V 1.0 — 2026

Verdik

// PROJECT VERDIK · SOLANA MAINNET

Trust, but verify. On-chain.

A privacy-first, code-level trust layer for Solana — auditing the developer and the source code before any capital moves.

CHAIN	NARRATIVE	LICENSE	DOCUMENT
Solana	Privacy + Trust	MIT	WP-2026-01

// 00 — DISCLAIMER

Legal Notice

This document is for informational purposes only and does not constitute an offer, solicitation, or recommendation to purchase, hold, or trade any digital asset. Project Verdik is an open-source, free, code-level trust scanner. No ticker has been assigned at the time of publication.

Nature of this Document

This whitepaper outlines the architecture, methodology, and roadmap of Project Verdik — including its on-chain privacy layer. It is published as a technical reference, not as financial guidance. Readers are expected to evaluate the document on its engineering merits.

No Investment Advice

Nothing in this whitepaper should be interpreted as investment, legal, accounting, or tax advice. Cryptographic systems, smart contracts, and on-chain assets carry inherent technical, economic, and regulatory risk. Always conduct independent research and consult qualified professionals before making decisions.

Forward-Looking Statements

This whitepaper contains forward-looking statements regarding planned features, architectural decisions, and roadmap milestones. These reflect current expectations and assumptions; actual outcomes may differ as the project evolves through community feedback, technical iteration, and on-chain conditions.

Open-Source Commitment

All MVP components of Project Verdik — including the Developer Trust Score engine, AST plagiarism scanner, and security rule database — are released under the MIT License. The engine runs without paid API dependencies, telemetry, or third-party data calls.

> // AUDIT PRINCIPLE
> Verdik treats every claim, score, and verdict as evidence-bound. If a signal cannot be cited from raw on-chain or repository data, it is not part of the verdict.

Regulatory Acknowledgment

The classification of digital assets and on-chain attestations varies by jurisdiction. Project Verdik makes no representation regarding the legal status of any future utility primitive in any given jurisdiction. Users are responsible for ensuring their interaction with the protocol complies with applicable laws.

Intellectual Property

The Verdik name, logo, brand assets, and all referenced trade dress are property of the Verdik project. The underlying source code is licensed under MIT and may be forked, audited, or self-hosted by any party.

// INDEX

Table of Contents

// 01

Executive Summary

Project Verdik is a code-level trust layer for Solana that audits the developer and the source code behind every token before capital is committed. Where existing tools chase on-chain symptoms after the rug, Verdik inspects the actual mechanism that enables it.

The project is built around two forensic engines — a Developer Trust Score (DTS) that profiles the GitHub identity behind a project, and an Abstract Syntax Tree (AST) plagiarism scanner that fingerprints source code against a curated database of 29 known rug-pull templates. Both engines run free, open-source, with no API keys, no telemetry, and no paid SaaS dependencies.

Layered on top of this scanning infrastructure is Verdik's privacy module — which moves attestations on-chain in a way that protects the identities of contributors and reporters while preserving public verifiability of the scan verdict.

29	24	<12s	4
RUG SIGNATURES	SECURITY RULES	SCAN TIME	LANGUAGES

Why this matters

The Solana ecosystem processes thousands of token deployments per week. The overwhelming majority of capital loss in the cycle traces to a small set of repeating code patterns — fee-rugs, mint-rugs, freeze-rugs, drain functions — many of which are textually similar to prior incidents. A scanner that can recognize structural similarity at the function level, combined with a transparent developer history check, converts what is currently a guessing game into a verifiable engineering decision.

What Verdik is not

Verdik is not a black-box AI oracle, not a paid SaaS product, not a centralized signal feed. Every penalty applied by the engine cites the underlying evidence. Every weight is documented. Every signature is open to public review. The endgame is a self-sustaining trust layer for Solana — community-curated, on-chain attested, with skin in the game from those who vouch for projects.

> // THESIS
> The rug-pull is, fundamentally, a software supply-chain problem. Solve it at the code, not at the chart.

// 02

Introduction

Solana is the highest-throughput chain in active retail use. It is also the chain where the gap between a token being deployed and that token being scrutinized has shrunk to near zero — often measured in seconds. Capital moves faster than any human auditor can read code.

The trust gap

Every new Solana token represents an unverified claim. The claim is structural: the deployer asserts that the code does what the marketing implies. In practice, the gap between intent and implementation is the surface where almost every retail loss originates. A token can be technically correct, structurally identical to a known scam, and still trade for hours before the pattern is recognized.

Traditional auditing — manual review of source by a paid firm — does not scale to the cadence of Solana launches. A typical audit takes days or weeks. A typical Solana token has a meaningful trading window of minutes. The two timelines have never overlapped.

The privacy paradox

The standard response to the trust gap is more disclosure — KYC the deployers, publish the source, attach a known auditor. This approach has two failure modes. First, it pushes legitimate anonymous developers out of the ecosystem. Second, public attestation creates a permanent on-chain record of who said what about whom, which itself becomes an attack surface.

Verdik's response — embodied in its on-chain privacy layer — is that trust and privacy are not opposing forces. A scan verdict can be public and verifiable while the contributor or reporter behind it remains pseudonymous. The verdict is what gets attested; the identity stays compartmentalized.

verdik-engine // overview.sh

```
$ verdik scan --target github.com/example/token-v3
[01] resolving developer profile ..... ok
[02] fetching repository tree ........ ok
[03] hashing AST signatures (n=412) ... ok
[04] security rule sweep (24 rules) ... 2 hits
[05] synthesizing verdict ............ done

# result
verdict: CAUTION (38/100)
match : presale-king-v2 (jaccard 0.71)
cite : /contracts/Token.sol:114-138
```

Scope of this document

This whitepaper covers Verdik's technical architecture, the two forensic engines (DTS and AST), the security rule database, the on-chain privacy module, the threat model, and a phased roadmap from open-source scanner to community-staked attestation network. It is intended for developers, auditors, launchpad operators, and serious investors who want to understand how the system arrives at its verdicts.

// 03

Problem Statement

Existing rug-pull defenses on Solana fall into three categories: post-mortem analytics, holder-graph heuristics, and centralized reputation databases. Each addresses a downstream symptom rather than the cause.

Failure mode 1 — Post-mortem analytics

Most "scam detector" tools operate after the deployment is on-chain. They monitor liquidity events, large transfers, and freeze authority calls. By the time these signals fire, the loss has already happened. The signal is correct but useless as a defense — it confirms a rug rather than preventing one.

Failure mode 2 — Holder-graph heuristics

Holder-distribution analysis (sniping wallets, insider clusters, suspicious early holders) catches some patterns but is easily defeated. A scammer with operational discipline distributes initial supply across hundreds of throwaway wallets, mimicking organic distribution. The on-chain footprint becomes indistinguishable from a legitimate launch.

Failure mode 3 — Centralized reputation

Curated lists — "trusted projects," "known scams" — are slow, opinionated, and a single point of failure. They cannot keep pace with token deployment cadence, and they create a privileged class of insiders who decide what is "legitimate." Once compromised, the list itself becomes an attack vector.

What is missing

None of the above examines the actual mechanism of the rug — the code. A scammer can fake holders, fake liquidity history, fake social presence, even fake an audit certificate. What they cannot easily fake is the structural similarity of their contract to prior rug templates, because rewriting a contract from scratch is harder than copying one. The economic incentive favors reuse.

// THE ASYMMETRY

Building a rug from scratch costs effort. Forking an existing one costs nothing. Over 70% of observed rug deployments share 5-gram signature overlap with prior incidents.

// THE OPPORTUNITY

Structural code matching closes the window. If a contract's function-level fingerprint matches a known rug template, that match is independent of token name, ticker, or marketing.

The pseudonymity problem

Any tool that demands real-world identity from project teams will not be adopted by the segment of the ecosystem that needs auditing the most. Anonymous teams are not inherently malicious — many legitimate Solana projects are pseudonymous by design. The defense must work without forcing identity disclosure. This is the constraint that motivates Verdik's on-chain privacy layer.

APPROACH	CATCHES CODE RUGS	PRE-LAUNCH	PRIVACY-PRESERVING
Post-mortem analytics	NO	NO	N/A
Holder-graph heuristics	NO	PARTIAL	NO
Centralized reputation lists			

	PARTIAL	PARTIAL	NO
Verdik (DTS + AST)	YES	YES	YES (VIA PRIVACY LAYER)

// 04

The Verdik Solution

Verdik runs every Solana project through two parallel forensic checks — one on the developer behind it, one on the code they shipped. The two outputs combine into a single verdict accompanied by an evidence trail any reviewer can audit independently.

Design principles

- **Evidence-bound:** No penalty is applied without a citable signal. Every score component links back to raw data.
- **Free and open:** No subscriptions, no API keys for the core engine, no paid dependencies. The entire pipeline is auditable.
- **Pre-launch:** A scan runs in under twelve seconds, fast enough to complete before a token deployment hits visibility.
- **Pseudonym-respecting:** The system audits developer behavior, not legal identity. Real-name disclosure is never required.
- **Self-hostable:** A team that distrusts hosted infrastructure can run Verdik entirely on their own machine.

The two-engine model

The DTS engine asks: `is the person shipping this code who they claim to be, and does their history match a legitimate developer profile?` The AST engine asks: `does this code structurally resemble a known scam?` Either signal alone is insufficient. A legitimate developer can ship bad code; a malicious actor can borrow a clean profile. The combination is what produces a defensible verdict.

// DTS — DEVELOPER TRUST SCORE

Account age, commit cadence, contributor graph, repository activity, bio risk signals. Cited evidence behind every penalty. Zero personal data leaves the engine.

// AST — CODE FINGERPRINTING

5-gram Jaccard similarity at the function level, normalized for variable renames and whitespace. 29 hand-curated rug templates. Token-level normalization makes renames meaningless.

Why combine them

A DTS hit with a clean AST suggests a high-risk developer who has not yet shipped a recognizable rug — useful as an early warning. An AST hit with a clean DTS suggests a developer who reused suspicious code patterns, possibly without malicious intent — useful as a remediation signal. Both hits firing simultaneously is the high-confidence rug case. Both clean is the high-confidence legitimate case.

> **// RULE OF TWO**
>
> The verdict is never decided by a single signal. Every published score reflects independent evidence from at least two of the three layers: developer history, code structure, and security rule sweep.

Verdict format

The engine outputs a numeric score (0–100), a categorical verdict (safe, caution, high-risk, critical), and a structured evidence list. The score is reproducible, and every penalty cites the line of code, commit, or profile signal that triggered it.

// 05

Technical Architecture

The Verdik engine is a streaming pipeline. Input enters at the top, evidence accumulates at each layer, and the verdict emerges synthesized from the entire evidence graph. Every step is observable in real time via Server-Sent Events.

Layered architecture

// L1 — INPUT	GitHub handle · repository URL · raw source paste · file upload
↓	
// L2 — FETCH	Repository resolver · profile resolver · raw blob loader · rate-limited HTTP client
↓	
// L3 — PARSE	Language detector · AST builder (Solidity, Rust/Anchor, JS, TS) · normalizer
↓	
// L4 — ANALYZE	DTS scorer · 5-gram Jaccard matcher · rule engine · category mapper
↓	
// L5 — SYNTHESIZE	Evidence aggregator · verdict synthesizer · risk heatmap · diff renderer
↓	
// L6 — OUTPUT	SSE stream · permalink · on-chain attestation · embed badge

The streaming pipeline

Unlike batch scanners that return a single response object, Verdik streams its work. Each step emits a typed event with its raw data, allowing the caller to surface progress, inspect intermediate signals, or abort early if a critical finding appears. The pipeline is auditable end-to-end: a reviewer can replay the exact sequence of events that produced any given verdict.

01	02	03	04	05
Resolve Profile	Fetch Repos	Hash AST	Rule Sweep	Synthesize Verdict

Component breakdown

COMPONENT	ROLE	LANGUAGE
Resolver	Maps GitHub handles to profile metadata and repository graphs.	TypeScript
AST Parser	Builds language-specific syntax trees for Solidity, Rust, Anchor, JS, TS.	TypeScript + native parsers
Normalizer	Strips identifiers and whitespace, producing a rename-invariant token stream.	TypeScript
Hasher	Computes 5-gram fingerprints for every parsed function.	TypeScript
Jaccard Matcher	Compares function fingerprints against the rug-template database.	TypeScript
Rule Engine	Applies the 24 security rules across all supported languages.	TypeScript
DTS Scorer	Computes the Developer Trust Score from profile signals.	TypeScript
Synthesizer	Aggregates layer outputs into a final verdict with cited evidence.	TypeScript
Attestation Bridge	Pushes scan-result hashes on-chain as privacy-preserving attestations.	Rust (Anchor)

Why TypeScript for the engine

The scanner targets predominantly TypeScript and JavaScript repositories on the input side, and increasingly Rust/Anchor on the Solana program side. Keeping the engine itself in TypeScript means contributors can read, audit, and extend the rules in the same language ecosystem they already work in. The Anchor on-chain component is the only Rust surface, scoped to the on-chain attestation primitive.

Self-hosting model

Verdik can run as a hosted service or entirely on a contributor's local machine. The only optional external dependency is an unauthenticated GitHub API endpoint for rate-limit-friendly fetching; supplying a personal access token raises the rate ceiling but is never required. There is no telemetry, no analytics call, no third-party data broker in the path.

verdik-engine // self-host.sh

```
$ git clone https://github.com/verdikcodes/Codegraph-Verdik
$ cd Codegraph-Verdik && pnpm install
$ cp .env.example .env # optional: add GITHUB_PAT
$ pnpm dev
▲ verdik engine ready on :3000 — offline mode supported
```

> // DESIGN CONSTRAINT
>
> The engine must produce a complete verdict on a typical Solana program (≤ 2 MB of source) in under 12 seconds on commodity hardware. Anything slower defeats the pre-launch use case.

// 06

Developer Trust Score (DTS)

The DTS engine treats the developer behind a project as the first piece of evidence. It quantifies how a GitHub identity behaves: how long has it existed, how have its commits clustered over time, who in the network has vouched for it, and does its bio match patterns associated with throwaway accounts?

Signal categories

DTS combines weighted signals from four categories, each with documented evidence requirements:

ACCOUNT-AGE WEIGHTING

Accounts younger than thirty days carry the highest baseline penalty. The penalty decays non-linearly toward zero as account age increases past one year. Reasoning: throwaway scam accounts are typically newly created; established developers rarely operate from a fresh profile.

COMMIT BURST DETECTION

Legitimate development has irregular commit cadence with meaningful intervals between pushes. A burst pattern — hundreds of commits in a single afternoon, no prior activity, repository deleted shortly after — is a strong adversarial signal. The detector flags clusters where commit density exceeds a configurable threshold and the surrounding timeline is empty.

CONTRIBUTOR-GRAPH SIGNALS

The follower and follow-back graph of legitimate developers tends to overlap with the broader open-source community. Isolated accounts whose graph consists entirely of similarly young, similarly empty accounts are flagged. This signal is privacy-aware: only graph topology is examined, never individual follower identities.

BIO RISK ANALYSIS

Profile bios containing promises of unrealistic returns, "presale" links, financial advice disclaimers, or known scam phrasings raise the bio-risk component. The rule set is published, auditable, and accepts community-submitted patterns.

Scoring formula

The DTS is a weighted sum where each component contributes both a numeric weight and a citable evidence object. The synthesizer never produces a final score without attaching the evidence list — a DTS of "62" with no breakdown is rejected as malformed.

dts-scorer.ts // sample output

```
// DTS breakdown for handle: 0x_anon_dev
account_age : -18 // 14 days old
commit_burst : -22 // 142 commits / 6 hours
graph_isolation : -12 // no overlap w/ known devs
bio_risk : -8 // match: "guaranteed x100"
repo_activity : +4 // non-zero history
─────────────────
final_dts : 24 / 100 // verdict: high-risk
```

Reading a DTS verdict

The categorical bands map roughly to actionable thresholds. A score above 80 is consistent with a long-established developer with a healthy contributor graph and no risk-flagged signals; this is a green light from the DTS layer alone, though the AST scan must also clear. A score between 60 and 80 indicates an established but unremarkable profile — neutral evidence. Below 40 indicates strong adversarial signals; this alone is sufficient to warrant pre-launch rejection by most launchpads.

BAND	SCORE	INTERPRETATION
SAFE	80 – 100	Established developer, healthy graph, no risk signals.
HEALTHY	60 – 79	Unremarkable profile, no specific red flags.
CAUTION	40 – 59	Mixed signals — at least one component is risk-flagged.
HIGH-RISK	20 – 39	Multiple adversarial signals firing simultaneously.
CRITICAL	0 – 19	Profile resembles throwaway accounts associated with prior rugs.

What DTS does not do

The DTS engine does not perform real-world identity matching. It does not attempt to link a GitHub handle to a person, company, or wallet. Its scope is intentionally narrow: behavioral signals from the public GitHub surface only. This keeps the layer pseudonymity-respecting and prevents DTS itself from becoming a privacy attack surface.

False positives and remediation

Every penalty is appealable in the sense that it cites its own evidence. A developer who believes a penalty is incorrect — for example, a legitimate burst caused by an initial open-source release — can publicly counter the evidence trail. The synthesizer accepts contextual evidence, allowing a verified counter-signal to neutralize the original penalty. This is not subjective override; it is evidence-on-evidence.

> // TRANSPARENCY PRINCIPLE
>
> If a DTS verdict cannot be reconstructed from the published evidence list, the verdict is wrong by definition. The score itself is a derivative of the evidence, not a primary judgment.

Evidence object format

Each DTS component emits a structured evidence record consisting of signal name, magnitude, the raw observation that triggered it, and a citation pointing back to the source data. These records are what get hashed and committed on-chain through the privacy layer — the verdict is reproducible because its inputs are immutable.

// DTS COST

Sub-second on cached profiles. ~3-5 seconds cold, dominated by repository graph fetching, not analysis.

// DTS SURFACE

Public GitHub data only. No private repos, no email leakage, no third-party data sources.

// 07

AST Plagiarism Scanner

The AST engine is the core defensive layer. Where DTS profiles behavior, AST examines the actual mechanism of the contract. It fingerprints each function and compares those fingerprints against a curated database of twenty-nine known rug-pull templates, returning a structural similarity score that is invariant to renaming.

Why ASTs, not regex

String-matching scanners are trivial to defeat. Rename a variable, reorder a statement, swap a function name — the regex misses. ASTs work at the syntactic structure level: the parser produces a tree of language constructs, and two functions that do the same thing produce structurally similar trees regardless of cosmetic edits.

Verdik's normalizer strips identifiers entirely. After normalization, two functions are compared on their tree shape and operator sequence alone. `variable_a + variable_b` and `foo + bar` reduce to identical fingerprints. This is the property that makes copy-paste rugs visible even when the deployer has put effort into superficial obfuscation.

The 5-gram Jaccard method

For each function, the normalizer produces a token stream. The hasher generates all overlapping 5-grams (sequences of five consecutive tokens). The fingerprint is the set of these 5-grams. Two functions are compared using Jaccard similarity — the size of the intersection of their fingerprint sets divided by the size of the union.

ast-matcher.ts // jaccard math

```
// Jaccard similarity
J(A, B) = |A ∩ B| / |A ∪ B|

// where A and B are the 5-gram fingerprint sets
// of the candidate function and the template, respectively.

// threshold bands
J < 0.30 → structural overlap unremarkable
J 0.30–0.55 → partial reuse — review flagged
J 0.55–0.75 → heavy structural match — high suspicion
J ≥ 0.75 → near-identical to known rug template
```

Why 5-grams specifically

Shorter n-grams (3-grams) produce too many spurious matches between unrelated functions — basic language constructs like loop initializations look similar everywhere. Longer n-grams (8-grams or above) become brittle to even small structural edits, missing genuine matches. Empirically, 5-grams sit at the inflection point: long enough to be semantically meaningful, short enough to survive minor refactoring.

The signature database

The 29 rug-pull signatures in the initial database are hand-curated from documented post-mortems across Solana, Ethereum, and EVM-compatible chains. Each signature is itself an open record — the source contract, the post-mortem reference, the contributor who added it. The database is versioned, signed, and publicly auditable.

Signature categories

The initial 29-template database spans seven category buckets, each capturing a distinct mechanism by which a contract can be weaponized against its holders:

CATEGORY	COUNT	MECHANISM
Fee-rug	6	Hidden transfer fees that escalate after liquidity is locked.
Mint-rug	5	Owner-only mint functions that allow unlimited supply inflation.
Freeze-rug	4	Per-account freeze authority retained on the mint.
Drain function	5	Privileged transfer or wallet-drain entry points.
Honeypot	4	Asymmetric transfer logic that blocks sells after buys.
Approval rug	3	Patterns soliciting MaxUint approvals against drainer contracts.
Phantom upgrade	2	Upgrade paths that route to a different implementation than advertised.

The diff viewer

When a function flags, the engine produces a side-by-side comparison: the candidate function on the left, the matched template on the right, with the structurally aligned tokens highlighted. This is the artifact that converts a numeric score into a reviewable claim. A reviewer who disagrees with the match can verify by inspection rather than by trust.

Beyond fingerprinting — the call graph layer

Function-level Jaccard matching catches direct reuse. Phase 3 of the roadmap extends this with cross-function call graph analysis: even if no single function matches a template, a call sequence that mirrors a known rug pattern is itself a signal. This catches the more sophisticated attacker who distributes the malicious logic across multiple cosmetically distinct functions.

// RECALL

High on direct forks of the 29 templates. Designed to catch copy-paste rugs first.

// PRECISION

Threshold-tunable. The default 0.55 cutoff balances recall against legitimate-pattern noise.

// EXTENSIBILITY

New signatures can be added by anyone. The DB is community-curated and versioned.

The contribution surface

The signature database grows fastest when the people closest to a rug — the early flaggers, the post-mortem authors — can add the offending pattern without going through a gatekeeper. The contribution model is open: any contributor can submit a signature, the matcher runs it against the existing database to detect duplicates, and the community reviews the inclusion through the same evidence-bound process used for verdicts.

> **// NETWORK EFFECT**
>
> Every rug that gets pattern-matched into the database raises the cost of the next rug. The attacker either rewrites from scratch — losing the economic advantage of reuse — or ships a signature-matched contract that flags before launch.

// 08

Security Rule Engine

Alongside fingerprint matching, Verdik runs a deterministic rule sweep — 24 pattern rules spanning Solidity, Rust/Anchor, JavaScript, and TypeScript. These rules catch the well-known anti-patterns that any auditor would flag on first reading, automated to scale.

Rule classes

The 24 rules are grouped into four severity classes, each corresponding to a level of certainty about the risk:

CLASS	EXAMPLES	SEVERITY
Privileged execution	`tx.origin` auth, `delegatecall`, `selfdestruct`, raw `assembly` blocks	CRITICAL
Authority retention	Unrevoked mint authority, unrevoked freeze authority, retained upgrade keys	HIGH
Approval/transfer abuse	MaxUint approvals, hidden routing through unverified contracts	HIGH
Frontend exfiltration	Hidden seed-phrase forms, wallet-drain UI patterns, fake claim flows	CRITICAL

Language-specific rules

Each language carries its own rule subset. Solidity rules target EVM-specific anti-patterns; Rust/Anchor rules focus on Solana-native authority management; JavaScript and TypeScript rules cover the wallet-facing frontend code where most user-side exfiltration patterns appear.

SOLIDITY

Eight rules covering `tx.origin` authentication, unsafe `delegatecall`, `selfdestruct` presence, raw assembly in non-library context, unbounded loops over user-controlled arrays, unsafe external calls, MaxUint approval patterns, and proxy-implementation mismatches.

RUST / ANCHOR

Six rules specific to Solana program development: unrevoked mint authority, unrevoked freeze authority, missing constraint checks on account validation, unsafe `invoke_signed` usage, transfer hooks routing through unverified programs, and upgrade authority retained post-launch.

JAVASCRIPT / TYPESCRIPT

Ten rules covering the frontend surface: hidden form fields requesting seed phrases, transaction-signing flows that obscure the actual instruction, fake "claim" buttons that route to drain contracts, MetaMask/wallet-popup patterns mismatched against the displayed intent, and DOM injection patterns associated with known drainer kits.

rule-sweep.ts // verdict fragment

```
// Rule sweep result for /contracts/Token.sol
✓ SOL-001 tx.origin auth ..............  clean
× SOL-003 selfdestruct present .........  L:217
× SOL-007 max-uint approval pattern ....  L:142
✓ SOL-011 unbounded user-array loop ....  clean
! SOL-014 upgrade authority retained ...  L:09 # review
```

```
// → 2 critical, 1 warning. evidence cited.
```

> // RULE PHILOSOPHY
>
> A rule is only included if it can be deterministically applied without false-positive flood. The bar is high: when Verdik flags a rule violation, the line of code that triggered it is part of the verdict.

// 09 — THE PRIVACY MODULE

The Privacy Layer

Verdik's privacy layer is the module that takes a scan verdict and commits it to Solana in a way that preserves public verifiability while protecting the identity of the scanner, the reporter, and any community attester. The verdict goes on-chain. The person behind the verdict does not.

The motivation

An on-chain attestation system creates two opposing pressures. On one hand, attestations are most valuable when they are public, durable, and easy to verify — anyone should be able to look up the verdict for a given contract. On the other hand, the act of publicly attesting carries personal risk: scammers retaliate against named flaggers, reputable contributors face doxxing, and legitimate teams may hesitate to scan if the act of being scanned leaks their identity.

The privacy layer's design goal is to separate the two dimensions: **the verdict is fully public; the attester is pseudonymous by default.** The chain knows the contract was scanned and what the result was. It does not know who initiated the scan.

Architecture

// STEP 1	Engine produces verdict object (score, category, evidence list).
↓	
// STEP 2	Verdict serialized canonically and hashed (SHA-256). Hash is the on-chain commitment.
↓	
// STEP 3	Stealth-address derivation produces a one-time attestation account. Original signer remains uncorrelated.
↓	
// STEP 4	Attestation PDA stores: target program ID, verdict hash, signature, timestamp, version.
↓	
// STEP 5	Public retrieval: anyone can fetch the attestation, recompute the hash from the verdict, and verify.

Key properties

- **Verifiability:** Anyone can recompute the verdict hash and confirm the on-chain commitment.
- **Unlinkability:** Stealth addresses prevent two attestations from the same originator being correlated.
- **Append-only:** Verdicts are permanent. Updates are new attestations, preserving history.
- **Selective disclosure:** The originator can choose to reveal themselves later to claim credit.

Why Solana for the attestation layer

Solana's account model is well suited to per-attestation Program Derived Addresses (PDAs). Each verdict gets its own deterministic address, derived from the target program ID and a stealth seed, allowing efficient lookup without enumerating an index. Transaction fees on Solana are low enough that attestation cost is not a barrier even at scale, which matters because the network value of the privacy layer scales with the number of recorded verdicts.

The attestation object

Each on-chain attestation stores a fixed-size record. The full verdict — including the evidence list — is published off-chain and addressable by its hash. The chain holds the commitment; the public archive holds the content. This split keeps on-chain costs constant per attestation while preserving the full audit trail.

verdik/attestation.rs // pda layout

```
// Solana PDA — fixed 96 bytes
pub struct Attestation {
  pub target_program : Pubkey, // 32 bytes
  pub verdict_hash : [u8; 32], // SHA-256
  pub stealth_pubkey : Pubkey, // one-time
  pub score : u8, // 0-100
  pub category : u8, // enum
  pub engine_version : u16,
  pub timestamp : i64,
  pub _reserved : [u8; 14],
}
```

What is on-chain — and what is not

ON-CHAIN	OFF-CHAIN
Verdict hash (32 bytes)	Full verdict object with evidence list
Target program ID	Source code, AST fingerprints
Score & category	Per-rule breakdowns, DTS components
Stealth one-time pubkey	Originating wallet (compartmentalized)
Engine version, timestamp	Scan transcript / SSE log

Privacy threat model — what the layer protects against

The privacy layer is designed against three adversaries. `Adversary A` is a scammer attempting to retaliate against the user who flagged their contract. `Adversary B` is a third-party data broker building behavioral profiles of who scans what. `Adversary C` is an on-chain observer attempting to correlate scan activity with a known wallet. The stealth-address layer breaks all three correlations at the on-chain surface.

The system does **not** claim privacy against an adversary with direct access to the originator's machine or network logs. The privacy boundary is the on-chain attestation surface.

> // DESIGN BOUNDARY
> The privacy layer is a publication-layer privacy primitive, not a network-layer anonymity system. Users who need stronger guarantees can layer a Tor relay or private RPC in front of scan calls.

// 10

Use Cases

Verdik's primary user is not a single archetype. The system produces evidence-bound verdicts that map onto multiple decision-making contexts across the Solana ecosystem.

Token founders & legitimate teams

Serious projects use Verdik to demonstrate clean code and a transparent developer history before public launch. The Pre-launch Verified Badge — earned by passing a scan above a threshold — becomes a visible signal investors and launchpads can verify independently. Continuous monitoring re-runs the scan on every commit, so the verdict stays accurate as the codebase evolves.

DAO treasuries & investment funds

Treasuries deploying capital into Solana primitives use Verdik as a pre-investment screen. The scan output integrates into existing diligence workflows: a failing scan becomes a documented reason to defer or decline an allocation, rather than an opinion. Aggregate risk dashboards span the entire portfolio, surfacing drift as scanned projects evolve.

Solana launchpads

Launchpads can require a pre-launch Verdik scan as part of their listing diligence. The badge becomes embeddable on the launchpad's listing page, transferring the audit burden from the launchpad's manual review onto a reproducible automated layer. Launches that fail the scan can be either declined or surfaced with their specific findings, putting the decision back in the investor's hands.

DeFi auditors & security researchers

Independent auditors use Verdik as a first-pass screen before committing time to manual review. A signature match against a known rug template is a strong prior — the auditor knows immediately whether the contract resembles prior incidents, and can focus their attention on the structurally novel parts.

Wallet apps & DEX frontends

End-user wallets and DEX UIs can query the on-chain attestation layer at the point of transaction signing, surfacing a verdict directly in the signing flow. Users see the scan result on the screen at the moment they would otherwise approve an unverified contract — converting a passive risk into an actionable warning.

// INDIVIDUAL INVESTOR

Paste a GitHub URL or program ID before aping. The verdict appears in under twelve seconds, cited and reviewable.

// CI/CD INTEGRATION

Project repos can fail builds when a security rule fires. The defense moves left into the development lifecycle.

Internal tooling for serious teams

Verdik is self-hostable. A token team can run a private instance of the engine, add internal signatures specific to their codebase, and gate merges on the in-house signature database. The system becomes both a public trust signal and an internal code-quality bar.

> **// ADOPTION THESIS**
>
> The same verdict format serves the investor, the launchpad, the auditor, and the wallet UI. Standardizing the artifact is what makes the trust signal portable across the ecosystem.

// 11

Security Model & Threat Surface

Every system that makes claims about trust must be transparent about what it does — and does not — protect against. Verdik's threat model is explicit, scoped, and documented; this section enumerates the assumptions and the corresponding mitigations.

Trust assumptions

Verdik operates under a small, explicit set of assumptions:

- The GitHub API returns accurate public profile data and repository contents.
- The Solana network reliably finalizes attestation transactions.
- The signature database is curated by reviewers who can be independently audited.
- SHA-256 is collision-resistant for verdict hashing.

Each of these assumptions is auditable. If any of them breaks, the failure mode is documented and the corresponding output is invalidated rather than silently corrupted.

Adversarial surface

THREAT	MITIGATION
Deployer obfuscates code via renaming	AST normalizer strips identifiers before fingerprinting; renames are invariant.
Deployer splits malicious logic across functions	Call-graph analysis (Phase 3) flags suspicious cross-function patterns.
Deployer farms an old GitHub account	Commit-burst and graph-isolation signals detect dormant-then-active patterns.
Deployer submits a forged signature to the DB	Community review process plus on-chain version pinning; reviewers can revert.
Attester is doxxed via attestation transaction	Stealth-address layer; attester pubkey is one-time and unlinkable.
Scan engine output is tampered in transit	Verdict hash on-chain; any consumer can re-derive and verify locally.
False positives damaging legitimate projects	Evidence list is part of the verdict; counter-evidence neutralizes the flag.

What Verdik cannot protect against

Honest acknowledgment is part of the trust model. Verdik does not protect against:

- **Genuinely novel rugs** with no structural overlap to any known template. The signature DB catches reuse, not novelty.
- **Off-chain social engineering** — phishing, fake support staff, Telegram impersonation.
- **Post-launch governance attacks** where a legitimate-at-deploy contract is later upgraded maliciously through retained authority. The continuous-monitoring layer mitigates this, but the window remains.

- **Marketing fraud** — false claims about partnerships, exchange listings, or token utility. Verdik audits code, not press releases.

Defense in depth

Verdik is one layer in a multi-layered defense. Holder-graph analysis, on-chain monitoring, manual auditing, and community vigilance all remain necessary. The contribution Verdik makes is to close the specific gap between deployment and review — to make code-level evidence available before capital moves, not after.

> // PRINCIPLE
>
> A scan is a signal, not a sentence. A clean Verdik verdict is not permission to skip diligence. A flagged verdict is reason to slow down — almost never the only reason.

// 12

Roadmap

Verdik's trajectory moves from open-source scanner to community-staked attestation network. Each phase ships independently usable software; nothing is gated on the next phase being live.

// PHASE 01 — SHIPPED

LIVE

DTS Forensics + AST Scanner (MVP)

- Developer Trust Score with transparent reasoning engine
- AST plagiarism scanner — 29 rug signatures, 24 security rules
- Live SSE streaming pipeline, code diff viewer, category heatmap
- Three input modes: GitHub repo, direct paste, file upload

// PHASE 02 — IN PROGRESS

ACTIVE

Privacy layer on Solana mainnet

- Solana mainnet deployment of the attestation primitive
- Scan-result attestations stored on-chain via Verdik-PDA
- Embeddable badges and shareable permalinks for verified projects
- Public scan archive (anonymized, opt-in)

// PHASE 03 — NEXT

PLANNED

Pre-launch verification & code intelligence

- Pre-launch Verified Badge for legitimate Solana projects
- Function call graph and self-similarity detection
- Local explanation layer (Ollama) — zero external API cost
- Cyclomatic complexity metrics and per-function quality profile

// PHASE 04 — FUTURE

FUTURE

Community-staked attestations

- Stake-to-vouch model: contributors with skin in the game can attest to projects
- Rugged projects burn staked positions, redistributing to early flaggers
- DAO-curated signature database with on-chain governance
- Cross-chain reputation lookup (Ethereum, Base, Bitcoin L2 surfaces)

// COMMITMENT

Every roadmap milestone is independently verifiable. MVP code is open-source from day one. No feature is described in this document that has not been either shipped or scoped into an implementation plan.

// 13 — FINAL NOTES

Conclusion

The rug-pull problem is not a market problem. It is a software supply-chain problem with predictable structural signatures. Verdik treats it as such — auditing the developer and the source code with reproducible, evidence-bound methods, and committing the resulting verdicts to Solana through a privacy-preserving attestation layer.

What we have built

The MVP is shipped: a forensic engine that scans a GitHub handle or repository in under twelve seconds, applies twenty-four deterministic security rules across four languages, and matches function-level fingerprints against twenty-nine known rug templates. Every verdict cites its evidence. Every signal is reproducible. The engine is open-source, free, and self-hostable.

What comes next

The on-chain privacy layer brings these verdicts to Solana mainnet. The Pre-launch Verified Badge gives serious teams a way to demonstrate clean code at launch. The community-staked attestation model — Phase 4 — closes the loop, aligning the incentives of those who vouch for projects with the outcomes those vouchings predict.

How to participate

Verdik is built for the people who use it. Contributing a new signature to the database, adding a security rule for a language, self-hosting the engine on a launchpad's infrastructure, or simply running a scan before committing capital — every action makes the trust layer stronger. The system grows fastest when its users treat it as shared infrastructure rather than as a vendor product.

The underlying claim

This whitepaper argues for a specific position: that the trust gap on Solana is closable, that the closure is technical rather than social, and that privacy and verifiability are compatible properties when designed for jointly from the start. Whether that position holds up depends on what the system catches, what it misses, and how the broader ecosystem chooses to use it.

> // CLOSING PRINCIPLE
>
> Trust, but verify. On-chain. Free, open, evidence-bound. Built for Solana — built to be useful to anyone who reads the code.

Engage with the project

Source code, signature database, security rules, and full documentation are public. The scan engine runs without setup, without accounts, and without paid dependencies. The audit is available the moment you ask for it.

verdik // get started

```
$ curl https://verdik.codes
→ dts forensics      # scan a developer
→ ast scanner         # scan a repository
→ leaderboard         # review recent verdicts
→ documentation      # every weight, every rule
```

```
// don't ape. verify first.
```

Trust, but verify. On-chain.

Project Verdik — Code-level rug protection for Solana. Built privacy-first.

```
$ verdik.engine.status
// live on solana mainnet
▲ 29 rug signatures  ·  24 security rules
▲ 4 languages supported  ·  <12s scan time
▲ 100% free  ·  open source  ·  mit license
// don't ape. verify first.
```

web // verdik.codes
code // github.com/verdikcodes
project // verdik on solana