Dear Reviewers,

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

Part 1. Statistics on 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 [12]. 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.

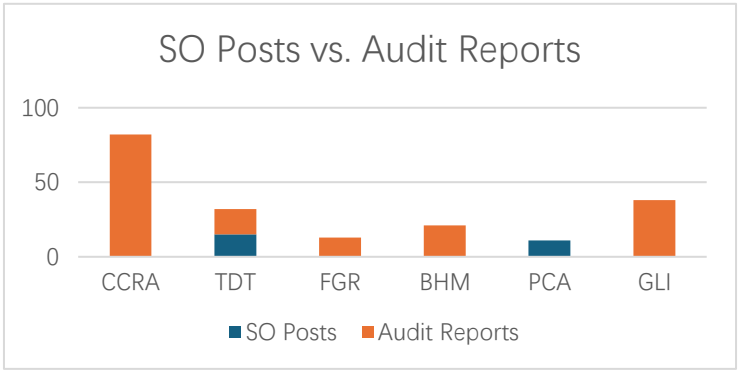

Figure 1: Comparison of Different Defect Information Sources

Figure 1 compares the sources of information for different defect types, showing their distribution across Stack Overflow (SO) posts and audit reports. Cross-Chain 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

Many developers prefer using block.number over block.timestamp to avoid miner manipulation attacks and front-running vulnerabilities, as block.number represents the current block number and does not rely on timestamps, enhancing the security of Solidity contracts. Additionally, estimating time with block.number is a common practice for implementing time

locks, conditional triggers, and reward distributions in smart contracts. While this method simplifies time estimation, it can lead to Time Discrepancy Trap (TDT) defects, for which automated analysis tools are sorely needed.

(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].

Part 2.2 Risk of Gas Limit Settings Being Too High or Too Low

Reusing smart contracts with a fixed Gas Limit across different blockchains can introduce Gas Limit Issues (GLI) due to varying gas costs, risking setting the limit too high or too low. Considering the volatility of gas costs, especially on Ethereum, ConsenSys recommends avoiding the use of transfer()/send() methods that forward a fixed gas amount (2300) and suggests switching to call() to mitigate these risks. Moreover, reusing contracts without accounting for the specific gas costs of each blockchain increases the likelihood of GLI attacks, underscoring the need for careful consideration of gas limits to ensure contract security and functionality.

(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 Mitigations

During the static analysis phase, EquivGuard utilizes pattern recognition technology based on Path Protection Technology (PPT) to detect common patterns such as mutexes, aiding in analyzing various mitigation measures and reducing false positives. Based on these preliminary analysis results, step 3 involves symbolic execution verification to confirm the reachability of potential paths, thus enhancing the accuracy of vulnerability detection. As demonstrated by the experiments in RQ2, the results indicate that EquivGuard achieves an overall accuracy rate of 95.29%.

Part 3.2 Higher Gas Limit

As discussed in Part 2.2, setting the gas limit requires considering the specific blockchain's gas costs, as too high a gas limit can easily lead to reentrancy attacks. As outlined in Section 2.1's Gas Mechanism, "Gas cost quantifies the amount of gas needed to perform each operation, e.g., Ethereum's ADD operation consumes 3 gas, according to the Gas cost standard [7]." Many EVM-compatible blockchains have their own gas cost standards and refund mechanisms to offer lower transaction fees, such as BSC's BEP [8]. Therefore, setting a fixed gas limit of 2300 for executing contracts on other EVM-compatible chains increases

the risk of reentrancy vulnerabilities. For example, due to the constant changes in gas costs, the security firm ConsenSys recommends avoiding the use of transfer & send [6].

Part 4. Risk Analysis of Hardcoded Uniswap Router Address

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]).

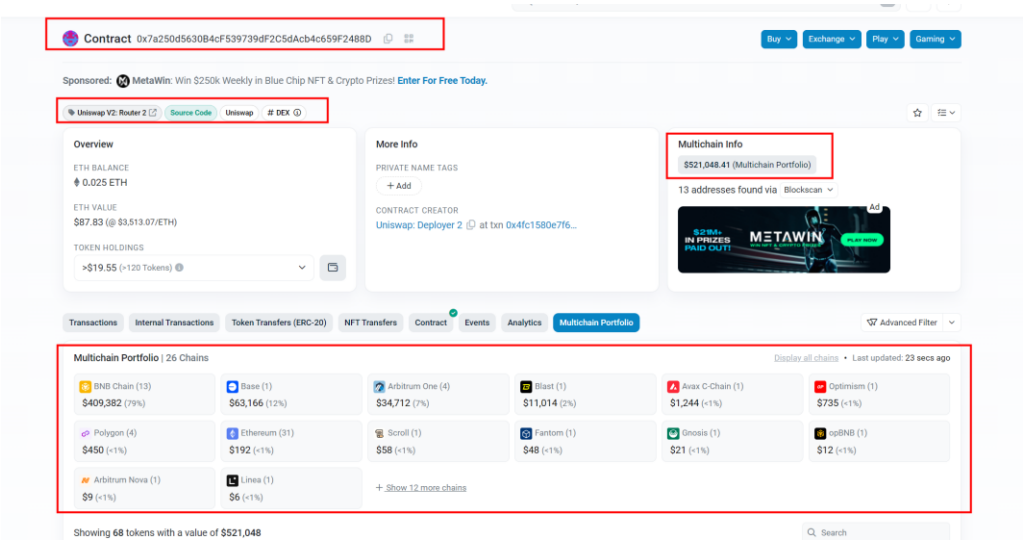

Figure 2 shows that due to PCA defects, assets received on EVM-compatible chains at the Uniswap router address have accumulated to $521,295.

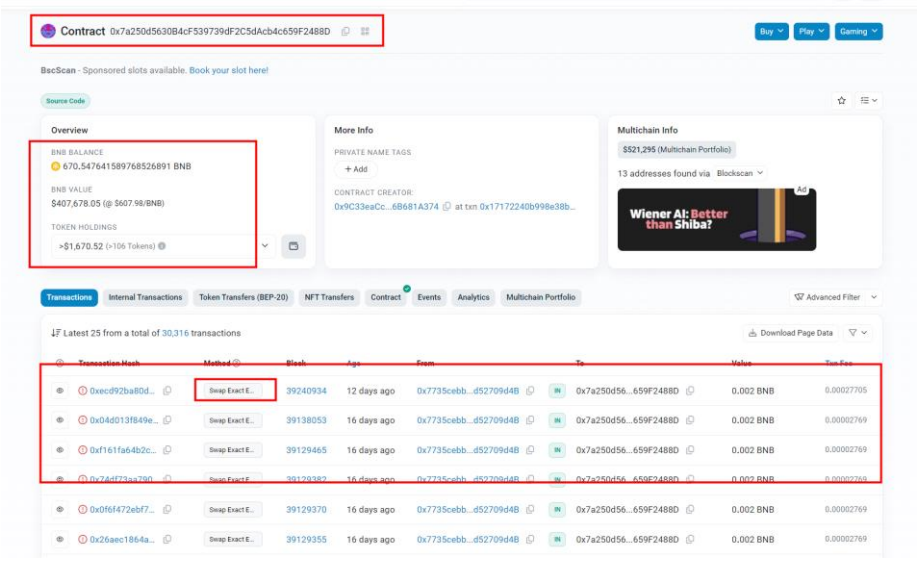

Figure 3 illustrates phishing attacks on the BSC chain caused by the Uniswap router address. Users or developers lacking awareness of PCA defect warnings who call the Swap Exact ETH For

Tokens function, transfer BNB but fail to swap for the corresponding tokens.

Currently, assets affected by the PCA defect from hardcoding the Uniswap router address have reached 26 chains, with the number of impacted assets continuously increasing. Although developers deploy contracts to prevent malicious attacks, the inconsistency of contract ownership across different chains complicates the recovery of affected assets [10]. As the first systematic study of EVM inequivalent defects, EquivGuard tool, designed to address these defect characteristics, enables effective detection of such defects. We have also actively contacted project teams to minimize user asset losses as much as possible.

Part 5. Comparative Analysis with Results from Other Tools

The reason for not including comparative experiments with other tools in this paper is due to the introduction of EVM Inequivalent Defects (EIDs) for the first time. Tools like Mythril and Slither lack the specific functionalities required for analyzing EIDs. To facilitate a horizontal comparison, we conducted comparative experiments of EquivGuard against mainstream smart contract detection tools Mythril and Slither. Since these tools do not directly support the detection of EVM Inequivalent Defects (EIDs), we used the positive data from the EquivGuard experiments in RQ2 as our benchmark data.

Table 2 Comparative Analysis with Slither and Mythril

	PCA	GLI	CCRA	FGR	TDT	BHM
Samples	95	14	88	96	94	57
Positives	95	14	86	91	88	48
Slither	0	0	0	11	0	0
Mythril	0	8	0	0	0	0

Our analysis of Mythril and Slither's official documentation regarding their supported vulnerability checks revealed that these tools currently lack effective analysis for EVM Inequivalent Defects. EquivGuard significantly outperformed Mythril and Slither in detecting various EIDs (PCA, GLI, CCRA, FGR, TDT, BHM), achieving an overall precision rate of over 90%. This superiority is attributed to EquivGuard's defect-specific analysis strategies, combining symbolic execution with taint analysis for effective EIDs detection. Slither identified security issues in FGR samples, mainly related to its reentrancy-unlimited-gas check [13], but its simple

pattern checking approach falls short in complex contexts. In contrast, Mythril detected issues in GLI samples related to SWC-134 [14], which involves fixed gas function calls, covering some GLI defects but failing in other samples. These findings highlight the current mainstream tools' lack of effective EVM Inequivalent Defect analysis, resulting in significantly inferior detection performance compared to EquivGuard. This experiment merely validates Mythril and Slither's capability to detect EVM Inequivalent Vulnerabilities on a basic level, with worse performance expected in large datasets or false positive analysis.

This work is the first systematic study of EVM Inequivalent Defects, introducing EquivGuard for efficient detection. EquivGuard's novelty lies in its combination of taint analysis and symbolic execution to effectively extract and verify key paths for EID analysis. It utilizes the Inter-contract Program Dependency Graph (I-PDG) for analyzing complex state dependencies in interactive contracts. This contributes technically in two main aspects: defining and systematically studying EIDs, and designing EquivGuard, which offers an effective defect detection method with practical applications.

References

[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-soliditys-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
[13] https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
[14] https://swcregistry.io/docs/SWC-134/
[15] https://detectors.auditbase.com/avoid-hardcoded-address-solidity