Decentralized Real Estate Marketplace

Project Technical Report

A Comprehensive Analysis of System Architecture, Optimization, and Security

April 2025

Ethereum Smart Contract Development

1 Project Overview

1.1 Objectives

The primary objectives of the decentralized real estate marketplace are:

- Enable property owners to list real estate assets with detailed information
- Facilitate secure property purchases using cryptocurrency
- Automate ownership transfers through smart contracts
- Maintain an immutable record of property transactions
- Minimize transaction costs through gas optimizations
- Provide a user-friendly interface for both buyers and sellers

1.2 System Architecture

The decentralized real estate marketplace follows a three-tier architecture:

1. **Smart Contract Layer**: Ethereum-based contract written in Solidity that handles property listings, ownership transfers, and payment processing.
2. **Web3 Integration Layer**: JavaScript middleware that connects the frontend with the blockchain through Web3.js.
3. **Frontend Interface**: React-based user interface that allows users to interact with the smart contract.

1.3 Technology Stack

- **Blockchain**: Ethereum
- **Smart Contract**: Solidity 0.8.19
- **Testing Framework**: Truffle with Ganache
- **Frontend**: React.js
- **Blockchain Interaction**: Web3.js
- **Security**: OpenZeppelin contracts
- **Development Environment**: Truffle Suite

2 Smart Contract Implementation

2.1 Core Data Structures

The smart contract utilizes the following data structures to organize property information and ownership:

```
struct Property {
    string name;           // Property name
    string location;       // Property location
    string description;    // Property description
    string imageUrl;       // IPFS hash or URL of the property image
    uint256 price;        // Price in wei
    uint256 size;         // Size in square feet
    uint8 bedrooms;       // Number of bedrooms
    uint8 bathrooms;      // Number of bathrooms
    address owner;        // Current owner
    bool isForSale;       // Whether the property is for sale
}
```

Listing 1: Property Struct

Key storage variables include:

```
// Array to store all properties
Property[] public properties;

// Mapping from property ID to owner
mapping(uint256 => address) public propertyOwner;
```

Listing 2: Storage Variables

2.2 Core Functions

2.2.1 Property Listing

The `listProperty` function allows property owners to list their real estate with detailed information. Key aspects include:

```
function listProperty(
    string memory _name,
    string memory _location,
    string memory _description,
    string memory _imageUrl,
    uint256 _price,
    uint256 _size,
    uint8 _bedrooms,
    uint8 _bathrooms
) public {...}
```

Listing 3: Property Listing Function Signature

Key implementation aspects:

- Input validation (non-empty fields, positive values)
- Creation of new Property struct
- Storage in properties array

- Ownership assignment in propertyOwner mapping
- Event emission for frontend tracking

2.2.2 Property Purchase

The `buyProperty` function enables buyers to purchase properties with cryptocurrency:

```
function buyProperty(uint256 _propertyId) public payable nonReentrant {
    // Key validation checks
    require(_propertyId < properties.length, "Property does not exist");
    require(property.isForSale, "Property is not for sale");
    require(property.owner != msg.sender, "Cannot buy your own property");
    require(msg.value >= property.price, "Not enough ETH sent");

    // Ownership transfer logic
    // ...

    // ETH transfer with security checks
    (bool sent, ) = payable(oldOwner).call{value: price}("");
    require(sent, "Failed to send ETH to seller");

    // Event emission
    // ...
}
```

Listing 4: Property Purchase Function

2.2.3 View Functions

The contract includes several view functions to retrieve property information:

```
// Get total number of properties
function getPropertyCount() public view returns (uint256);

// Get a specific property by ID
function getProperty(uint256 _propertyId) public view returns (
    string memory name,
    string memory location,
    string memory description,
    string memory imageUrl,
    uint256 price,
    uint256 size,
    uint8 bedrooms,
    uint8 bathrooms,
    address owner,
    bool isForSale
);

// Check if sender is the owner of a property
function isOwner(uint256 _propertyId) public view returns (bool);
```

Listing 5: View Function Signatures

2.2.4 Property Management Functions

The contract allows property owners to update their listings:

```
// Update property price
function updatePropertyPrice(uint256 _propertyId, uint256 _newPrice)
    public;

// Toggle property sale status
function toggleForSale(uint256 _propertyId) public;

// Update property image
function updateImageUrl(uint256 _propertyId, string memory _newImageUrl
    ) public;

// Update property details
function updatePropertyDetails(
    uint256 _propertyId,
    uint256 _price,
    string memory _description,
    string memory _imageUrl
) public;
```

Listing 6: Management Function Signatures

Common security patterns in management functions:

```
require(_propertyId < properties.length, "Property does not exist");
require(propertyOwner[_propertyId] == msg.sender, "Only owner can
    update");
```

Listing 7: Ownership Verification Pattern

2.3 Events

The contract emits events to track property listings, sales, and updates:

```
// Events
event PropertyListed(
    uint256 indexed propertyId,
    string name,
    string location,
    uint256 price,
    address owner
);

event PropertySold(
    uint256 indexed propertyId,
    address oldOwner,
    address newOwner,
    uint256 price
);

event PropertyUpdated(
    uint256 indexed propertyId,
    uint256 newPrice,
    bool isForSale
);
```

Listing 8: Contract Events

3 Security Implementations

3.1 Reentrancy Protection

The contract inherits from OpenZeppelin's `ReentrancyGuard` to prevent reentrancy attacks during property purchases:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract RealEstateMarketplace is ReentrancyGuard {
    // Contract code...

    function buyProperty(uint256 _propertyId) public payable
    nonReentrant {
        // Function implementation...
    }
}
```

Listing 9: Reentrancy Protection

The `nonReentrant` modifier prevents malicious contracts from re-entering the function before it completes execution, protecting against recursive calls that could manipulate contract state or drain funds.

3.2 Ownership Verification

The contract implements strict ownership verification for property management functions:

```
require(propertyOwner[_propertyId] == msg.sender, "Only owner can
    update price");
```

Listing 10: Ownership Verification

This pattern is used across all property management functions to ensure that only legitimate property owners can modify their listings.

3.3 Secure ETH Transfer

The contract uses the recommended pattern for ETH transfers to prevent potential vulnerabilities:

```
// Transfer exact price to the seller
(bool sent, ) = payable(oldOwner).call{value: price}("");
require(sent, "Failed to send ETH to seller");
```

Listing 11: Secure ETH Transfer

This approach:

- Uses `call` instead of deprecated `transfer` or `send`
- Checks for successful transfer
- Is protected by the nonReentrant modifier

3.4 Input Validation

The contract implements extensive input validation to prevent invalid property listings:

```
require(bytes(_name).length > 0, "Name cannot be empty");
require(_price > 0, "Price must be greater than zero");
require(_propertyId < properties.length, "Property does not exist");
```

Listing 12: Input Validation Examples

4 Gas Optimization Techniques

4.1 Data Type Selection

The contract uses appropriate data types to minimize gas consumption:

- `uint8` for small values (bedrooms, bathrooms)
- `uint256` for monetary values and larger numbers
- `bool` for binary states (isForSale)

4.2 Storage vs. Memory Usage

The contract optimizes gas usage by carefully managing storage and memory:

```
// Use storage for state-modifying operations
Property storage property = properties[_propertyId];

// Use memory for read-only operations
Property memory newProperty = Property({
    // Property initialization...
});
```

Listing 13: Storage vs. Memory Usage

4.3 Minimizing On-Chain Data

The contract stores large media files off-chain and only maintains references:

```
string imageUrl;  // IPFS hash or URL of the property image
```

Listing 14: Off-Chain Storage

4.4 Event Usage

The contract emits events for off-chain tracking instead of storing historical data on-chain:

```
emit PropertySold(_propertyId, oldOwner, msg.sender, price);
```

Listing 15: Event Usage

5 Testing Framework

5.1 Test Coverage

The contract testing suite covers the following scenarios:

1. Property listing functionality
2. Ownership verification
3. Property purchase with correct ETH amount
4. Rejection of insufficient ETH payments
5. Rejection of non-existent property purchases
6. Property price updates by owners
7. Rejection of unauthorized price updates
8. Toggle of property sale status

5.2 Test Implementation

The tests use Truffle's testing framework with OpenZeppelin's test helpers. Key testing patterns include:

```
// Event verification
expectEvent(tx, 'PropertyListed', {
    propertyId: new BN(0),
    location: propertyLocation,
    price: propertyPrice,
    owner: owner
});

// State verification
const property = await realEstateMarketplace.getProperty(0);
assert.equal(property.owner, buyer, "Property owner should be updated
   to buyer");

// Balance verification
const finalOwnerBalance = new BN(await web3.eth.getBalance(owner));
const expectedBalance = initialOwnerBalance.add(new BN(propertyPrice));
assert.equal(finalOwnerBalance.toString(), expectedBalance.toString(),
   "Seller should receive ETH payment");

// Transaction rejection testing
await expectRevert(
    realEstateMarketplace.updatePropertyPrice(0, newPrice, { from:
   unauthorized }),
    "Only owner can update price"
);
```

Listing 16: Test Patterns

6 Frontend Architecture

6.1 Component Structure

The frontend is built with React.js and organized into the following components:

1. `PropertyList.js`: Displays all available properties in the marketplace
2. `PropertyCard.js`: Individual property display component
3. `PropertyForm.js`: Form for listing new properties
4. `UserProperties.js`: Displays properties owned by the current user
5. `UpdatePriceDialog.js`: Dialog for updating property prices
6. `ConfirmDialog.js`: Confirmation dialogs for transactions
7. `LoadingSpinner.js`: Loading indicator for blockchain transactions

6.2 Web3 Integration

The frontend integrates with the Ethereum blockchain using Web3.js. Key integration patterns include:

```
// Connect to Web3 provider
const connectWallet = async () => {
    if (window.ethereum) {
        try {
            // Request account access
            await window.ethereum.request({ method: '
   eth_requestAccounts' });
            // Initialize Web3 instance and contract
            // ...
        } catch (error) {
            console.error("Error connecting to MetaMask", error);
        }
    }
};
```

Listing 17: Web3 Connection Pattern

```
// Example of contract interaction
const buyProperty = async (propertyId, price) => {
    try {
        await contract.methods.buyProperty(propertyId)
            .send({ from: account, value: price });
        // Handle success
    } catch (error) {
        // Handle error
    }
};
```

Listing 18: Contract Interaction Pattern

7 Design Decisions and Tradeoffs

7.1 On-Chain vs. Off-Chain Data

One of the key design decisions was determining which data to store on-chain versus off-chain. The implementation follows these guidelines:
On-Chain Data:

- Property ownership records
- Price information
- Basic property details (bedrooms, bathrooms, size)
- Sale status

Off-Chain Data:

- Property images (stored on IPFS with hash references)
- Extended property descriptions
- Historical transaction data (tracked via events)

This approach minimizes gas costs while maintaining the integrity of essential ownership and transaction data.

7.2 Security vs. Gas Cost

The implementation balances security and gas optimization:
Security Priorities:

- Use of nonReentrant modifier for buyProperty function
- Strict ownership verification
- Proper validation of inputs
- Secure ETH transfer patterns

Gas Optimization:

- Appropriate data types (uint8 for small numbers)
- Minimal on-chain storage
- Efficient use of memory vs. storage

In cases where tradeoffs were necessary, security was prioritized over gas optimization.

8 Future Enhancements

8.1 Tokenization of Properties

Future iterations could implement ERC-721 tokens to represent property ownership, allowing for:

- Fractional ownership of properties
- Integration with the broader NFT ecosystem
- More sophisticated property rights management

8.2 Escrow Mechanism

A more advanced version could implement:

- Multi-signature escrow contracts
- Time-locked transactions
- Dispute resolution mechanism

8.3 Oracle Integration

Integration with oracles could enable:

- Real-world property verification
- Automatic price updates based on market conditions
- Integration with traditional property databases

9 Conclusion

The decentralized real estate marketplace successfully implements a secure, gas-efficient platform for property transactions on the Ethereum blockchain. By leveraging smart contracts, the system eliminates intermediaries and reduces friction in real estate transactions while maintaining security and usability.

The implementation balances on-chain and off-chain data storage to optimize for both cost and functionality. Security measures, including reentrancy protection and ownership verification, safeguard user funds and property records.

The comprehensive testing suite ensures the reliability of all core functions, and the React-based frontend provides an intuitive interface for users to interact with the blockchain. This project demonstrates the practical application of blockchain technology to solve real-world problems in the real estate sector.