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

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

Blockchain Foundation LatAm
JACK Research Group
research@lukas.lat

2026

Abstract

The rapid fragmentation of liquidity, execution venues, and state across heterogeneous blockchain ecosystems has produced an execution-layer bottleneck for decentralized finance. While bridges, aggregators, and routers enable cross-chain value movement, they do not offer a unified, programmable execution abstraction nor a policy-enforced settlement layer.

This paper introduces JACK (Just-in-time Autonomous Cross-chain Kernel), a protocol-level execution kernel that transforms high-level user intents into verifiable, privacy-preserving, and policy-constrained cross-chain execution plans. JACK decouples intent expression, solver-based execution, cryptographic constraint enforcement, and venue-specific settlement adapters. We formalize an execution model in which off-chain autonomous agents coordinate cross-chain execution under encrypted constraints and on-chain programmable market policies, enabling Uniswap v4 hooks and similar venues to act as autonomous execution controllers.

We present a formal intent language, solver competition model, encrypted constraint evaluation layer, and a cryptographically verifiable execution pipeline. We further describe a new DeFi execution algorithm that combines private constraint evaluation with public settlement validation, enabling programmable market policy enforcement without revealing execution strategies prior to settlement.

Contents

1 Introduction

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

JACK proposes 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 cryptographically enforced constraints. Final settlement is performed by programmable on-chain execution venues equipped with policy logic (e.g., Uniswap v4 hooks) [4].

JACK is designed as infrastructure, not as an application or market. It provides a general execution substrate upon which specialized financial primitives—such as regional currencies, treasury automation, or market making agents—can be built.

2 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.

3 System Architecture

JACK is decomposed into five orthogonal layers:

1. **Intent Layer**
2. **Solver and Coordination Layer**
3. **Privacy and Constraint Enforcement Layer**
4. **Execution Routing Layer**
5. **Settlement Adapter Layer**

3.1 Kernel Model

The JACK kernel is formally defined as the tuple:

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

where:

- $\mathcal{I}$ denotes the intent representation system,
- $\mathcal{S}$ denotes the solver set,
- $\mathcal{C}$ denotes cryptographic constraint enforcement mechanisms,
- $\mathcal{R}$ denotes cross-chain routing primitives,
- $\mathcal{V}$ denotes settlement venues.

Each layer operates independently but exposes standardized interfaces to the kernel.

4 Intent Model

4.1 Formal Intent Definition

An intent is defined as:

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

where:

- U is the user identifier,
- A is the target asset or asset vector,
- T is the destination execution environment,
- Φ is the encrypted constraint vector,
- Ω is the public execution envelope.

4.2 Public and Private Components

The intent is split into:

- a public descriptor I_{pub} containing routing compatibility information,
- a private descriptor I_{priv} containing execution bounds and preferences.

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

This separation allows solvers to construct execution plans without access to sensitive strategy parameters.

4.3 Constraint Vector

The private constraint vector contains:

- maximum slippage bounds,
- execution deadlines,
- minimum output guarantees,
- market policy restrictions,
- execution venue requirements.

5 Solver-Based Execution

5.1 Solver Role

Solvers act as autonomous agents which attempt to satisfy intents. A solver produces a candidate execution plan:

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

where each r_i is a routing or bridging primitive and v is a settlement venue.

5.2 Competition Model

Solvers compete by submitting commitments to execution plans. The kernel verifies:

1. compatibility with public intent envelope,
2. cryptographic satisfaction of encrypted constraints,
3. verifiability of final settlement.

6 Privacy and Constraint Enforcement

6.1 Encrypted Constraint Evaluation

JACK employs fully homomorphic evaluation over encrypted constraint vectors [1, 2].

Let c denote a private constraint and x a solver-generated execution parameter. Solvers must prove correctness of an encrypted evaluation such that:

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

without revealing c.

The evaluation function is executed inside a privacy execution environment compatible with encrypted computation.

6.2 Constraint Proof Object

A solver produces a proof:

$$\Pi_{priv} \leftarrow \mathsf{Prove}(\mathsf{Enc}(c), x)$$

which can be verified by the kernel without decrypting c.

6.3 FHE-Based Enforcement Layer

The kernel defines a constraint circuit F such that:

$$F(c, x) \to \mathbb{B}$$

The solver publishes an encrypted evaluation:

$$\mathsf{Eval}(\mathsf{Enc}(F), \mathsf{Enc}(c), x)$$

and a validity witness.

7 Cross-Chain Routing Layer

7.1 Routing Abstraction

JACK defines a routing graph:

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

Each edge contains:

- execution cost,
- settlement latency,

- risk weight.

Routing is performed under encrypted cost preferences.

7.2 Multi-Hop Cross-Domain Execution

Execution plans may traverse heterogeneous environments:

$$Chain_i \to Bridge_j \to Chain_k$$

without exposing path selection strategy to observers.

8 Settlement Adapter Layer

8.1 Venue Interface

Each settlement venue v implements:

$$Execute(v, \pi) \to tx$$

and exposes:

$$Verify(v, tx) \to \mathbb{B}$$

8.2 Programmable Policy Venues

Venues may embed on-chain programmable logic that enforces market and policy constraints during execution.

In JACK, Uniswap v4 pools equipped with hooks act as policy-enforced settlement venues [4].

9 Policy-Enforced Market Execution

9.1 Hook as Policy Agent

Let P denote a market policy function:

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

where:

- s_{pool} is current pool state,
- s_{market} is reference state,
- θ is policy configuration.

Hooks are invoked during execution and operate as autonomous agents enforcing policy decisions.

9.2 Dual-Agent Architecture

JACK explicitly separates:

- off-chain autonomous solvers,
- on-chain autonomous policy agents.

This creates a two-layer agentic execution system.

10 Execution Algorithm

Algorithm 1 JACK Kernel Execution

```
User submits intent I
Kernel publishes I_pub and stores Enc(I_priv)
Solvers generate candidate plans π
for all solver submissions do
    Verify public compatibility
    Verify encrypted constraint proof Π_priv
end for
Select winning solver π*
Execute routing steps
Submit settlement to venue v
Enforce policy via venue logic
Verify settlement
```

11 Cryptographic Verification Pipeline

11.1 Execution Correctness

An execution is valid if and only if:

$$\mathsf{Verify}(\Pi_{priv}) = 1 \ \wedge \ \mathsf{Verify}(v, tx) = 1$$

11.2 Public Verifiability

Observers can independently verify:

- settlement correctness,
- policy execution,
- venue execution trace.

They cannot recover private intent parameters.

12 Adversarial Model

We consider:

- malicious solvers,
- adversarial observers,
- malicious routing infrastructure,
- partially malicious settlement venues.

We assume cryptographic hardness of FHE schemes and correctness of venue execution environments.

13 Security Properties

1. **Intent Privacy:** execution constraints are hidden prior to settlement.
2. **Solver Non-Censorship:** multiple solvers compete.
3. **Policy Enforceability:** settlement cannot bypass on-chain policy logic.
4. **Execution Integrity:** cryptographic verification binds execution to intent.
5. **Venue Agnosticism:** kernel does not depend on specific market implementations.

14 New DeFi Primitive: Policy-Constrained Private Execution

We define a new primitive: *Policy-Constrained Private Execution (PCPE).*

A PCPE system satisfies:

1. private execution strategy,
2. public settlement verifiability,
3. programmable execution rejection or modification,
4. cryptographically enforced constraint satisfaction.

This primitive generalizes market execution beyond swaps and enables policy-aware financial automation.

15 Implementation Notes

- Frontend: TypeScript, React, intent encoding
- Kernel coordination: off-chain services
- Smart contracts: Solidity settlement adapters
- Venue policies: Uniswap v4 hooks
- Encrypted constraint layer: FHE-compatible runtime (prototype may use confidential execution to approximate encrypted constraint handling)
- Routing: multi-chain aggregation SDKs

16 Evaluation and Benchmarks

We measure:

- constraint evaluation latency,
- solver competition throughput,
- settlement overhead,
- policy execution gas cost.

Preliminary experiments show that encrypted constraint evaluation dominates off-chain cost, while on-chain policy enforcement adds bounded overhead relative to standard execution.

17 Limitations and Future Work

- scalability of FHE constraint circuits,
- decentralized solver reputation systems,
- formal verification of venue policies,
- cross-venue composability of policy logic,
- on-chain dispute resolution mechanisms.

18 Conclusion

JACK introduces a kernel abstraction for decentralized execution across heterogeneous environments. By combining encrypted intent constraints, solver-based execution, and programmable settlement venues, JACK enables a new class of autonomous, privacy-preserving and policy-aware DeFi systems. The architecture elevates execution itself into a programmable primitive and positions market venues as enforceable execution controllers rather than passive liquidity providers.

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.