Title: Decentralized Crypto Payment Gateway with Virtual Card Integration

Author: Muhammad Hassan

Date: July 24, 2025

Abstract

This whitepaper introduces a next-generation decentralized payment gateway designed for the Web3 ecosystem, built on the Solana blockchain. The system enables seamless, secure crypto transactions using a virtual card interface without relying on traditional financial institutions. The payment mechanism ensures user privacy, complete control, and scalability for e-commerce and application developers.

Overview

Muhammad Hassan presents an innovative payment model that replaces fiat-based card systems with crypto-native virtual cards. Every wallet created through this gateway automatically generates a virtual card (with Card Number, CVC, and Expiry Code) directly tied to the user's wallet.

Key Features

- Built on Solana: Ensures low fees and high-speed transactions
- Virtual Card Generation: Auto-generated on wallet creation
- Socket-Based Server Logic: For real-time payment flows
- First-Time Manual Approval: All first-time payments are user-approved
- Auto-Approval Option: Users can toggle auto/manual approval per merchant
- Fraud Protection: Card data is stored only on user-side (local memory), not on the central server
- One Wallet = One Card: No multi-card clutter, full crypto-native freedom
- Regeneration: Users can regenerate Expiry and CVC anytime if compromised
- Contract-Centric Logic: All approval and validation checks are encoded in smart contracts

Technical Architecture

- Blockchain Layer: Solana SPL Token used for all transactions.

- Smart Contract: Handles card validation, address checking, approval flow, and expiry regeneration.

- Socket Server: Node.js-based real-time connection between client site and backend wallet/contract.

- Security Layer: Heavy encryption, no server-side card storage. All matching is hash-based and session-bound.

- Device/Identity Check: Device-specific parameters used to block reused stolen data.

User Experience

- Wallet creation triggers automatic card creation.

- On first transaction to a new merchant, manual approval is required.

- Approval data (hashed) is saved on the merchant's socket server.

- If user enables auto-approval, future transactions from the same device/card are instant.

- User can always revoke or limit permissions through smart contract functions.

Revenue Model

1. Token Value Appreciation: A native utility token with a total supply of 10 billion will serve as the backbone of the payment system. As adoption grows, the tokens demand and trade value increase, offering long-term revenue potential to early holders and the project treasury.

2. Socket Integration Fees: Merchants and platforms integrating the socket will be charged a small percentage-based fee per transaction. This fee will be automatically routed via smart contracts.

3. Premium Merchant Access: Optionally, high-volume merchants can subscribe to premium plans for higher API limits, faster support, and analytics dashboards.

4. Token-Based Access: Some platform features or integrations may require staking or holding a specific number of native tokens.

How This Project Is Unique

This project stands apart from other crypto card providers and payment platforms through its deep

decentralization, unique architecture, and direct Web3-native approach:

- OKX, Bybit, Binance Cards: These rely on centralized exchanges and third-party partners like Mastercard/Visa. Our system is 100% decentralized with smart contract-based card logic.
- Stripe/PayPal: Fiat-based platforms that do not offer crypto-native cards or decentralized approvals. Our socket-based model is crypto-to-crypto only.
- Crypto Wallet Apps: Traditional wallets like MetaMask or Phantom offer payments but not virtual card structures. This system integrates both a wallet and a card in one flow.
- Local Banks: These depend on KYC, fiat bridging, and limited crypto support. Our model operates outside of banking rails, but still encourages fiat-to-crypto entry via exchanges.
- Security & Control: Only our system allows dynamic card regeneration, device verification, and manual/auto toggle no one else offers this level of flexibility.

Legal & Copyright Protection

- This concept and structure are original and legally protected under international copyright law.
- Dated documentation and publication history will act as intellectual property proof.
- Any resemblance or partial replication of this idea by other parties will be recognized as infringement.
- A future trademark, patent filing, and IP licensing model is under consideration.

Traditional Banking Impact

While this system challenges the way fiat payments are handled, it does not replace local banks. Instead, it works alongside them:

- Users can still purchase crypto using fiat via exchanges connected to banks.
- Our focus is on simplifying crypto use post-purchase, especially for e-commerce.
- The aim is not to remove banks but to decentralize crypto utility for the masses.

Use Cases

- E-Commerce Platforms (pure crypto)

- Subscription Services

- Mobile Apps

- NFT Marketplaces

Future Scope

- Integration with physical cards (optional)

- Multi-chain expansion (Ethereum, Polygon)

- Merchant dashboard for analytics and management

Copyright Notice

End of Whitepaper

Appendix: Smart Contract (Demo)

The following Anchor-based Solana smart contract showcases the decentralized logic behind wallet creation, virtual card generation, and

```
use anchor_lang::prelude::*;
use anchor_lang::solana_program::pubkey::Pubkey;

declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkgBfv3ho9SMz");

#[program]
pub mod crypto_card_gateway {
    use super::*;

    pub fn initialize_wallet(ctx: Context<InitializeWallet>) -> Result<()> {
        let user_wallet = &mut ctx.accounts.user_wallet;
        user_wallet.owner = *ctx.accounts.owner.key;
        user_wallet.card_number = random_u64();
        user_wallet.cvc = random_u8();
        user_wallet.expiry = get_expiry_timestamp();
        user_wallet.approval_required = true;
        Ok(())
    }

    pub fn toggle_approval(ctx: Context<UpdateWallet>, approve: bool) -> Result<()> {
        let user_wallet = &mut ctx.accounts.user_wallet;
        require!(user_wallet.owner == *ctx.accounts.owner.key, CustomError::Unauthorized);
        user_wallet.approval_required = approve;
        Ok(())
    }

    pub fn regenerate_card(ctx: Context<UpdateWallet>) -> Result<()> {
        let user_wallet = &mut ctx.accounts.user_wallet;
        require!(user_wallet.owner == *ctx.accounts.owner.key, CustomError::Unauthorized);
        user_wallet.cvc = random_u8();
        user_wallet.expiry = get_expiry_timestamp();
        Ok(())
    }

    pub fn authorize_payment(ctx: Context<AuthorizePayment>, device_hash: u64) -> Result<()> {
        let user_wallet = &ctx.accounts.user_wallet;
        let merchant = &ctx.accounts.merchant;

        require!(user_wallet.device_hash == device_hash, CustomError::DeviceMismatch);

        if user_wallet.approval_required {
            require!(merchant.approved == true, CustomError::ApprovalRequired);
        }

        Ok(())
    }
}

#[derive(Accounts)]
pub struct InitializeWallet<'info> {
    #[account(init, payer = owner, space = 8 + 64)]
    pub user_wallet: Account<'info, Wallet>,
    #[account(mut)]
    pub owner: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct UpdateWallet<'info> {
    #[account(mut)]
    pub user_wallet: Account<'info, Wallet>,
    pub owner: Signer<'info>,
}

#[derive(Accounts)]
pub struct AuthorizePayment<'info> {
    pub user_wallet: Account<'info, Wallet>,
    pub merchant: Account<'info, Merchant>,
}

#[account]
pub struct Wallet {
    pub owner: Pubkey,
    pub card_number: u64,
    pub cvc: u8,
    pub expiry: i64,
    pub approval_required: bool,
    pub device_hash: u64,
}
```

```
#[account]
pub struct Merchant {
    pub approved: bool,
}

#[error_code]
pub enum CustomError {
    #[msg("Unauthorized access.")]
    Unauthorized,
    #[msg("Approval required for this transaction.")]
    ApprovalRequired,
    #[msg("Device mismatch detected.")]
    DeviceMismatch,
}

// Dummy generators for demo (you'd replace these with real secure randomness in production)
fn random_u64() -> u64 {
    1234567890123456
}

fn random_u8() -> u8 {
    123
}

fn get_expiry_timestamp() -> i64 {
    Clock::get().unwrap().unix_timestamp + 31536000  // 1 year from now
}
```