

HacdsDeFi Whitepaper

2026-10-21 Version 2

Before we start

hacds is an on-chain asset on the Hachash mainnet, serial 2000. Its swap, market making, staking, lending and treasury functions all live in a single HVM contract. The oracle, the indexer and the wallet are written by us too, and run on our own servers.

It runs on Hachash's first DeFi contract. Hachash activated its HVM (a Turing-complete smart contract VM) at block height 765,432, and this contract is the first on mainnet to put all four DeFi primitives — swap and market making, staking, lending, treasury — into a single contract. Our own deployment checklist was written as "the first Turing-complete smart contract on mainnet". The oracle, indexer and wallet are ours as well, running on our servers.

This document has four parts: what went wrong with the previous version, how this version is designed, its four characteristics, and where things stand.

The earlier "Hacds Inscription Whitepaper" is void. We built that design, shipped it, and shut it down.

1. The previous version

1.1 What it was

The previous version defined hacds as an inscription carved onto HACD diamonds. It had four parts.

Issuance by inscription. Each HACD carries one Hacds inscription, and only one per diamond is valid. An ordinary diamond was worth 10,000 hacds; how much a special one was worth (500,000 for a heart, 300,000 for four consecutive letters, 200,000 for a fancy shape) was decided by community vote.

Trading by plugin. The decentralized exchange was built as a plugin wallet called MoneyNex. Users installed the plugin and traded through its wallet.

Listing as the goal. The first listing target written into the whitepaper was xeggex.com.

Discovery by chain scan. Every client scanned the whole chain for inscriptions itself; third-party servers could additionally provide search.

1.2 We built it, and it got listed

None of this was a paper design. The inscription system was built, the MoneyNex plugin ran, and hacds was listed on two exchanges: xeggex.com and nonkyc.io.

Then it was suspended.

1.3 What went wrong

The root cause is the first point. The other three follow from it.

A coin carved onto diamonds cannot move. Holding 10,000 hacds meant holding a specific set of HACD diamonds, and transferring meant moving the diamonds along with them. One diamond carries one inscription, so total supply was pinned to the number of diamonds — indivisible and unsplittable. Something that cannot be split is not a currency, and exchanges cannot run normal deposits, withdrawals or depth on it.

The price had no anchor. Issuance ratios came from a vote, which meant the number of hacds each class of diamond was worth could be changed by voting at any time. Exchanges had no reproducible conversion basis, market makers had no stable quotes, and depth never developed.

Trading depended on other people. The decentralized exchange was a third-party plugin — if the plugin stopped being maintained, the exchange stopped. The centralized exchanges were more direct: trading and circulation sat entirely with xeggex and nonkyc.

There was no fallback. Those two exchanges were the entire circulation channel. When they stopped, there was nowhere else on-chain to trade.

Put together: the coin could not circulate freely, so it could not be priced; with no price it could not be market-made; without market making it depended entirely on exchanges; when the exchanges stopped, everything stopped.

Being listed was never the point. A listing does not remove a design flaw — it just delays it until the day the listing is suspended. That is why this version changed the design.

1.4 What changed

One thing: the coin came off the diamonds.

hacds became a standard on-chain asset, attached to no diamond, freely divisible and transferable. The financial logic moved out of plugins and exchanges and into the contract. Circulation no longer depends on a listing — the on-chain AMM trades on its own, and a listing becomes optional rather than a precondition for survival.

2. How this version is designed

2.1 Where the supply number comes from

The hacds supply was not picked out of the air. It was computed from every diamond on the chain. Each HACD is scored on the five HIP-5 dimensions (shape, style, colour, naming, number). One point converts to one hacds (`SCORE_SCALE = 1e8` in the contract, i.e. 1 point = 1e8 atoms). The total is the exact mean score across all diamonds:

```
mean score × total HACD supply
= 3.55440306 × 16,777,216
```

= 59,632,987 hacds
= 5,963,298,700,000,000 atoms

Anyone can recompute this: pull the explorer API, score every diamond, and you get the same number. This is exactly what the previous version lacked — a basis that does not depend on anyone's decision.

Scoring happens off-chain and is signed; the contract recomputes it on-chain from the certificate before accepting anything. Off-chain saves gas, on-chain keeps it correct, and both sides use the same algorithm, so the results must agree. We verified this against more than 600 real diamonds.

2.2 Asset specification

Item	Value
serial	2000
ticket / name	HACDS / hacds
decimal	8
total supply	5,963,298,700,000,000 atoms
issuer	VKNwXFFMCEn9XfMrhyHhd5rHqBnEyM3m1 (the contract)
issuance cost	8 HAC (protocol fee, varies with block reward)

Note: the ticket of serial #1028 on chain is also HACDS (its name is "HACD Fragments" and it belongs to someone else). The protocol does not check name or ticket uniqueness — it only checks whether the serial is taken. Go by serial 2000.

3. Four characteristics

3.1 Open source

The contract (fitsh source), the oracle, the indexer and the wallet will all be published.

A DeFi protocol runs other people's money, and the only way users can protect themselves is to be able to read the code and check the arithmetic. The previous wallet was a third-party plugin, so users could only choose to trust it or not. This version puts the logic in the open.

Once the code is public, two things become possible:

- **Scoring can be independently recomputed.** Anyone can pull every diamond on the chain, rescore with the same algorithm, and check the supply figure. We did this ourselves on over 600 diamonds.
- **Contract behaviour can be audited.** What the admin can and cannot do is written in the function list — no need to take our word for it.

3.2 Decentralization

This one needs its boundaries stated. Saying "fully decentralized" without qualification is meaningless.

What is already in place:

Rules and execution are on-chain. AMM pricing, the conditions for a loan to pass, the liquidation threshold, the fee split — all of it is contract code. Participants trade under the same rules; we are not in the middle, and cannot insert ourselves.

The contract cannot be upgraded and has no back door. There is no self-destruct function, no upgrade function, and no function that freezes an individual user. What was deployed is what stays.

The admin cannot touch user funds. This is directly verifiable from the function list: among the six functions the admin can call, there is no withdrawal, no balance change, no collateral change, no debt change, and no ability to move any asset. It only changes parameters.

Admin powers have hard caps. Not by promise, but by checks in the code:

Operation	Cap
<code>set_fee</code> — AMM fee rate	≤ 300 bps (3%)
<code>set_loan_rate</code> — lending rate	≤ 2000 bps (20%)
<code>set_liquidate_bonus</code> — liquidation bonus	≤ 1000 bps (10%)

The collateral ratio is not a parameter. `COLL_RATIO` is a contract constant, hard-coded to 2. The admin cannot change it, so the question "could the collateral ratio be quietly lowered" does not exist.

The admin cannot be replaced. It is fixed at deployment by `init_admin`, and there is no `set_admin`. The address cannot be changed.

Participation is permissionless. Swapping, market making, staking and lending are open to anyone — no whitelist, no approval, no need for our consent. The treasury address is a hard-coded constant too.

Assets stay in your hands. `hacds` is a standard on-chain asset held in the user's own address. We do not custody it and cannot freeze it.

What is not yet in place, stated plainly:

The oracle is currently a single point, run by us. It cannot decide what your assets are worth — the contract recomputes the score on-chain — but it does decide whether swapping works at all. If it goes down, swapping stops.

The admin holds `set_paused`, which can pause the contract globally. He cannot stop your assets or take anyone's money, but he can make the protocol temporarily refuse operations.

Those two are the real boundaries of decentralization today. Removing them requires contract changes (multiple oracles, a timelock, multisig), which will go in with the next contract update.

3.3 Security

Two sides: how user funds are held, and how the protocol's rules are guaranteed.

The private key never touches the network. The wallet is split into cold and hot sides. The offline machine builds and signs transactions; the online machine only queries and broadcasts. The private key appears exactly once, in main-process memory, and is injected into the child process as an environment variable — never written to disk. This works because Hachash transactions have no nonce and no expiry, so a transaction signed offline stays valid and can be broadcast much later.

Signing happens locally. Transactions are signed on the local machine by the bundled Rust program, not through any node. The node is used only to translate, verify and broadcast. The node's `/util/transaction/sign` endpoint requires sending your private key over HTTP — we do not use it.

Wallet files are encrypted. The passphrase is stretched with scrypt ($N=2^{15}$, $r=8$, $p=1$); encryption is AES-256-GCM with the format version bound as AAD.

The UI process has no system access. The renderer runs with `contextIsolation` on and `nodeIntegration` off, and can only call the main process through the preload allowlist. Even with malicious content in the renderer, it cannot reach the filesystem or the private key.

The oracle is not blindly trusted. The signed certificate is only a transport credential. On receipt, the contract recomputes the score with its own engine and compares. A compromised oracle still cannot forge a score.

Assets are conserved. The on-chain identity: user holdings plus contract holdings equals total supply. Every operation preserves it, and anyone can check at any time.

Guardrails exist. The per-bucket gas limit means only one diamond can be deposited per transaction, so the wallet hard-blocks this and auto-splits multi-diamond selections; after a batch completes, the reverse button has a cooldown so you do not accidentally deposit again right after redeeming.

3.4 Complete DeFi

The previous version had two financial functions: issue a coin, and get it listed. No primitives — no market making, no collateral, no borrowing, and no yield except price appreciation.

This version fills that in. One contract holds four pieces:

Module	What it does
AMM	Two-way swap between HAC and hacds. Constant product $x \times y = k$, 0.5% fee split LP 50% / staking pool 30% / treasury 20%
Staking	LP shares lock into the staking pool to earn fees, distributed via rpu fixed-point, settled lazily — unclaimed rewards are never lost
Lending	Borrow HAC against hacds collateral. 50% collateral ratio, 5% APR, repayable, liquidated if the ratio falls too low
Treasury	Fees accumulate here and are swept periodically

Two more tools come with it.

Atomic transactions. Multiple operations can go into a single transaction — all succeed or all fail — up to the protocol limit of 200 actions. "Deposit a diamond + claim hacds immediately" can be one transaction.

Liquidation. Lending has a defined liquidation line and bonus. Anyone who sees a position below a 200% collateral ratio can liquidate it; we do not need to be involved. So that people can find liquidatable accounts, we wrote the indexer (see 4.3), because the contract cannot enumerate borrowers.

Interest is computed lazily: no scheduled job, no keeper dependency. Whenever someone interacts with the contract, interest is caught up by block height difference. It saves gas and keeps working even if nobody maintains it.

4. System composition

Four layers: the contract and assets on-chain, the node, oracle and indexer on the server side, the wallet on the client, and the cold/hot split for how it is used.



Rationale: Hacash transactions have **no nonce and no expiry** — a signed tx stays valid indefinitely and can be broadcast later.

⑤ Interface contract (wallet ↔ server)

Endpoint	Purpose	Notes
GET /cert?name=X (:8123)	HACD scoring certificate	126 B; required to deposit/redeem
sandboxcall (:8082)	Read contract state	Per-IP concurrency ≤ 2 ; serialize
GET /watchlist (:8124)	Borrower ledger + health	One snapshot; replaces local scanning
POST /submit/transaction	Broadcast	ret:0 only means it reached the mempool; 300 s blocks

Single source of truth: [wallet-app/设计文档.md §0](#) | Units: HAC in HAC, hacds in raw atoms everywhere (no $\div 1e8$) | VM bucket caps: compute 18,009 / resource 6,100 / storage 111,911

4.1 The contract

The contract is written in fitsh on the Hacash VM. The rules of the four modules were covered above; here are a few implementation trade-offs.

Collateral is the `due` quota — the claim created by depositing diamonds into the contract — not the diamonds themselves. Users do not have to lock up diamonds in order to borrow.

Collateral is valued directly at the AMM's live pool price (`coll_to_hac = coll × amm:hac / amm:ds`). No external price feed is introduced, which removes a whole piece of oracle infrastructure and shrinks the surface for price-feed attacks.

The treasury address is hard-coded in the contract and cannot be changed.

The contract does not expose an endpoint for reading configuration, so the admin UI cannot show the current fee rate or lending rate. This is deliberate: upgrading a live contract just to echo values is not worth it.

There is a hard limit worth knowing up front. Hacash VM gas is split into three independent buckets, unrelated to the total budget `gas_max`:

Bucket	Cap
compute	18,009
resource	6,100
storage	111,911

The resource row is what forces **at most one HACD per deposit transaction**. Verifying a diamond's certificate requires recomputing its score on-chain, which costs roughly 3,000 resource; two diamonds come to 6,101, just over 6,100. This has nothing to do with `gas_max` or the fee — raising them does not help, only splitting the transaction does. This is a consensus parameter; changing it would be a fork.

4.2 The oracle

The contract has to verify what a diamond is worth, but scoring involves SHA3 and a five-dimension evaluation, which is too expensive to run in full on-chain. So: compute off-chain, verify

on-chain.

Call `GET /cert?name=KWTVSH`. The server fetches the diamond's genes, derives `visual_gene`, computes the five-dimension score, and packs it into a 126-byte certificate (29-byte message + 97-byte signature), signed with its own private key. The contract recomputes the score on-chain and compares; the signature exists to prevent forgery.

The oracle runs at 43.251.101.101:8123 and depends only on the Python standard library (`hashlib.sha3_256` is built in). The private key stays server-side; clients only receive certificates.

`cfg:oracle` in the contract recognises exactly one fixed address. A certificate signed by an oracle you run yourself will not be accepted — this is deliberate, otherwise scores could be forged at will.

4.3 The indexer

Here we hit a problem with no way around it: the contract cannot enumerate borrowers. Loan records are stored under keys built from the address (`loan:debt:<address>`); the contract has no list, writes no log, and the node cannot enumerate contract KV by prefix. So the question "who can be liquidated right now" cannot be answered on-chain.

The answer is a ledger built by scanning the chain off-chain: walk blocks, pick out contract calls that hit a lending function selector, take the transaction's main address as the borrower, record `lastScanned` incrementally, and persist the ledger. For a snapshot, query `loan_status` for each address in the ledger.

The wallet needs exactly one endpoint: `GET /watchlist`, which returns the summary and the ledger with health values. The others — `/snapshot`, `/health`, `/sync`, `/watch/add`, `/watch/remove` — are auxiliary.

The indexer has to live server-side. Measured scan throughput is 2.6 blocks/s serially and 6.6 blocks/s at concurrency 4; walking back 500 blocks takes 15 to 75 seconds and over a thousand HTTP requests. A phone cannot afford that in data or battery. With it server-side, the wallet makes one request. Two side benefits: the Python standard library has SHA3 (Electron's Node runtime does not, see the wallet section below), and there is only one ledger, so PC and mobile see the same thing.

4.4 The wallet

The wallet is an Electron app, cold/hot combined: the offline machine builds and signs, the online machine queries and broadcasts.

The UI is plain HTML/CSS/JS with no framework. It includes asset overview, protocol status, a persistent loan status card, live AMM quotes, the HACD gallery, offline signing, atomic transactions, a liquidation console and an admin page. Copy is localised into Chinese, English and Turkish, 479 entries.

The main process handles the IPC allowlist, the wallet keystore, iterative fee estimation, and HTTP to the three services.

Transaction building and signing are delegated to a bundled Rust program, `hacds_txbuild`, with eight modes. We did not rewrite it because the logic is complex (200+ actions, Type-3 wire format,

fee iteration), rewriting is risky, and this code already runs.

A few things in the wallet worth mentioning:

- The lending page shows the borrow limit and the collateral this loan needs, live, under the input boxes; there is also a persistent loan status card on the home screen. Previously, a user who just clicked "borrow" would only get the contract's `return 40` with no explanation.
- Multi-selecting HACDs and batching auto-splits into one diamond per transaction.
- Offline signing: build and sign on the cold side, produce a chunked QR or a file, scan and broadcast on the hot side.
- Atomic transactions: multiple actions in one transaction.

5. Old vs new

	Previous	Now
Token form	Inscription on HACD	Standard on-chain asset (serial 2000)
Supply	Limited by diamond count, one inscription per diamond	Issued once at 59,632,987, freely divisible
Value source	Community vote	Diamond scoring algorithm, recomputable
Trading venue	MoneyNex plugin wallet	On-chain AMM contract
Listing	Listed on xeggex.com and nonkyc.io, later suspended	Trades without a listing; can be listed on the standard asset interface
Financial primitives	None	AMM / staking / lending / treasury
Scoring source	Client-side computation	Oracle-signed certificate + on-chain recomputation
Data discovery	Every client scans the chain for inscriptions	Server-side indexer, one call from the wallet
Wallet	Third-party plugin	Our own Electron console
Private key	Handed to the plugin	Held locally, never on the wire
Source code	Closed	Planned to be fully public
Who can change the rules	Plugin author, exchanges	Only the on-chain contract; the admin can only change capped parameters

6. Status

What is live:

Item	Where
Contract	VKNwXFFMCEn9XfMrhyHhd5rHqBnEyM3m1
hacds asset	serial 2000, supply 5,963,298,700,000,000 atoms
Full node	43.251.101.101:8082
Oracle	43.251.101.101:8123
Indexer	43.251.101.101:8124
Wallet	HacdsDeFi-Setup-0.1.0-x64.exe / HacdsDeFi-0.1.0-x64.exe

There is a conservation identity on chain: user holdings plus contract holdings equals total supply. What users hold is what has been redeemed out of the contract.

Things we have verified:

- Scoring agrees with the oracle on 152 diamonds and with the contract's on-chain engine on 610 diamonds, all matching
- A one-diamond deposit broadcasts successfully
- A two-diamond deposit reports `OutOfGas(31): resource ... used 6101 > limit 6100`, as expected
- A batch deposit of 3 diamonds on a real machine: 3/3 succeeded; transactions built with the same timestamp have distinct hashes
- Packaged self-test reports `RESULT = ALL OK`, with all eight engine modes present
- 479 entries in each of the three languages, with matching key sets

7. Known limitations

Some of these are not omissions but trade-offs we accepted for now. They are written down so the next person does not step on them.

The oracle is a single point. We run it, its address is hard-coded in the contract, and it cannot be swapped. We chose trustworthy scoring over distributed scoring: scores are objective and recomputable, and the contract recomputes them on-chain as a backstop — but the machine has to stay online.

The admin can pause the contract. He cannot move funds, but he can make the protocol temporarily refuse operations. Parameter control is also concentrated with the admin, though with numeric caps.

Deposits are limited to one diamond per transaction. The reason is in the gas section above. Larger amounts have to be split across transactions, and the UI says so.

300-second blocks. A successful broadcast does not mean it is on-chain; `ret:0` only means it reached the mempool. The wallet refills a few times on the block rhythm (15, 45, 90, 180, 320 seconds).

Interest is not shown separately. The contract accrues lazily, and the accrual start point is not exposed, so the currently payable interest cannot be computed. It is only shown as the on-chain settlement result when you repay.

The first two are the boundaries of decentralization; the other three come from the contract already being live. Changing them would mean redeploying the contract and reissuing the asset, which costs far more than it gains, so they stay for now. Writing them down beats hiding them.

8. Next

The Android client is in progress — Kotlin with Compose, reusing the same Rust engine so transaction bytes match. The aarch64 cross-compile is already working.

On decentralization, the next step is removing the first two items from the previous section: multiple oracle instances, and a timelock on key parameters. Both require contract changes and will go in with the next contract update.

In the queue: pre-submit parameter validation, automatic balance refresh after each block, transaction history, batch operations.

Directions we are not pursuing: a locally run oracle (the contract will not accept it), multi-node broadcast, auto-update.

Appendix A: Addresses and parameters

Item	Value
Deployer / admin	15fv6cNEapTNqYhigMP38pC1XRKn3Wh5q
Contract	VKNwXFFMCEn9XfMrhyHhd5rHqBnEyM3m1
Treasury	1FbwjELGMmaomNQq3ZZ6hdgQ17GRGwzAzK
Oracle account	1H3sFZ53HzNVUVN97mjpC9JT8s2PqKMyRu
hacds serial	2000 (decimal 8, ticket HACDS)
SCORE_SCALE	1e8 (1 point = 1 hacds)
COLL_RATIO	2 (50% collateral ratio, liquidation line 200, contract constant, not changeable)
AMM fee	0.5%, LP 50% / staking 30% / treasury 20%
Lending rate / liquidation bonus	5% APR / 2%
Admin parameter caps	fee 3% / rate 20% / liquidation bonus 10%
Certificate length	126 bytes (message 29 + signature 97)
VM bucket caps	compute 18,009 / resource 6,100 / storage 111,911

Item	Value
Block target	300 seconds

Appendix B: Units

hacds is a decimal-8 asset. The integer on chain is in atoms, and it is easy to mix up the unit and the count when writing a UI.

The wallet's approach is simple: hacds is always shown in atoms, with no extra conversion layer. Balances, pools, collateral, input boxes, quotes — all in atoms, matching the explorer and exchange interfaces. Input labels say atoms, and the default value is 1000000000 so the user can see directly how many atoms one hacds is.

Convert to a count yourself by dividing by 1e8. 7000000000 atoms is 70 hacds.

Three rules:

1. On chain, in exchange interfaces, and in the wallet, always use atoms, unchanged. A consumer that needs a count divides by 1e8 itself.
2. Do not fill an input box using the divided-by-1e8 reading. If the user sees 70 and types 70, they have transferred 70 atoms — off by a factor of 1e8. Input labels must say atoms.
3. Total supply 5,963,298,700,000,000 is already 66% above JS `Number.MAX_SAFE_INTEGER`. Use `BigInt` in JS, `BigInteger` in Kotlin; `Long` is not enough.

HAC works differently: the on-chain unit is zhu (1 HAC = 1e8 zhu), and the UI always uses HAC. So the two parameters of `add_liquidity(hac_amt, ds_amt)` have different units — one is zhu, the other atoms. This is the easiest place to make a 1e8 error; we made it ourselves on the first liquidity injection.