In-Depth Analysis of Alternative Solutions in Solidity Smart Contracts

SHAO Yuning
Department of Computing
The Hong Kong Polytechnic University
Hong Kong SAR, China
Email: 24045081g@connect.polyu.hk

***Abstract*—In this paper, we provide an in-depth analysis of similar implementations with different functionalities in smart contracts, based on the Official Developer's Guide for Ethernet. Starting in Chapter 2, we provide a multi-dimensional comparative analysis, followed by a physical storage and logical level security analysis in Chapter 3, and a detailed gas cost comparison in Chapter 4. Finally, we provide recommendations for novice developers.**

Keywords: Development, Smart Contract, Security, Gas Optimization

I. Introduction

Solidity is an object-oriented high-level programming language specifically designed to implement smart contracts on various blockchain platforms (especially the Ethernet blockchain). It was developed by core contributors to the Ethernet network and is statically typed with support for inheritance, libraries, and complex user-defined types [1]. The language is influenced by C++, Python and JavaScript, and is designed specifically for the Ethernet Virtual Machine (EVM). As Ether continues to dominate the smart contract platform space, more and more developers are adopting Solidity as their primary language for smart contract development. However, for newcomers to blockchain development, Solidity presents unique challenges, especially when it comes to distinguishing between similar implementations that serve the same purpose. This article provides a detailed analysis of these alternatives to help developers write smart contracts using the right approach.

II. Comparative Analysis

A. Require vs. Assert vs. Revert

1) Background and Implementation: A lot of dapps use ownable, we need to do owner validation before calling the function, if it does not meet the rollback occurs. Starting with version 0.4.10 [2], the traditional throw keyword was replaced by three specialized functions, require(), assert() and revert(), which marked a significant shift in error handling. While all three mechanisms enable transaction rollback, each has different responsibilities.

```
if(msg.sender != owner) { revert("Not owner!"); }
     // We can set the return value
assert(msg.sender == owner);
require(msg.sender == owner, "Not owner!"); // We
     can set the return value
```

While these implementations yield identical results, their intended use cases and implications differ significantly in contract design and gas optimization.

2) Comparative Analysis: require() is mainly used to handle input validation and business logic checking, and its concise syntax and gas return mechanism make it ideal for everyday validation; assert() focuses on invariant checking, and plays an important role especially when dealing with overflow/underflow detection and stateful validation; and revert() provides a more flexible approach to error handling, and is especially suited to handling complex conditional branching and custom error scenarios.

In Solidity 0.8.0, the implementation of assert() underwent a significant change from using INVALID to REVERT, an improvement that significantly improves the efficiency of gas while maintaining the ability to roll back state, i.e., it does not consume all the gas, but instead returns it, as with require and reverse. This evolution reflects Solidity's continued efforts to strike a balance between security and efficiency. In practice, developers need to choose the right error handling mechanism according to specific scenarios: for regular input validation, require() is the best choice; for critical invariant checking, assert() should be used; and when complex logic or custom error messages need to be handled, revert() provides the necessary flexibility. This structured error handling strategy not only improves the robustness and gas efficiency of the contract, but also effectively guards against potential security vulnerabilities through a reasonable error handling mechanism.

B. Modifier vs. Direct Function Validation

1) Background and Implementation: As mentioned above, there are a number of ways in which we can add conditional restrictions [3]. Although these two approaches serve the same purpose, they show significant differences in terms of code reusability, readability and gas consumption.

```
// Using modifier
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner!!");
    _;
}
```

```
function transferOwnership(address newOwner)
    public onlyOwner {
    owner = newOwner;
}

// Using direct function validation
function transferOwnership(address newOwner)
    public {
    require(msg.sender == owner, "Not owner!!");
    owner = newOwner;
}
```

- **Code Reusability**
  - Modifiers provide reusable validation logic across multiple functions
  - Direct validation requires repeating the same checks in each function

  Consider this example of modifier reuse:

```
// Multiple functions using the same
    modifier
contract Token {
    modifier onlyOwner() {
        require(msg.sender == owner, "Not
            owner");
        _;
    }

    function transferOwnership(address
        newOwner)
        public onlyOwner {
        owner = newOwner;
    }

    function mint(address to, uint amount)
        public onlyOwner {
        _mint(to, amount);
    }

    function pause()
        public onlyOwner {
        _pause();
    } // We can see that we simpilify the
        coding by reusing modifier
}
```

2) Comparative Analysis: From a technical perspective, modifiers are transformed into inline code during compilation, where the compiler simply inserts the modifier's code directly into each function that uses it. The resulting bytecode is nearly identical to direct validation, with only an additional jump instruction added. This leads to a very small gas difference of about 20-50 gas.

Given this minimal gas overhead and the significant benefits in code organization that modifiers provide, they are generally the recommended choice for validation logic. The slight increase in gas cost is a reasonable trade-off for better code structure and easier maintenance. However, for very simple contracts with only one or two validation checks, direct validation remains a viable option.

C. SafeMath vs. Native Checks vs. Unchecked

1) Background and Implementation: Adding, subtracting, multiplying and dividing are very routine operations in any programming code, but there are some security issues that we must not overlook, and the arithmetic security mechanisms have undergone a significant shift over successive releases. Prior to the 0.8.0 release of Solidity, developers relied heavily on the SafeMath library [4] to protect against arithmetic overflow and underflow vulnerabilities, a defensive programming model implemented through OpenZeppelin's audited contract, which rolls back the transaction through a method wrapper when a numerical anomaly is detected. With the 0.8.0 release, Solidity introduces a built-in overflow checking mechanism [5], while providing the unchecked keyword to flexibly control this feature. Table I shows a detailed comparison of these three approaches.

Basic implementation examples:

```
// Using SafeMath (pre-0.8.0)
using SafeMath for uint256;
a = a.add(b);

// Using native checks (0.8.0+)
a = a + b;

// Using unchecked (0.8.0+)
unchecked { a = a + b; }
```

2) Comparative Analysis: From an implementation perspective, this evolution reflects the maturity of Solidity as a smart contract language. Security checks that originally needed to be implemented through external libraries are integrated into the language itself, which not only simplifies the syntactic structure, but also improves gas efficiency. In particular, the introduction of unchecked blocks provides optimisation options for scenarios where overflow is mathematically impossible (e.g. loop counters).

By comparatively analysing the features of these three mechanisms (as shown in Table I), we can clearly see the trade-offs between them in different aspects: In terms of version support, SafeMath mainly serves versions prior to 0.8.0, while native checking and unchecked are features of newer versions. In terms of gas efficiency, there is a clear gradient: SafeMath consumes the highest (28K gas), native checking is in the middle (24K gas), and unchecked is the lowest (21K gas).

In terms of security, SafeMath and native checking both provide a high level of protection, just implemented differently, while unchecked forgoes overflow protection altogether.

In terms of best application scenarios, SafeMath is primarily used for legacy system compatibility, native checking is suitable for standard operations, and unchecked is targeted at performance-critical code.

This evolution of arithmetic security mechanisms reflects Solidity's efforts to find a balance between security and performance optimisation. For developers, choosing the right mechanism takes into account the specific needs of the project: SafeMath may be needed when dealing with legacy systems, native checking is both safe and convenient for day-to-day development, and unchecked provides the necessary flexibility for specific performance-optimisation scenarios. This multi-layered security mechanism not only enhances the usefulness

TABLE I
COMPARISON OF ARITHMETIC SAFETY MECHANISMS IN SOLIDITY

Feature	SafeMath	Native Checks	Unchecked
Version Support	Pre-0.8.0	0.8.0 and later	0.8.0 and later
Gas Efficiency	Highest cost (28K gas)	Medium cost (24K gas)	Lowest cost (21K gas)
Security	High with external validation	High with built-in checks	No overflow protection
Implementation	Requires library import	Built-in feature	Explicit block syntax
Best Use Case	Legacy compatibility	Standard operations	Performance-critical code

of the language, but also provides appropriate solutions for development needs in different scenarios.

D. Transfer vs. Send vs. Call

1) Background and Implementation: Transaction transfers have an integral part in defi. Solidity provides three main ETH transfer methods: transfer(), send() and call() [6] [7], each with its own unique characteristics in terms of gas limits, return value handling and security features. Contracts can receive ETH via the receive() and fallback() functions with payable modifiers, while the sender needs to choose the appropriate transfer method for a specific scenario.

```
// Contract receiving ETH
contract Receiver {
    // Receive ETH through direct transfers
    receive() external payable {
        // Handle plain ETH transfers
    }

    // Fallback function for handling ETH with
        data
    fallback() external payable {
        // Handle ETH transfers with data
    }
}

// Methods for sending ETH
contract Sender {
    // Using transfer
    payable(recipient).transfer(amount);

    // Using send
    bool success = payable(recipient).send(amount
        );
    require(success, "Transfer failed");

    // Using call
    (bool success, bytes memory data) = recipient
        .call{value: amount}("");
    require(success, "Call failed");
}
```

2) Comparative Analysis: From a technical implementation perspective, both transfer() and send() implement a fixed limit of 2300 gas, a design originally intended to prevent reentry attacks. However, this limit can be a constraint in the complex interaction scenarios of modern smart contracts, especially when the cost of gas for EVM operations varies with network upgrades. transfer() automatically rolls back on failure, providing a safe default behaviour, while send() returns a boolean value, requiring explicit error handling. In contrast, the call() method provides greater flexibility, supports configurable gas limits, and can return success status and response data, but this flexibility also requires greater care in implementation to ensure security.

In practice, the choice of these three methods needs to be based on specific scenarios: transfer() is suitable for simple ETH transfers, especially if the recipient is the base wallet address; send() still has its application in scenarios where custom error handling is required and 2300 gas limits are sufficient; and call() becomes the recommended method for interacting with complex contracts, although this requires a more careful security implementation.

This evolution of transfer mechanisms reflects the continued development of the Ethernet ecosystem, which has gradually shifted from the initial pursuit of strict security constraints to a more flexible and adaptable direction. This shift not only reflects the growing maturity of smart contract applications, but also illustrates developers' efforts to find a balance between security and functionality. For modern smart contract development, understanding the characteristics and applicable scenarios of these approaches is crucial for building secure and efficient decentralised applications.

E. Call vs. Delegatecall vs. Staticcall

1) Background and Implementation: The call() function in Solidity serves two primary purposes: transferring ETH (as discussed in the previous section) and executing functions in other contracts. Building upon this dual functionality, Solidity provides three low-level methods for cross-contract interactions: call(), delegatecall(), and staticcall() [8]. Each method offers different ways to execute code in other contracts, with distinct context handling and security implications. While call() maintains its versatility for function calls, delegatecall() specializes in maintaining caller context for proxy patterns, and staticcall() ensures read-only operations.

```
// Target contract to be called
contract Target {
    address public owner;
    uint public value;

    function updateValue(uint _value) public {
        owner = msg.sender;  // Will differ based
            on call type
        value = _value;
    }

    function getValue() public view returns (uint
        ) {
        return value;
    }
}

// Caller contract demonstrating different call
    types
contract Caller {
    address public owner;
    address public target;

    constructor(address _target) {
        owner = msg.sender;
        target = _target;
    }

    // Using call: msg.sender in Target will be
        Caller's address
    function executeCall() public {
        (bool success, bytes memory data) =
            target.call(
            abi.encodeWithSignature("updateValue(
                uint256)", 100)
        );
        require(success, "Call failed");
        // In Target: owner will be Caller's
            address
    }

    // Using delegatecall: preserves original msg
        .sender
    function executeDelegateCall() public {
        (bool success, bytes memory data) =
            target.delegatecall(
            abi.encodeWithSignature("updateValue(
                uint256)", 100)
        );
        require(success, "DelegateCall failed");
        // In Caller: owner will be original msg.
            sender (EOA)
    }

    // Using staticcall: only for reading state
    function executeStaticCall() public view
        returns (uint) {
        (bool success, bytes memory data) =
            target.staticcall(
            abi.encodeWithSignature("getValue()")
        );
        require(success, "StaticCall failed");
        return abi.decode(data, (uint));
        // Cannot modify state, only read
    }
}
```

TABLE II
COMPARISON OF LOW-LEVEL CALL METHODS

Feature	call	delegatecall	staticcall
Context	Target	Caller	Target
State Changes	Allowed	Caller's State	Not Allowed
msg.sender	Caller	Original Sender	Caller
Storage	Target's	Caller's	Read-only
Use Case	General	Upgrades	View Functions

2) *Comparative Analysis:* As shown in Table II, these three methods differ in several key aspects. The execution context and storage access patterns vary significantly: while call() operates in the target contract's context with access to its storage, delegatecall() uniquely executes in the caller's context using the caller's storage. staticcall() follows call()'s context but restricts all state modifications.

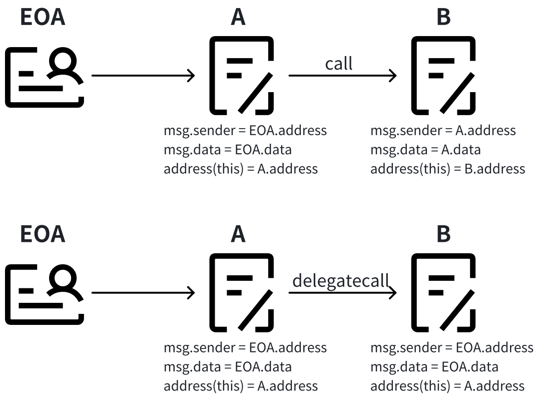

Fig. 1. Comparison of Call and Delegatecall Execution Context

As shown in Figure 1, call() and delegatecall() handle contract addresses and data differently. In call(), the msg.sender becomes the calling contract's address, and the target contract receives this address. However, in delegatecall(), the original caller's address (EOA) is preserved and passed to the target contract. This means that when using delegatecall(), the target contract sees the transaction as if it came directly from the original sender, not from the intermediate contract.

These three call types also differ in their handling of msg.sender:

- call(): Changes msg.sender to the calling contract's address
- delegatecall(): Keeps the original transaction sender's address
- staticcall(): Works like call() but only allows reading data

Each method has its specific use case as shown in the table: call() is suitable for general-purpose interactions, delegatecall() is specifically designed for upgrade patterns, and staticcall() is optimized for view functions where state modifications are not needed.

Notice: When implementing upgradeable smart contracts through proxy patterns, delegatecall() is the recommended method as it uniquely maintains the caller's storage context while executing the target contract's logic. This preservation of context is crucial for contract upgradeability, as it allows the proxy contract to maintain its state while delegating the execution of logic to the implementation contract. The method's ability to preserve the original msg.sender ensures proper access control mechanisms remain intact throughout the upgrade process. However, developers must exercise extreme caution when implementing upgradeable contracts using delegatecall(): the storage layout must remain consistent across all contract versions to prevent state corruption, implementation contracts must undergo thorough security audits, and robust access controls must be established for the upgrade functionality to prevent unauthorized modifications.

F. Memory Array vs Storage Array

1) Background and Implementation: In Solidity, there are various data types including integers (uint/int), addresses (address), booleans (bool), and more. When dealing with multiple values of the same type, arrays provide an efficient way to store and manage these values instead of declaring multiple individual variables. There are two types of arrays: memory arrays and storage arrays, each with distinct characteristics in terms of persistence and mutability.

```
contract ArrayComparison {
    // Storage array examples with different
        types
    uint256[] public numbers;      // Array of
        integers
    address[] public addresses;    // Array of
        addresses
    bool[] public flags;          // Array of
        booleans

    // Memory array operations
    function memoryArrayOps() public pure returns
         (uint256[] memory) {
        uint256[] memory memArray = new uint256
            [](3);
        memArray[0] = 1;
        memArray[1] = 2;
        memArray[2] = 3;
        return memArray;
    }

    // Dynamic storage array operations
    function storageArrayOps() public {
        numbers.push(1);     // Dynamic sizing
        numbers.push(2);
        numbers.pop();       // Can remove
            elements
    }
}
```

2) Implementation Mechanisms: Storage arrays are implemented using Ethereum's persistent storage, where elements are stored in non-contiguous slots calculated using the array's base position and element index [9]. This allows for dynamic resizing but incurs higher gas costs. In contrast, memory arrays are allocated in a contiguous block of memory during function execution, with a fixed size specified at creation time. The memory layout starts with a 32-byte length field, followed by the array elements, each occupying 32 bytes.

```
Storage Layout:            Memory Layout:
slot p: Length            0x00-0x20: Array length
keccak256(p): Item 0      0x20-0x40: First element
keccak256(p)+1: Item 1    0x40-0x60: Second element
```

3) Key Differences: The fundamental distinction between memory and storage arrays lies in their persistence and mutability. Storage arrays persist between function calls and can be dynamically resized using push() and pop() operations, making them suitable for contract state management. Memory arrays, while more gas-efficient for read and write operations, are temporary and fixed in size, making them ideal for function-scoped computations and return values.

G. Receive vs. Fallback

1) Background and Implementation: In Solidity, contracts can receive ETH through two special functions: receive() and fallback(). While receive() is specifically designed for plain ETH transfers with no additional data, fallback() serves as a more general-purpose function that handles both function calls with data and ETH transfers when receive() is not present. This dual-purpose nature of fallback() means it can effectively replace receive() in many cases, though at the cost of reduced code clarity and slightly higher gas consumption.

```
contract PaymentHandler {
    event Received(address sender, uint amount);

    // Specific handler for plain ETH transfers
    receive() external payable {
        emit Received(msg.sender, msg.value);
    }

    // General-purpose handler for all cases
    fallback() external payable {
        // Can handle both ETH transfers and
            function calls
        // Has access to msg.data if needed
    }
}
```

2) Comparative Analysis: The function selection mechanism in Solidity prioritizes receive() for plain ETH transfers when both functions are present. When a contract receives a transaction with no calldata, the EVM first attempts to execute receive(). Only if receive() is not defined will it fall back to the fallback() function. For calls with data, fallback() is always used unless a matching function signature is found. This hierarchy reflects the design philosophy of having specialized functions for common cases (receive() for plain transfers) while maintaining flexibility through a general-purpose fallback mechanism.

In the Solidity smart contract, receive() and fallback() functions have significant differences in functional scope and application scenarios. receive() function adopts a stricter but purposeful design concept, which is exclusively used to process pure ETH transfer transactions and has no access to transaction data. This specialisation limits the scope of its functionality, but provides a clearer expression of the code's intent when dealing with simple transfer scenarios. In contrast, the fallback() function offers more comprehensive flexibility, not only in terms of being able to handle all types of call requests, but also in terms of accessing msg.data, making it ideal for implementing generic processing logic and custom function routing. However, this flexibility comes at the cost of potentially reducing the clarity of the code's intent, as well as a slightly higher gas cost due to the need to handle call data.

In practice, the choice between implementing both functions or relying only on the fallback() function will largely depend on the contract's specific trade-off between clarity and flexibility needs. The explicit intent of the receive() function may be more valuable for contracts that are primarily concerned with simple ETH receiving functionality, while the flexibility of the fallback() function may be more important for contracts that need to handle complex inbound calls. It is worth noting that in modern smart contract development practice, many contracts choose to implement both functions to ensure functional integrity while keeping concerns separate. This approach not only meets the needs of different scenarios, but also provides better code organisation.

III. Security Analysis

A. Overflow Vulnerabilities

In Solidity, arithmetic operations and array bounds can lead to critical security vulnerabilities through overflow or underflow conditions [10]. Understanding these vulnerabilities and their prevention mechanisms is essential for secure smart contract development.

1) Arithmetic Overflow: Before Solidity 0.8.0, arithmetic operations could silently overflow, leading to unexpected results. For example:

- **uint8 Overflow**: Maximum value is 0xff (binary: 11111111):

  $$0xff + 0x01 = 100000000 \rightarrow 00000000$$

  When adding 1 to 0xff, the result requires 9 bits (100000000), but since uint8 can only store 8 bits, the leading 1 is truncated, resulting in 0x00.
- **uint8 Underflow**: Minimum value is 0x00:

  $$0x00 - 0x01 = 11111111 = 0xff$$

  When subtracting 1 from 0x00, it borrows from a higher bit, wrapping around to the maximum 8-bit value.

To prevent these vulnerabilities, OpenZeppelin's SafeMath library implements checks before each arithmetic operation:

```
library SafeMath {
    function add(uint256 a, uint256 b) internal
        pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition
            overflow");
        return c;
    }

    function sub(uint256 a, uint256 b) internal
        pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction
            overflow");
        return a - b;
    }

    function mul(uint256 a, uint256 b) internal
        pure returns (uint256) {
        if (a == 0) return 0;
        uint256 c = a * b;
        require(c / a == b, "SafeMath:
            multiplication overflow");
        return c;
    }
}
```

To prevent these vulnerabilities, OpenZeppelin's SafeMath library implements checks before each arithmetic operation. Since Solidity 0.8.0, these checks have been built into the compiler, automatically reverting transactions when overflow occurs. This built-in protection eliminates the need for SafeMath in most cases. However, as discussed in our earlier comparison of SafeMath vs. Native Checks vs. Unchecked, there are scenarios where bypassing these checks through unchecked blocks can optimize gas consumption, particularly in mathematically bounded operations like loop counters.

2) Array Bounds: Memory arrays in Solidity have fixed sizes and require careful bounds checking [9]. Attempting to access elements beyond the array's length will cause the transaction to revert. While the EVM automatically enforces these bounds checks, developers should still implement proper validation to prevent unnecessary transaction failures and gas waste.

For dynamic storage arrays, the main security concerns lie in proper length management and access control. While .push() operations are inherently safe, developers need to be cautious about unbounded array growth which could lead to excessive gas costs. Additionally, when deleting elements, using delete directly on array elements can leave gaps in storage, making the swap-and-pop pattern a more efficient and secure approach.

Static arrays, while providing compile-time size guarantees and potentially better gas efficiency, still require careful index validation to prevent out-of-bounds access attempts. Their fixed size nature provides some inherent safety but can be limiting in scenarios requiring dynamic sizing. When copying between static arrays, developers should ensure size compatibility to prevent data truncation.

In both cases, implementing proper bounds checking and size validation is crucial for maintaining contract security, though the specific approaches may differ based on the array type chosen.

B. Replay Attacks

In the field of smart contract security, replay attacks are a way of achieving unintended effects by maliciously repeating valid transactions, and Solidity mitigates the risk of such attacks by providing multiple layers of protection through its design features and built-in mechanisms.

The 2300 gas limit enforced by the transfer() and send() functions, while originally designed to prevent reentry attacks, also constitutes an effective barrier against replay attacks. This gas limit ensures that the receiving contract can only perform minimal logical operations, typically limited to recording the receipt of funds, and cannot initiate external calls or complex state changes that could be exploited by a replay attack.

Similarly, the gas limitations of the receive() and fallback() functions provide inherent protection mechanisms. receive() functions are specifically designed to handle pure ETH transfers, and their limited execution environment inherently reduces the risk of replay attacks. The fallback() function, on the other hand, while offering greater flexibility, can be effectively prevented from becoming an attack vector through proper gas management and implementation models.

However, the more flexible call() method, which does not enforce gas restrictions, requires developers to take additional security measures in its implementation. When using call(), it is recommended to implement the following protection mechanisms:

- Maintain counters for each operation using a nonce tracking mechanism
- Implement reasonable transaction time window validation
- Establish fine-grained function access control to limit the calling privileges of specific functions

These security considerations demonstrate how Solidity's built-in features, when used appropriately, can effectively protect against replay attacks while maintaining contract functionality. By using these mechanisms together, developers can build more secure and robust smart contract systems.

C. *Storage Layout Conflicts*

When using delegatecall, one of the most critical security vulnerabilities arises from storage layout conflicts [12]. Since delegatecall executes the target contract's code in the context of the calling contract's storage, mismatched storage layouts can lead to unintended state modifications [13]. Consider this example:

```
// Original Contract
contract Logic {
    address public owner;    // slot 0
    uint256 public value;    // slot 1
}

// Updated Contract (Incorrect Layout)
contract LogicV2 {
    uint256 public value;    // slot 0 - Will
        overwrite owner!
    address public owner;    // slot 1
}
```

In this example, if a proxy contract using delegatecall upgrades from Logic to LogicV2, any operation modifying the value variable in LogicV2 will accidentally corrupt the owner variable in the proxy's storage due to the misaligned storage slots. This type of vulnerability can lead to severe security issues, including loss of contract ownership or corruption of critical state variables.

To prevent such vulnerabilities:

- Always maintain consistent storage layouts across contract versions
- Implement thorough testing of storage layouts before deployment

IV. Gas Analysis

A. *environment*

We use multiple tools and platforms for testing:

- Tenderly platform: for contract deployment and trade simulation (https://dashboard.tenderly.co/explorer)
- Remix IDE: for contract compilation and initial testing (https://remix.ethereum.org)
- MetaMask: for testing network deployment
- Python matplotlib: for data visualization

Due to limited testnet tokens, we utilized Tenderly for simulated transactions, which provides detailed transaction information. For each simulated transaction, we performed 5 iterations to obtain average data and minimize variance.

B. *Analysis*

Firstly,We tested the gas cost data for assert, require, and revert methods across versions 0.7.0 and 0.8.0. Figure 2 shows our findings.

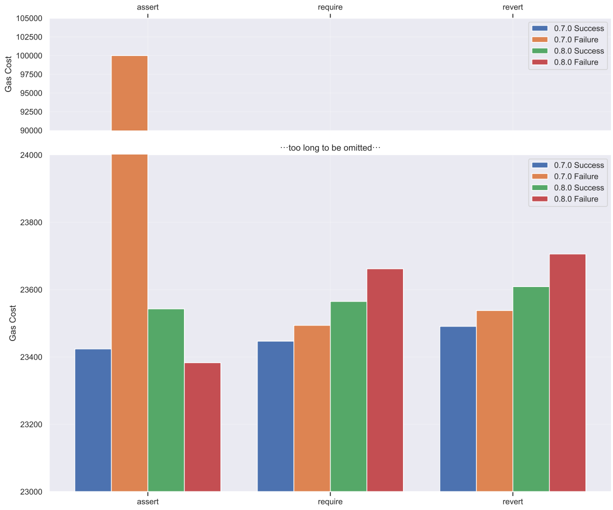

Fig. 2. Gas Cost Comparison Across Solidity Versions

We can see that failures tend to consume more gas than successes, this is due to the fact that the rollback operation after a failure consumes a certain amount of gas fee. Due to the extra security checks added to these functions in version 0.8.0, the compiler generates more opcodes to handle boundary cases, so higher versions consume more gas for the same state. An exception, we found that failing an assert transaction consumes all the gas in versions below 0.8.0, so developers should be more willing to spend a little bit more gas to get a more secure review mechanism to avoid excessive losses.

Next, We also compared gas costs between using modifiers and direct function validation, as shown in Table III.

TABLE III
Gas Cost Comparison: Modifiers vs Direct Validation

Validation Method	Success (gas)	Failure (gas)
Direct Validation	23,542	23,639
Modifier	23,586	23,683

The result shows that although modifiers consume a little bit more, this is due to the fact that modifiers are essentially code inlining, and the compiler will insert the modifier code directly into the function. The final compiled bytecode is very similar, except for the extra jump instruction, it usually only consumes about 20-50 gas more, but in view of the advantages of modifiers analyzed above, this article suggests that they can all be expressed as modifiers, unless there are too many a priori conditions, and nesting modifiers in multiple layers may lead to errors, so we can use in-function conditions.

Since the problem of overflow above and below a function is very serious and can lead to contractually fatal errors, we have analysed the cost of gas consumed by safely adding and subtracting maths, and the results are shown in Figure 3.

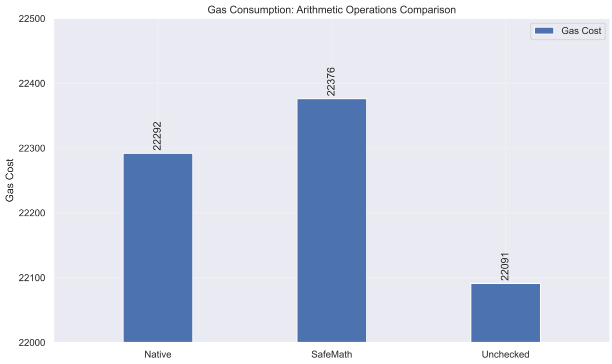

Fig. 3. Gas Cost Comparison: Arithmetic Safety Mechanisms

Because we are using version 0.8.0 or above, the compiler has been optimized, although it is not as big as the above comparative analysis of the gas difference, but the size of the gas cost can still reflect the performance of different methods, as we said above, unchecked consumes the smallest cost, safemath consumes the largest cost.

In Ether smart contracts, ETH is closely related to the method of sending ETH, so we have also analyzed transfer methods: Figure 4.

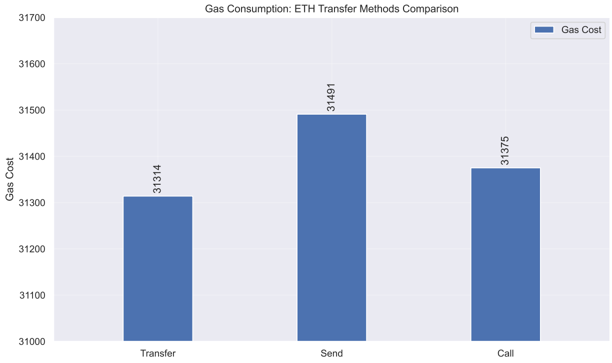

Fig. 4. Gas Cost Comparison: ETH Transfer Methods

The reason why transfer is lowest is that we use a fixed sequence of opcodes, which the compiler can optimize, and do not need to handle return values, and failure is rolled back directly without additional error handling logic, whereas send needs to handle boolean return values and includes an additional return value checking mechanism.

Notice: Of course, first of all, to ensure that the contract security premise, the developer needs to further consider the gas optimisation problem, and different versions have different methods of application, developers need to always pay attention to the

V. Conclusion

This paper provides an in-depth analysis of the different functional implementation alternatives commonly encountered by Solidity beginners. By examining seven pairs of key alternatives, we observe that the built-in compiler checks for smart contracts have become increasingly comprehensive and sophisticated with major upgrades to the Solidity release. However, developers should not be overly reliant on these automated checking mechanisms, but should instead keep an eye on the functionality changes in each major version update. This in-depth understanding of version changes enables developers to make appropriate adjustments in different versions of smart contracts to ensure contract security.

In addition to security considerations, developers need to maintain a balance between code reusability and efficiency, focusing on gas optimization. By following the design principles of high cohesion and low coupling, developers can establish good programming practices and lay a solid foundation for advanced smart contract development. This systematic approach not only improves the maintainability of contracts, but also provides the necessary technical support for building complex decentralized applications.

References

[1] Wikipedia Contributors, “Solidity,” Wikipedia, The Free Encyclopedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Solidity

[2] Solidity Team, “Solidity 0.8.0 Breaking Changes,” Solidity Documentation, 2020. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/080-breaking-changes.html

[3] Solidity Team, “Solidity Modifiers,” Solidity Documentation, 2020. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/070-inheritance.html#modifiers

[4] OpenZeppelin Team, “SafeMath Library,” OpenZeppelin Documentation, 2020. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol

[5] Solidity Team, “Arithmetic in Solidity,” Solidity Documentation, 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.0/control-structures.html#checked-or-unchecked-arithmetic

[6] Solidity Team, “Address Members,” Solidity Documentation, 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.17/types.html?highlight=send#members-of-addresses

[7] Z. Mohammed, “Solidity — transfer vs send vs call function,” Coinmonks, Dec. 2021. [Online]. Available: https://medium.com/coinmonks/solidity-transfer-vs-send-vs-call-function-64c92cfc878a

[8] Solidity Team, “Contract ABI Specification,” Solidity Documentation, 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.17/abi-spec.html#function-selector-and-argument-encoding

[9] Solidity Team, “Solidity Storage and Memory,” Solidity Documentation, 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.17/internals/layout_in_memory.html

[10] N. Atzei, M. Bartoletti, and T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts,” Principles of Security and Trust, 2017. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[11] G. Chen et al., “Understanding Ethereum via Graph Analysis,” IEEE INFOCOM, 2018. [Online]. Available: https://doi.org/10.1109/INFOCOM.2018.8486401

[12] F. Zhang et al., “Smart Contract Security: A Software Lifecycle Perspective,” IEEE Access, vol. 7, pp. 150184-150202, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2946988

[13] OpenZeppelin Team, “Proxy Upgrade Pattern,” OpenZeppelin Documentation, 2021. [Online]. Available: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies

[14] SHAO Yuning, “In-Depth Analysis in Solidity Smart Contracts,” GitHub Repository, 2024. [Online]. Available: https://github.com/RootCoinT/In-Depth-Analysis-in-Solidity-Smart-Contracts