset</code>	25,993	

Every transaction requests an explicit compute limit of 1.4M, the per-transaction maximum, and stays below 1,232 bytes as a legacy transaction, so every wallet can sign it.

7. Bounty vaults

Each rung has one bounty vault: a USDC token account owned by the rungs program's bounty authority for that rung. The vault is what makes a rung worth claiming. It receives sponsors' deposits and a share of market fees, pays its whole balance to the solver when the rung falls, and after the 2040-01-01 sunset lets sponsors reclaim what remains.

7.1 Where the money sits

Account	Address	Controlled by
Bounty vault	PDA "collateral", p	The bounty authority PDA "bounty", p, which only the rungs program can sign for
Receipt mint	PDA "receipt", p	Mint authority: the bounty authority; no freeze authority
A sponsor's receipts	The sponsor's Token-2022 associated token account	The sponsor, but the tokens cannot be transferred
SponsorPosition	PDA "sponsor", p, sponsor	Written only by fund_bounty and reclaim_after_sunset

No instruction moves USDC out of a bounty vault except the three payout instructions of Section 5.8 and reclaim_after_sunset. There is no administrator withdrawal, no early withdrawal and no fee taken on deposits.

7.2 Funding

fund_bounty(p, amount) accepts a deposit while the rung is Open and the clock is before the sunset:

```
require amount > 0, status(p) = Open, now < 2,208,988,800 (2040-01-01T00:00:00Z)
transfer_checked(amount, 6 decimals): sponsor → bounty vault
mint amount receipts to the sponsor
```

The first deposit by a sponsor creates their SponsorPosition and receipt account and increments the rung's sponsor count. Every deposit adds to total_funded on the rung and on the position.

Field of SponsorPosition	Meaning
owner	The sponsor
total_funded	Everything this sponsor has deposited on the rung
total_reclaimed	Everything this sponsor has reclaimed after the sunset
first_funded_ts, first_funded_slot	When the sponsor first funded the rung

7.3 Receipts

Receipts are Token-2022 tokens with the NonTransferable extension, minted one receipt unit per USDC unit deposited, both at 6 decimals. A sponsor who deposits 25,000 USDC holds 25,000 receipts.

Property	Consequence
NonTransferable	A reclaim right stays with the sponsor who funded the rung; it cannot be sold, lent or pooled
Minted 1:1	Receipts measure contribution exactly; together they are each sponsor's share of the vault
No freeze authority	Nobody can freeze a sponsor's receipts
Mint authority is a PDA	New receipts exist only against new deposits

A receipt is also a public, verifiable record: anyone can read who funded a rung, and how much, from the mint and the positions.

7.4 Growth from market fees

The markets program's `sweep_bounty` transfers each rung's accrued share of settlement fees (Section 8.7) directly into the rung's bounty vault. Fees mint no receipts. They raise the vault's balance and so both the bounty a solver receives and, after the sunset, every receipt's reclaim value. The API reports the cumulative amount a rung has received this way as `feesReceived`.

7.5 Payout on a fall

When a claim succeeds, the payout instruction transfers the vault's entire balance at that moment, deposits and fees together, to the recipient's USDC account, and sets the rung Solved:

```
paid = balance(vault) at the payout instruction total_paid ← paid
```

From then on the rung accepts no deposits, and the vault holds only what arrives later from fee sweeps. A Solved rung whose receipts are all redeemed is flagged by the markets program as having no exit for further fees, which then go to the other rungs (Section 8.8).

7.6 Reclaim after the sunset

From the sunset on, any receipt holder can call `reclaim_after_sunset(p, r)` to burn r of their receipts for a proportional share of the vault:

```
amount = ⌊ balance(vault) × r ÷ supply(receipts) ⌋ (computed in 128-bit arithmetic)
```

The receipts are burned and the amount is transferred to the holder's USDC account in the same instruction. A holder can reclaim part of their receipts and keep the rest. Rounding down means each reclaim leaves any fractional unit in the vault; the last holder to reclaim receives what remains.

Example	Value
Deposits on the rung	1,000,000 USDC, so 1,000,000 receipts outstanding
Fees swept in since	200,000 USDC
Vault balance at the reclaim	1,200,000 USDC
A sponsor's receipts	250,000
Reclaim	$1,200,000 \times 250,000 \div 1,000,000 = 300,000$ USDC

Reclaim does not check the rung's status. If the rung has fallen, the vault holds only fees that arrived after the payout, and receipts reclaim those. If it is still Open, the rung remains claimable: a claim pays whatever is in the vault when it executes, so after the sunset claims and reclaims draw on the same balance.

7.7 What sponsors see

Endpoint	Returns
<code>GET /v1/bounties</code>	Every rung's vault address and balance, totals funded and paid, fees received, sponsor and receipt counts, and the sunset
<code>GET /v1/rungs/{rungId}</code>	The rung, its state and its bounty; <code>/events</code> and <code>/claim-jobs</code> give its fall and its claims in flight

Endpoint	Returns
GET /v1/wallets/{wallet}/receipts	A wallet's receipts per rung: amount funded and when, share units, and the reclaim's availability, estimated amount and amount already claimed

Share units are the wallet's remaining receipt balance. A sponsor's current share of a vault is therefore their receipts divided by the receipt supply, both of which are on chain.

7.8 Measured costs

Instruction	Compute units	Transaction bytes
fund_bounty	58,550–64,550	623
reclaim_after_sunset, partial	25,993	

8. Rung-event markets

A rung-event market is a binary contract on one question: does rung p fall by time T ? It is two SPL tokens, YES and NO, collateralized one-for-one in USDC, and it resolves by reading the rung's `RungState` account. This section defines the contract precisely; Section 9 gives the program that implements it.

8.1 The contract

A market m is the tuple (p, T, c) : a rung p , an expiry T (a Unix timestamp; v0 expiries are 23:59:59 UTC on 2027-12-31, 2028-12-31 and 2030-12-31) and a collateral mint c (USDC in v0). Let `event_ts(p)` be the rung's recorded event time, the time its successful claim began.

YES pays 1 unit of $c \Leftrightarrow p$ reads Solved and `event_ts(p)  $\leq$  T`

NO pays 1 unit of $c \Leftrightarrow p$ reads Solved with `event_ts(p)  $>$  T`, or T has passed and p still reads Open 300 slots after the first post-expiry observation

Exactly one side pays. Because a claim may take several transactions to verify, the rung's event time is the start of the claim, not its end: a solver who begins before the expiry has fallen before the expiry, even if verification completes after it. The claim window, at most 300 slots between `begin_claim` and `finalize_claim`, bounds how long after T the outcome can remain undecided.

8.2 Pairs and collateral

Any holder of c can mint pairs while the market is open:

`mint_pair(a)`: lock a units of $c \rightarrow$ receive a YES and a NO

`merge_pair(a)`: burn a YES and a NO $\rightarrow$ receive a units of c

Both are exact at c 's 6-decimal precision. The market's collateral account therefore always holds exactly the outstanding supply:

`collateral(m) = supply(YES) = supply(NO)` while m is unresolved

`mint_pair` is accepted only while the market is unresolved, the clock is at or before T and the target rung reads Open, and only within the rung's open-interest cap (Section 8.6). `merge_pair` has the same gates: once T has passed or the target rung reads Solved, a pair can no longer be merged, and the only exit is redemption after resolution. Closing merge at the event means the side that won cannot avoid the settlement fee by buying the losing side cheaply and merging at par.

YES and NO are classic SPL tokens with the market's program address as their sole mint authority and no freeze authority. They trade on any Solana venue that lists SPL tokens; Canary runs no exchange.

8.3 Resolution

`resolve` is permissionless: anyone may call it, and Canary's crank calls it for every market as soon as it can succeed. It reads the rung account and the clock and sets the outcome once:

Condition at the call	Outcome
p reads Solved, its claim finalized within 300 slots of its event slot, and <code>event_ts(p) $\leq$ T</code>	YES
p reads Solved and <code>event_ts(p) $>$ T</code>	NO

Condition at the call	Outcome
p reads Open and the clock is past T, on the first such call	Records the observation slot; no outcome yet
p reads Open and the slot is more than 300 past the recorded observation	NO
Otherwise	No change

The 300-slot barrier after expiry gives a claim that began before the expiry time to finish its verification before NO can be declared.

The rung account is read with full validation: its owner must be `canary_rungs`, its address must be the program-derived address of p, and its discriminator and layout version must match. A spoofed account cannot resolve a market. Once set, the outcome never changes.

8.4 The quantum-signal market

The quantum-signal market is a market on the pair (Q-160, C-160) rather than one rung. It answers the attribution question of Section 3.5:

YES $\Leftrightarrow$ Q-160 fell by T, and C-160 had not fallen at or before Q-160's event (compared by event time, then event slot)

The market waits until 300 slots after Q-160's event slot before resolving YES, so a C-160 claim begun at the same moment can land first. Ties resolve NO. If Q-160 and C-160 fall together, classical money or a classical algorithm explains both, and the quantum-signal market pays NO even though both rung-event markets pay YES. The quantum-signal market is created with C-160 Open as its control rung and counts against Q-160's open-interest cap; it opens when C-160 is initialized.

8.5 Redemption and the settlement fee

After resolution, holders of the winning side redeem. Let g be the gross amount redeemed, in atomic units of c. The settlement fee is 1.0 %, that is 100 basis points of the winning redemption, computed on the market's cumulative winning redemptions so that splitting a redemption into pieces never adds fee:

$$R' = R + g \text{ fee} = \lfloor R' \div 100 \rfloor - F \quad F' = F + \text{fee net} = g - \text{fee}$$

where R is the market's total redeemed before this call and F its total fees. The losing side redeems for nothing. Minting and merging carry no fee. The fee stays in the market's collateral account as escrow; nothing leaves the market at redemption except the net payout.

8.6 Open-interest caps

Each rung has a cap on the open interest that markets on it may carry. The cap is set at no more than 1 % of the low end of the rung's classical break cost, so that buying YES and then funding a brute-force attack never pays:

$$\begin{aligned} \text{cap}(p) &\leq 1 \% \times \text{classical cost}(p) \\ \text{Q-160: } &1 \% \times \$1 \text{ billion} \approx \$10 \text{ million} \end{aligned}$$

Open interest counts only unresolved markets: once a market resolves, its pairs no longer count against the rung's cap, whether or not their holders have redeemed. A rung's cap is set once, when the markets program enables the rung, and can't be changed afterwards. Q-160's default is 10,000,000 USDC; the caps of rungs with a higher classical cost are set at deployment. The quantum-signal market counts against Q-160's cap.

8.7 Sweeping fees

At each redemption, the fee is split into a bounty share and a protocol share, both computed on cumulative totals across the whole protocol:

$$\text{budget} = \lfloor (\Phi + \text{fee}) \times 30 \div 100 \rfloor - B \text{ protocol} = \text{fee} - \text{budget}$$

where Φ is the protocol's total fees before this redemption and B the total bounty budget so far. The bounty share therefore rounds down and the protocol share is the remainder, and over many redemptions the split converges to exactly 30 % and 70 %.

Share	Basis points	Swept by	Destination
Protocol	7,000	<code>sweep_treasury</code>	The buyback vault, the markets program's own USDC account under <code>["buyback"]</code> ; it buys CANARY and burns it (Section 15.5)
Bounties	3,000	<code>sweep_bounty</code> , per rung	The rung's bounty vault

Both sweeps are permissionless and independent, and both destinations are derived by the program, never supplied by the caller. The crank calls both. Pending amounts are tracked per market and per rung, so a sweep moves exactly what has accrued since the last one.

8.8 Splitting the bounty share by time-weighted open interest

The 30 % bounty share is split across rungs in proportion to the open interest their markets carried over time. For each rung p the markets program maintains an accumulator of open interest multiplied by the slots it was outstanding:

$$\begin{aligned} A(p) &+= OI(p) \times (\text{slot_now} - \text{slot_last}(p)); \text{slot_last}(p) \leftarrow \text{slot_now} \text{ before every change of open interest} \\ W &= \sum A(\sigma) \text{ over enabled rungs } \sigma \text{ not flagged no-exit} \\ \text{share}(p) &= \lfloor \text{budget} \times A(p) \div W \rfloor, \text{ the remainder (fewer than 16 units) to the largest fractional parts} \end{aligned}$$

The accumulators are 128-bit and run over each rung's whole lifetime in the markets program. A position opened and closed in the same slot adds nothing to any accumulator, so the split cannot be steered by flash positions. The split routes funds only to bounty vaults from which they can still leave: a rung that reads Solved and has no receipts outstanding is flagged no-exit, its pending share returns to a deferred pool, and `allocate_deferred` later re-splits that pool across the remaining rungs. If no rung has accumulated weight yet, the whole budget is deferred until one has.

8.9 Reading prices as a date

YES and NO are classic SPL tokens and trade wherever Solana tokens trade; the indexer reads their prices from Jupiter's price API (Section 13.2). A YES price at expiry T is the market's risk-neutral probability that the rung falls by T . Because winners on either side redeem net of the 1 % fee, the cleanest estimate normalizes by both sides' prices:

$$F(T) = p_{\text{YES}}(T) \div (p_{\text{YES}}(T) + p_{\text{NO}}(T))$$

Across expiries $T_1 < T_2 < T_3$, F traces a cumulative distribution for the rung's fall date. The chance of a fall within an interval, given none before it, and the corresponding constant hazard rate, are:

$$\begin{aligned} h(T_{i-1}, T_i) &= (F(T_i) - F(T_{i-1})) \div (1 - F(T_{i-1})) \\ \lambda_i &= -\ln((1 - F(T_i)) \div (1 - F(T_{i-1}))) \div (T_i - T_{i-1}) \end{aligned}$$

With illustrative prices for Q-160 of 8.5 %, 14 % and 22 % at the three v0 expiries, measured from October 2026:

Interval	Years	F at end	Conditional chance h	Hazard λ per year
Now → 2027-12-31	1.21	0.085	0.085	0.073
2027-12-31 → 2028-12-31	1.00	0.140	0.060	0.062
2028-12-31 → 2030-12-31	2.00	0.220	0.093	0.049

The site's landing page draws these curves for each rung as an hourglass of bars whose widths are the YES prices. Read rung by rung, the curves give a market-implied date for each threshold of quantum capability; read across rungs, they give the market's view of how fast capability climbs the ladder.

8.10 Who can trade

Markets exclude US persons in v0. Eligibility is checked by the API from the deployment's configured source, and the app asks the user to confirm their status where no source is configured. Bounties, Q-Vaults and RungState carry no such restriction.

9. The canary_markets program

`canary_markets` implements the markets of Section 8. It holds one global configuration, one account per market, each market's collateral and YES and NO mints, and the fee escrow. It reads `RungState` from the rungs program and never writes to it. This section gives its accounts, its instruction set, the order in which state changes, and the accounting it maintains.

9.1 Accounts and addresses

Account	Seeds	Size	Purpose
<code>MarketsConfig</code>	<code>"config"</code>	1,215 bytes	Collateral, the CANARY mint and buyback key, fee and buyback totals, and the per-rung weights
<code>Market</code>	<code>"mkt"</code> , <code>p</code> , expiry (i64 LE), kind, collateral mint	477 bytes	One market
Market collateral	<code>"collateral"</code> , market	SPL token account	The market's USDC, including its fee escrow
YES mint	<code>"yes"</code> , market	SPL mint, 6 decimals	Mint authority: the market
NO mint	<code>"no"</code> , market	SPL mint, 6 decimals	Mint authority: the market
Buyback vault	The associated USDC account of <code>"buyback"</code>	SPL token account	The protocol share, until a buyback spends it
Buyback CANARY account	The associated CANARY account of <code>"buyback"</code>	SPL token account	The CANARY a buyback buys, burned in the same transaction

Because the market's address includes the rung, the expiry to the second, the kind and the collateral mint, there is exactly one market for each combination, and anyone can derive it. YES and NO mints have no freeze authority.

9.2 MarketsConfig

Field	Meaning
<code>rungs_program</code>	The rungs program id the markets program reads
<code>collateral_mint</code>	USDC, a classic SPL mint with 6 decimals
<code>administrator</code>	The account that may enable rungs with <code>configure_rung</code> and set the buyback with <code>configure_buyback</code>
<code>canary_mint</code>	The CANARY mint, set once by <code>configure_buyback</code> and never changed
<code>buyback_authority</code>	The key that may send a buyback; the administrator can replace it
<code>pending_treasury</code> , <code>total_treasury_swept</code>	Protocol share accrued, and swept to the buyback vault, across all markets
<code>total_fees_accrued</code>	Φ , cumulative settlement fees across all markets
<code>total_bounty_budget</code>	B , the cumulative bounty share, $[\Phi \times 30 \div 100]$ at the last redemption
<code>total_bounty_swept</code>	Bounty share swept to vaults
<code>total_buyback_usdc</code> , <code>total_canary_burned</code>	USDC spent by buybacks and CANARY burned, cumulative
<code>weights[16]</code>	One <code>RungWeight</code> per enabled rung

Field	Meaning
<code>unallocated_bounty</code>	Bounty share deferred until a rung can receive it
<code>buyback_*</code>	The open buyback, 37 bytes: written by <code>buyback_begin</code> , checked and cleared by <code>buyback_finish</code> in the same transaction

Each `RungWeight` holds the rung id, an enabled flag, the open-interest cap, the rung's current open interest, its pending bounty share, the amount swept to it so far, the 128-bit accumulator $A(p)$ of open interest $\times$ slots, the slot of its last update and the no-exit flag. Up to 16 rungs can be enabled.

9.3 Market

A market's first 237 bytes form a discovery header that indexers scan without reading the rest:

Header field	Meaning
<code>rung_id, kind</code>	The target rung; 0 rung-event, 1 quantum-signal
<code>outcome</code>	0 unresolved, 1 YES, 2 NO
<code>expiry</code>	Unix time T , at most 9999-12-31T23:59:59Z
<code>no_observation_slot</code>	The slot of the first post-expiry observation with the rung still Open
<code>resolved_slot</code>	When the market resolved
<code>rung, control_rung</code>	The target <code>RungState</code> , and C-160's for a quantum-signal market
<code>collateral_mint, collateral_vault, yes_mint, no_mint</code>	The market's token accounts
<code>open_interest</code>	Outstanding pairs while unresolved; the remaining winning liability after

The body records the market's totals and its share of the fee escrow: `total_minted`, `total_merged`, `total_redeemed` (R), `total_fees` (F), `total_fees_swept`, `pending_treasury`, `pending_bounty[16]` and `unallocated_bounty`, with its creator and creation time.

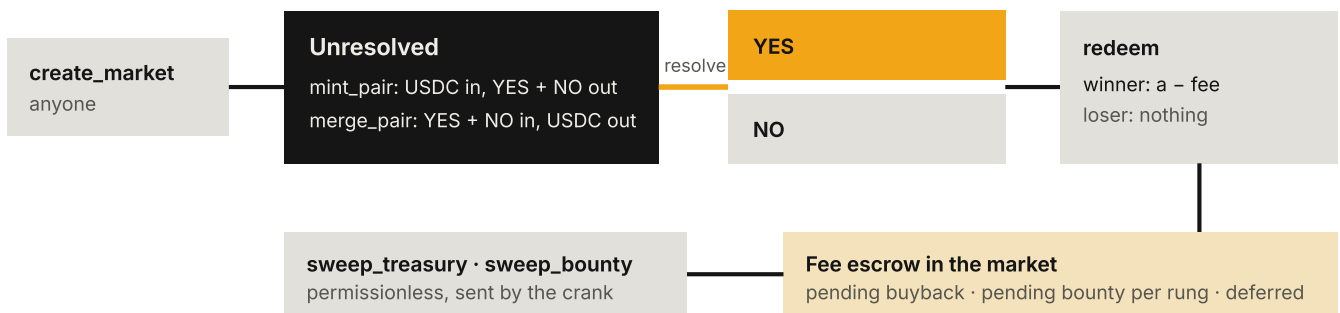
9.4 Instructions

Instruction	Caller	Gates	Effect
<code>initialize_config()</code>	Upgrade authority, once	6-decimal mint; the configured USDC	Creates the configuration
<code>configure_rung(p, cap)</code>	Administrator	Rung initialized with the same collateral; $cap > 0$; not yet enabled; a free slot	Enables p with an immutable cap
<code>create_market(p, T, kind)</code>	Anyone	$now < T$; p Open and enabled; for kind 1, $p = Q-160$ and C-160 Open	Creates the market and its mints
<code>mint_pair(a)</code>	Anyone eligible	Unresolved; $now \leq T$; p Open; $OI(p) + a \leq cap$	Takes a USDC, mints a YES and a NO
<code>merge_pair(a)</code>	A holder	Unresolved; $now \leq T$; p Open	Burns a YES and a NO, returns a USDC
<code>resolve()</code>	Anyone	Market unresolved	Records the observation or sets the outcome (Section 8.3)
<code>redeem(a)</code>	A holder of the winning side	Resolved	Burns a winning tokens, pays a – fee
<code>allocate_deferred()</code>	Anyone		Re-splits the market's deferred bounty share

Instruction	Caller	Gates	Effect
<code>sweep_bounty(p)</code>	Anyone		Moves the market's pending share for p into p's bounty vault
<code>sweep_treasury()</code>	Anyone		Moves the market's pending protocol share to the buyback vault
<code>sweep_fees(p)</code>	Anyone		Both sweeps in one call
<code>configure_buyback(mint, key)</code>	Administrator	A classic SPL mint other than USDC; no mint set yet, or the same one; no buyback open	Sets the CANARY mint for good, and the buyback key
<code>buyback_begin(a, m)</code>	The buyback key	CANARY mint set; a, m > 0; a ≤ the vault's USDC; called directly by the transaction; a matching <code>buyback_finish</code> later in it	Records the balances and advances a USDC to the key's USDC account
<code>buyback_finish()</code>	The buyback key	The open buyback's pair; the key holds no more USDC than before; at least m CANARY arrived	Burns all CANARY in the program's account and adds to the totals

There is no instruction that moves collateral anywhere except to a token holder redeeming or merging, to a bounty vault, to the buyback vault, or out of the buyback vault into a swap whose CANARY is burned in the same transaction. No instruction can change a market's outcome once set, its expiry, or a rung's cap.

9.5 A market from creation to redemption



Mint and merge close at expiry and when the target rung falls; redemption opens at resolution.

The instructions a market passes through, and where its fees wait.

9.6 Resolution in code order

`resolve()` reads the market, its target rung and, for a quantum-signal market, its control rung, and applies these rules in order:

1. If the market is already resolved, do nothing.
2. If the target rung reads Solved, require $\text{event_slot} \leq \text{finalize_slot} \leq \text{event_slot} + 300$ for it, and for a Solved control rung as well; then:
 - if $\text{event_ts} > T$, resolve NO;
 - else, for a rung-event market, resolve YES;
 - else, for a quantum-signal market, wait until the slot exceeds the target's $\text{event_slot} + 300$; then resolve NO if the control rung is Solved with $(\text{event_ts}, \text{event_slot})$ at or before the target's, and YES otherwise.
3. If the target rung reads Open:
 - if $\text{now} \leq T$, do nothing;
 - if no observation is recorded, record $\text{no_observation_slot} = \text{the current slot}$;

- if the current slot exceeds `no_observation_slot + 300`, resolve NO.

On an outcome, the program settles the weight accumulators, subtracts the market's whole open interest from the rung's, and records the outcome, slot and time. From then on the market's `open_interest` counts the winning tokens still unredeemed.

9.7 Settlement arithmetic

Every redemption updates the market's and the configuration's cumulative totals:

```
R' = R + a
fee = [R' ÷ 100] - F
F' = F + fee
Φ' = Φ + fee
budget = [Φ' × 30 ÷ 100] - B
B' = B + budget
protocol = fee - budget
payout = a - fee
```

Because every quantity is computed from cumulative totals, splitting a redemption into pieces never changes the total fee or the total split. For example, a holder who redeems 1,000,000 units as 999,999 and then 1 pays 9,999 and then 1: a total of 10,000, exactly 1 %, of which 3,000 goes to the bounty share and 7,000 to the buyback vault.

The bounty share of each redemption is credited to rungs by weight at once (Section 8.8): each rung's part goes to the market's `pending_bounty` for that rung, and any amount that cannot yet be allocated goes to the market's `unallocated_bounty`. The protocol part goes to `pending_treasury`.

9.8 Accounting identities

The program maintains these identities at every instruction boundary:

Identity	Holds because
<code>collateral(market) = open_interest + fees_not_yet_swept</code>	Mint and merge move exactly a USDC for a pairs; a redemption of a pays <code>a - fee</code> and keeps the fee in the market
<code>total_fees = total_fees_swept + pending_treasury + Σ pending_bounty + unallocated_bounty</code>	Every fee unit is credited to exactly one bucket and leaves only by a sweep
<code>Σ Ol(markets_on_p, unresolved) = Ol(p) ≤ cap(p)</code>	Mint adds, merge subtracts and resolution subtracts each market's open interest once
<code>payout + fee = a</code> , for every redemption	The fee is taken from the redemption itself

Counters only grow. A sweep moves exactly what its bucket holds and zeroes it, so repeating a sweep moves nothing.

9.9 Weight arithmetic

The accumulator `A(p)` is a 128-bit integer. The sum `W` over all eligible rungs is computed in a 256-bit accumulator built from four 64-bit words, and each share is computed by bitwise long division:

```
share(p) = [ budget × A(p) ÷ W ]
leftover = budget - Σ share(p) < 16
```

The leftover, fewer units than there are rungs, goes one unit each to the rungs with the largest remainders, ties broken by the lower rung id, so the split is exact and deterministic. If `W` is zero, because no rung has accumulated weight yet, the whole budget is deferred to `unallocated_bounty` and `allocate_deferred` splits it later. A clock that moves backwards fails the instruction with `InvalidWeightClock` rather than corrupting an accumulator.

9.10 Measured costs

Instruction	Compute units	Transaction bytes
<code>initialize_config</code>	22,520	384
<code>configure_rung</code>	14,392	293
<code>create_market</code>	55,529	559
<code>mint_pair</code>	44,495	589
<code>merge_pair</code>	44,986	589
<code>resolve</code>	17,368–17,671	349
<code>redeem</code>	39,810–40,459	490
A crank sweep transaction	25,429–28,842	415–482

The program emits no events. Its accounts are the record, and the indexer reads them.

10. Q-Vault

10.1 Why a vault, and why on Solana

Every Solana address that is not program-derived is an Ed25519 public key [24]. Every balance held at such an address is behind a public key, exposed the moment Ed25519 can be broken. A program-derived address, by contrast, has no private key: funds held there move only under the rules of the program that derives it [16]. A program can therefore hold funds that no Ed25519 key alone can move and release them under a post-quantum signature.

Q-Vault combines that property with Canary's trigger. Before the trigger rung falls, the vault behaves much like a wallet with a safety delay: the owner's Ed25519 key spends within a budget, and larger spends wait. After the trigger falls, Ed25519 is rejected and only the owner's Falcon-512 key can move funds. The owner never has to notice the event or act in time: the vault reads the rung on every Ed25519 spend.

10.2 Keys and modes

A vault has an owner (an Ed25519 public key, the wallet), a Falcon-512 public key, a trigger rung, a delay, an instant budget and a mode:

Mode	Ed25519	Falcon-512
<code>Ed25519AndPq</code>	The vault's instant asset only: instant spends within the budget; queued spends after the delay	Spends any supported asset at any time; cancels queued Ed25519 spends; rotates the key
<code>Lockdown</code>	Rejected	Spends at any time; rotates the key

A vault enters `Lockdown` in two ways, and never leaves it:

- `trip`, callable by anyone once the trigger rung reads Solved. Before `trip` is called, every Ed25519 path already rejects a Solved trigger; `trip` records the lockdown permanently in the vault.
- `Lockdown`, called by the owner at any time.

10.3 Falcon-512

Falcon is a lattice-based signature scheme over NTRU lattices, selected by NIST for standardization [35]. Canary uses Falcon-512:

	Size
Public key	897 bytes
Signature	666 bytes (padded encoding)
Verification on Solana	365,000–390,000 compute units, measured

Neither a key nor a signature fits beside the rest of a transaction within a legacy 1,232-byte transaction, so both are written to chain in chunks. The public key is stored in the vault's `PqKey` account. A signature is staged in an authorization account before the instruction that consumes it, so every transaction stays within the legacy size every wallet can sign [22].

10.4 Creating a vault and confirming its key

Creation commits to the hash of the Falcon public key, writes the key, and then proves possession with a Falcon signature over the vault itself:

#	Transaction	Purpose
1	<code>init_vault</code>	Creates the vault: owner, index, trigger rung, delay, instant limit, instant asset, a hash of the Falcon public key, a random non-zero instance salt
2	<code>init_pq_key</code>	Creates the <code>PqKey</code> account at epoch 0, committed to the key hash
3– 4	<code>write_pq_key</code> × 2	Writes the 897-byte key in order; the account becomes immutable when complete and must match the committed hash
5	<code>init_authorization</code>	Creates the staging account for a signature: the action (here, confirm) and the SHA-256 of the signature
6– 7	<code>write_authorization</code> × 2	Writes the 666-byte signature
8	<code>confirm_pq_key</code>	Verifies the Falcon signature over the vault's confirmation digest and marks the key confirmed

The app prepares all eight for one wallet approval. Until the key is confirmed, the vault refuses Falcon actions; possession is proven before any value depends on the key, so a mistyped or corrupted key can't brick the vault. A vault funded before confirmation is flagged by the API.

10.5 The instant budget

The Ed25519 path moves only the vault's one instant asset, chosen at creation (SOL in the app). Its instant spends draw on a budget that limits how much the wallet key can move without waiting:

```
previous = 0 if spent = 0 or now ≥ last_instant_ts + delay, else spent
accept a ⇔ 0 < a ≤ instant_limit and previous + a ≤ instant_limit
then spent ← previous + a, last_instant_ts ← now
```

The budget refills only after a full delay passes with no instant spend; a failed call changes nothing. Someone who steals the wallet key can take at most one budget's worth before the owner sees it. A spend within the limit that exceeds what is left must wait for the refill, and any spend larger than the limit must go through the queue, where the Falcon key can veto it.

10.6 The queue

An Ed25519 spend beyond the budget is queued:

```
queue_ed25519(amount > instant_limit, recipient, destination, asset) → QueuedSpend { unlock_ts = now + delay, status =
Queued }
execute_queued: accepted ⇔ now ≥ unlock_ts ∧ status = Queued ∧ mode = Ed25519AndPq ∧ key confirmed ∧ trigger Open
```

Queuing does not draw on the instant budget, and `execute_queued` is permissionless: once the delay has passed, anyone can send it, and it moves exactly the destination, mint, amount and fee the queue recorded.

Execution rechecks the mode and the trigger at the moment it runs, so a spend queued before the trigger fell can't execute after. `cancel_queued_pq`, signed by the Falcon key, sets the status to Cancelled. A queued spend executes at most once.

10.7 Spending with Falcon

A Falcon spend is four transactions: initialize an authorization, write the signature in two chunks, then `spend_pq`, which verifies the signature against the vault's key and executes the transfer. The signature covers a digest of the exact action:

```
digest = SHA-256("canary.vault.action.v2" || program id || vault || instance salt || Action)
Action = operation || key epoch || nonce || mint || token program || recipient || destination || amount || queue || new key || new key
hash || expected fee
```

The preimage is 375 bytes. The nonce is consumed by the action, so a signature can be used once; Ed25519 activity uses a separate nonce, so it can never invalidate a pending Falcon veto. The instance salt is random and fixed at creation, so a signature made for a vault on one network, or for a vault since closed and re-created at the same address, is invalid for any other. The SDK's offline signer computes this digest itself from the action's fields and returns those fields alongside the signature, so an offline signer can show exactly what it is signing.

10.8 Rotation

`rotate_pq_key` replaces the Falcon key. It requires two signatures: the current key authorizes the rotation, and the candidate key proves possession over the vault, so the vault never ends up with a key nobody holds. The key epoch increments with each rotation, which invalidates any signature staged under the old key.

10.9 Assets

Asset	Deposits	Ed25519	Falcon
SOL	Plain transfer to the vault's SOL account	If it is the instant asset	Yes
Classic SPL tokens, USDC included	Transfer to the vault's associated token account	If it is the instant asset	Yes
Token-2022 tokens	Transfer to the vault's associated token account	If it is the instant asset and its extensions are supported	Where its extensions are supported

Token-2022 mints are checked against an extension allowlist: transfer fees, close authority, confidential-transfer configuration, default account state, interest-bearing configuration, permanent delegate, metadata and group pointers, and a transfer hook only when it names no program. A transfer fee must equal the fee the mint charges at the current epoch and is moved with `transfer_checked_with_fee`; every other mint moves with a zero fee. Any other extension, a non-transferable mint for example, is refused, as is a transfer hook pointing at a program, because executing it would require calling an arbitrary program. The API reports, per asset, whether a vault's balance is spendable.

10.10 What a vault reads

On every Ed25519 instruction, the vault reads its trigger rung's `RungState` with the full validation of Section 12.2. While a claim on the trigger is in flight, the rung still reads Open and Ed25519 still works; the API reports the in-flight claim on the vault, so an owner can choose to lock down early.

11. The `canary_vault` program

`canary_vault` implements Q-Vault (Section 10). It has no administrator and no instruction that can move a vault's funds without its owner's Ed25519 signature on the rung-gated path or its owner's Falcon-512 signature on the post-quantum path. This section gives its accounts, its seventeen instructions, the authorization format, and the transaction sequences of each flow.

11.1 Accounts and addresses

Account	Seeds	Size	Purpose
<code>Vault</code>	"qv", owner, index (u64 LE)	332 bytes	Owner, trigger, policy, key commitment, nonces, budget
SOL custody	"sol", vault	0 bytes, System-owned	Holds the vault's SOL
Token custody	The associated token account of (vault, mint, token program)	Token account	Holds each SPL or Token-2022 asset
<code>PqKey</code>	"pq", vault, payer, epoch (u64 LE), key hash	1,014 bytes	One Falcon-512 public key, written in chunks
<code>PqAuthorization</code>	"auth", vault, payer, auth id (u64 LE)	1,040 bytes	One staged Falcon signature and the action it signs
<code>QueuedSpend</code>	"queue", vault, Ed25519 nonce (u64 LE)	227 bytes	One delayed Ed25519 spend

An owner can hold any number of vaults, distinguished by index. The SOL custody account keeps its rent-exempt minimum; only lamports above it are spendable.

11.2 `Vault`

Field	Meaning
<code>owner</code>	The Ed25519 key: the owner's wallet
<code>index</code>	Distinguishes the owner's vaults
<code>trigger_rung</code> , <code>trigger_rung_id</code>	The <code>RungState</code> address and id the vault reads
<code>mode</code>	0 <code>Ed25519AndPq</code> , 1 <code>Lockdown</code>
<code>lockdown_reason</code>	0 none, 1 the owner locked it, 2 the trigger fell
<code>asset_policy</code>	1, one instant asset
<code>instant_mint</code> , <code>instant_token_program</code>	The one asset the Ed25519 path can move; the default key with the System program means SOL
<code>instant_limit</code>	The instant budget, in the asset's atomic units
<code>delay_seconds</code>	The queue delay and the budget's quiet period
<code>pq_key_hash</code>	SHA-256 of the 897-byte Falcon public key
<code>active_pq_key</code> , <code>key_epoch</code> , <code>key_confirmed</code>	The current <code>PqKey</code> account, its epoch, and whether possession has been proven
<code>nonce</code>	The Falcon nonce, consumed by each Falcon action
<code>ed_nonce</code>	The Ed25519 nonce, the index of the next queued spend
<code>instance_salt</code>	32 random non-zero bytes fixed at creation

Field	Meaning
<code>instant_spent</code> , <code>last_instant_ts</code>	The budget checkpoint
<code>created_ts</code> , <code>created_slot</code> , <code>locked_ts</code> , <code>locked_slot</code>	When the vault was created and locked

11.3 Instructions

Path	Instruction	Signer	Effect
Lifecycle	<code>init_vault</code>	Owner	Creates the vault and its SOL account; starts in <code>Lockdown</code> if the trigger already fell
	<code>lockdown</code>	Owner	Locks the vault permanently (reason 1)
	<code>trip</code>	Anyone	Locks the vault permanently once the trigger reads Solved (reason 2)
Key	<code>init_pq_key(epoch, key_hash)</code>	Payer	Creates a <code>PqKey</code> ; epoch 0 for the first key, <code>key_epoch + 1</code> for a candidate
	<code>write_pq_key(offset, bytes)</code>	The key's payer	Appends up to 450 bytes in order; the last chunk must hash to <code>key_hash</code>
	<code>confirm_pq_key</code>	Anyone	Verifies a possession proof by the key itself and marks it confirmed
	<code>rotate_pq_key</code>	Anyone	Switches to a confirmed candidate under the active key's signature
	<code>reset_initial_pq_key_hash</code>	Owner	Replaces the first key's commitment before it is confirmed
	<code>close_unconfirmed_pq_key</code>	The key's payer	Closes an unconfirmed key account
Staging	<code>init_authorization(auth_id, action, signature_hash)</code>	Payer	Fixes an action and the hash of its signature
	<code>write_authorization(offset, bytes)</code>	The payer	Appends up to 450 signature bytes; the last chunk must match the hash
	<code>close_authorization</code>	The payer	Closes a staging account
Ed25519	<code>spend_ed25519(amount, recipient, expected_fee)</code>	Owner	An instant spend within the budget
	<code>queue_ed25519(...)</code>	Owner	Queues a spend above the instant limit
	<code>execute_queued</code>	Anyone	Executes a queued spend after its delay
Falcon	<code>spend_pq</code>	Anyone	Executes the staged, Falcon-signed spend
	<code>cancel_queued_pq</code>	Anyone	Cancels a queued spend under a Falcon signature

Instructions marked "Anyone" carry their authority in a staged Falcon signature or in on-chain state, so the transaction itself can be sent by any fee payer. That lets an owner sign offline and have the result relayed, and lets anyone execute a matured queue or trip a vault whose trigger fell.

11.4 Gates on the Ed25519 path

Every Ed25519 instruction, including `execute_queued`, passes the same gate:

$\text{ed_allowed} \Leftrightarrow \text{trigger is the authenticated RungState} \wedge \text{key_confirmed} \wedge \text{mode} = \text{Ed25519AndPq} \wedge \text{trigger reads Open}$

and moves only the instant asset. The trigger is authenticated as in Section 12.2: owned by the rungs program, at the PDA for its id, with the current layout version, and, if it reads Solved, with a finalize slot inside the claim window. A Solved trigger blocks Ed25519 whether or not `trip` has been called.

The Falcon path has no mode or trigger gate. After the trigger falls, it is the vault's only way out, and it works the same before and after.

11.5 The authorization format

A Falcon signature signs a 32-byte digest of a 375-byte preimage:

Bytes	Field	Notes
0–21	"canary.vault.action.v2"	Domain tag
22–53	Program id	One deployment
54–85	Vault address	One vault
86–117	Instance salt	One instance of that address
118	Operation	0 confirm, 1 spend, 2 cancel, 3 rotate
119–126	Key epoch	The signing key's epoch
127–134	Nonce	Must equal the vault's Falcon nonce
135–166	Mint	Spend: the asset; SOL is the default key
167–198	Token program	Spend: System, Token or Token-2022
199–230	Recipient	Spend: the SOL recipient or the token account's owner
231–262	Destination	Spend: the account credited
263–270	Amount	Spend: gross atomic units
271–302	Queue	Cancel: the <code>QueuedSpend</code>
303–334	New key	Confirm or rotate: the <code>PqKey</code> account
335–366	New key hash	Confirm or rotate: its SHA-256
367–374	Expected fee	Spend: the Token-2022 transfer fee, else 0

digest = SHA-256(preimage) integers little-endian, unused fields zero

Every field an operation does not use must be zero, and the program checks the operation's fields against the accounts the instruction actually touches: a spend's destination, mint and token program must be the accounts passed in, a cancel's queue must be the queue being cancelled, and a confirmation's new key must be the key being confirmed. A signature therefore authorizes exactly one action, once.

11.6 Staging

A Falcon signature is 666 bytes and a public key 897 bytes; neither fits in a transaction beside its instruction. Both are staged:

1. `init_authorization` fixes the action and the SHA-256 of the signature that will be written. It requires the action's nonce to equal the vault's current nonce.
2. `write_authorization` appends the signature in order: 450 bytes, then 216. The final chunk must make the staged bytes hash to the committed value.
3. The consuming instruction rechecks everything: not consumed, all 666 bytes written, the hash, the nonce, the key epoch, the key complete and matching its commitment, and the Falcon verification itself. It then marks the authorization consumed and increments the nonce.

Staging reserves nothing. Anyone may stage the same signed action under another payer or authorization id; only one can be consumed, because consumption increments the nonce. A staging account that is no longer needed can be closed by its payer to recover its rent; closing one does not restore a nonce.

11.7 Moving assets

The vault calls exactly three kinds of instruction in other programs:

Call	Program	Signed by
<code>transfer</code>	System	The SOL custody PDA
<code>transfer_checked</code>	Token or Token-2022	The vault PDA
<code>transfer_checked_with_fee</code>	Token-2022	The vault PDA

The source is always the vault's own custody account, checked by address. The destination's mint and owner are checked against the action. No remaining accounts are forwarded and no other program is invoked, so a token's extensions cannot route a transfer through code the vault did not choose.

11.8 Flows in transactions

Flow	Transactions	Instructions
Create a vault and register its key	8	<code>init_vault</code> , <code>init_pq_key</code> , <code>write_pq_key</code> × 2, <code>init_authorization</code> , <code>write_authorization</code> × 2, <code>confirm_pq_key</code>
Deposit	1	A plain SOL or token transfer to the custody account
Instant Ed25519 spend	1	<code>spend_ed25519</code>
Queue an Ed25519 spend	1	<code>queue_ed25519</code>
Execute it after the delay	1	<code>execute_queued</code>
Falcon spend	4	<code>init_authorization</code> , <code>write_authorization</code> × 2, <code>spend_pq</code>
Falcon veto of a queued spend	4	<code>init_authorization</code> , <code>write_authorization</code> × 2, <code>cancel_queued_pq</code>
Stage and confirm a candidate key	7	<code>init_pq_key</code> , <code>write_pq_key</code> × 2, <code>init_authorization</code> , <code>write_authorization</code> × 2, <code>confirm_pq_key</code>
Rotate to it	4	<code>init_authorization</code> , <code>write_authorization</code> × 2, <code>rotate_pq_key</code>
Lock down	1	<code>lockdown</code>
Trip	1	<code>trip</code>

The client prepares each flow as an ordered list of transactions for one wallet approval. Every stage carries a distinct memo, so a wallet that signs all at once cannot deduplicate two stages with identical instructions.

11.9 Measured costs

Instruction	Compute units	Transaction bytes
<code>init_vault</code>	21,497	502
<code>init_pq_key</code>	10,085	357
<code>write_pq_key</code> (450 / 447 bytes)	2,183 / 2,761	708 / 705
<code>init_authorization</code>	11,506–19,006	615
<code>write_authorization</code> (450 / 216 bytes)	3,154 / 3,617	708 / 474

Instruction	Compute units	Transaction bytes
<code>confirm_pq_key</code> (Falcon verification)	364,041–374,332	316
<code>spend_ed25519</code>	12,717	433
<code>queue_ed25519</code>	14,585–16,085	503
<code>execute_queued</code>	12,806	417
<code>spend_pq</code> (Falcon verification)	368,736–389,390	417–481
<code>cancel_queued_pq</code> (Falcon verification)	365,428–365,478	349
<code>trip</code>	7,029	283

The largest vault transaction is 708 bytes, a full signature or key chunk; the Falcon verification dominates compute and stays well within the 1.4M limit.

12. Security design

Canary's programs move real money on a signal that, by construction, fires rarely and all at once. The security design follows from that: every reader of a rung must read the real rung, every payout must follow from verified state, and every path out of a vault must be bound to exactly the action its owner authorized. This section collects the properties the programs enforce and how they were checked.

12.1 Principles

Principle	In practice
Read the real account	Every cross-program read checks owner, program-derived address, discriminator and layout version, and fails closed
One source of truth	A market's outcome and a vault's mode follow from <code>RungState</code> and the clock only
State moves one way	Open → Solved, unresolved → resolved, Ed25519-and-PQ → lockdown: none of these transitions can be reversed
Exact arithmetic	Checked integer arithmetic in atomic units; the protocol's share of any split rounds down
No arbitrary calls	Vaults invoke only the System, Token and Token-2022 programs
Bound every signature to its action	Schnorr challenges commit to the recipient; Falcon digests commit to the program, the vault instance, the operation, a nonce and every field of the action
Minimal, immutable trigger code	The rungs program's upgrade authority is revoked after audit

12.2 Reading RungState

Both the markets program and the vault program read rungs through a typed account constraint that requires:

1. the account's owner to be the `canary_rungs` program;
2. its address to be the program-derived address for the rung id, derived under the rungs program's id;
3. its discriminator to be `RungState`'s;
4. its layout version to be the current one.

A rung that fails any check is not treated as Open or as Solved: the instruction fails. For the vault this fail-closed rule means a malformed trigger read blocks Ed25519 spending rather than allowing it. For markets it means no spoofed account can resolve YES. The markets program additionally requires that a Solved rung's claim finalized within the claim window measured from its event slot, so it does not rely on the rungs program alone for that bound.

12.3 The claim

Property	How it holds
The discrete logarithm is never revealed	The claim is a Schnorr proof; only R and s go on chain
A claim can't be retargeted	The challenge commits to the recipient; a different recipient needs a new s, which needs k
A proof can't be replayed on another rung or deployment	The challenge commits to the domain tag <code>canary.claim.v1</code> , the program id, the rung id, G, Q and the rung's transcript hash
A proof can't be replayed for a second payout	A rung pays once: finalize sets Solved and a Solved rung accepts no claim
Verification can't be left half done to block others	Each claim job is its own account keyed by recipient and nonce; a stale job expires after the claim window and does not block a new one

Property	How it holds
Event time can't be backdated	<code>event_ts</code> is the clock at <code>begin_claim</code> , recorded by the program, not supplied by the solver

For the terminal rungs, verification uses Solana's own curve syscalls (`sol_secp256k1_recover` and `sol_curve_multiscalar_mul`), which avoids instruction-introspection patterns and their offset-validation pitfalls.

12.4 Custody and accounting in markets

- **Exact collateral.** Minting locks exactly the pair count in collateral; merging and redeeming release exactly what the tokens entitle. The collateral account always equals outstanding supply while a market is unresolved.
- **Mint authority.** YES and NO mints have the market's program-derived address as sole mint authority and no freeze authority.
- **Collateral program.** Collateral mints are classic SPL Token mints from the deployment's allowlist (USDC in v0).
- **Fixed destinations.** Fees go only to the buyback vault and to bounty vaults, both derived by the program; the destinations are not supplied by the caller.
- **Cumulative fee counters.** Accrued, swept-to-buyback and swept-to-bounties counters only grow, so `sweep_fees` and `resolve` are idempotent.
- **Merge closes at the event.** A pair can't be merged after expiry or after the rung (or Q-160, for the quantum-signal market) falls, so the winning side can't avoid the settlement fee by merging.
- **No steering of the bounty split.** The split uses open interest $\times$ slots, so a position that exists for no slots carries no weight.
- **Caps that don't erode.** Only unresolved markets count against a rung's cap; resolved but unredeemed tokens free their capacity.
- **Independent sweeps.** `sweep_bounty` and `sweep_treasury` are separate instructions, and the crank sends them in separate transactions, so a failed protocol sweep can't roll back a bounty transfer.
- **Atomic buyback.** USDC leaves the buyback vault only inside a `buyback_begin` and `buyback_finish` pair, both called directly by one transaction. Finish requires every advanced unit spent and at least the quoted CANARY received, and burns it; otherwise the whole transaction fails, swap included, and the USDC stays in the vault.
- **Canonical labels.** The API marks whether a market uses one of the v0 expiries and shows a non-canonical expiry to the second, so look-alike markets are distinguishable.

12.5 Bounty vaults

Deposits are irrevocable until the sunset; there is no instruction that withdraws before it. The payout on a fall is computed and transferred in the same instruction that sets the rung Solved, to the recipient the verified proof names. After the sunset, a receipt reclaims exactly its share-unit fraction of the vault, and receipts are Token-2022 NonTransferable, so a reclaim right can't be traded away from the sponsor who funded it.

12.6 Q-Vault

Property	How it holds
A stolen Ed25519 key can't drain the vault	Instant spends draw on a budget that refills only after a full delay passes without an instant spend; anything larger is queued for the delay, during which the Falcon key can veto it
The trigger closes Ed25519 permanently	Every Ed25519 path checks the live trigger rung; <code>trip</code> persists lockdown, and lockdown has no inverse
A queued spend executes at most once	Its account records status; execution rechecks mode and trigger after the delay
Falcon signatures can't be replayed or retargeted	The digest binds program id, vault, operation, key epoch, nonce, asset, recipient, amount, queue or new key, and a per-vault random instance salt fixed at creation; nonces are consumed

Property	How it holds
A Falcon signature can't be blocked mid-staging	A staged authorization commits to the signature's hash at creation and checks it on the final chunk
A wrong key can't brick the vault	Possession is proven by a Falcon signature over the vault before the key is confirmed; rotation needs both the current key's signature and the new key's proof of possession
Key bytes can't be altered	Key chunks are written in order, become immutable when complete, and are hash-checked against the commitment
Rent can't be spent	The SOL custody account keeps its rent-exempt reserve
Funds can only leave from the vault's own accounts	Sources are the vault's own token accounts, checked by address and owner
Tokens with unsafe extensions can't trap or redirect	Mint extensions are checked against a policy: extensions that need no arbitrary call are supported, transfer fees are handled exactly, and a transfer hook with a non-null program is refused

12.7 The Falcon-512 verifier

The vault verifies Falcon-512 with an SBF implementation (Blueshift's `solana-falcon512`, version 0.1.2, MIT licence) over raw 897-byte public keys. It rejects malformed key and signature headers (the padded signature header is `0x39`), coefficients out of range, the encoding of negative zero and non-zero padding. It hashes to the lattice with SHAKE256 over the nonce and message and applies the standard norm bound of 34,034,726. Verification costs 365,000 to 390,000 compute units as measured, within the 1.4M per-transaction limit. Offline signing in the SDK uses `@noble/post-quantum`'s padded Falcon-512.

12.8 Off-chain components

- **The indexer** isolates failures per account: one unreadable vault or market does not change the freshness reported for the rest, and every vault carries its own indexed slot.
- **The API** serves bounded, paginated snapshots and reports freshness honestly: indexed slot, lag and whether the chain was reachable.
- **The crank** holds only a fee-paying key. Every transaction it sends is permissionless and idempotent, so a duplicate or a crash never changes an outcome.
- **The browser clients** never hold user keys; they prepare transactions for the user's wallet to sign. The Falcon secret lives only in the page's memory and in the file the user saves.
- **No secret** — RPC key, crank key, deployer key — appears in source, logs or API responses.

12.9 How the code was checked

Each program was reviewed independently after it was built, and every finding of medium severity or above was fixed with a test that failed before the fix:

Program	Review outcome
<code>canary_rungs</code>	Account validation, claim binding and payout paths reviewed; findings fixed
<code>canary_markets</code>	No critical or high findings; fee-avoidance, split-steering, cap-erosion and sweep-coupling findings fixed
<code>canary_vault</code>	One high finding (an unbounded instant limit) fixed with the delay-gated budget; queue visibility, indexer isolation, Token-2022 handling, signature binding and staging findings fixed

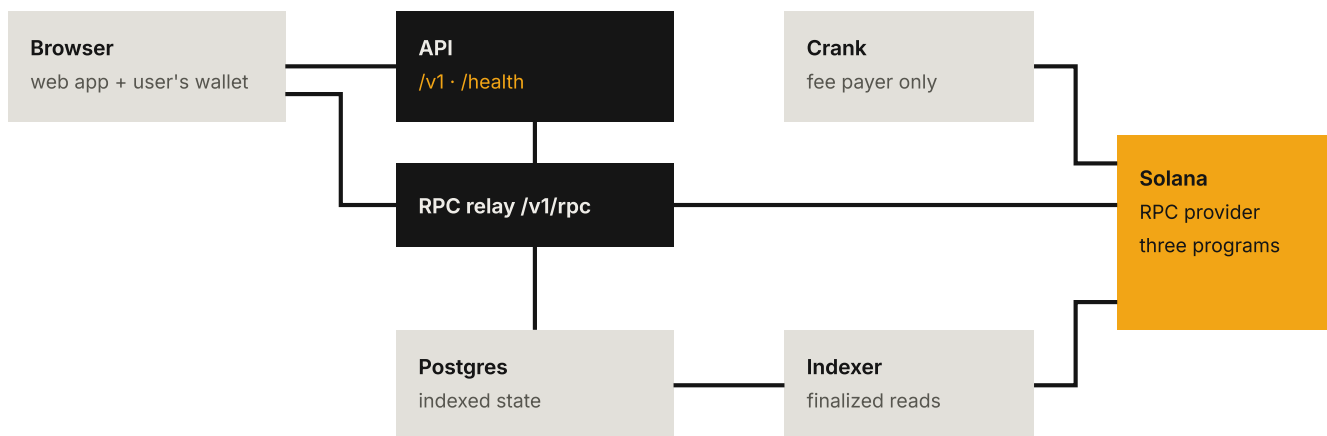
A reference verifier in a second language checks the same proofs off chain, and the parameter generator's transcript is reproducible bit for bit. Before mainnet, the rungs program receives two external audits and the markets and vault programs one each, and the Q-rung verifier gains a second, independently written implementation that must agree with the first.

13. Off-chain infrastructure

Everything that decides an outcome or moves money runs on chain. The off-chain services exist to make that state easy to read and easy to act on: an indexer that mirrors the programs' accounts into a database, an API that serves them with an honest account of their freshness, a relay that lets browsers reach Solana without exposing an RPC key, a crank that sends the permissionless instructions nobody else has an incentive to send, and the web app. None of them holds a key that can move user funds, and any of them can be replaced by anyone who reads the chain directly.

13.1 Components

Component	Runs	Reads	Writes	Holds
The three programs	Solana		All protocol state	Nothing off chain
Indexer	A long-running worker	Solana RPC, finalized; Jupiter's price API	Postgres	Nothing secret beyond the RPC URL
Postgres	A managed database			Indexed state, prices, checkpoints
API	An HTTP service	Postgres, Solana RPC for readiness	Nothing on chain	The RPC URL, which never reaches a client
RPC relay	Inside the API, at <code>/v1/rpc</code>	Solana RPC	Relays signed transactions only	No signing key
Crank	A long-running worker	Solana RPC; Jupiter's swap API	Permissionless transactions and buybacks	A fee-paying key, which is also the buyback key
Web app	A Next.js service	The API and the relay	Transactions the user's wallet signs	No key



Services and the direction of their reads and writes.

13.2 The indexer

The indexer runs a loop. Each pass reads the chain at `finalized` commitment, validates what it reads, and writes one consistent snapshot to Postgres inside a transaction that holds an advisory lock, so two indexers never interleave. A pass begins by reading each program's latest transaction signature and the Clock; only a program whose signature has changed is scanned again, so a quiet chain costs a few reads per pass.

Read	How
The chain's identity	Genesis hash, compared with the deployment's
Rungs	<code>getProgramAccounts</code> filtered by discriminator for <code>RungState</code> , <code>SponsorPosition</code> and Verified or Finalized <code>VerifyJob</code> accounts
Balances	Batched <code>getMultipleAccounts</code> of at most 100 keys: the Clock sysvar, the USDC mint, bounty vaults, receipt mints and sponsors' receipt accounts
Program authority	Each program's ProgramData account
Markets	A scan of the 237-byte market headers, then bodies for 32 markets per pass (the 16 largest and 16 in rotation) and up to 512 holders per side
Vaults	<code>Vault</code> accounts and active <code>QueuedSpend</code> accounts, in pages of 32 vaults
Prices	Jupiter's price API, at most once a minute: the YES and NO mints of up to 24 markets in rotation, and USDC to convert them
Transaction signatures	Best effort: up to 64 recent signatures per account, at most 8 candidates, within 1.5 seconds

Every rung is checked against the published manifest and its PDA before it is stored. Two safety rules hold across passes: a rung recorded as Solved is never recorded as Open again, and two different Solved records for the same rung stop the indexer rather than overwrite either.

The poll interval is 10 seconds in production (2 seconds on a local validator), configurable up to 60 seconds. After a failure the delay doubles, up to 30 seconds by default. u64 values are stored as `Decimal(20,0)` so no amount is ever rounded through a floating-point type.

Table group	Tables
Indexer state	<code>IndexerCheckpoint</code> , <code>ProgramDeployment</code> , <code>Asset</code>
Rungs	<code>Rung</code> , <code>GenerationTranscript</code> , <code>RungEvent</code> , <code>BountyBalance</code> , <code>SponsorReceipt</code> , <code>VerifyJob</code>
Markets	<code>Market</code> , <code>MarketProtocol</code> , <code>MarketPrice</code> , <code>PositionBalance</code>
Vaults	<code>QVault</code> , <code>PqKey</code> , <code>VaultBalance</code> , <code>QueuedSpend</code>
Fee sweeps	<code>FeeSweep</code> , <code>FeeSweepAllocation</code>

13.3 The API

The API serves JSON over HTTPS under `/v1`. Every response is an envelope `{data, meta}`; lists are pages `{items, nextCursor}` with a limit from 1 to 100 (default 50). Section 14.3 and the developer documentation list the routes; in summary:

Group	Routes
Configuration	<code>/v1/app-config</code> , <code>/v1/programs</code> , <code>/v1/eligibility</code>
Rungs and bounties	<code>/v1/rungs</code> , <code>/v1/rungs/{rungId}</code> , <code>/v1/rungs/{rungId}/events</code> , <code>/v1/rungs/{rungId}/claim-jobs</code> , <code>/v1/claim-jobs/{jobId}</code> , <code>/v1/bounties</code> , <code>/v1/transcripts/{hash}</code>
Markets	<code>/v1/markets</code> , <code>/v1/markets/{marketId}</code> , <code>/v1/markets/{marketId}/prices</code> , <code>/v1/q-date</code> , <code>/v1/totals</code>
Wallets and vaults	<code>/v1/wallets/{wallet}/receipts</code> , <code>/v1/wallets/{wallet}/positions</code> , <code>/v1/wallets/{wallet}/vaults</code> , <code>/v1/vaults/{vaultId}</code> , <code>/v1/vaults/{vaultId}/queued-spends</code>
Feed and health	<code>/v1/status-feed</code> , <code>/v1/landing</code> , <code>/health/live</code> , <code>/health/ready</code>
Relay	<code>POST /v1/rpc</code>

Responses are built from snapshots cached for one second with single-flight loading, so a burst of identical requests costs one database read. Each client address may make 1,200 requests per minute by default. Errors carry a stable code: `VALIDATION_FAILED`, `NOT_FOUND`, `UPSTREAM_UNAVAILABLE`, `RATE_LIMITED`, `CONFIGURATION_REQUIRED`, `FORBIDDEN` or `INTERNAL_ERROR`, with a request id and whether a retry may succeed.

Waiting and ready. In production the API checks, every ten seconds, that the chain it is pointed at matches the configured genesis hash, that the three programs are deployed and executable, that the USDC mint exists with 6 decimals, and that every rung in the published manifest is initialized with its transcript hash. Until all hold, `/v1/app-config` and `/health/ready` report state: `"waiting"` with the reasons, and the app shows the service as not yet open; once they hold, the state becomes `ready` without a restart. `/health/ready` answers 200 in both states, so a platform health check does not cycle the service while a deployment is still being initialized.

13.4 Freshness

Every snapshot reports how current it is:

Field	Meaning
<code>commitment</code>	<code>finalized</code>
<code>indexedSlot</code> , <code>indexedAt</code>	The slot and time of the snapshot
<code>chainHeadSlot</code> , <code>lagSlots</code>	The chain's latest slot and the distance to it
<code>chainTime</code>	The Clock sysvar's time at the snapshot
<code>rpcReachable</code> , <code>lastSuccessfulRpcAt</code>	Whether the last pass reached the RPC
<code>state</code>	<code>fresh</code> , <code>indexer-behind</code> or <code>rpc-unreachable</code>

The state is `rpc-unreachable` when the last pass could not reach the chain, `indexer-behind` when the snapshot is older than 60 seconds or more than 150 slots behind, and `fresh` otherwise. If the database is unavailable, the API serves its last snapshot marked `indexer-behind` rather than nothing. Each vault carries its own indexed slot, so one unreadable vault does not change the freshness reported for the others.

13.5 The RPC relay

Browsers need an RPC endpoint to read accounts, fetch a blockhash and submit signed transactions. Production RPC endpoints carry an API key that must not be published. The relay at `POST /v1/rpc` forwards a fixed set of JSON-RPC methods to the deployment's RPC provider and returns the result, so the browser never sees the provider's URL.

Limit	Value
Methods	<code>getVersion</code> , <code>getGenesisHash</code> , <code>getSlot</code> , <code>getBlockHeight</code> , <code>getLatestBlockhash</code> , <code>isBlockhashValid</code> , <code>getEpochInfo</code> , <code>getAccountInfo</code> , <code>getMultipleAccounts</code> , <code>getBalance</code> , <code>getTokenAccountBalance</code> , <code>getTokenAccountsByOwner</code> , <code>getTokenSupply</code> , <code>getSignatureStatuses</code> , <code>getTransaction</code> , <code>getMinimumBalanceForRentExemption</code> , <code>getFeeForMessage</code> , <code>simulateTransaction</code> , <code>sendTransaction</code> , <code>getSignaturesForAddress</code>
Request body	At most 65,536 bytes; a batch of 1 to 5 requests
Addresses	Must be valid public keys; <code>getMultipleAccounts</code> at most 100
Signatures	<code>getSignatureStatuses</code> at most 32; <code>getSignaturesForAddress</code> limit 1 to 20
Transactions	Base64, 65 to 1,232 bytes, must deserialize as a Solana transaction
Rate	600 requests and 60 transactions per client per minute by default

The relay has no signing key. It can only forward transactions a wallet has already signed, and it validates their size and shape before forwarding. A client that prefers its own RPC can use one; nothing in the protocol depends on the relay.

13.6 The crank

Several instructions exist to be sent by anyone: `resolve`, `sweep_bounty`, `sweep_treasury` and `allocate_deferred`. The crank sends them so that markets resolve and fees reach the bounty vaults and the buyback vault without anyone having to remember to. It also sends the buyback.

Action	When the crank sends it
<code>resolve</code>	A market's target rung is Solved (a quantum-signal market waits 300 slots after the event); or the expiry has passed and the observation is unset or more than 300 slots old
<code>sweep_bounty</code>	A market holds a pending bounty share for a rung
<code>sweep_treasury</code>	A market holds a pending protocol share
<code>allocate_deferred</code>	A market holds deferred bounty share and some rung now has weight
A buyback	The buyback vault holds at least 100 USDC: one transaction of <code>buyback_begin</code> , a Jupiter swap of the whole balance quoted at 1 % slippage, and <code>buyback_finish</code>

Each pass plans at most 64 actions with a rotating cursor, so no market starves; sends them in batches of at most 4 instructions with at most 3 in flight; and polls every 10 seconds, backing off to 30 seconds on failure. The crank's key pays transaction fees and signs buybacks as the configured buyback key. Every other instruction it sends is permissionless and idempotent, so a duplicate, a crash or a second crank run by someone else never changes an outcome; a buyback that fails, or finds no route, leaves the USDC in the vault for a later pass. Claim steps and `trip` are left to the clients that need them: a solver's client drives its own claim, and anyone can trip a vault whose trigger has fallen.

13.7 The status feed

`GET /v1/status-feed` serves a compact, unwrapped document for integrators that poll:

```
{ schemaVersion: "canary.status.v1", contractVersion, mode, network, generatedAt, freshness, programId, rungs: [{ id, account, state, transcriptHash }] }
```

Each rung's `state` is either `{status: "Open"}` or `{status: "Solved", eventTs, eventSlot, recipient, proofHash, finalizedSlot, finalizedAt, transactionSignature}`. `generatedAt` is the time of the indexed snapshot, not of the request. Appendix E gives the full schema. The feed is a convenience; a program or a cautious integrator reads `RungState` directly (Section 14).

13.8 The app configuration

`GET /v1/app-config` tells the web app and any other client which deployment it is talking to:

Field	Production value
<code>schemaVersion</code>	<code>canary.app-config.v1</code>
<code>appEnv</code> , <code>state</code> , <code>waitingReasons</code>	<code>production</code> , <code>waiting</code> or <code>ready</code> , and why it is waiting
<code>cluster</code> , <code>genesisHash</code> , <code>commitment</code>	The Solana cluster, its genesis hash, <code>finalized</code>
<code>rpcUrl</code>	The API's relay, <code>{API}/v1/rpc</code>
<code>programIds</code>	The three program ids
<code>collateral</code>	The USDC mint, 6 decimals
<code>rungs</code>	Per rung: id, label, state address, transcript hash, the public parameters a claim needs, and whether it is claimable now
<code>faucet</code>	Disabled

13.9 Market eligibility

Markets exclude US persons in v0. The API's `/v1/eligibility` answers from the deployment's location source: either a country header set by a trusted proxy and authenticated with a shared token, or the visitor's own attestation of their country. The web app asks before showing market actions. This is an interface policy; the markets program itself has no notion of location.

13.10 Hosting

The reference deployment runs on Railway as five services in one project:

Service	Start	Health	Notes
web	<code>next start</code> on the injected port	<code>/</code>	Static pages and the app; reads the API's public origin at build
api	Migrations, then the API server	<code>/health/live</code>	Public; CORS limited to the web origin
indexer	Migrations, then the indexer loop		Private
crank	Migrations, then the crank loop		Private; holds the fee-paying key
Postgres	Managed		Private; referenced by the other services

Schema migrations run when a service starts, never during the build, because a build cannot reach the private network the database lives on. Every service reads its port from the platform, and cross-service values such as the database URL are platform references rather than copied secrets. Section 16 and the deployment guide give the exact settings.

14. Integration

Canary's output is one account per rung. A program that wants to react to a fall reads that account in its own instruction; a service that wants to watch the ladder reads it over RPC or through the API. This section shows all three, and the rules that make a read safe.

14.1 What to read

Question	Read
Has the quantum signal fired?	Q-160's <code>RungState.status</code>
Did it fire for quantum reasons?	The quantum-signal market's outcome, once C-160 is on the ladder (Section 8.4)
How far up the ladder has it reached?	The highest Solved rung among Q-160, Q-192 and Q-224
Can secp256k1 or Ed25519 keys be taken?	T-256k's or T-255e's status
When did it happen?	<code>event_ts</code> and <code>event_slot</code> of the Solved rung
What does the market expect?	YES prices across expiries: <code>/v1/q-date</code> (Section 8.9)

The status says whether; `event_ts` says when. A reader that only needs a switch reads `status`. A reader that settles anything against a date, as the markets program does, also reads `event_ts` and checks the claim window.

14.2 The rules of a safe read

1. **Owner.** The account must be owned by the deployment's `canary_rungs` program id.
2. **Address.** It must be the PDA of `["rung", p]` under that program, for the rung id you mean.
3. **Type.** Its first 8 bytes must be `RungState`'s discriminator, `67 56 8b 18 51 ff d0 fa`.
4. **Version.** Byte 8 must be 2, and byte 9 must be `p`.
5. **Window.** If Solved, $\text{event_slot} \leq \text{finalize_slot} \leq \text{event_slot} + 300$.
6. **Fail closed.** If any check fails, treat the read as an error, not as Open and not as Solved.

Rules 1 and 2 together rule out every spoofed account: only the rungs program can create an account at its PDA. Rule 6 is what makes the read safe in both directions. A vault that treated a malformed account as Open would let a stolen key spend; a market that treated it as Solved would pay the wrong side.

14.3 From a Solana program

The markets and vault programs read rungs through Anchor's typed account constraint, using the `canary-rungs` crate with its `cpi` feature:

```
#[derive(Accounts)]
#[instruction(rung_id: u8)]
pub struct WatchRung<'info> {
    #[account(
        seeds = [b"rung", &[rung_id]],
        bump = rung.bump,
        seeds::program = canary_rungs::ID,
    )]
    pub rung: Account<'info, canary_rungs::RungState>,
}
```

`Account<RungState>` checks the owner and the discriminator; the seeds constraint checks the address. The handler then checks the version, the id and, when Solved, the window. Without Anchor, the same read is a few lines over the raw account:

```
pub enum Rung { Open, Solved { event_ts: i64, event_slot: u64 } }

const RUNG_STATE: [u8; 8] = [0x67, 0x56, 0x8b, 0x18, 0x51, 0xff, 0xd0, 0xfa];

pub fn read_rung(account: &AccountInfo, rungs_program: &Pubkey, rung_id: u8) → Result<Rung, ProgramError> {
    let (address, _) = Pubkey::find_program_address(&[b"rung", &[rung_id]], rungs_program);
    if account.key ≠ &address || account.owner ≠ rungs_program {
        return Err(ProgramError::InvalidAccountData);
    }
    let d = account.try_borrow_data()?;
    if d.len() ≠ 271 || d[..8] ≠ RUNG_STATE || d[8] ≠ 2 || d[9] ≠ rung_id {
        return Err(ProgramError::InvalidAccountData);
    }
    let u64_at = |o: usize| u64::from_le_bytes(d[o..o + 8].try_into().unwrap());
    match d[11] {
        0 ⇒ Ok(Rung::Open),
        1 ⇒ {
            let (event_slot, finalize_slot) = (u64_at(151), u64_at(263));
            if finalize_slot < event_slot || finalize_slot > event_slot + 300 {
                return Err(ProgramError::InvalidAccountData);
            }
            Ok(Rung::Solved { event_ts: u64_at(143) as i64, event_slot })
        }
        _ ⇒ Err(ProgramError::InvalidAccountData),
    }
}
```

Most of a read's cost is the address derivation; storing the bump (byte 12) and using `create_program_address` makes it cheaper. The account is passed read-only, so any number of programs can read the same rung in the same slot without contention.

14.4 Over RPC

```
import { Connection, PublicKey } from '@solana/web3.js'

const RUNG_STATE = Buffer.from('67568b1851ffd0fa', 'hex')

export async function readRung(connection: Connection, rungsProgram: PublicKey, id: number) {
    const [address] = PublicKey.findProgramAddressSync([Buffer.from('rung'), Buffer.from([id])], rungsProgram)
    const account = await connection.getAccountInfo(address, 'finalized')
    if (!account || !account.owner.equals(rungsProgram)) throw new Error('Not a rung')
    const d = account.data
    if (d.length ≠ 271 || !d.subarray(0, 8).equals(RUNG_STATE) || d[8] ≠ 2 || d[9] ≠ id) throw new Error('Unexpected state')
    if (d[11] === 0) return { status: 'Open' as const }
    const eventSlot = d.readBigUInt64LE(151)
    const finalizeSlot = d.readBigUInt64LE(263)
    if (finalizeSlot < eventSlot || finalizeSlot > eventSlot + 300n) throw new Error('Outside the claim window')
    return {
        status: 'Solved' as const,
        eventTs: new Date(Number(d.readBigInt64LE(143)) * 1000),
        eventSlot,
        recipient: new PublicKey(d.subarray(159, 191)).toBase58(),
    }
}
```

Read at `finalized` commitment. A Solved status at `confirmed` can, in principle, be rolled back with its fork; at `finalized` it cannot. To watch rather than poll, subscribe to the six rung addresses with `accountSubscribe`; the accounts change only when a rung is funded, claimed or reclaimed from.

14.5 Through the API

For dashboards, alerts and services that do not want to run RPC infrastructure, the status feed returns every rung in one request, with its freshness:

```
curl -s "$CANARY_API/v1/status-feed"
```

where `$CANARY_API` is the deployment's API origin. Check `freshness.state` before acting: `fresh` means the snapshot is at most 60 seconds and 150 slots old. Each rung's `account` field is its `RungState` address, so a client can confirm anything important against the chain with the read in Section 14.4.

14.6 Client libraries

The web app is built on TypeScript clients that prepare transactions without signing or sending them; a wallet signs, and the caller sends. They are part of the Canary source and usable by any integrator:

Client	Prepares
<code>RungsClient</code>	Funding a bounty and a claim from a secret scalar, which never leaves the caller; decoding rung, sponsor and job accounts
<code>MarketsClient</code>	Minting, merging and redeeming pairs; decoding markets
<code>VaultTransactionClient</code>	Creating a vault and registering its key, deposits, Ed25519 spends and queues, executing queues, Falcon spends and vetoes, tripping
Falcon SDK	Key generation, the vault action digest and offline signing, with export and import of the secret key file
<code>createCanaryClient</code>	Typed calls to the API's routes

Every prepared transaction carries an explicit compute limit, the wallet as fee payer, a recent blockhash and a distinct memo, and is at most 1,082 bytes so a wallet can add its own instructions within the 1,232-byte limit.

14.7 Patterns

Integrator	Reads	Reaction
A bridge or a lending protocol	Q-160 or Q-192 in its own instruction	Pause withdrawals that rely on elliptic-curve signatures, or route them through a delay
A treasury	Its Q-Vault, which reads its chosen trigger	Ed25519 stops; the Falcon key moves the funds
A custodian	The status feed, then <code>RungState</code>	Start the planned key migration with the lead time the ladder gives (Section 3.6)
A fund or an insurer	YES and NO tokens, which are plain SPL tokens	Hold, lend or price them like any other token; they redeem against the same rung
A multisig	A trigger rung, through a voter that reads it (Section 17.3)	Require a post-quantum approval after the fall

Each of these needs no permission from Canary and no account with it: the rung is a public account, and reading it is free.

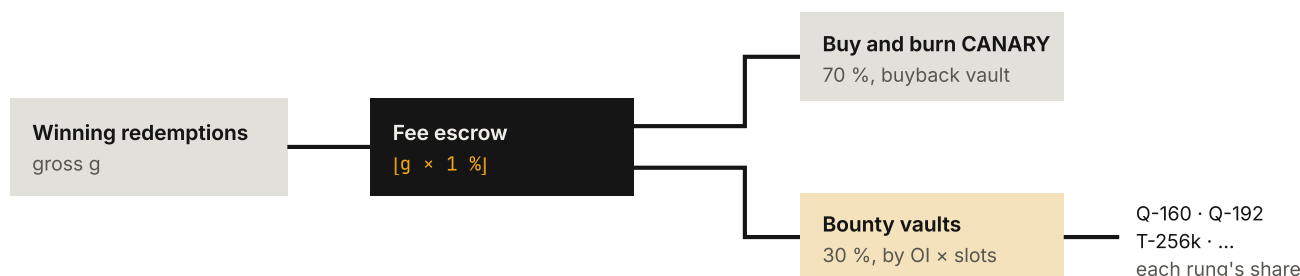
15. Economics

15.1 Fee lines

Line	Rate	Paid by	Note
Settlement fee	1.0 % of winning redemptions	The winning side, at redemption	In years without an event, mostly NO holders
Minting and merging pairs	None		
Sponsorship	0 % protocol cut		Every unit funds the bounty
Q-Vault	Free		No deposit, spend or assets-under-management fee
Status feed	Free		The public feed and API
Collateral yield skim (v1)	10 % of yield on collateral		With yield-bearing collateral; winners receive principal plus the remaining yield
Status attestation (planned)	Annual contract	Custodians, fund issuers	A service-level API and periodic monitoring attestations

15.2 Where fees go

Every settlement fee is swept 70 % to the buyback vault and 30 % to the bounty vaults, split across rungs by time-weighted open interest (Section 8.8). The buyback vault's USDC buys CANARY on the market and burns it (Section 15.5). A rung that the market hedges against grows its own bounty in proportion to how long and how heavily it is hedged, which strengthens the incentive to claim exactly where the demand for the signal is.



Settlement fees, from redemption to the CANARY burn and the bounty vaults.

15.3 Fee flow at three levels of open interest

The table assumes open interest turns over once a year at expiry and, for the v1 skim, a 4 % yield on collateral. These are assumptions for scale, not forecasts.

Open interest	Settlement fees	To the CANARY burn (70 %)	To bounties (30 %)	v1 yield skim	Total per year
\$5M	\$50k	\$35k	\$15k	\$20k	~\$70k
\$50M	\$500k	\$350k	\$150k	\$200k	~\$0.7M
\$500M	\$5M	\$3.5M	\$1.5M	\$2M	~\$7M

15.4 The bounty layer is a public good

Bounties are funded by sponsors and by the 30 % fee share. Sponsors — exchanges, custodians, treasury companies and foundations — receive what a public good can offer: a named, verifiable rung, early warning, and a Token-2022 NonTransferable receipt that records their contribution and carries their share of any unclaimed bounty after 2040-01-01. Because sponsorship carries no protocol cut, every sponsored unit is visibly at work in the bounty.

15.5 The CANARY token

CANARY is Canary's token. Its design follows from the same requirement as everything else in the protocol: the trigger must stay credibly neutral, so the token sits entirely on the revenue side.

Supply	1,000,000,000, fixed; no mint authority remains after launch
Launch	100 % to one public launch on equal terms
Allocations	None: no presale, team, investor, airdrop or reserved pools, and no vesting
Buyback	The protocol's revenue buys CANARY on the market: the protocol's 70 % of every settlement fee, and in v1 the yield skim
Burn	Every CANARY bought is burned, so supply only falls

The buyback runs on the USDC in the buyback vault. `sweep_treasury` moves each market's pending protocol share there (Section 8.7). Whenever the vault holds at least 100 USDC, the crank sends one transaction that advances the whole balance with `buyback_begin`, swaps it for CANARY through Jupiter into the program's own CANARY account, and ends with `buyback_finish`, which requires every unit spent and at least the CANARY quoted at 1 % slippage, then burns it all (Section 9.4). Each burn shows on chain as a lower supply of the mint, and `MarketsConfig` counts the USDC spent and the CANARY burned. With fee F per settlement, the amount burned is

$$\text{burned} = \text{CANARY bought with } 0.70 \times F, F = [1 \% \times \text{winning redemptions}]$$

so the burn tracks settlement, not trading: minting, merging, sponsorship and Q-Vaults carry no fee. At the open-interest levels of Section 15.3, the protocol share is \$35k, \$350k and \$3.5M a year.

CANARY has no role in the trigger or in any payout:

CANARY is never	Why
Collateral	A rung falling would most likely move crypto prices exactly when payouts come due; a crypto-native asset as collateral would lose value at the moment it is needed. Markets and bounties settle in USDC, and SOL, staked SOL and BTC are excluded for the same reason
A resolution input	Markets resolve from <code>RungState</code> and the clock alone
A vote on rung parameters, resolution or vault rules	The rungs program is immutable after audit; markets and vault move to a timelocked multisig and then freeze
A payout backstop	Bounties pay from their vaults and markets from their own collateral

The other functions a token is sometimes given are already filled without one:

Function	Filled by
Funding bounties	Sponsor deposits plus 30 % of settlement fees
Underwriting	NO holders, directly, in USDC
Recognizing sponsors	Token-2022 NonTransferable receipts carrying the post-sunset reclaim right
Incentivizing liquidity	Market-maker fee rebates in USDC

15.6 Market scale

Anchor	Figure	Source
BTC behind exposed keys	~6M BTC (~30 %, ~\$400bn) to ~6.9M BTC	[3][4]
Value secured by elliptic-curve cryptography in digital assets	> \$2.5T	[5]
Collateralized tail-risk capital (cat bonds and ILS)	\$64.7B outstanding	[30]
On-chain cover precedent	Nexus Mutual: \$7.00B protected, 13,000 covers since 2019	[29]

The first market Canary serves directly is Solana-native: DAOs, foundations and funds holding Ed25519-exposed balances, who can both hedge with rung-event markets and move funds into Q-Vaults, and traders expressing a view on the date curve. Bitcoin-side proposals can reference RungState off chain, since rung state is public.

16. Upgrade authority and deployment

A Solana program is deployed with an upgrade authority, a key that can replace its code [25]. Canary's design goal of a credibly neutral trigger (Section 1.4) means that authority must end where it matters most. This section describes the authority each program has, how it is retired, how a deployment is built and initialized, and how anyone can check a deployment against its sources.

16.1 Authority in each program

Program	What its upgrade authority can do in the program	Instructions that need it	Plan
<code>canary_rungs</code>	Initialize a rung	<code>initialize_rung</code>	Revoked after audit; the rungs program becomes immutable
<code>canary_markets</code>	Create the configuration; the configuration's administrator enables rungs with their caps, sets the CANARY mint once and names the buyback key	<code>initialize_config</code> , <code>configure_rung</code> , <code>configure_buyback</code>	A 3-of-5 multisig with a 7-day timelock; revoked in v1
<code>canary_vault</code>	Nothing: the program has no privileged instruction	none	The same multisig policy; later a Falcon-gated multisig voter

No instruction in any program lets an authority or administrator move user funds, change a rung's parameters, change an outcome, change a cap once set, or alter a vault. The authority's only other power is the platform's own: deploying new program code. That is why the rungs program's authority is revoked, and why every program's current authority is public.

16.2 Revocation and what it fixes

Revoking the rungs program's upgrade authority sets its ProgramData authority to none. From then on:

- the verifier, the curve constants and the transcript hashes can never change;
- no new rung can be initialized, because `initialize_rung` requires the authority;
- the bounty vaults' rules, including the sunset, are final.

The indexer reads each program's ProgramData account on every pass, and `/v1/programs` reports whether each authority is revoked or still held. A revocation is a single transaction that anyone can see on chain.

16.3 Building a deployment

A production build compiles the deployment's identity into the programs:

Build input	Where it ends up
The three program ids	Each program's <code>declare_id</code> , and the rungs program id inside the markets and vault programs
The USDC mint	Required by <code>initialize_rung</code> and <code>initialize_config</code>
The canonical parameter suite	The rungs program's curve constants and transcript hashes

The parameter suite is generated from the published seed (Section 4.8) as part of the build. The build refuses test rungs and any parameter source containing test material, so a production program cannot be built with a rung whose logarithm is known.

16.4 Initializing it

After the three programs are deployed, a one-time initialization brings the ladder up:

#	Step	Instruction	Signer
1	Create each rung in the suite	<code>initialize_rung(p)</code>	Rungs upgrade authority
2	Create the markets configuration	<code>initialize_config</code>	Markets upgrade authority
3	Create the buyback vault and the buyback CANARY account	Associated token accounts of <code>["buyback"]</code>	Any payer
4	Set the CANARY mint and the buyback key	<code>configure_buyback(mint, key)</code>	Markets administrator
5	Enable Q-160, Q-192 and T-256k with their caps	<code>configure_rung(p, cap)</code>	Markets administrator
6	Create the rung-event markets at the three v0 expiries	<code>create_market(p, T, 0)</code>	Any payer

Q-160's cap defaults to 10,000,000 USDC; the Q-192 and T-256k caps are set explicitly at this step, and none can change afterwards. The deployment scripts are idempotent: each step checks whether its account already exists, so an interrupted initialization resumes where it stopped. When step 1 completes, the API's readiness check passes (Section 13.3) and the service opens.

16.5 The deployment record

Deployment writes a public record, `canary.deployment.v1`, that the API reads:

Field	Content
<code>cluster</code> , <code>genesisHash</code>	The Solana cluster and its genesis hash
<code>programIds</code>	The three program ids
<code>programs</code>	Per program: id, ProgramData address, SHA-256 and length of the deployed ELF, upgrade authority, and the slot at which it was verified
<code>collateral</code>	The USDC mint, 6 decimals
<code>parameterManifestPath</code>	The published parameter manifest
<code>rungs</code>	Each rung's id, label, <code>RungState</code> address and transcript hash
<code>policy</code>	Settlement fee 100 bps, shares 7,000 / 3,000 bps, the sunset, the claim window of 300 slots, and the caps
<code>buyback</code>	The buyback signer and vault, the CANARY mint, the buyback key, and the USDC spent and CANARY burned so far

The record contains no RPC credential and no key. It is the same information anyone can read from the chain, collected in one place.

16.6 Checking a deployment

Anyone can check a Canary deployment without trusting its operators:

1. **Code.** Fetch each program's ELF from its ProgramData account and compare its SHA-256 with a build from the published source and the published build inputs.
2. **Authority.** Read each program's upgrade authority from its ProgramData account. For the rungs program after audit, it must be none.
3. **Parameters.** Regenerate the parameter suite from the published seed (Section 4.8); every transcript hash must equal the one in the corresponding `RungState`.
4. **Collateral and buyback.** Read `MarketsConfig`: the collateral mint must be USDC and `canary_mint` the published CANARY mint. The buyback vault is derived from `["buyback"]`, and `total_buyback_usdc` and `total_canary_burned` count every buyback so far.

5. **Caps.** Read each enabled rung's cap from `MarketsConfig.weights`.

6. **Bounties.** Read each rung's bounty vault balance and receipt supply.

Every value in this list is on chain or derived from published inputs; none depends on the API.

17. Roadmap

17.1 v0

Canary v0 consists of:

- the three programs: `canary_rungs` with bounty vaults and claim verification, `canary_markets` with USDC collateral, and `canary_vault` with a Falcon-512 path and a rung-gated Ed25519 path;
- the parameter generator and its transcript;
- the claim CLI and the in-browser claim builder;
- the indexer, the permissionless crank and the public API with the status feed;
- the web app for sponsors, traders, vault users and solvers;
- the browser clients and the offline Falcon-512 SDK;
- the CANARY token: 1,000,000,000 fixed supply, launched in full to the public, bought and burned with the protocol's fee share.

Markets run at three expiries on Q-160, Q-192 and T-256k with per-rung open-interest caps. C-160 and the quantum-signal market join when C-160's point count is certified.

17.2 Toward the mainnet launch

Step	Done when
Independent reproductions of the rung parameters	At least three parties reproduce the transcript bit for bit
A second, independently written Q-rung verifier	Both verifiers accept the same proofs and reject the same malformed ones across differential fuzzing
Audits	The rungs program twice; markets and vault once each
Launch sponsors	Three to five sponsors fund named rungs
Mainnet with the rungs authority revoked	The revocation is visible on chain

17.3 v1

Item	What it adds
Yield-bearing collateral	Markets collateralized in yield-bearing USD (for example USDY for non-US users [31]); winners receive principal plus yield, less a 10 % skim
A second post-quantum key in Q-Vault	An optional Winternitz one-time signature key alongside Falcon-512 [19]
A trigger-aware multisig voter	A Falcon-gated Squads voter PDA that reads a trigger rung [17]
Frozen markets program	<code>canary_markets</code> upgrade authority revoked
Underwriting pool	Pool shares as plain SPL claims, with no governance rights
Readers on other chains	Rung state referenced from other ecosystems

17.4 Metrics

Horizon	Target
Launch	≥ 3 independent parameter reproductions; ≥ \$250k in bounties from ≥ 3 sponsors; verification within the measured compute budgets
Six months	≥ \$2M open interest; ≥ 2 market makers; ≥ \$5M in Q-Vaults; ≥ 1 integrator reading RungState

References

1. BitMEX Research, "Avoiding An Unnecessary Quantum Freeze", April 2026. <https://www.bitmex.com/blog/Avoiding-An-Unnecessary-Quantum-Freeze>
2. E. Aronesty, "Weak Quantum Bounty Ceremony", bitcoin-dev mailing list, 2026-05-31, with replies (secp160k1, QCAP). <https://groups.google.com/g/bitcoind/c/d7o74e-teNo/m/ILKjufQZAgAJ>
3. Allianz Research, quantum finance special, 2026-07-29. https://www.allianz.com/en/economic_research/insights/publications/specials_fmo/260729-quantum-finance.html
4. The Block, Project Eleven Q-Day report, 2026-05-06. <https://theblock.co/post/400274/project-eleven-warns-quantum-moment-all-at-once-q-day-early-2030>
5. The Quantum Insider, "15-Bit ECC Key Broken on Quantum Hardware Wins Q-Day Prize", 2026-04-24. <https://thequantuminsider.com/2026/04/24/project-eleven-q-day-prize-quantum-ecc-attack/>
6. Google Research blog, "Safeguarding cryptocurrency by disclosing quantum vulnerabilities responsibly", 2026-03-31. <https://www.research.google/blog/safeguarding-cryptocurrency-by-disclosing-quantum-vulnerabilities-responsibly/>
7. R. Babbush et al., "Securing Elliptic Curve Cryptocurrencies against Quantum Vulnerabilities", arXiv:2603.28846. <https://arxiv.org/abs/2603.28846>
8. M. Cain et al., "Shor's algorithm is possible with as few as 10,000 reconfigurable atomic qubits", arXiv:2603.28627. <https://arxiv.org/abs/2603.28627>
9. M. Roetteler, M. Naehrig, K. M. Svore, K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms", arXiv:1706.06752. <https://arxiv.org/abs/1706.06752>
0. P.-L. Dallaire-Demers, W. Doyle, T. Foo, "Brace for impact: ECDLP challenges for quantum cryptanalysis", arXiv:2508.14011. <https://arxiv.org/abs/2508.14011>
1. Trail of Bits, "We beat Google's zero-knowledge proof of quantum cryptanalysis", 2026-04-17. <https://blog.trailofbits.com/2026/04/17/we-beat-googles-zero-knowledge-proof-of-quantum-cryptanalysis/>
2. J. Bos, "112-bit prime ECDLP solved", cryptography mailing list, 2009-07-09. <https://www.metzdowd.com/pipermail/cryptography/2009-July/015476.html>
3. "Solving 114-bit ECDLP for a Barreto-Naehrig curve", Springer LNCS. https://link.springer.com/chapter/10.1007/978-3-319-78556-1_13
4. Bitcoin.com News, Bitcoin puzzle status, 2026-06-23. <https://news.bitcoin.com/key-hunters-eye-58-87m-bitcoin-puzzle-as-916-btc-sits-unsolved-in-78-addresses/>
5. Bitcoin-Puzzle-Info: RCKangaroo throughput, solved and unsolved list. <https://github.com/roadhero/Bitcoin-Puzzle-Info>
6. Jump Crypto, "Quantum migration paths for Solana", 2026-04-27. <https://jumpcrypto.com/resources/quantum-migration-paths-for-solana>
7. Solana Compass, Squads Falcon-512 multisig test and Blueshift Vector, 2026-10-13. <https://solanacompass.com/news/squads-builder-says-falcon-512-post-quantum-multisig-voting-has-been-tested-on-solana-mainnet>
8. SIMD-0461, Falcon syscall proposal. <https://github.com/solana-foundation/solana-improvement-documents/pull/461>
9. solana-winternitz-vault. <https://github.com/deanmlittle/solana-winternitz-vault>
0. The Quantum Insider, Solana Winternitz vault, 2025-01-04. <https://thequantuminsider.com/2025/01/04/solana-takes-a-step-toward-pqc-era-with-quantum-resistant-vault/>
1. Agave execution budget constants (compute-unit limit, syscall costs). https://github.com/anza-xyz/agave/blob/master/program-runtime/src/execution_budget.rs
2. Solana documentation, Transactions (the 1,232-byte limit). <https://solana.com/docs/core/transactions>
3. Solana documentation, Versioned transactions. <https://solana.com/docs/core/transactions/versioned-transactions>
4. Solana documentation, Accounts (Ed25519 addresses). <https://solana.com/docs/core/accounts>

5. Solana documentation, Programs (upgrade authority). <https://solana.com/docs/core/programs>
6. Solana documentation, Syscall reference. <https://solana.com/docs/core/programs/syscall-reference>
7. groth16-solana (compute-unit costs). <https://github.com/Lightprotocol/groth16-solana>
8. SECG, SEC 2: Recommended Elliptic Curve Domain Parameters, version 2.0. <https://www.secg.org/sec2-v2.pdf>
9. Nexus Mutual. <https://nexusmutual.io/> and <https://docs.nexusmutual.io/overview/>
0. Artemis, catastrophe bond and ILS market dashboard, viewed 2026-10-15. <https://www.artemis.bm/dashboard/>
1. Ondo Finance, USDY. <https://ondo.finance/usdy>
2. S. Josefsson, I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, 2017. <https://www.rfc-editor.org/rfc/rfc8032>
3. C. P. Schnorr, "Efficient signature generation by smart cards", Journal of Cryptology 4, 1991. <https://doi.org/10.1007/BF00196725>
4. D. Pointcheval, J. Stern, "Security arguments for digital signatures and blind signatures", Journal of Cryptology 13, 2000. <https://doi.org/10.1007/s001450010003>
5. P.-A. Fouque et al., "Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU", specification v1.2, 2020. <https://falcon-sign.info/>
6. SECG, SEC 2: Recommended Elliptic Curve Domain Parameters, version 1.0, 2000 (secp160k1). <https://www.secg.org/SEC2-Ver-1.0.pdf>

Appendix A. Account layouts

Offsets are in bytes from the start of the account data. Every account begins with Anchor's 8-byte discriminator, `SHA-256("account:<Name>")[0..8]`; fields follow in Borsh encoding, integers little-endian, `Pubkey` as 32 bytes, `bool` as one byte.

A.1 Discriminators

Account	Program	Discriminator (hex)	Size
<code>RungState</code>	<code>canary_rungs</code>	<code>67568b1851ffd0fa</code>	271
<code>SponsorPosition</code>	<code>canary_rungs</code>	<code>87a4bc5ee0829c8f</code>	75
<code>VerifyJob</code>	<code>canary_rungs</code>	<code>22b650087c46810c</code>	360
<code>MarketsConfig</code>	<code>canary_markets</code>	<code>cce2e2a67c8f9e15</code>	1,130
<code>Market</code>	<code>canary_markets</code>	<code>dbbed53700e3c69a</code>	477
<code>Vault</code>	<code>canary_vault</code>	<code>d308e82b02987577</code>	332
<code>PqKey</code>	<code>canary_vault</code>	<code>3d88b0e71ba21567</code>	1,014
<code>PqAuthorization</code>	<code>canary_vault</code>	<code>f5ba432cf637f62f</code>	1,040
<code>QueuedSpend</code>	<code>canary_vault</code>	<code>81457801b66224c7</code>	227

A.2 RungState

Seeds `["rung", rung_id]`. Version 2.

Offset	Field	Type
0	discriminator	<code>[u8; 8]</code>
8	version	<code>u8</code>
9	rung_id	<code>u8</code>
10	family	<code>u8</code>
11	status	<code>u8</code>
12	bump	<code>u8</code>
13	bounty_bump	<code>u8</code>
14	receipt_bump	<code>u8</code>
15	collateral_mint	<code>Pubkey</code>
47	collateral_vault	<code>Pubkey</code>
79	receipt_mint	<code>Pubkey</code>
111	transcript_hash	<code>[u8; 32]</code>
143	event_ts	<code>i64</code>
151	event_slot	<code>u64</code>
159	recipient	<code>Pubkey</code>
191	proof_hash	<code>[u8; 32]</code>
223	total_funded	<code>u64</code>

Offset	Field	Type
231	total_paid	u64
239	total_reclaimed	u64
247	sponsor_count	u32
251	receipt_count	u32
255	initialized_slot	u64
263	finalize_slot	u64

family: 0 quantum, 1 classical control, 2 terminal. status: 0 Open, 1 Solved.

A.3 SponsorPosition

Seeds ["sponsor", rung_id, sponsor]. Version 1.

Offset	Field	Type
8	version	u8
9	rung_id	u8
10	bump	u8
11	owner	Pubkey
43	total_funded	u64
51	total_reclaimed	u64
59	first_funded_ts	i64
67	first_funded_slot	u64

A.4 VerifyJob

Seeds ["job", rung_id, recipient, nonce (u64 LE), proof_hash]. Version 2.

Offset	Field	Type	Notes
8	version	u8	
9	rung_id	u8	
10	bump	u8	
11	status	u8	0 Verifying, 1 Verified, 2 Finalized, 3 Rejected
12	recipient	Pubkey	
44	nonce	u64	
52	begin_ts	i64	Becomes the rung's event_ts
60	begin_slot	u64	Becomes the rung's event_slot
68	proof_hash	[u8; 32]	
100	r	[u8; 64]	R as x y, big-endian
164	s	[u8; 32]	Big-endian
196	e	[u8; 32]	The challenge, computed at begin
228	acc_x	[u32; 8]	Jacobian X, Montgomery form

Offset	Field	Type	Notes
260	acc_y	[u32; 8]	Jacobian Y
292	acc_z	[u32; 8]	Jacobian Z; 0 is the point at infinity
324	bit_cursor	u16	Bits processed
326	steps	u16	Step instructions executed
328	payer	Pubkey	Receives the rent at <code>close_job</code>

A.5 MarketsConfig

Seeds `["config"]`. Version 3, 1,215 bytes.

Offset	Field	Type
8	version	u8
9	bump	u8
10	rungs_program	Pubkey
42	collateral_mint	Pubkey
74	administrator	Pubkey
106	canary_mint	Pubkey
138	buyback_authority	Pubkey
170	pending_treasury	u64
178	total_treasury_swept	u64
186	total_fees_accrued	u64
194	total_bounty_budget	u64
202	total_bounty_swept	u64
210	total_buyback_usdc	u64
218	total_canary_burned	u64
226	weights	[RungWeight; 16]
1170	unallocated_bounty	u64
1178	buyback_active	u8
1179	buyback_amount	u64
1187	buyback_min_out	u64
1195	buyback_pre_usdc	u64
1203	buyback_pre_canary	u64
1211	buyback_begin_index	u16
1213	buyback_finish_index	u16

`canary_mint` holds the default key until `configure_buyback` sets it. The seven `buyback_` fields from offset 1178 describe an open buyback and are zero outside one.

A.6 RungWeight

59 bytes, sixteen of them from offset 226 of `MarketsConfig`. Offsets are within the element.

Offset	Field	Type	Notes
0	run_id	u8	
1	enabled	u8	
2	cap	u64	Unresolved open-interest cap; immutable
10	open_interest	u64	Current unresolved exposure
18	pending_bounty	u64	
26	total_bounty_swept	u64	Fees received by the rung's bounty vault
34	oi_slots	u128	$A(p)$, open interest $\times$ slots
50	last_weight_slot	u64	
58	no_exit	u8	1 for a Solved rung with no receipts outstanding

A.7 Market

Seeds ["mkt", run_id, expiry (i64 LE), kind, collateral_mint]. Version 3. Bytes 0–236 are the discovery header.

Offset	Field	Type	Notes
8	version	u8	
9	bump	u8	
10	run_id	u8	Target rung
11	kind	u8	0 rung-event, 1 quantum-signal
12	outcome	u8	0 unresolved, 1 YES, 2 NO
13	expiry	i64	
21	no_observation_slot	u64	
29	resolved_slot	u64	
37	run	Pubkey	Target RungState
69	control_run	Pubkey	C-160's RungState for kind 1; else zero
101	collateral_mint	Pubkey	
133	collateral_vault	Pubkey	Seeds ["collateral", market]
165	yes_mint	Pubkey	Seeds ["yes", market]
197	no_mint	Pubkey	Seeds ["no", market]
229	open_interest	u64	
237	total_minted	u64	
245	total_merged	u64	
253	total_redeemed	u64	
261	total_fees	u64	
269	created_slot	u64	
277	resolved_ts	i64	
285	total_fees_swept	u64	
293	pending_treasury	u64	

Offset	Field	Type	Notes
301	pending_bounty	[u64; 16]	Indexed like <code>MarketsConfig.weights</code>
429	creator	Pubkey	
461	created_ts	i64	
469	unallocated_bounty	u64	

A.8 Vault

Seeds `["qv", owner, index (u64 LE)]`. Version 2.

Offset	Field	Type	Notes
8	version	u8	
9	bump	u8	
10	sol_bump	u8	Seeds <code>["sol", vault]</code>
11	owner	Pubkey	
43	index	u64	
51	trigger_rung	Pubkey	
83	trigger_rung_id	u8	
84	mode	u8	0 Ed25519AndPq, 1 Lockdown
85	lockdown_reason	u8	0 none, 1 owner, 2 trigger
86	asset_policy	u8	1
87	instant_mint	Pubkey	Default key for SOL
119	instant_token_program	Pubkey	
151	instant_limit	u64	
159	delay_seconds	u32	
163	pq_key_hash	[u8; 32]	
195	active_pq_key	Pubkey	
227	key_epoch	u64	
235	key_confirmed	bool	
236	nonce	u64	Falcon nonce
244	ed_nonce	u64	Ed25519 nonce
252	created_ts	i64	
260	created_slot	u64	
268	locked_ts	i64	
276	locked_slot	u64	
284	instance_salt	[u8; 32]	
316	instant_spent	u64	
324	last_instant_ts	i64	

A.9 PqKey

Seeds ["pq", vault, payer, epoch (u64 LE), key_hash]. Version 1. The first 117 bytes form a scan header.

Offset	Field	Type
8	version	u8
9	bump	u8
10	vault	Pubkey
42	payer	Pubkey
74	epoch	u64
82	key_hash	[u8; 32]
114	written	u16
116	confirmed	bool
117	bytes	[u8; 897]

A.10 PqAuthorization

Seeds ["auth", vault, payer, auth_id (u64 LE)]. Version 2.

Offset	Field	Type
8	version	u8
9	bump	u8
10	vault	Pubkey
42	payer	Pubkey
74	auth_id	u64
82	action	Action (257 bytes)
339	written	u16
341	consumed	bool
342	signature	[u8; 666]
1008	signature_hash	[u8; 32]

A.11 Action

Embedded in PqAuthorization at offset 82; also the last 257 bytes of the Falcon digest preimage (Section 11.5).

Offset	Field	Type
0	operation	u8
1	key_epoch	u64
9	nonce	u64
17	mint	Pubkey
49	token_program	Pubkey
81	recipient	Pubkey
113	destination	Pubkey

Offset	Field	Type
145	amount	u64
153	queue	Pubkey
185	new_key	Pubkey
217	new_key_hash	[u8; 32]
249	expected_fee	u64

A.12 QueuedSpend

Seeds ["queue", vault, ed_nonce (u64 LE)]. Version 2.

Offset	Field	Type	Notes
8	version	u8	
9	bump	u8	
10	vault	Pubkey	
42	nonce	u64	
50	recipient	Pubkey	
82	destination	Pubkey	
114	mint	Pubkey	
146	token_program	Pubkey	
178	amount	u64	
186	queued_ts	i64	
194	unlock_ts	i64	
202	queued_slot	u64	
210	status	u8	0 Queued, 1 Executed, 2 Cancelled
211	completed_slot	u64	
219	expected_fee	u64	

Appendix B. Instruction reference

S marks a signer and W a writable account. "Permissionless" means the instruction needs no signer beyond the transaction's fee payer. Instruction discriminators are `SHA-256("global:<name>")[0..8]`.

B.1 canary_rungs

`initialize_rung(rung_id: u8)`

Account	S	W	Constraint
payer	✓	✓	The program's current upgrade authority
rung	✓		Created at <code>["rung", rung_id]</code>
bounty_authority			<code>["bounty", rung_id]</code>
collateral_mint			The configured USDC mint, 6 decimals
collateral_vault	✓		Created at <code>["collateral", rung_id]</code> , authority the bounty PDA
receipt_mint	✓		<code>["receipt", rung_id]</code> , unused and System-owned
token_program, token_2022_program, system_program, rent			
program, program_data			The rungs program and its ProgramData

Checks: parameters compiled for `rung_id`; collateral decimals 6; family mapping. Effects: creates the NonTransferable receipt mint (6 decimals, mint authority the bounty PDA, no freeze authority) and `RungState` (Open, transcript hash, initialized slot). Emits `RungInitialized`.

`fund_bounty(rung_id: u8, amount: u64)`

Account	S	W	Constraint
sponsor	✓	✓	
rung		✓	
collateral_mint			
source	✓		The sponsor's USDC account
collateral_vault	✓		The rung's bounty vault
receipt_mint		✓	
receipt_account	✓		The sponsor's Token-2022 associated account, created if needed
sponsor_position	✓		<code>["sponsor", rung_id, sponsor]</code> , created if needed
bounty_authority			
token programs, associated token program, system_program			

Checks: `amount > 0`; rung Open; `now < sunset`. Effects: `transfer_checked(amount, 6)` from source to vault; mints `amount` receipts; updates totals and, on the first deposit, the sponsor and receipt counts. Emits `BountyFunded`.

begin_claim(rung_id: u8, recipient: Pubkey, nonce: u64, r: [u8; 64], s: [u8; 32])

Account	S	W	Constraint
payer	✓	✓	Pays the job's rent
rung			
job		✓	Created at ["job", rung_id, recipient, nonce, proof_hash]
system_program			

Checks: rung Open; a staged rung; transcript hash matches the compiled one; engine setup valid; $0 < s < n$; R on the curve with coordinates below p. Effects: computes e; creates the job with the accumulator at infinity, `begin_ts`, `begin_slot` and payer. Emits `ClaimBegun`.

step_claim(max_bits: u16)

Account	S	W	Constraint
rung			
job		✓	

Permissionless. Checks: $\text{slot} \leq \text{begin_slot} + 300$; rung Open; job Verifying; $1 \leq \text{max_bits} \leq 16$. Effects: processes $\min(\text{max_bits}, \text{scalar_bits} - \text{cursor})$ bits; after the last, sets Verified if the result equals R, else Rejected. Emits `ClaimAdvanced`.

finalize_claim()

Account	S	W	Constraint
rung		✓	
job		✓	
bounty_authority			
collateral_mint			
collateral_vault		✓	
recipient_token		✓	A USDC account owned by the job's recipient
token_program			

Permissionless. Checks: $\text{slot} \leq \text{begin_slot} + 300$; rung Open; job Verified. Effects: transfers the vault's whole balance to `recipient_token`; sets the rung Solved with event time and slot from the job, the recipient, the proof hash, `total_paid` and `finalize_slot`; sets the job Finalized. Emits `RungSolved`.

close_job()

Account	S	W	Constraint
rung			
job		✓	Closed
refund		✓	Equal to the job's payer

Permissionless. Checks: the job is Finalized or Rejected, or the rung is Solved, or $\text{slot} > \text{begin_slot} + 300$; otherwise `JobStillActive`. Effects: closes the job and returns its rent to its payer.

claim_secp256k1(rung_id, recipient, nonce, r: [u8; 64], s: [u8; 32])

Account	S	W	Constraint
rung		✓	
bounty_authority			
collateral_mint			
collateral_vault		✓	
recipient_token		✓	Owned by <code>recipient</code>
token_program			

Permissionless. Checks: rung Open; T-256k; transcript; $s < n$; no degenerate scalar; $e \cdot Q + R = s \cdot G$. Effects: pays the vault's balance; sets Solved with event time and slot from the clock. Emits `RungSolved`.

claim_ed25519(rung_id, recipient, nonce, r: [u8; 32], s: [u8; 32])

Accounts as for `claim_secp256k1`. Permissionless. Checks: rung Open; T-255e; transcript; R not the identity; $0 < s < n$; $s \cdot G + (n - e) \cdot Q$ encodes to R. Effects as for `claim_secp256k1`.

reclaim_after_sunset(rung_id: u8, receipt_amount: u64)

Account	S	W	Constraint
sponsor	✓		
rung		✓	
sponsor_position		✓	Owned by the sponsor
bounty_authority			
collateral_mint			
collateral_vault		✓	
receipt_mint		✓	
receipt_account		✓	The sponsor's associated receipt account
destination		✓	A USDC account owned by the sponsor
token_programs			

Checks: $\text{now} \geq \text{sunset}$; $\text{receipt_amount} > 0$; $\text{receipt_amount} \leq \text{the sponsor's balance}$; $\text{supply} > 0$. Effects: $\text{amount} = \lfloor \text{vault} \times \text{receipt_amount} \div \text{supply} \rfloor$; burns the receipts; pays `amount`; updates `total_reclaimed` on the rung and the position. Emits `SponsorReclaimed`.

B.2 canary_markets

initialize_config()

Account	S	W	Constraint
payer	✓	✓	The program's current upgrade authority; becomes the administrator
config		✓	Created at <code>["config"]</code>
collateral_mint			The configured USDC mint, 6 decimals
system_program, program, program_data			

configure_rung(rung_id: u8, cap: u64)

Account	S	W	Constraint
administrator	✓		Equal to <code>config.administrator</code>
config		✓	
rung			<code>["rung", rung_id]</code> under the rungs program

Checks: rung version 2, matching id and collateral; cap > 0; rung not yet enabled; a free weight slot. Effects: enables the rung with its cap and starts its accumulator at the current slot.

create_market(rung_id: u8, expiry: i64, kind: u8)

Account	S	W	Constraint
payer	✓	✓	
config			
rung			The target rung
control_rung			C-160 for kind 1
market		✓	Created at <code>["mkt", rung_id, expiry, kind, mint]</code>
collateral_mint			
collateral_vault		✓	Created
yes_mint, no_mint		✓	Created, 6 decimals, authority the market
token_program, system_program, rent			

Permissionless. Checks: now < expiry ≤ 253,402,300,799; target Open and enabled; kind ≤ 1; for kind 1, target Q-160 and control C-160, authenticated and Open.

mint_pair(amount: u64) and merge_pair(amount: u64)

Account	S	W	Constraint
owner	✓		
config		✓	
market		✓	
rung			The market's target
collateral_mint			
collateral_vault		✓	
yes_mint, no_mint		✓	
source		✓	The owner's USDC account
yes_token, no_token		✓	The owner's YES and NO accounts
token_program			

Checks: amount > 0; unresolved; now ≤ expiry; target Open; for minting, `OI + amount` ≤ cap. Effects: settles weights; minting takes `amount` USDC and mints `amount` of each side; merging burns `amount` of each and returns `amount` USDC; updates open interest and totals.

resolve()

Account	S	W	Constraint
config		✓	
market		✓	
rung			The market's target
control_rung			Equal to <code>market.control_rung</code> for kind 1

Permissionless. Applies Section 9.6.

redeem(amount: u64)

Account	S	W	Constraint
owner	✓		
config		✓	
market		✓	
collateral_mint			
collateral_vault		✓	
winning_mint		✓	YES if the outcome is YES, NO if NO
winning_token		✓	
destination		✓	The owner's USDC account
token_program			

Checks: amount > 0; resolved; the winning mint matches the outcome. Effects: Section 9.7.

allocate_deferred(), sweep_bounty(rung_id: u8), sweep_treasury(), sweep_fees(rung_id: u8)

Instruction	Accounts	Effect
<code>allocate_deferred</code>	config W, market W	Re-splits the market's deferred bounty share by weight
<code>sweep_bounty</code>	config W, market W, collateral_mint, collateral_vault W, rung, bounty_vault W (the rung's), receipt_mint (the rung's), token_program	Pays the rung's pending share into its bounty vault, or flags it no-exit and defers the share
<code>sweep_treasury</code>	config W, market W, collateral_mint, collateral_vault W, buyback_signer ([<code>"buyback"</code>]), buyback_vault W (its associated USDC account), token_program	Pays the pending protocol share into the buyback vault
<code>sweep_fees</code>	The union of the two above	Both

All four are permissionless.

configure_buyback(canary_mint: Pubkey, buyback_authority: Pubkey)

Account	S	W	Constraint
administrator	✓		Equal to <code>config.administrator</code>
config		✓	
canary_mint			An initialized classic SPL mint, equal to <code>canary_mint</code>

Checks: no buyback open; `canary_mint` is neither the default key nor the collateral mint; `buyback_authority` is not the default key; no CANARY mint is set yet, or the same one is. Effects: sets `canary_mint`, which can never change again, and `buyback_authority`, which the administrator can replace.

`buyback_begin(amount: u64, min_out: u64)` and `buyback_finish()`

Both take the same accounts, and they must appear in one transaction with the same accounts, begin first.

Account	S	W	Constraint
authority	✓		Equal to <code>config.buyback_authority</code>
config		✓	Version 3, with the CANARY mint set
collateral_mint			
canary_mint		✓	Equal to <code>config.canary_mint</code>
buyback_signer			<code>["buyback"]</code>
buyback_vault		✓	The signer's associated USDC account
authority_usdc		✓	The authority's associated USDC account
canary_vault		✓	The signer's associated CANARY account
instructions			The instructions sysvar
token_program			

`buyback_begin` checks: `amount > 0`; `min_out > 0`; no buyback open; `amount ≤` the vault's balance; it is called directly by the transaction, not by another program; a later instruction of the transaction is a `buyback_finish` with the same accounts. Effects: records `amount`, `min_out`, the authority's USDC balance, the CANARY account's balance and both instruction positions, then transfers `amount` from the vault to `authority_usdc`.

`buyback_finish` checks: a buyback is open; this is the recorded finish, after a begin with the recorded arguments and the same accounts; `authority_usdc` holds no more than its recorded balance; the CANARY account gained at least `min_out`. Effects: burns the CANARY account's whole balance with `burn_checked`, adds `amount` to `total_buyback_usdc` and the burn to `total_canary_burned`, and clears the open buyback. Any failure reverts the whole transaction, including the swap between the two.

B.3 `canary_vault`

`init_vault(index, trigger_rung_id, delay_seconds, instant_limit, asset_policy, instant_mint, instant_token_program, pq_key_hash, instance_salt)`

Account	S	W	Constraint
owner	✓	✓	
vault		✓	Created at <code>["qv", owner, index]</code>
trigger_rung			The rung's PDA under the rungs program
sol_vault		✓	Created at <code>["sol", vault], 0 bytes</code>
system_program			
instant_mint_account			The instant asset's mint, or the System program for SOL

Checks: trigger authenticated; policy 1; the asset's mint, owner, initialization and extensions; salt, delay and key hash non-zero. Effects: creates the vault, in Lockdown if the trigger already fell.

Key instructions

Instruction	Accounts	Checks and effects
<code>init_pq_key(epoch, key_hash)</code>	payer S W, vault, pq_key W (created), system_program	Epoch 0 with the committed hash for the first key, else key_epoch + 1
<code>write_pq_key(offset, bytes)</code>	payer S (the key's payer), pq_key W	In order; 1–450 bytes; at most 897 in all; the final chunk must hash to key_hash
<code>confirm_pq_key()</code>	vault W, pq_key W, authorization W	Permissionless. A confirm action for this key, verified with this key; marks it confirmed and, for the first key, makes it active and increments the nonce
<code>rotate_pq_key()</code>	vault W, pq_key (active), authorization W, new_pq_key	Permissionless. Candidate confirmed with epoch key_epoch + 1; a rotate action verified with the active key; switches the key and increments the nonce
<code>reset_initial_pq_key_hash(key_hash)</code>	owner S, vault W, trigger_rung	First key unconfirmed, vault not locked, trigger Open; replaces the commitment
<code>close_unconfirmed_pq_key()</code>	payer S W, pq_key W	The key is unconfirmed; closes it to its payer

Staging instructions

Instruction	Accounts	Checks and effects
<code>init_authorization(auth_id, action, signature_hash)</code>	payer S W, vault, authorization W (created), system_program	Operation ≤ 3 ; action nonce equals the vault's; non-zero hash
<code>write_authorization(offset, bytes)</code>	payer S, authorization W	Not consumed; in order; at most 450 bytes; at most 666 in all; the final chunk must match the hash
<code>close_authorization()</code>	payer S W, authorization W	Closes it to its payer

Spending instructions

Instruction	Accounts	Checks and effects
<code>spend_ed25519(amount, recipient, expected_fee)</code>	owner S, vault W, trigger_rung, sol_vault W, asset_mint, source W, destination W, token_program	Ed25519 gate; $0 < \text{amount} \leq \text{limit}$; the instant asset; the budget; transfers and updates the budget
<code>queue_ed25519(queue_nonce, amount, recipient, destination, mint, token_program, expected_fee)</code>	owner S W, vault W, trigger_rung, queue W (created), system_program	Ed25519 gate; the instant asset; amount $> \text{limit}$; queue_nonce = ed_nonce; unlock at now + delay

Instruction	Accounts	Checks and effects
<code>execute_queued()</code>	vault, trigger_rung, queue W, sol_vault W, asset_mint, source W, destination W, token_program	Permissionless. Ed25519 gate rechecked; Queued; now $\geq$ unlock; accounts match the queue; transfers and marks Executed
<code>spend_pq()</code>	vault W, pq_key, authorization W, sol_vault W, asset_mint, source W, destination W, token_program	Permissionless. A spend action verified with the active key; accounts match the action; transfers, consumes and increments the nonce
<code>cancel_queued_pq()</code>	vault W, pq_key, authorization W, queue W	Permissionless. Queue active; a cancel action for this queue verified with the active key; marks Cancelled

Lock instructions

Instruction	Accounts	Checks and effects
<code>lockdown()</code>	owner S, vault W	Locks with reason 1; a locked vault keeps its first reason
<code>trip()</code>	vault W, trigger_rung	Permissionless. Trigger authenticated and Solved; locks with reason 2

Appendix C. Error codes

Each program numbers its errors from 6000, Anchor's first custom error code. A failed instruction returns the code in the transaction's error, and the web app and the clients show the name.

C.1 canary_rungs

Code	Name	Raised when
6000	ParameterUnavailable	No parameters are compiled for the rung id
6001	AlreadySolved	The rung is Solved
6002	InvalidCollateral	The mint is not the configured USDC, or its decimals are not 6
6003	AlreadyInitialized	The rung or its receipt mint already exists
6004	ZeroAmount	An amount is zero
6005	SunsetReached	A deposit is attempted at or after the sunset
6006	ArithmeticOverflow	A checked operation overflows
6007	WrongClaimMethod	The claim instruction does not match the rung's family
6008	NonCanonicalScalar	s is zero or not below n
6009	InvalidPoint	R is not a valid point, or a coordinate is not below p
6010	JobNotVerifying	A step is sent to a job that is not Verifying
6011	InvalidStepSize	max_bits is outside 1 to 16
6012	InvalidJob	The job does not belong to this rung or is malformed
6013	JobNotVerified	Finalize is sent to a job that is not Verified
6014	InvalidProof	A terminal proof fails verification
6015	DegenerateProof	A scalar multiplication would use a zero scalar
6016	InvalidRecipient	The payout account is not owned by the recipient
6017	BeforeSunset	A reclaim is attempted before the sunset
6018	InsufficientReceipt	The reclaim exceeds the holder's receipts, or no receipts exist
6019	UnauthorizedInitializer	The signer is not the upgrade authority
6020	ClaimExpired	The claim window has passed
6021	JobStillActive	A job is closed while still within its window

C.2 canary_markets

Code	Name	Raised when
6000	InvalidCollateral	The mint or its decimals do not match
6001	UnauthorizedInitializer	The signer is not the upgrade authority or the administrator
6002	CapUnset	The rung has not been enabled with a cap
6003	AlreadyConfigured	The rung is already enabled
6004	TooManyRungs	All sixteen weight slots are in use

Code	Name	Raised when
6005	<code>InvalidRung</code>	The rung account is not the authenticated <code>RungState</code>
6006	<code>ExpiryPassed</code>	The expiry is in the past, or minting or merging after it
6007	<code>RungAlreadySolved</code>	The target rung is Solved
6008	<code>InvalidKind</code>	The market kind is not 0 or 1
6009	<code>InvalidQuantumRung</code>	A quantum-signal market targets a rung other than Q-160
6010	<code>ControlAlreadySolved</code>	The control rung is Solved at creation
6011	<code>ZeroAmount</code>	An amount is zero
6012	<code>AlreadyResolved</code>	Minting or merging on a resolved market
6013	<code>OpenInterestCap</code>	Minting would exceed the rung's cap
6014	<code>NotResolved</code>	Redeeming before resolution
6015	<code>WrongWinningMint</code>	The redeemed token is not the winning side
6016	<code>ArithmeticOverflow</code>	A checked operation overflows
6017	<code>NoOpenInterest</code>	Reserved
6018	<code>UnsupportedExpiry</code>	The expiry is beyond 9999-12-31T23:59:59Z
6019	<code>InvalidClaimWindow</code>	A Solved rung's finalize slot is outside its claim window
6020	<code>InvalidWeightClock</code>	The slot is earlier than a rung's last weight update
6021	<code>InvalidCanaryMint</code>	The CANARY mint is the default key or USDC, or differs from the configured one
6022	<code>CanaryMintImmutable</code>	A different CANARY mint is set after one was configured
6023	<code>InvalidBuybackAuthority</code>	The buyback key is the default key
6024	<code>CanaryMintUnset</code>	A buyback before the CANARY mint is set
6025	<code>UnauthorizedBuyback</code>	The signer is not the configured buyback key
6026	<code>BuybackInProgress</code>	A buyback is already open
6027	<code>NoBuybackInProgress</code>	<code>buyback_finish</code> without an open buyback
6028	<code>InsufficientBuybackBalance</code>	The amount exceeds the buyback vault's USDC
6029	<code>InvalidMinimumOutput</code>	The minimum CANARY output is zero
6030	<code>MissingBuybackFinish</code>	No matching <code>buyback_finish</code> follows in the transaction
6031	<code>InvalidBuybackPair</code>	Begin and finish differ in arguments, accounts or position
6032	<code>BuybackCpiForbidden</code>	Begin or finish is called by another program
6033	<code>BuybackUsdcNotSpent</code>	The buyback key kept some of the advanced USDC
6034	<code>InsufficientCanaryOutput</code>	Less than the minimum CANARY arrived
6035	<code>InvalidConfigVersion</code>	The configuration is not version 3

C.3 `canary_vault`

Code	Name	Raised when
6000	<code>AssetPolicyUnset</code>	The asset policy is not 1
6001	<code>InvalidDelay</code>	The delay is zero
6002	<code>InvalidTrigger</code>	The trigger account is not the authenticated <code>RungState</code> , or its window check fails

Code	Name	Raised when
6003	<code>TriggerStillOpen</code>	<code>trip</code> is called while the trigger is Open
6004	<code>Ed25519Disabled</code>	An Ed25519 instruction on a locked vault or after the trigger fell
6005	<code>InvalidKey</code>	A key account or hash does not match
6006	<code>KeyNotConfirmed</code>	An action needs a confirmed key
6007	<code>KeyAlreadyConfirmed</code>	A confirmed key is confirmed or reset again
6008	<code>InvalidKeyEpoch</code>	The epoch does not follow the active key's
6009	<code>InvalidChunk</code>	A chunk is out of order, empty or too long
6010	<code>ImmutableProof</code>	Writing to a complete key or a consumed authorization
6011	<code>InvalidAction</code>	The action's fields do not match the instruction and its accounts
6012	<code>InvalidNonce</code>	The action's nonce is not the vault's current nonce
6013	<code>InvalidAuthorization</code>	The staged signature is incomplete, consumed or does not match its hash
6014	<code>InvalidFalconProof</code>	Falcon-512 verification fails
6015	<code>AboveInstantLimit</code>	An instant spend exceeds the instant limit
6016	<code>UseInstantPath</code>	A queued spend is not above the instant limit
6017	<code>DelayNotElapsed</code>	A queued spend is executed before it unlocks
6018	<code>QueueNotActive</code>	The queued spend is executed or cancelled already
6019	<code>InvalidAsset</code>	The asset is not supported, or not the instant asset on the Ed25519 path
6020	<code>InvalidDestination</code>	The destination's owner or mint does not match
6021	<code>UnsupportedTokenExtension</code>	The mint has an extension outside the allowlist
6022	<code>ZeroAmount</code>	An amount is zero
6023	<code>InsufficientBalance</code>	The custody account holds too little
6024	<code>ArithmeticOverflow</code>	A checked operation overflows
6025	<code>InstantBudgetExceeded</code>	The instant spend exceeds what is left of the budget
6026	<code>InvalidInstanceSalt</code>	The salt is zero
6027	<code>InvalidTransferFee</code>	The expected fee does not equal the mint's fee for the current epoch

Appendix D. Rung parameters

The constants below are the curve parameters each rung verifies against, in hexadecimal. They are the standard values of SEC 2 [28][36] and RFC 8032 [32]. The points G and Q are not standard: each rung's pair is derived from the published seed by the procedure of Section 4.3 and recorded, with the counters used, in the rung's transcript. The transcript hashes are listed by `/v1/programs`, stored in each `RungState`, and served with their transcripts at `/v1/transcripts/{hash}`.

D.1 Summary

Rung	Id	Curve	Field bits	Subgroup bits	Cofactor	Scalar bits	Claim transactions
Q-160	0	secp160k1	160	161	1	161	13
Q-192	1	secp192k1	192	192	1	192	14
Q-224	2	secp224k1	224	225	1	225	17
C-160	3	joins when certified	256	~160			
T-256k	4	secp256k1	256	256	1	256	1
T-255e	5	Ed25519	255	253	8	253	1

D.2 Q-160 (secp160k1)

The curve $y^2 = x^3 + 7$ over the prime field of order p , with $a = 0$ and a group of prime order n .

```
p = ffffffff ffffffff ffffffff ffffffff ffffac73
n = 1 00000000 00000000 0001b8fa 16dfab9a ca16b6b3
a = 0
b = 7
h = 1
```

D.3 Q-192 (secp192k1)

The curve $y^2 = x^3 + 3$ over the prime field of order p , with $a = 0$ and a group of prime order n .

```
p = ffffffff ffffffff ffffffff ffffffff ffffefe ffffee37
n = ffffffff ffffffff ffffffff 26f2fc17 0f69466a 74defd8d
a = 0
b = 3
h = 1
```

D.4 Q-224 (secp224k1)

The curve $y^2 = x^3 + 5$ over the prime field of order p , with $a = 0$ and a group of prime order n .

```
p = ffffffff ffffffff ffffffff ffffffff ffffffff ffffefe ffffe56d
n = 1 00000000 00000000 00000000 0001dce8 d2ec6184 caf0a971 769fb1f7
a = 0
b = 5
h = 1
```

D.5 T-256k (secp256k1)

The curve $y^2 = x^3 + 7$ over the prime field of order p , with $a = 0$ and a group of prime order n .

```
p = ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff2f
n = ffffffff ffffffff ffffffff ffffffff baaedce6 af48a03b bfd25e8c d0364141
a = 0
b = 7
h = 1
```

D.6 T-255e (Ed25519)

The curve $-x^2 + y^2 = 1 + d \cdot x^2 \cdot y^2$ over the prime field of order $p = 2^{255} - 19$, with $a = -1$ and a prime-order subgroup of order n .

```
p = 7fffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffffed
n = 100000000 000000000 000000000 000000000 14def9de a2f79cd6 5812631a 5cf5d3ed
d = 52036cee 2b6ffe73 8cc74079 7779e898 00700a4d 4141d8ab 75eb4dca 135978a3
a = -1
h = 8
```

D.7 Classical and quantum costs

Rung	Pollard rho steps, $\log_2$	Logical qubits [9]
Q-160	79.0	1,466
Q-192	95.0	1,754
Q-224	111.0	2,042
T-256k	127.0	2,330
T-255e	125.8	~2,330

The step counts use $\sqrt[3]{(\pi n/12)}$ for the $j = 0$ curves and $\sqrt[3]{(\pi n/4)}$ for Ed25519 (Section 2.2); the qubit counts use $9n + 2\lceil \log_2 n \rceil + 10$ for an n -bit prime field (Section 2.4).

Appendix E. Status feed and app configuration

Both documents are served by the API as JSON. Integers that can exceed 2^{53} , such as slots and atomic amounts, are encoded as decimal strings; times are ISO 8601 in UTC.

E.1 canary.status.v1

GET /v1/status-feed returns the feed unwrapped, without the usual {data, meta} envelope.

Field	Type	Meaning
schemaVersion	"canary.status.v1"	
contractVersion	"1.0.0"	The API contract version
mode	"live"	
network	"devnet" or "mainnet-beta"	
generatedAt	time	The indexed snapshot's time
freshness	Freshness	Section 13.4
programId	ConfigurationValue<address>	The rungs program
rungs	array	One entry per rung

Each entry of rungs:

Field	Type	Meaning
id	string	Q-160, Q-192, Q-224, C-160, T-256k or T-255e
account	address	The RungState address
state	RungState	Below
transcriptHash	ConfigurationValue<hex>	The parameter transcript's SHA-256

state is one of:

```
{ "status": "Open", "eventTs": null, "eventSlot": null, "recipient": null, "proofHash": null }
```

```
{
  "status": "Solved",
  "eventTs": "2029-03-14T09:26:53.000Z",
  "eventSlot": "412345678",
  "recipient": "<address>",
  "proofHash": "<64 hex characters>",
  "finalizedSlot": "412345690",
  "finalizedAt": "2029-03-14T09:27:01.000Z",
  "transactionSignature": "<signature>"
}
```

The example values are illustrative.

A ConfigurationValue is either {"status": "set", "value": ..., "provenance": ...} or {"status": "unset", "key": ..., "reason": ...}, so a client can tell a value the deployment has not configured yet from a missing one.

E.2 Freshness

Field	Type	Meaning
<code>commitment</code>	<code>"finalized"</code>	
<code>indexedSlot</code>	integer string	The slot the snapshot was read at
<code>indexedAt</code>	time	When the snapshot was written
<code>observedAt</code>	time	When the API served it
<code>chainTime</code>	time	The Clock sysvar at the snapshot
<code>chainHeadSlot</code>	integer string	The latest slot the indexer saw
<code>lagSlots</code>	integer string	<code>chainHeadSlot</code> – <code>indexedSlot</code>
<code>state</code>	<code>"fresh"</code> , <code>"indexer-behind"</code> or <code>"rpc-unreachable"</code>	
<code>rpcReachable</code>	boolean	
<code>lastSuccessfulRpcAt</code>	time	

E.3 `canary.app-config.v1`

GET `/v1/app-config` returns `{data, meta}` with `data`:

Field	Type	Meaning
<code>schemaVersion</code>	<code>"canary.app-config.v1"</code>	
<code>mode</code>	<code>"live"</code>	
<code>appEnv</code>	<code>"production"</code>	
<code>state</code>	<code>"waiting"</code> or <code>"ready"</code>	Section 13.3
<code>waitingReasons</code>	string[]	Empty when ready
<code>cluster</code>	<code>"devnet"</code> or <code>"mainnet-beta"</code>	
<code>rpcUrl</code>	URL	The API's relay, <code>{apiUrl}/v1/rpc</code>
<code>wsUrl</code>	null	Clients poll signature status through the relay
<code>genesisHash</code>	string	The cluster's genesis hash
<code>commitment</code>	<code>"finalized"</code>	
<code>apiUrl</code>	URL	The API's public origin
<code>programIds</code>	<code>{rungs, markets, vault}</code>	Addresses
<code>collateral</code>	<code>{mint, decimals: 6, symbol: "USDC"}</code>	
<code>protocolPolicy</code>	object	Below
<code>rungs</code>	array	Below
<code>faucet</code>	<code>{enabled: false}</code>	

Each entry of `rungs`:

Field	Type	Meaning
<code>id</code> , <code>rungId</code>	string, integer	<code>Q-160</code> and 0, and so on
<code>claimable</code>	boolean	True when the deployment is ready and the rung is Open

Field	Type	Meaning
knownDiscreteLog	false	Always false in production
stateAddress	address	The RungState address
transcriptHash	hex	
parameters	object	id, label, model, pHex, nHex, aHex, bHex, gHex, qHex, transcriptSha256, parameterRole, scalarBits: what a client needs to build a claim

protocolPolicy:

Field	Value
settlementFeeBps	100
treasuryShareBps	7000
bountyShareBps	3000
sunsetTs	"2208988800" (2040-01-01T00:00:00Z)
maxClaimSlots	300
capsAtomic	Per enabled rung, keyed by rung id as a string ("0" is Q-160), the open-interest cap in atomic USDC

E.4 canary.deployment.v1

The deployment record (Section 16.5) is published with the deployment and read by the API:

```
{
  "schemaVersion": "canary.deployment.v1",
  "cluster": "mainnet-beta",
  "genesisHash": "<genesis hash>",
  "programIds": { "rungs": "<address>", "markets": "<address>", "vault": "<address>" },
  "collateral": { "mint": "<USDC mint>", "decimals": 6, "symbol": "USDC" },
  "parameterManifestPath": "<path to canonical.json>",
  "programs": {
    "rungs": {
      "programId": "<address>",
      "programDataAddress": "<address>",
      "elfSha256": "<64 hex characters>",
      "elfLength": 0,
      "upgradeAuthority": "<address or null>",
      "verifiedSlot": 0
    }
  },
  "rungs": [{ "rungId": 0, "id": "Q-160", "stateAddress": "<address>", "transcriptHash": "<64 hex characters>" }],
  "policy": {
    "settlementFeeBps": 100, "treasuryShareBps": 7000, "bountyShareBps": 3000,
    "sunsetTs": "2208988800", "maxClaimSlots": 300, "capsAtomic": { "0": "100000000000000" }
  },
  "buyback": {
    "authority": "<buyback signer address>", "vault": "<its USDC account>",
    "canaryMint": "<address or null>", "buybackAuthority": "<address or null>",
    "totalBuybackUsdcAtomic": "0", "totalCanaryBurnedAtomic": "0"
  }
}
```

The record never contains an RPC URL or a key; the API rejects a record that does. `/v1/programs` serves the same `buyback` object with the totals read from `MarketsConfig` at the indexed slot. `treasuryShareBps` keeps its name and is the protocol share that goes to the buyback vault.

E.5 Eligibility

GET /v1/eligibility returns:

Field	Type	Meaning
markets.allowed	boolean	Whether the interface offers market actions
markets.reason	"eligible-region", "us-restricted" or "location-unknown"	
markets.countryCode	string or null	Two letters
checkedAt	time	
source	"trusted-header", "self-attestation" or "unconfigured"	
policy	string	US users excluded from markets in v0

With self-attestation, the client passes countryCode and attestNonUs=true as query parameters.

Appendix F. Glossary

Term	Meaning
Atomic unit	The smallest unit of a token; USDC, receipts and YES and NO tokens have 6 decimals, so 1 USDC is 1,000,000 atomic units
Bounty vault	A rung's USDC token account, paid in full to the solver when the rung falls
Buyback vault	The markets program's USDC account under <code>["buyback"]</code> ; it receives the protocol's 70 % of fees, and its USDC leaves only in a buyback that burns the CANARY bought
BPSW	Baillie-PSW, a probable-prime test with no known counterexample
C-160	The classical control rung: as costly classically as Q-160, as costly for a quantum computer as a 256-bit curve
CANARY	Canary's token: 1,000,000,000 fixed supply, launched in full to the public; the protocol's 70 % of settlement fees buys and burns it
Claim window	300 slots from a claim's begin slot, within which a staged claim must finish
Cofactor	The ratio h of a curve's group order to its prime subgroup order n
Commitment	How settled a Solana read is; <code>finalized</code> reads cannot be rolled back
Compute unit (CU)	Solana's measure of execution cost; a transaction may use at most 1.4M
CRQC	Cryptographically relevant quantum computer: one that can solve discrete logarithms at the sizes used in practice
Discriminator	The first 8 bytes of an Anchor account or instruction, identifying its type
ECDLP	The elliptic-curve discrete-logarithm problem: given G and $Q = k \cdot G$, find k
<code>event_ts</code> , <code>event_slot</code>	The time and slot of a rung's fall: when the winning claim began, or the terminal claim executed
Expiry	A market's settlement time T ; a fall with <code>event_ts</code> $\leq T$ resolves YES
Falcon-512	A lattice-based signature scheme selected by NIST; Q-Vault's post-quantum key
Jacobian coordinates	A projective representation $(X : Y : Z)$ of the point $(X/Z^2, Y/Z^3)$, which avoids field inversions
Lockdown	A Q-Vault mode in which only the Falcon key can move funds; permanent
Logical qubit	An error-corrected qubit, the unit in which quantum resource estimates are stated
Montgomery form	A representation of field elements that makes modular multiplication cheap
No-exit	A Solved rung with no receipts outstanding, which receives no further fee share
NonTransferable	A Token-2022 extension that makes a token impossible to transfer; used for sponsor receipts
NUMS	Nothing up my sleeve: values derived from a public seed by a fixed procedure, so no party could have chosen them with hidden knowledge
Open interest	Outstanding YES-and-NO pairs in unresolved markets, counted per target rung
PDA	Program-derived address: an address with no private key, controlled only by the program that derives it
Pollard's rho	The best known classical algorithm for a generic prime-order group, taking about $\sqrt{(\pi n/4)}$ steps with the negation map
Q-date	The date the market implies for a rung's fall, read from YES prices across expiries
Q-Vault	A vault that accepts Ed25519 spends while its trigger rung is Open and only Falcon-512 after
Quantum-signal market	A market that resolves YES if Q-160 falls while C-160 has not
Receipt	A sponsor's NonTransferable token, minted one per USDC unit deposited, reclaimable after the sunset

Term	Meaning
Recipient	The owner bound into a claim's challenge; the bounty goes only to an account it owns
Rung	One discrete-logarithm instance with its bounty: Q-160, Q-192, Q-224, C-160, T-256k or T-255e
RungState	The on-chain account that says whether a rung is Open or Solved, and when it fell
Schnorr proof	A proof of knowledge of a discrete logarithm: (R, s) with $s \cdot G = R + e \cdot Q$
Settlement fee	1 % of winning redemptions, split 70 % to the buyback vault and 30 % to bounty vaults
Shor's algorithm	The quantum algorithm that solves discrete logarithms in polynomial time
Slot	Solana's unit of time for block production, nominally 400 ms
Staged claim	A claim on a Q-rung, verified over several transactions by <code>begin_claim</code> , <code>step_claim</code> and <code>finalize_claim</code>
Sunset	2040-01-01T00:00:00Z, after which deposits stop and receipts can reclaim
Terminal rung	T-256k or T-255e: the curves of Bitcoin and Ethereum keys and of Solana addresses
Time-weighted open interest	Open interest multiplied by the slots it was held, accumulated per rung; the basis of the bounty split
Transcript	The canonical JSON record of a rung's parameters and the checks they passed; its SHA-256 is stored on chain
Trip	The permissionless instruction that locks a Q-Vault after its trigger falls
Upgrade authority	The key that can replace a Solana program's code; revoked for the rungs program after audit
YES, NO	A market's two outcome tokens; a pair is minted for one USDC and merges back into one USDC