Dear Reviewers,

In this attachment, we present complementary details to support our response in the rebuttal letter.

Part 1. Statistics on the Real-World Distribution of Each Defect Type

After analyzing 1,379 security audit reports and 326 Stack Overflow posts, we screened for security issues arising from the reuse of contract code on EVM-compatible blockchains. We identified 197 security analyses covering related defects and ultimately defined six types of EVM Inequivalent Defects: Cross-Chain Replay Attack (CCRA), Time Discrepancy Trap (TDT), Fixed Gas Reentrancy (FGR), Block Height Misalignment (BHM), Phishing Contract Address (PCA), and Gas Limit Imbalance (GLI). To better analyze the real-world distribution of these defects, we examined the distribution and sources of the 197 materials covering these defects. For specific titles and URLs, please see [xx]. Table 1 provides statistics on the quantity and percentage of different defects.

Table 1 Quantity and Percentage of Different Defects

Type	**CCRA**	**TDT**	**FGR**	**BHM**	**PCA**	**GLI**
Quantity	82	32	13	21	11	38
Percentage	41.62%	16.24%	6.60%	10.66%	5.58%	19.29%

Table 1 shows the quantity and percentage of six different defects. The CrossChain Replay Attack (CCRA) type leads with 82 instances, making up 41.62% of the total, indicating it as the most common issue. Following closely is the Gas Limit Imbalance (GLI) type with 38 defects, accounting for 19.29%, marking it as another major defect. The Time Discrepancy Trap (TDT) ranks in the middle with 32 instances (16.24%), while Block Height Misalignment (BHM) has 21 instances but still reaches a 10.66% share, showing it is comparatively less common. Fixed Gas Reentrancy (FGR) and Phishing Contract Address (PCA) types have lower numbers and percentages, with 13 (6.60%) and 11 (5.58%) instances respectively, indicating these defects are relatively rare. This data analysis clearly shows that CCRA and GLI are the defect types developers need to pay most attention to, together constituting over half of the total, with TDT also not to be overlooked. Developers need to thoroughly understand the distribution of defects and tailor their improvement measures based on the severity of the vulnerabilities.

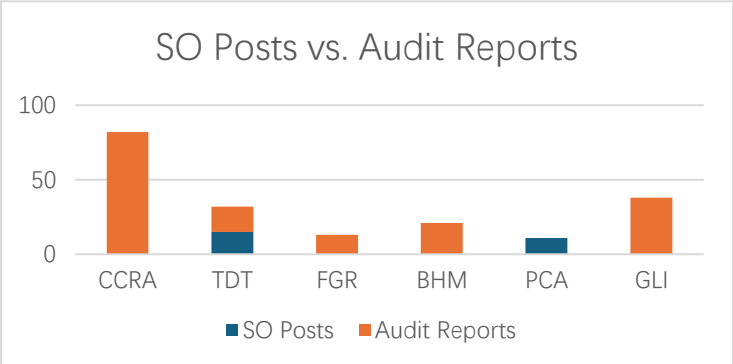

Figure 1: Comparison of Different Defect Information Sources

Figure 1 compares the sources of information for different defect types, showing their distribution across Stack Overflow (SO) posts and audit reports. CrossChain Replay Attack (CCRA) defects are found exclusively in audit reports, with 82 instances primarily from contract security audits, indicating these issues are closely related to vulnerability security in the

development process and not commonly discussed in developer communities. Time Discrepancy Trap (TDT) defects are present in both SO posts (15 instances) and audit reports (17 instances), highlighting this defect type as both a focus in the developer community and in audits. Fixed Gas Reentrancy (FGR) and Block Height Misalignment (BHM) defects are only recorded in audit reports, with 13 and 21 instances respectively, suggesting these issues are more about specialized and internal concerns, less known to the external developer community. Meanwhile, Phishing Contract Address (PCA) defects are only noted in SO posts (11 instances), indicating these are problems developers likely encounter and seek help for in actual programming. Gas Limit Imbalance (GLI) defects appear solely in audit reports (38 instances), with no records in SO posts, suggesting GLI issues might be more related to internal audits rather than public technical discussions. These data reveal significant differences in the sources of various defect types, helping us understand the common issues developers face when reusing contract codes.

Part 2. Detailed Response to Reviewer#C Comments

Part 2.1 Using block.number to Estimate Time

(1) Using block.number helps avoid miner manipulation attacks. As demonstrated in the report by security company AuditBase [1], block.timestamp is susceptible to miner manipulation, potentially leading to front-running vulnerabilities. To ensure the security of Solidity contracts and mitigate risks associated with block.timestamp, it is recommended to use block.number. block.number does not rely on timestamps but represents the current block number.

```
contract TimeBasedContract {
    uint256 public expirationBlock;

    function setTimeLock(uint256 _duration) public {
        expirationBlock = block.number + _duration;
    }

    function isExpired() public view returns (bool) {
        return block.number >= expirationBlock;
    }
}
```

In the example above, block.number is used to calculate the expiration block based on a given duration. AuditBase auditors suggest that using block.number instead of block.timestamp can prevent vulnerabilities related to miner manipulation and front-running.

(2) Using block.number to estimate time is a common programming practice, especially on blockchain platforms like Ethereum, mainly for time locks, conditional triggers, and reward distributions in smart contracts. Ethereum contract developers consider the block generation time to be relatively stable, aiming for a block every 15 seconds, for example. Therefore, developers might use block.number multiplied by the block time (e.g., 15 seconds) to estimate the current time [2]. Most developers are not aware of this during development. Although block.number offers convenience in time estimation, this practice

can lead to Time Discrepancy Trap (TDT) defects in contract reuse scenarios, along with a lack of automated analysis for such defects during contract reuse.

(3) As shown in Figure 4, these cases mainly involve crowdfunding, voting autonomy, and profit calculation via block numbers, covering a wide range of use cases. We emphasize to developers the importance of addressing widespread unsafe programming practices. At the same time, correct use of block.timestamp averts Time Discrepancy Trap (TDT) defects [3].

Part2.2 Risk of gas limit settings being too high or too low

(1) Reusing contracts with a fixed Gas Limit can lead to GLI defects. Gas Costs vary across different blockchains, and using the same Gas Limit in reused contracts poses risks of setting the gas limit too high or too low. Additionally, given the fluctuating Gas Costs on Ethereum, the security firm ConsenSys has already recommended stopping the use of statements like transfer/send that use a fixed Gas Limit [4].

```
contract Vulnerable {
    function withdraw(uint256 amount) external {
        // This forwards 2300 gas, which may not be enough if the recipient
        // is a contract and gas costs change.
        msg.sender.transfer(amount);
    }
}

contract Fixed {
    function withdraw(uint256 amount) external {
        // This forwards all available gas. Be sure to check the return value!
        (bool success, ) = msg.sender.call.value(amount)("");
        require(success, "Transfer failed.");
    }
}
```

If gas costs are subject to change, smart contracts cannot rely on any specific gas costs. Any smart contract employing transfer() or send() inherently depends on gas costs by forwarding a fixed amount of gas: 2300. Their recommendation is to cease using transfer() and send() in your code and to switch to using call() instead. Apart from the amount of gas forwarded, these two methods are equivalent.

(2) The gas limit should not be set too high or too low. The gas limit defines the maximum amount of gas that can be consumed in executing a transaction or smart contract operation. Such low-level operations can pose significant attack risks, especially when contracts are reused across different blockchain environments. Moreover, the setting of Gas limits must account for the specific Gas costs of the blockchain. Setting it too low may lead to DoS attacks, while too high could facilitate reentrancy attacks. [5] When reusing contracts without considering the specific Gas costs on each chain, using a fixed value increases the likelihood of GLI defect attacks.

Part 3. Detailed Response to Reviewer#B Comments

Part 3.1 Analyzing Mutexes

EquivGuard utilizes pattern recognition techniques based on Path Protection Techniques (PPTs) [6] during the static analysis phase to detect reentrancy protections such as mutexes, aiding in the analysis of various mitigation measures and reducing false positives. For preliminary analysis results, Step 3: Symbolic Execution Verification is implemented to verify the reachability of potential paths, thereby enhancing the accuracy of vulnerability detection. As demonstrated in the experiments for RQ2, the results indicate that EquivGuard achieves an overall precision rate of 95.29%.

Part 3.2 Higher Gas Limit

As mentioned in Section 2.1's Gas Mechanism, "Gas cost quantifies the amount of gas required to execute each operation, e.g., Ethereum's ADD operation consumes 3 gas, according to the Gas cost standard [7]." Many blockchains compatible with EVM have their own Gas cost standards and refund mechanisms to offer lower transaction fees, such as BSC's BEP [8]. A fixed Gas limit of 2300 for executing contracts on other EVM-compatible chains increases the risk of reentrancy vulnerabilities. For instance, due to the constant changes in Gas Costs, ConsenSys security firm recommends avoiding the use of transfer() and send()[6].

Part 4. Risk Analysis of Hardcoded Uniswap Router Address (0xFb13...FF7C)

Hardcoding the Uniswap router address to 0xFb13...FF7C poses risks, as router addresses vary across different EVM-compatible chains, potentially leading to failed token swaps. Currently, asset losses due to such misconceptions on non-Ethereum chains with the address 0xFb13...FF7C are on the rise (details: [9]). Although developers deploy contracts to prevent malicious attacks, the inconsistency of contract ownership across different chains makes it difficult to recover affected assets [10].

Part 5. Comparative Analysis with Results from Other Tools

[1] https://detectors.auditbase.com/blocknumber-vs-timestamp-solidity
[2] https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3546
[3]https://ethereum.stackexchange.com/questions/18576/how-time-difference-can-be-calculated-by-block-number-and-how-it-is-different-fr
[4] https://consensys.io/diligence/blog/2019/09/stop-using-solidity s-transfer-now/
[5] https://learn.bybit.com/glossary/definition-gas-limit/
[6] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings. 274–275.
[7] evm.storage. 2024. An Ethereum Virtual Machine Opcodes Interactive Reference. https://www.evm.codes/
[8] BNB Chain community. 2024. About the BEP Category. https://forum.bnbchain. org/t/about-the-bep-category/624
[9]https://etherscan.io/address/0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D#multichain-portfolio
[10]https://ethereum.stackexchange.com/questions/99740/how-to-change-uniswap-router-address-

after-deployment-of-contract
[11]https://stackoverflow.com/questions/78057643/is-there-a-uniswap-v2-router-for-arbitrum-how-to-trade-arbitrum-using-ethers-js
[12] https://github.com/EquivGuard-SC/EquivGuard/blob/main/Experimental_data/statistics.csv