

face/id

Drain-resistant tokens on Solana
from on-chain passkey verification

The face/id authors

Version 2, October 2026

Abstract

A wallet drain succeeds when the attacker obtains one valid signature from the holder's wallet key. Biometric locks in wallet software do not change this, since the chain never observes them. We describe face/id, a token launchpad on Solana in which every launched token is held in a per-holder vault owned by the launchpad program. Spending a vault's SOL, moving tokens or SOL out of it, and changing who may do so all require a WebAuthn assertion from the holder's passkey, verified on-chain through the secp256r1 precompile and bound to one action and one vault state. Selling requires only the wallet key and credits the proceeds to the same vault. Value leaves only toward destinations registered at least Δ earlier, and SOL spent from the vault on purchases is capped per window of length Δ .

We give a game-based reduction showing that any passkey-gated action the holder did not approve yields an existential forgery of P-256 ECDSA or a collision in SHA-256. An adversary holding the wallet key can only sell the holder's tokens into the holder's own vault, at no less than $(1 - \varphi)(1 - \gamma)$ of the slot's opening price, or request a key rotation that the holder cancels with one assertion within Δ ; an unattended vault cannot lose value to such an adversary in less than 2Δ . Against an adversary that also chooses what the holder's passkey signs, the only loss not announced Δ in advance is purchases of at most $2L$ per interval of length Δ , where L is the holder's limit. For the market, the bonding curve is solvent under full redemption, and the fee-funded burn of the platform token sets a price floor $P_0/(1 - b)^2$ that does not decrease, where b is the burned fraction of supply.

1 Introduction

A Solana token account follows one rule: a valid signature for its owner moves its balance. The drains reported against wallet users reduce to obtaining such a signature. A phishing page requests a transaction whose effect differs from what it displays; a malicious approval grants a delegate; a leaked seed phrase discloses the key; malware on the signing machine can do any of the three. Hardware wallets address key extraction and leave the first two cases in place, because the holder still signs a transaction the attacker built.

Wallet software increasingly gates the key behind Face ID or a fingerprint. The gate is local to the device. The chain receives an Ed25519 signature and has no means to distinguish a signature the holder intended from one obtained by deception or produced from an exported seed.

face/id moves the requirement into the asset. Each token launched on face/id is held in a vault owned by the launchpad program, one vault per holder. The vault records a passkey, a P-256 key pair generated by the holder's device whose private key is usable only through the device's authenticator after local user verification. The launchpad releases value from a vault only against a WebAuthn assertion from that passkey over the exact action, verified on-chain through Solana's secp256r1 precompile [2]. The wallet key still pays fees and trades, and no launchpad instruction accepts it as authority to move value out of a vault.

Two further rules make the scheme usable as a trading venue and bound what a compromised signing context can do. Selling needs no passkey, and the proceeds are credited to the seller’s vault. Value leaves a vault only toward destinations that have been on its list for at least Δ (24 hours), and purchases funded from the vault are capped at a holder-chosen L per window of length Δ .

Contributions.

1. *Asset-level custody* (Section 4). Every balance of every face/id token is held by an address of the launchpad program. This holds by program-derived ownership and, independently, by a Token-2022 transfer hook that rejects any transfer not issued by a top-level launchpad instruction (Lemma 2, Appendix B).
2. *An on-chain WebAuthn verifier* that binds one assertion to one action, one vault state and one origin by exact byte comparison (Section 4.4, Appendix A).
3. *Security results* (Section 5): a reduction to P-256 unforgeability and SHA-256 collision resistance (Theorem 1); an exact list of what the wallet key alone can do (Theorem 2); a 2Δ lower bound on the time to drain an unattended vault (Corollary 1); and a loss bound against a compromised signing context (Theorem 4).
4. *Market results* (Section 6): solvency under full redemption, a per-slot lower bound on sale proceeds with exact integer rounding, and a burn-driven price floor that does not decrease.
5. *An implementation and measurements* (Section 7).

2 Background

Accounts and program-derived addresses. Every Solana account names an owner program, and only the owner may modify the account’s data or debit its lamports. A program-derived address (PDA) is derived from seeds and a program id and lies off the Ed25519 curve, so no private key exists for it; the deriving program signs for it through `invoke_signed`. Cross-program invocation forbids reentrancy except direct self-recursion: a call chain $A \rightarrow B \rightarrow A$ is refused.

Token-2022. Debiting a token account requires the signature of its owner or of a delegate the owner approved. The *TransferHook* mint extension names a program that Token-2022 invokes on every checked transfer of the mint with the source, mint, destination and authority, followed by extra accounts listed in a validation account [3]; the hook can fail the transfer. During the call the source and destination carry a *transferring* flag. *ImmutableOwner* forbids reassigning an account’s owner and is present on every associated token account created under Token-2022. Mint extensions are fixed when the mint is initialized.

WebAuthn assertions. An assertion [1] consists of authenticator data AD (SHA-256 of the relying party id, a flags byte, a signature counter, optional extensions), client data CD (a JSON string written by the browser or operating system, containing the ceremony type, the challenge and the origin of the requesting page), and an ECDSA signature over $AD \parallel \text{SHA256}(CD)$. The client writes the origin itself. An authenticator uses a credential only for the relying party it was created for, and sets the user-verified flag (UV) only after verifying the user locally. WebAuthn Level 3 fixes the order of the leading members of CD and specifies a limited verification procedure that compares a byte prefix in place of a JSON parser.

The secp256r1 precompile. Since SIMD-0075 [2] a transaction may contain instructions to `Secp256r1SigVerify1111...`. The instruction data lists signature records, each giving offsets to a signature, a compressed public key and a message, and for each of the three the index of the instruction that holds the bytes. The runtime verifies every record, rejects signatures with s above half the group order, and rejects the transaction if any record fails. A program confirms that verification took place by reading the instruction through the instructions sysvar.

3 Model

A holder H has a wallet key sk_w (Ed25519) and a passkey K (P-256) created for the launchpad's relying party id RP ; the launchpad's web origin is O .

Adversaries.

- $\mathcal{A}_{\text{key}}$ knows sk_w and can obtain H 's signature under sk_w on any transaction. It may run pages on any origin other than O , or embed O in a frame of its own page, and obtain every assertion H 's client produces there. It orders transactions within a slot as the slot leader would and reads all chain state.
- $\mathcal{A}_{\text{ctx}}$ is $\mathcal{A}_{\text{key}}$ that in addition chooses the content of the assertions H performs on O , for example by replacing the code served at O . Each assertion still requires H to perform user verification, but H cannot inspect what it authorizes.

Assumptions.

- (A1) ECDSA over P-256 is existentially unforgeable under chosen-message attack (EUF-CMA) [4].
- (A2) SHA-256 is collision resistant.
- (A3) *Platform integrity.* The client reports the origin and cross-origin status of the requesting page truthfully; the authenticator uses K only for RP and sets UV only after local verification; the private key of K cannot be used outside the authenticator.
- (A4) *Honest registration.* sk_w is uncompromised when the vault is created.
- (A5) *Liveness.* A transaction submitted by H is included within a time much shorter than Δ .
- (A6) *Code identity.* The deployed programs are those specified here.

Goals. (G1) A passkey-gated action executes only if H approved that exact action, and at most once. (G2) With sk_w alone, no value leaves a vault. (G3) Value reaches only addresses announced on-chain at least Δ earlier. (G4) Losses that are not announced in advance are bounded by a quantity the holder sets. (G5) Holders can always sell, at a price bounded below relative to the market.

4 Protocol

4.1 State

A *vault* is the PDA $\text{PDA}(\text{"vault"}, w)$ for wallet w . It stores the passkey K , a nonce n , a list D of four destination slots (d, t_d) where t_d is the activation time, a purchase limit L with the window state (s, t_w) and an optional pending increase (L', t_L) , and an optional pending

Instruction	Authorized by	Effect
<code>register(K, d_0, L)</code>	wallet and passkey	create vault; d_0 (optional) active at once
<code>buy($m, v, \underline{a}$)</code>	wallet	SOL from the wallet; tokens into the vault
<code>sell($m, a, \underline{v}$)</code>	wallet	tokens from the vault; SOL into the vault's cash
<code>buy_cash($m, v, \underline{a}$)</code>	passkey, within L	the vault's cash buys tokens into the vault
<code>withdraw(d, v)</code>	passkey, d active	cash to address d
<code>send(m, d, a)</code>	passkey, d active	tokens to the vault of d
<code>add_destination(d)</code>	passkey	d active from now + Δ
<code>remove_destination(d)</code>	passkey	d removed
<code>set_limit(L')</code>	passkey	lower at once, or raise from now + Δ
<code>request_rotation(K')</code>	wallet	K' pending until now + Δ
<code>cancel_rotation</code>	passkey	pending rotation cleared
<code>confirm_rotation</code>	anyone, after Δ	$K \leftarrow K', D \leftarrow \emptyset$
<code>launch(meta)</code>	wallet with a vault	new mint and pool
<code>claim(m)</code>	anyone	creator fees into the creator's vault
<code>burn</code>	anyone	treasury buys the platform token and burns it

Table 1: Launchpad instructions. $\underline{a}$ and $\underline{v}$ are slippage bounds. “Passkey” means the instruction passes (V1) to (V5) of Section 4.4 for its intent (Definition 1).

rotation (K', t_K). The vault's *cash* is its lamport balance above the rent-exempt minimum. For each mint the vault owns one associated token account, created on first use.

A *pool* is the PDA `PDA("pool", m)` for mint m . It owns the token reserve, whose balance we write y , and records the real SOL R held for traders, the creator's accrued fees F , and the reserves (x_o, y_o) observed before the first trade of the current slot. The pool prices against $x = V + R$, where V is a constant virtual reserve. Appendix A gives the byte layouts.

4.2 Instructions

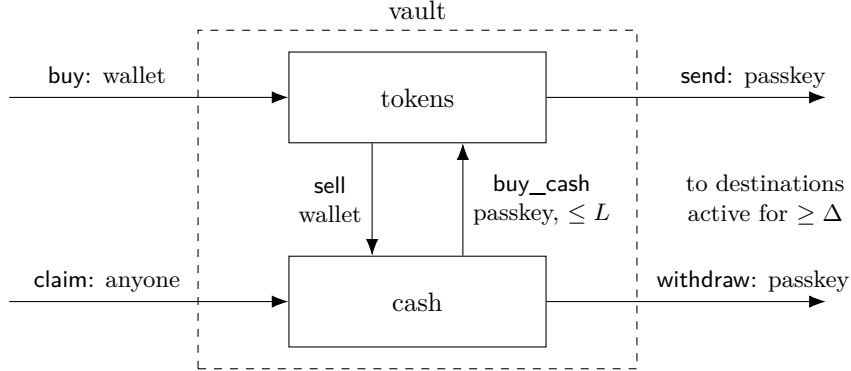


Figure 1: Value flows of one vault and the authority each needs.

Table 1 lists the instructions. Value may enter a vault, or change form inside it, under the wallet key (Figure 1). Spending the cash, moving anything out, and changing the destination list, the limit or the passkey require the passkey. Changes that widen what the passkey can do take effect only after Δ .

4.3 Intents

Definition 1 (Intent). For an action α with fields f and the vault's nonce n ,

$$I(\alpha, f, n) = \text{SHA256}(\text{enc}(\alpha, f, n)), \quad \text{enc}(\alpha, f, n) = \text{"faceid/v1"} \parallel \text{tag}(\alpha) \parallel id \parallel f \parallel \text{le}_{64}(n),$$

where id is the launchpad's program id and $\text{tag}(\alpha)$ the ASCII name of the action. Fields have fixed widths: 32 bytes for an address, 33 for a passkey, 8 for an amount. Table 5 in Appendix A lists every encoding.

Lemma 1 (Injective encoding). *enc is injective. Consequently two distinct triples (α, f, n) have distinct intents unless SHA-256 collides.*

Proof. The eight tags form a prefix-free set, so $\text{tag}(\alpha)$ is recovered by reading the string after the 9-byte domain until a tag matches. The tag determines the widths of all remaining fields, so f and n are recovered by position. $\square$

Every successful passkey-gated instruction increments n ; no other instruction changes it. An assertion therefore commits to one action, field for field, in one vault state.

4.4 Verifying an assertion

A passkey-gated instruction at index j of a transaction carries (AD, CD) and computes its intent c . It succeeds only if all of the following hold.

- (V1) $j \geq 1$, and instruction $j - 1$ is a secp256r1 precompile instruction with exactly one signature record.
- (V2) The record's three instruction indices equal 0xFFFF: signature, key and message lie in instruction $j - 1$.
- (V3) The record's public key equals the vault's K , and its message equals $AD \parallel \text{SHA256}(CD)$ in length and bytes.
- (V4) CD begins with the bytes

`{"type": "webauthn.get", "challenge": " $\beta(c)$ ", "origin": "O"`

and continues with `}` or `,"crossOrigin": false`, where β is unpadded base64url and $|\beta(c)| = 43$.

- (V5) $|AD| \geq 37$, the first 32 bytes of AD equal $\text{SHA256}(RP)$, and the flags byte has UP (0x01) and UV (0x04) set.
- (V2) prevents a record from verifying bytes in another instruction while the program compares its own copy. (V4) fixes the ceremony type, so a self-attestation produced with K at registration cannot be replayed as an assertion; it fixes the origin exactly, which excludes look-alike domains and subdomains; and it rejects framed pages, whose client data carry `"crossOrigin": true`. A prefix match leaves no room for a second `origin` member later in the string. (V5) requires the UV flag in addition to UP.

4.5 Custody

`launch` creates the mint with exactly two extensions, *TransferHook* pointing at a separate program called the *gate*, and *MetadataPointer*, both initialized without an authority. The freeze authority is empty. The pool PDA mints the supply S into its reserve, initializes the gate's validation account while it is the mint authority, sets the mint authority to none, and sets the metadata update authority to none.

The gate accepts a transfer only if (i) the source carries the *transferring* flag, (ii) the destination has *ImmutableOwner*, and (iii) the stack height is 3 and the top-level instruction at the current index belongs to the launchpad, so that the call chain is launchpad, Token-2022, gate. A transfer issued directly by a user places the gate at height 2; a launchpad call wrapped by another program places it at height 4.

4.6 Market

Each pool is a constant-product market [5] over (x, y) with $x = V + R$; all divisions round down.

- **buy** and **buy_cash** with v lamports take fees $\lfloor \varphi_c v \rfloor$ for the creator and $\lfloor \varphi_b v \rfloor$ for the burn treasury, and deliver $\lfloor yv'/(x + v') \rfloor$ tokens for the remainder v' .
- **sell** of a tokens computes $g = \lfloor xa/(y + a) \rfloor$, takes the same two fees from g , and credits the remainder to the seller's vault. It succeeds only if

$$\frac{x - g}{y + a} \geq (1 - \gamma) \frac{x_o}{y_o}, \quad (1)$$

evaluated as $100(x - g)y_o \geq 95x_o(y + a)$.

- **burn** spends the treasury's balance above rent, T , on $\lfloor yT/(x + T) \rfloor$ platform tokens from their pool and burns them from the reserve.

Table 2 lists the parameters.

Symbol	Meaning	Value
S	supply per token, base units	10^{15} (10^9 tokens, 6 decimals)
V	virtual SOL reserve	$3 \cdot 10^{10}$ lamports (30 SOL)
$P_0 = V/S$	opening price	30 lamports per token
φ_c, φ_b	creator fee, burn fee	0.5% each
φ	total fee	1%
γ	largest price drop within a slot	5%
Δ	delay for destinations, limit raises, rotation; window	86,400 s
L	purchase limit per window	chosen by the holder

Table 2: Protocol parameters.

5 Security

Lemma 2 (Custody closure). *At every reachable state, every token account with a nonzero balance of a face/id mint is the pool's reserve or a vault's associated token account, and every change of such a balance happens inside a top-level launchpad instruction.*

Proof. By induction over transactions. After **launch** the supply sits in the reserve and the mint authority is removed, so balances can afterwards decrease only through the operations of Table 6 (Appendix B). Each requires a signature by the debited account's owner or by a delegate it approved, or an authority held by a mint extension. The reserve and all vault accounts are owned by launchpad PDAs, for which only the launchpad signs; the launchpad never approves a delegate; the mint carries no extension that holds such an authority, and its extension set cannot change. Hence every debit is initiated by the launchpad, whose transfer destinations are reserves and vault accounts derived from fixed seeds, which preserves the hypothesis. Independently, the gate fails every transfer not issued from a top-level launchpad instruction (Section 4.5). $\square$

Lemma 3 (Binding). *If an instruction that satisfies (V1) to (V5) belongs to an executed transaction, that transaction carried a valid P-256 signature with low s by the vault's K on $AD \parallel \text{SHA256}(CD)$.*

Proof. The runtime rejects a transaction in which a precompile record fails. By (V2) the verified bytes are bytes of instruction $j - 1$, and by (V3) they equal K and the expected message. $\square$

Security game. The challenger generates K inside an honest authenticator and gives the adversary $\mathcal{A}$ everything else, including sk_w and control of transaction order. $\mathcal{A}$ has two oracles. $\text{APPROVE}(\alpha, f)$ makes H produce, on origin O , an assertion for $l(\alpha, f, n)$ at the vault's current nonce; the adversary receives it and may submit, withhold or reorder it. $\text{OTHER}(\cdot)$ returns any assertion H 's client produces for a page whose origin is not O or which frames O ; by (A3) its client data name that origin or carry `"crossOrigin":true`. $\mathcal{A}$ *wins* if a passkey-gated instruction executes whose intent differs from every intent produced by APPROVE .

Theorem 1 (Authorization). *For every adversary $\mathcal{A}$ making q oracle queries,*

$$\Pr[\mathcal{A} \text{ wins}] \leq \text{Adv}_{\text{P256}}^{\text{euf-cma}}(\mathcal{B}) + \text{Adv}_{\text{SHA256}}^{\text{cr}}(\mathcal{C}),$$

where $\mathcal{B}$ makes q signing queries and $\mathcal{B}, \mathcal{C}$ run in time close to that of $\mathcal{A}$. Except with the same probability, each approved action executes at most once.

Proof. Let the winning instruction carry (AD^*, CD^*) with intent c^* , and set

$$m^* = AD^* \parallel \text{SHA256}(CD^*).$$

By Lemma 3 the transaction carried a valid signature σ^* on m^* under K . Let $m_1, \dots, m_q$ be the messages the authenticator signed in answer to oracle queries.

Case $m^ \notin \{m_i\}$.* $\mathcal{B}$ receives K and a signing oracle, generates sk_w and all other state, runs $\mathcal{A}$, and answers each oracle query by forming AD and CD as an honest client and authenticator would and querying its own oracle. When $\mathcal{A}$ wins, $\mathcal{B}$ outputs (m^*, σ^*) , a valid signature on an unqueried message.

Case $m^ = m_i$.* The digest occupies the last 32 bytes of the message, so $AD^* = AD_i$ and $\text{SHA256}(CD^*) = \text{SHA256}(CD_i)$. If $CD^* \neq CD_i$, $\mathcal{C}$ outputs the pair. Otherwise $CD_i = CD^*$ satisfies (V4) for c^* . The challenge member is delimited by quotation marks, which `base64url` does not produce, so CD_i carries exactly $\beta(c^*)$, origin O , and no cross-origin flag. By (A3) query i was an APPROVE query with intent c^* , contradicting the winning condition.

For the last claim, an approved action is bound to the nonce at approval, and its execution increments the nonce. Executing it again requires the same message to verify against a different intent, which by Lemma 1 requires a SHA-256 collision. $\square$

The bound concerns the content of approvals. Phishing pages, look-alike domains, framed copies of the site and replays of past assertions fall under OTHER or under the replay clause. An adversary that submits an approved assertion before H does executes exactly the approved action. The wallet key cannot advance the nonce, so it cannot invalidate approvals H has already produced.

Theorem 2 (Wallet key). *Against $\mathcal{A}_{\text{key}}$, except with the probability of Theorem 1 and while the vault's passkey is H 's, every change to a vault that H did not approve is one of: (i) `sell`, which removes tokens and credits the proceeds to the same vault; (ii) a deposit through `buy`, `claim`, or a `send` from another vault; (iii) `request_rotation`, which has no other effect until `confirm_rotation` runs Δ later and which H cancels with one assertion. In particular the vault's cash decreases only by approved actions, and its token balances decrease only by approved actions and sells.*

Proof. In Table 1, the instructions that sk_w authorizes alone are `buy`, `sell`, `launch` and `request_rotation`. Of the instructions that need no signature, `claim` and `confirm_rotation` touch a vault. All other instructions are passkey-gated and fall under Theorem 1. By Lemma 2 no other program moves vault tokens, and the cash is the lamport balance of a launchpad-owned account, which only the launchpad debits. $\square$

Theorem 3 (Announced exits). *Let value leave a vault toward an address d at time t through `withdraw` or `send`. Then either d was named at registration and no rotation has been confirmed since, or `add_destination( $d$ )` executed at some $t_a \leq t - \Delta$ under the passkey current at t_a and no rotation was confirmed in $(t_a, t]$.*

Proof. Both instructions require $d \in D$ with $t_d \leq t$. Entries enter D only at registration, with t_d equal to the registration time, or through `add_destination`, with $t_d = t_a + \Delta$; `confirm_rotation` empties D . $\square$

Theorem 3 holds against every adversary in the model, $\mathcal{A}_{\text{ctx}}$ included, because it does not depend on who approved the addition.

Corollary 1 (Unattended vault). *Against $\mathcal{A}_{\text{key}}$, let t_0 be the time of the adversary's first `request_rotation`. No value reaches an address of the adversary's choice before $t_0 + 2\Delta$, and one `cancel_rotation` by H at any time in $[t_0, t_0 + \Delta)$ returns the vault to its prior state.*

Proof. By Theorem 2, rotation is the adversary's only route to passkey authority; it confirms no earlier than $t_0 + \Delta$ and empties D . By Theorem 3 an exit toward a new address then requires an addition at some $t_a \geq t_0 + \Delta$ and executes no earlier than $t_a + \Delta$. By (A5), $\mathcal{A}_{\text{key}}$ cannot keep H 's cancellation from landing. $\square$

Because confirmation empties D , the bound holds even when a destination H registered earlier is the compromised wallet itself. Rotation is also the recovery path: a holder who loses the device requests a new passkey with sk_w , waits Δ , adds a destination and waits Δ again.

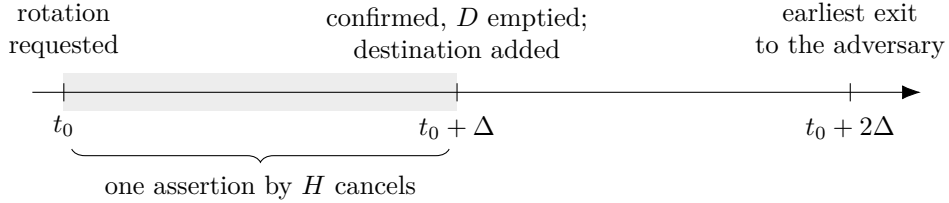


Figure 2: An unattended vault under a stolen wallet key (Corollary 1). Each step is an on-chain event.

Compromised signing context. Against $\mathcal{A}_{\text{ctx}}$ the passkey signs what the adversary chooses whenever H performs user verification, so Theorem 1 no longer constrains the content of approvals. The remaining guarantees come from the delays and the limit.

Lemma 4 (Purchase limit). *Let $\bar{L}$ be the largest limit in force during an interval of length Δ . The cash spent through `buy_cash` within the interval is at most $2\bar{L}$. Every increase of the limit takes effect no earlier than Δ after its `set_limit`.*

Proof. `buy_cash` keeps a window start t_w and a running total s ; at time $t \geq t_w + \Delta$ it sets $t_w \leftarrow t$ and $s \leftarrow 0$, and it requires $s + v \leq L$ before adding v . Windows are disjoint and each spans at least Δ , so an interval of length Δ meets at most two. Within one window the total never exceeds the largest limit in force there: a decrease applies at once and blocks further spending above it, and an increase recorded at t_L applies at the first purchase at or after $t_L + \Delta$. $\square$

Theorem 4 (Compromised context). *Against $\mathcal{A}_{\text{ctx}}$, every decrease of a vault's holdings is one of: (i) a sale, with proceeds credited to the vault as in Theorem 6; (ii) cash spent through `buy_cash`, bounded by Lemma 4; (iii) an exit toward a destination announced at least Δ earlier (Theorem 3). Consequently the loss that is not announced on-chain at least Δ in advance is, over any interval of length Δ , at most $2\bar{L}$ plus the shortfall of sales.*

Proof. Enumerate Table 1. `withdraw` and `send` are covered by Theorem 3; `buy_cash` by Lemma 4; `sell` by Theorem 6. `add_destination`, increases through `set_limit`, and `request_rotation` change no balance and become effective only Δ after their event. `remove_destination`, decreases through `set_limit`, and `cancel_rotation` only reduce what can leave. The remaining instructions only add value to the vault. $\square$

The bound counts the full amount of every purchase. Exits require two assertions separated by at least Δ , the first of which is a public event.

6 Market

Throughout, $k = xy$ and $k_0 = VS$. For a pool, C denotes the tokens held in vaults and B the tokens burned, so $y + C + B = S$; only the platform token has $B > 0$.

Lemma 5. *No instruction decreases k .*

Proof. A purchase of net amount v' gives $y' = y - \lfloor yv'/(x+v') \rfloor \geq yx/(x+v')$ and $x' = x + v'$, so $x'y' \geq xy$. A sale gives $x' = x - \lfloor xa/(y+a) \rfloor \geq xy/(y+a)$ and $y' = y + a$. A burn is a purchase whose output is destroyed. Fees are taken outside the reserves. $\square$

Theorem 5 (Solvency). *At every reachable state and for every $a \leq C$, a sale of a tokens pays out $g \leq R$. Hence the pool pays every holder, in any order, until all tokens are back in its reserve.*

Proof. By Lemma 5 and $y + a \leq y + C = S - B \leq S$,

$$g \leq \frac{xa}{y+a} = x - \frac{xy}{y+a} \leq x - \frac{k_0}{y+a} \leq x - \frac{VS}{S} = R. \quad \square$$

Theorem 6 (Sale proceeds). *Let a sale of a tokens execute in slot σ with opening price $P_o = x_o/y_o$. The amount credited to the vault satisfies*

$$out \geq (1 - \varphi)(1 - \gamma) P_o a - 2.$$

Proof. Let (x, y) be the reserves before the sale, $g^* = xa/(y+a)$ and $g = \lfloor g^* \rfloor$. Since $x - g \leq x - g^* + 1 = xy/(y+a) + 1$, the guard (1) gives

$$(1 - \gamma)P_o \leq \frac{xy}{(y+a)^2} + \frac{1}{y+a}.$$

Multiplying by a and using $a/(y+a) < 1$ yields $(1 - \gamma)P_o a < xy a/(y+a)^2 + 1$. Also $g^* = \frac{xya}{(y+a)^2} \cdot \frac{y+a}{y} \geq \frac{xya}{(y+a)^2}$, so $g \geq g^* - 1 > (1 - \gamma)P_o a - 2$. Both fees round down, so together they are at most φg , and $out \geq (1 - \varphi)g > (1 - \varphi)(1 - \gamma)P_o a - 2$. $\square$

With the parameters of Table 2, $(1 - \varphi)(1 - \gamma) = 0.9405$. The bound holds for every order of transactions in the slot, including an order chosen by a leader who trades before and after the sale.

Corollary 2 (Forced sales). *If $\mathcal{A}_{\text{key}}$ forces a vault to sell a tokens in slot σ , the vault receives at least $0.9405 P_o a - 2$ lamports. Measured at the opening price, the holder's loss is at most 5.95% of the sale's value plus two lamports, whatever trades the adversary places around the sale.*

Proposition 1 (Depth per slot). *After any sale in slot σ , with $k_o = x_o y_o$,*

$$y \leq y_o \sqrt{\frac{k}{(1-\gamma)k_o}}.$$

Since k/k_o exceeds 1 only through rounding, the net inflow of tokens to the reserve within one slot is at most about $(1-\gamma)^{-1/2} - 1 \approx 2.6\%$ of the opening reserve.

Proof. After the sale, $x/y \geq (1-\gamma)x_o/y_o$ by (1), and $x = k/y$, so $k/y^2 \geq (1-\gamma)k_o/y_o^2$. $\square$

A larger sale needs several slots. Each slot opens at the price left by the previous one, so holding the price down across a slot boundary leaves the manipulator’s position open to every other buyer.

Theorem 7 (Burn floor). *For the platform token, at every reachable state,*

$$P = \frac{x}{y} \geq P_{\text{floor}} = \frac{P_0}{(1-b)^2}, \quad b = \frac{B}{S},$$

and P_{floor} does not decrease over time.

Proof. By Lemma 5, $xy \geq VS$, and $y = S - B - C \leq S - B$. Therefore

$$\frac{x}{y} = \frac{xy}{y^2} \geq \frac{VS}{(S-B)^2} = \frac{V/S}{(1-B/S)^2}.$$

No instruction mints, and only burn changes B , upward. $\square$

P_{floor} is the price the curve reaches when every holder has sold and $k = k_0$; it requires no intervention. The same inequality bounds the SOL that remains.

Corollary 3 (Locked SOL). *The platform token’s pool always holds $R \geq Vb/(1-b)$.*

Proof. $x \geq VS/y \geq VS/(S-B) = V/(1-b)$ and $R = x - V$. $\square$

burned share b	P_{floor}/P_0	locked SOL $Vb/(1-b)$
1%	1.0203	0.303
5%	1.1080	1.579
10%	1.2346	3.333
25%	1.7778	10.000
50%	4.0000	30.000

Table 3: Floor and locked SOL for $V = 30$ SOL.

Every trade of every face/id token sends $\varphi_b = 0.5\%$ of its SOL to the treasury, and burn is permissionless. A burn of T lamports at reserves (x, y) destroys $\lfloor yT/(x+T) \rfloor$ tokens, so activity on any pool raises the platform token’s floor.

7 Implementation and evaluation

Programs. The launchpad and the gate are separate Anchor programs, built with the Agave 3.1.10 toolchain. The separation is required: with the hook inside the launchpad, every launchpad transfer would produce the chain launchpad, Token-2022, launchpad, which the runtime refuses. The gate does not call back; it reads the instructions sysvar to identify the top-level program.

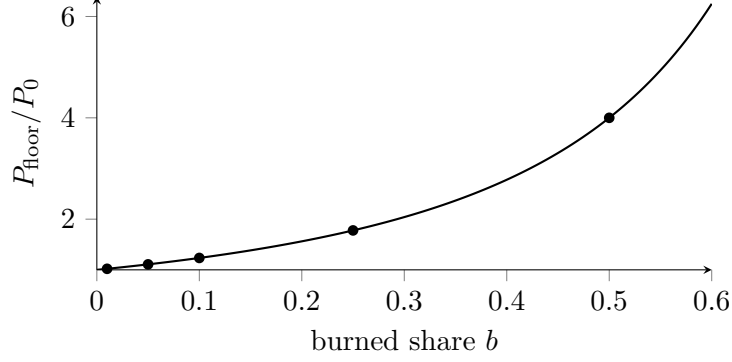


Figure 3: The floor as a function of the burned share. It is convex in b .

Two properties of the runtime shaped the code. A cross-program invocation checks that lamports are conserved over the accounts it receives, so lamports the launchpad moves by direct assignment must move after the token transfer it invokes; `sell` and `buy_cash` both transfer tokens first and settle SOL afterwards. And the account validation generated for `send`, which loads two vaults, exceeded the 4 KB stack frame of the SBF target by 16 bytes; the two vaults are therefore boxed onto the heap.

Client data in practice. Chrome inserts, at random, an extra member `other_keys_can_be_added_here` after `crossOrigin`, precisely to break verifiers that compare the whole string against a template. The rule (V4) compares only the prefix that WebAuthn Level 3 fixes, so these assertions verify, and they still fit: the largest transaction leaves about 200 bytes of headroom.

Adversarial tests. The suite runs both programs on a Solana validator (Agave 3.1.10), with the launchpad built for $\Delta = 20$ s so that every delay is observed within the run; all 95 checks pass, and every refusal is asserted by its error code. Acting with a victim’s wallet key, the suite attempts: a withdrawal to the attacker’s address; a withdrawal with no precompile instruction; assertions from the attacker’s own passkey that add a destination, remove the victim’s destination, or spend the victim’s cash on a token the attacker launched; the victim’s assertion produced on a look-alike origin, inside a frame on the attacker’s page, or for another relying party; an assertion without UV; the victim’s genuine withdrawal and send with the amount or the receiver changed; a replay of an executed assertion; a plain transfer, an approval, a burn and an owner change on the victim’s vault account; a direct invocation of the gate; and a second vault for the victim’s wallet. Each is refused. A forced sale credits its proceeds to the victim’s vault, and a withdrawal cannot reach into the vault’s rent-exempt balance.

Delays. For a vault left unattended under a stolen wallet key, the rotation confirmed after Δ , the destination list was empty afterwards, and the destination the adversary then added was refused until a second Δ had passed. The earliest exit came 53s after the rotation request, against $2\Delta = 40$ s; the difference is transaction latency, so Corollary 1 is tight up to that latency. Additions of destinations and raises of the limit took effect only after Δ , lowering the limit applied at once, and purchases beyond the limit were refused within a window.

Cost. Table 4 lists the cost of each instruction as a complete legacy transaction with a compute-budget instruction. Compute varies between runs because freshly generated keys change the number of bump searches for program-derived and associated token addresses; the table gives the range over three runs. The base fee is 5,000 lamports per signature, and

the secp256r1 record counts as one, so every passkey-gated action costs 10,000 lamports. The largest transactions, `buy_cash` and `send`, use 1,031 and 1,023 of the 1,232 bytes Solana permits, leaving room for the additional client-data member that Chrome inserts in some assertions.

Instruction	Compute units	Bytes	Fee (lamports)
<code>register</code>	15,248 to 19,748	789	10,000
<code>launch</code>	85,020	641	10,000
<code>buy</code>	73,148 to 91,148	630	5,000
<code>sell</code>	41,285 to 63,785	564	5,000
<code>buy_cash</code>	53,375 to 74,375	1,031	10,000
<code>withdraw</code>	12,410	725	10,000
<code>send</code>	49,702 to 69,202	1,023	10,000
<code>add_destination</code>	11,536	716	10,000
<code>remove_destination</code>	11,322	716	10,000
<code>set_limit</code>	11,424	692	10,000
<code>request_rotation</code>	4,830	284	5,000
<code>cancel_rotation</code>	11,249	684	10,000
<code>confirm_rotation</code>	2,816	250	5,000
<code>claim</code>	7,202	283	5,000
<code>burn</code>	14,019	382	5,000

Table 4: Measured cost per transaction. `launch` carries two signatures (creator and mint); passkey-gated instructions carry the fee payer’s and the passkey’s.

Browser. An assertion produced by Chrome’s WebAuthn implementation with user verification was accepted on-chain by the same verification routine; the same request from a look-alike origin was refused by the browser before any signature was produced.

Market arithmetic. We replayed 120 randomized histories of purchases, sales and burns, each with up to 300 operations and random slot boundaries, through the exact integer formulas of the implementation, followed by a full redemption. Every statement of Section 6 held at every step. The smallest observed ratio $out/(P_o a)$ was 0.9457 (bound 0.9405), and the largest reserve ratio y/y_o after a sale was 1.025969 (bound $(0.95)^{-1/2} = 1.025978$).

8 Parameters

We set Δ to one day, the time within which an on-chain event reaches the holder as a push notification or a message from an alert bot. With $\gamma = 5\%$, one slot absorbs a net sale of about 2.6% of the reserve (Proposition 1) at no worse than 94.05% of the opening price (Theorem 6). L is a holder setting; a holder who never buys from the vault sets it to zero, and the vault then has no purchase channel at all.

9 Related work

Smart-contract wallets on several chains accept passkey signatures, verifying P-256 in contract code or through precompiles such as RIP-7212 on EVM rollups [6] and SIMD-0075 on Solana. Spending limits and delayed recovery are established wallet features, for example Safe’s allowance module [7] and guardian-based recovery with a security period; Bitcoin vault proposals such as BIP-345 [8] add a withdrawal delay with a recovery path. In each case the protection applies to holders who choose that wallet. In face/id the requirement belongs to

the asset: since a face/id token cannot exist outside a gated vault (Lemma 2), every holder of every launched token is covered.

10 Conclusion

face/id requires, for every movement of value out of a vault, a second signature bound to the holder’s device, the launchpad’s origin and the exact action, and verifies it on-chain. With the wallet key alone an adversary can sell tokens into the holder’s vault or request a rotation; the holder cancels the rotation with one assertion, and it cannot lead to an exit before 2Δ . The surrounding market is solvent by construction, and its burn sets a floor that does not decrease.

References

- [1] W3C. *Web Authentication: An API for accessing Public Key Credentials, Level 3*. <https://www.w3.org/TR/webauthn-3/>
- [2] Solana Improvement Document SIMD-0075, *Precompile for verifying secp256r1 signatures*. <https://github.com/solana-foundation/solana-improvement-documents>
- [3] Solana Program Library. *Token-2022 extensions: transfer hook*. <https://spl.solana.com/token-2022/extensions>
- [4] S. Goldwasser, S. Micali, R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing* 17(2):281–308, 1988.
- [5] H. Adams, N. Zinsmeister, D. Robinson. *Uniswap v2 Core*, 2020.
- [6] Rollup Improvement Proposal RIP-7212, *Precompile for secp256r1 curve support*.
- [7] Safe. *Allowance module*. <https://github.com/safe-global/safe-modules>
- [8] J. O’Beirne, G. Sanders. *BIP-345: OP_VAULT*, 2023.

A Encodings and layouts

Action	Fields after domain, tag and program id	Bytes hashed
register	$w \parallel K \parallel d_0 \parallel \text{le}_{64}(L) \parallel \text{le}_{64}(0)$	162
buy_cash	$m \parallel w \parallel \text{le}_{64}(v) \parallel \text{le}_{64}(\underline{a}) \parallel \text{le}_{64}(n)$	137
withdraw	$w \parallel d \parallel \text{le}_{64}(v) \parallel \text{le}_{64}(n)$	129
send	$m \parallel w \parallel d \parallel \text{le}_{64}(a) \parallel \text{le}_{64}(n)$	157
add_destination	$w \parallel d \parallel \text{le}_{64}(n)$	128
remove_destination	$w \parallel d \parallel \text{le}_{64}(n)$	131
set_limit	$w \parallel \text{le}_{64}(L') \parallel \text{le}_{64}(n)$	98
cancel_rotation	$w \parallel K' \parallel \text{le}_{64}(n)$	129

Table 5: Intent encodings. Each is prefixed by the 9-byte domain `faceid/v1`, the tag, and the 32-byte program id; the last column is the full length of `enc`.

Precompile instruction. The client builds one record: byte 0 holds the count 1, byte 1 is padding, bytes 2 to 15 hold seven little-endian 16-bit fields (signature offset 49 with index 0xFFFF; key offset 16 with index 0xFFFF; message offset 113, message length, index 0xFFFF), followed by the 33-byte compressed key, the 64-byte signature with low s , and the message $AD \parallel \text{SHA256}(CD)$. With a 37-byte AD the message has 69 bytes and the instruction data 182.

Client data. The accepted prefix has $92 + |O|$ bytes: 36 bytes up to the opening quote of the challenge, 43 for $\beta(c)$, 12 for `", "origin": "`, the origin, and a closing quote.

Accounts. A vault occupies 323 bytes: an 8-byte discriminator, wallet (32), passkey (33), nonce (8), four destination slots of 40 bytes, limit, spent amount, window start, pending limit and its activation time (8 each), pending passkey (33), its activation time (8) and a bump byte. A pool occupies 161 bytes.

B Token-2022 operations that decrease a balance

Operation	Authority required	For a face/id account
TransferChecked	owner or delegate	owner is a launchpad PDA; gate runs
Transfer	owner or delegate	refused for accounts of a hooked mint
Burn, BurnChecked	owner or delegate	owner is a launchpad PDA
Approve, ApproveChecked	owner	the launchpad never approves
SetAuthority (owner)	owner	blocked by <i>ImmutableOwner</i>
CloseAccount	owner, zero balance	moves no tokens
permanent delegate	mint extension	extension absent
withheld fee withdrawal	mint extension	extension absent
confidential transfer	mint extension	extension absent
freeze, pause	mint authority or extension	authority empty; extension absent

Table 6: Paths by which Token-2022 can decrease a token balance, and why none is available outside the launchpad. Mint extensions cannot be added after initialization.