JACK: Just-in-time Autonomous Cross-chain Kernel

A Formal Architecture for Intent-Based, Privacy-Aware, and Policy-Enforced DeFi Execution

v1.0.1

Blockchain Foundation LatAm
research@lukas.lat

February 2026

Abstract

Liquidity, execution venues, and state have fragmented across heterogeneous blockchain ecosystems, creating a usability and safety bottleneck for decentralized finance. While bridges, aggregators, and routers enable cross-domain value movement, they do not provide a unified *execution abstraction* with explicit policy enforcement and a rigorous failure model.

This paper introduces JACK, a protocol-level execution kernel that transforms high-level user intents into verifiable, policy-constrained execution plans and settles them on programmable venues (e.g., Uniswap v4 hooks). JACKseparates (i) intent representation, (ii) solver coordination, (iii) private constraint handling, (iv) routing, and (v) settlement adapters. We formalize an execution model in which off-chain solvers compete to satisfy intents while on-chain hooks enforce market policy at settlement time.

Scope note (v1). JACK v1.0.1is a *hackathon-grade* specification aligned with practical implementation constraints: private constraints are handled via a pluggable *Confidential Constraint Module (CCM)* interface (e.g., confidential execution / coprocessors) rather than claiming production-ready FHE+ZK enforcement. Fully homomorphic evaluation and proof-carrying constraints remain a forward-looking research direction.

Contents

1 Versioning, Scope, and Non-Goals

1.1 Change Log

- **v1.0.1** (Feb 2026): Clarifies v1 scope; replaces hard FHE claims with a CCM interface; adds explicit non-goals for cross-chain atomicity; adds a minimal solver economic model; expands threat model and hook security considerations.
- **v1.0.0** (Feb 2026): Initial formal architecture draft.

1.2 What is in-scope for v1

- A concrete intent format with public envelope and private constraints.
- Solver competition and a minimal economic security primitive (bonded execution).
- Routing via existing infrastructure (e.g., LI.FI) with explicit failure handling.
- Settlement on a single destination chain using programmable venues (Uniswap v4 hooks).
- A pluggable privacy interface (CCM) to reduce information leakage about constraints.

1.3 Non-goals for v1

- **Atomic cross-chain settlement** across heterogeneous finality domains.
- **Trustless bridge security** (bridges are treated as external dependencies with allowlists and circuit breakers).
- **Production-ready FHE+ZK** constraint proofs (research track only).
- **Global solver decentralization** guarantees (v1 focuses on correctness and observability).

1.4 v1 vs. vNext

Component	**v1 (this document)**	**vNext (research / roadmap)**
Private constraints	CCM interface; confidential execution / coprocessor possible	FHE+ZK proof-carrying constraints; threshold key mgmt
Cross-chain execution	Best-effort routing + explicit failure states	Stronger atomicity / dispute games / optimistic safety
Solver economics	Minimal bonded execution + fees	Auctions, staking, slashing, reputation, anti-collusion
Venue policy	Uniswap v4 hooks enforce on-chain policy	Formal verification of policies; registries and attestations

2 Introduction

Decentralized finance has evolved from single-chain composability into a multi-domain execution environment. The dominant interaction paradigm remains transaction-centric: users manually select routes, bridges, and execution venues. This model fails to scale across heterogeneous ecosystems and exposes execution strategies to adversarial observation and manipulation [3].

JACKproposes a kernel-level abstraction in which users express execution *intents* rather than transactions. Execution is delegated to autonomous solvers that compete to satisfy the intent under constraints, while final settlement is performed by programmable on-chain execution venues equipped with policy logic (e.g., Uniswap v4 hooks) [4]. The core design goal is to make policy-enforced settlement explicit and verifiable, while enabling privacy-aware constraint handling via a modular interface.

3 Notation and Preliminaries

We denote by $\mathbb{B} = \{0, 1\}$ the Boolean domain. For a probabilistic polynomial-time algorithm A, we write $y \leftarrow A(x)$ to denote randomized execution. Let λ denote the security parameter.

We denote a public-key encryption scheme by $\mathsf{PKE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. For fully homomorphic encryption, we denote $\mathsf{FHE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$. For a statement s and witness w, we denote a zero-knowledge proof system by $\mathsf{ZK} = (\mathsf{Prove}, \mathsf{Verify})$.

All cryptographic primitives are assumed to be secure against probabilistic polynomial-time adversaries.

4 System Architecture

JACKis decomposed into five orthogonal layers:

1. **Intent Layer**
2. **Solver and Coordination Layer**
3. **Privacy / Constraint Layer (CCM)**
4. **Execution Routing Layer**
5. **Settlement Adapter Layer**

4.1 Kernel Model

The JACKkernel is defined as:

$$\mathcal{K} = \langle \mathcal{I}, \mathcal{S}, \mathcal{C}, \mathcal{R}, \mathcal{V} \rangle$$

where $\mathcal{I}$ denotes intent representation, $\mathcal{S}$ solvers, $\mathcal{C}$ constraint/privacy enforcement, $\mathcal{R}$ routing, and $\mathcal{V}$ settlement venues.

Each layer operates independently but exposes standardized interfaces. In v1, the privacy/constraint layer is explicitly modeled as a *pluggable module*.

5 Intent Model

5.1 Formal Intent Definition

An intent is:

$$I = \langle U, A, T, \Phi, \Omega \rangle$$

where U is user identifier, A target asset(s), T destination execution environment, Φ private constraint payload, and Ω public execution envelope.

5.2 Public and Private Components

We split:

- I_{pub}: routing compatibility and execution metadata
- I_{priv}: constraints and preferences

$$I = (I_{pub}, \mathsf{Enc}(I_{priv}))$$

In v1, *encryption is optional* depending on CCM implementation; at minimum, the design requires that sensitive constraints need not be publicly broadcast in cleartext.

5.3 Constraint Vector

The constraint vector may contain:

- maximum slippage bounds,
- execution deadlines,
- minimum output guarantees,
- venue policy requirements,
- routing allowlists / deny lists,
- optional compliance / disclosure flags (future work).

6 Solver-Based Execution

6.1 Solver Role

Solvers produce candidate execution plans:

$$\pi = \langle r_1, r_2, \ldots, r_n, v \rangle$$

where r_i are routing/bridging steps and v is the settlement venue.

6.2 Competition Model

The kernel verifies:

1. compatibility with the public envelope,
2. satisfaction of constraints via CCM evidence,
3. verifiability of final settlement on v.

6.3 Minimal Economic Security (v1)

To reduce griefing and align incentives, v1 specifies a minimal bonded execution model:

- Users specify a **max fee** and **deadline** inside I_{pub} and sign the intent.
- Solvers register execution by posting a small **bond** b (testnet ETH in v1).

- If the solver fails to reach settlement before deadline (or submits invalid evidence), the bond can be **slashed** (v1: retained by the protocol; vNext: redistributed to user / challengers).
- Winning solver receives the fee upon successful settlement (v1: paid by user off-chain or via a fee vault).

This model is intentionally minimal and is designed to be replaced by auctions, staking, and slashing in vNext.

7 Privacy / Constraint Layer: CCM

7.1 Design Objective

The constraint layer aims to reduce information leakage (e.g., slippage bounds, intent size, routing preferences) that can be exploited for MEV or censorship, while keeping settlement verifiable.

7.2 CCM Interface (v1)

We define a Confidential Constraint Module (CCM) interface that produces evidence of constraint satisfaction without requiring public disclosure of raw constraints:

$$\mathsf{CCM.Verify}(I_{pub}, \Phi, x) \to \mathbb{B}$$

where x are solver execution parameters (route, expected outputs, timing).

A CCM may be implemented by:

- **Confidential execution / coprocessor** producing signed attestations (v1 implementable path).
- **FHE evaluation** of constraint circuits (research track) [1, 2].
- **ZK proofs** over committed constraints (research track).

7.3 FHE+ZK as Research Track

For completeness, a future proof-carrying variant can model constraint evaluation as:

$$\mathsf{Dec}(\mathsf{Eval}(\mathsf{Enc}(c), x)) = 1$$

and produce a proof $\Pi_{priv} \leftarrow \mathsf{Prove}(\mathsf{Enc}(c), x)$ that the kernel can verify. This document does **not** claim that such a system is production-ready in v1.

8 Cross-Chain Routing Layer

8.1 Routing Abstraction

JACKdefines a routing graph:

$$G = (V_{chains}, E_{bridges})$$

Edges encode cost/latency/risk attributes. In v1, routing is delegated to external routing infrastructure (e.g., LI.FI) and is constrained by allowlists and safety policies.

8.2 Failure Handling (v1)

Cross-domain execution is treated as *best-effort* and modeled explicitly as a state machine:

$$\texttt{CREATED} \to \texttt{QUOTED} \to \texttt{EXECUTING} \to (\texttt{SETTLED} \mid \texttt{ABORTED} \mid \texttt{EXPIRED})$$

If a bridge or step fails, the execution transitions to `ABORTED` with a reason code; partial completion is not treated as atomic and must be handled by application-level recovery procedures (future work: dispute games / insurance / rollback primitives).

8.3 Safety Controls (v1)

- Bridge/route allowlists.
- Value caps per execution.
- Circuit breaker (pause new executions).
- Timeouts aligned to destination-chain finality.

9 Settlement Adapter Layer

9.1 Venue Interface

Each settlement venue v implements:

$$Execute(v, \pi) \to tx \qquad \text{and} \qquad Verify(v, tx) \to \mathbb{B}$$

9.2 Programmable Policy Venues

Uniswap v4 pools equipped with hooks act as policy-enforced settlement venues [4]. Hooks are the on-chain controller for settlement-time policy.

10 Policy-Enforced Market Execution (Uniswap v4 Hooks)

10.1 Hook as Policy Agent

Let P denote a policy function:

$$P(s_{pool}, s_{ref}, \theta) \to \{allow, reject, modify\}$$

Hooks can enforce constraints such as max slippage, min-out guarantees, allowlisted assets, and oracle-based deviation checks.

10.2 Hook Security Considerations (v1)

Hooks increase attack surface. v1 requires:

- Strict access control: only PoolManager may call hook entrypoints.
- No unbounded loops; bounded gas usage.
- Minimal external calls (prefer none) during hook execution.
- Correct delta accounting and reentrancy-safe design.

In vNext, we propose an audited registry of approved hooks and optional on-chain attestations.

11 Execution Algorithm

Algorithm 1 JACKKernel Execution (v1)

```
User constructs and signs intent $I = (I_{pub}, \Phi)$
Kernel publishes $I_{pub}$ and stores $\Phi$ (optionally encrypted)
Solvers generate candidate plans $\pi$ and parameters $x$
for all solver submissions do
    Verify public compatibility with $I_{pub}$
    Verify CCM evidence: $\mathsf{CCM.Verify}(I_{pub}, \Phi, x) = 1$
end for
Select winning solver $\pi^\star$ (by fee / time / policy)
Execute routing steps (best-effort) with explicit failure handling
Submit settlement to venue $v$ (e.g., Uniswap v4 pool)
Enforce policy via hook logic
Verify settlement and emit public events
```

12 Verification and Observability

12.1 Execution Correctness

An execution is valid if:

$$\mathsf{CCM.Verify}(\cdot) = 1 \ \wedge \ \mathsf{Verify}(v, tx) = 1$$

12.2 Public Verifiability

Observers can verify:

- settlement correctness and venue trace,
- policy hook decision path (events / revert codes),
- execution state transitions.

Private constraints remain opaque to the extent provided by the CCM.

13 Adversarial Model (Expanded)

We consider:

- **Malicious solvers**: invalid routes, censorship, griefing, collusion, Sybil attacks.
- **Adversarial observers**: MEV extraction, timing inference, metadata leakage.
- **Routing/bridge failures**: exploits, message verification bugs, compromised keys.
- **Malicious venues/hooks**: access control failures, reentrancy, DoS via gas exhaustion.
- **Oracle manipulation**: reference price distortion affecting policy checks.

We assume cryptographic hardness where used; in v1, trust assumptions depend on CCM implementation (e.g., coprocessor attestations).

14 Security Properties (v1)

1. **Policy Enforceability:** settlement cannot bypass on-chain hook policy.
2. **Execution Integrity:** kernel binds settlement to a signed intent and verified evidence.
3. **Fail-Closed Behavior:** executions abort on missing evidence or policy violations.
4. **Risk Containment:** allowlists, caps, and circuit breakers limit blast radius.
5. **Privacy-Awareness:** sensitive constraints need not be publicly broadcast in cleartext.

15 PCPE: Policy-Constrained Private Execution

We define *Policy-Constrained Private Execution (PCPE)* as an execution primitive that provides:

1. private constraints (via CCM),
2. public settlement verifiability,
3. programmable policy rejection or modification at settlement,
4. bounded failure handling semantics.

16 Implementation Notes (v1)

- Frontend: TypeScript, React / Next.js, wallet connection.
- Kernel coordination: off-chain services (intent store, solver coordination).
- Smart contracts: Solidity settlement adapters + Uniswap v4 hooks.
- Privacy/constraints: CCM interface; prototype may use confidential execution attestations.
- Routing: LI.FI or equivalent aggregation SDK; explicit allowlists and caps.

17 Evaluation Plan (v1)

We measure:

- end-to-end latency (intent → settlement),
- policy hook gas overhead and revert rate,
- solver throughput and failure modes,
- CCM verification latency (attestation/proof verification),
- routing success rate under degraded conditions.

18 Limitations and Future Work

- Production-grade economic security (auctions, staking, slashing).
- Decentralized solver sets and anti-collusion mechanisms.
- Formal verification and auditing frameworks for hook policies.
- Stronger cross-domain atomicity / dispute resolution / insurance primitives.
- Proof-carrying private constraints (FHE+ZK) with practical performance.

19 Conclusion

JACKpositions execution itself as a programmable primitive: users submit intents, solvers compete to execute them, and settlement venues enforce policy via hooks. JACK v1.0.1explicitly scopes v1 to practical, verifiable components while preserving a research path toward stronger private constraint proofs and cross-domain safety guarantees.

References

[1] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, 2009.

[2] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. *Journal of Cryptology*, 33(1), 2020.

[3] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Ittay Eyal, and Emin Gün Sirer. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. In *IEEE Symposium on Security and Privacy*, 2020.

[4] Uniswap Labs. Uniswap v4 Core Architecture. https://github.com/Uniswap/v4-core, 2024.

[5] Flashbots. MEV-Boost: Scaling Blockspace by Separating Proposers and Builders. https://docs.flashbots.net/flashbots-mev-boost/, 2022.

[6] CoW Protocol. CoW Protocol: Batch Auctions and Solver Competition (Docs). https://docs.cow.fi/, 2024.

[7] Uniswap Labs. UniswapX (Docs). https://docs.uniswap.org/uniswapx/overview, 2024.