

Dear Reviewers,

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

Part 1. Toy Contract

To verify whether the high defect count is due to the presence of numerous toy contracts, we analyzed the blockchain's smart contracts and retrieved the transaction data for each contract address. Contracts with **fewer than 5 transactions** were classified as toy contracts. For open-source contracts, we analyzed the lines of code (LOC), excluding blank lines and standard library dependencies (e.g., `stdlib.fc`). Contracts with **fewer than 100 LOC** were also classified as toy contracts. Based on these criteria, we identified **762 (total 1,640) toy contracts**. The detailed data is shown in the **Figure 1** and **Figure 2** respectively, where the boundaries used for classification are highlighted. More than half of the on-chain contracts have executed no more than 5 transactions, and more than one-third of open-source contracts have no more than 100 lines of code. Therefore, it can be concluded that one of the reasons why TONScanner detected a large number of defects is that there are currently a large number of toy contracts on TON.

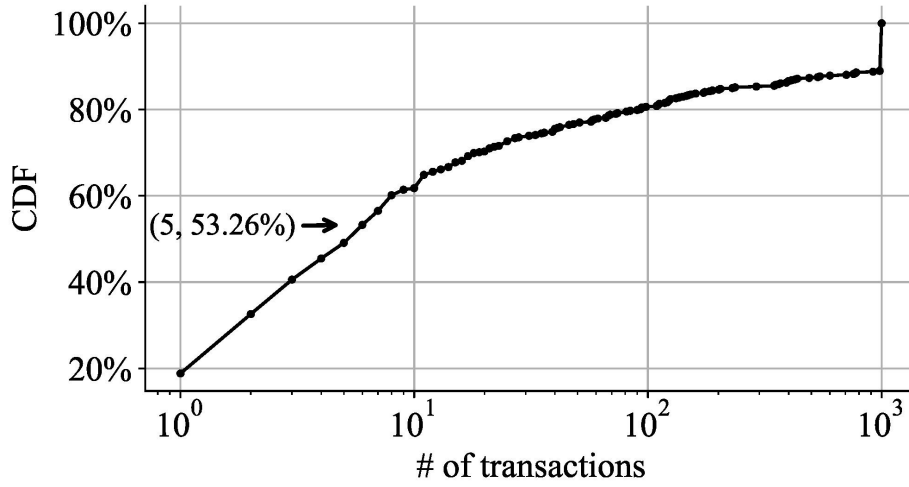


Figure 1: Number of transactions in on-chain contracts

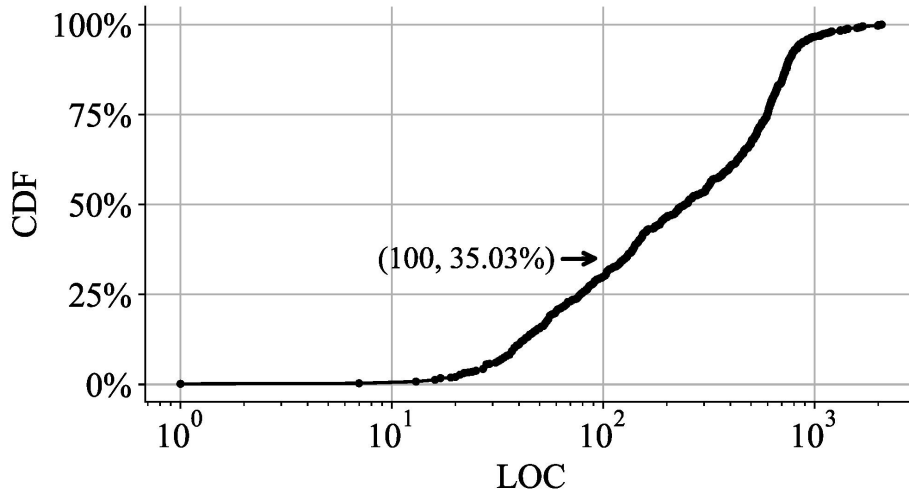


Figure 2: LOC in open-source contracts

Part 2. Case Study of Harm Caused by Defects

We illustrate the potential damage caused by the defined defects and the necessity of our tool for detecting them through a real-world FunC smart contract example.

Contract [EQCajaUU1XXSAjTD-xOV7pE49fGtg4q8kF3ELCOJtGvQFQ2C](#) is the wton token contract, which provides functions such as minting, destruction, deposit and withdrawal. Upon receiving the *op::deposit*

message, the contract initiates the deposit operation (**line 19**). Based on the specified *ton_amount* (TONs, **line 23**) and *jetton_amount* (tokens, **line 24**), the *mint_tokens* function is called to mint new tokens (**line 26**). Then, the *save_data* function updates the *total_supply* of tokens (**line 27**). Finally, the corresponding TONs are deducted, and the remaining TONs are returned to the sender via the *op::excesses* message (non-bounced).



example.js

```
1 () recv_internal(int my_balance, int msg_value, cell in_msg_full, slice
  in_msg_body) impure {
2   if (in_msg_body.slice_empty?()) { ;; ignore empty messages
3     return ();
4   }
5   slice cs = in_msg_full.begin_parse();
6   int flags = cs~load_uint(4);
7
8   if (flags & 1) { ;; ignore all bounced messages
9     return ();
10  }
11  slice sender_address = cs~load_msg_addr();
12  int fwd_fee = cs~load_coins(); ;; we use message fwd_fee for estimation of
  provide_wallet_address cost
13
14  int op = in_msg_body~load_uint(32);
15  int query_id = in_msg_body~load_uint(64);
16
17  (int total_supply, slice admin_address, slice next_admin_address, cell content,
  cell jetton_wallet_code) = load_data();
18
19  if (op == op::deposit) {
20    msg_value -= deposit_gas_consumption();
21    throw_if(74, msg_value < 0);
22
23    int ton_amount = msg_value;
24    int jetton_amount = in_msg_body~load_coins();
25    throw_if(75, ton_amount < jetton_amount);
26    mint_tokens(sender_address, jetton_wallet_code, jetton_amount);
27    save_data(total_supply + jetton_amount, admin_address, next_admin_address,
  content, jetton_wallet_code);
28
29    msg_value -= jetton_amount;
30    if (msg_value > 0) {
31      var msg = begin_cell()
32        .store_uint(0x10, 6) ;; nobounce - int_msg_info$0 ihr_disabled:Bool
  bounce:Bool bounced:Bool src:MsgAddress → 010000
33        .store_slice(sender_address)
34        .store_coins(msg_value)
35        .store_uint(0, 1 + 4 + 4 + 64 + 32 + 1 + 1)
36        .store_uint(op::excesses, 32)
37        .store_uint(query_id, 64);
38      send_raw_message(msg.end_cell(), 1 + 2); ;; pay transfer fees together,
  ignore errors
39    }
40
41    return ();
42  }
43 }
```

Figure 3: `recv_internal` function

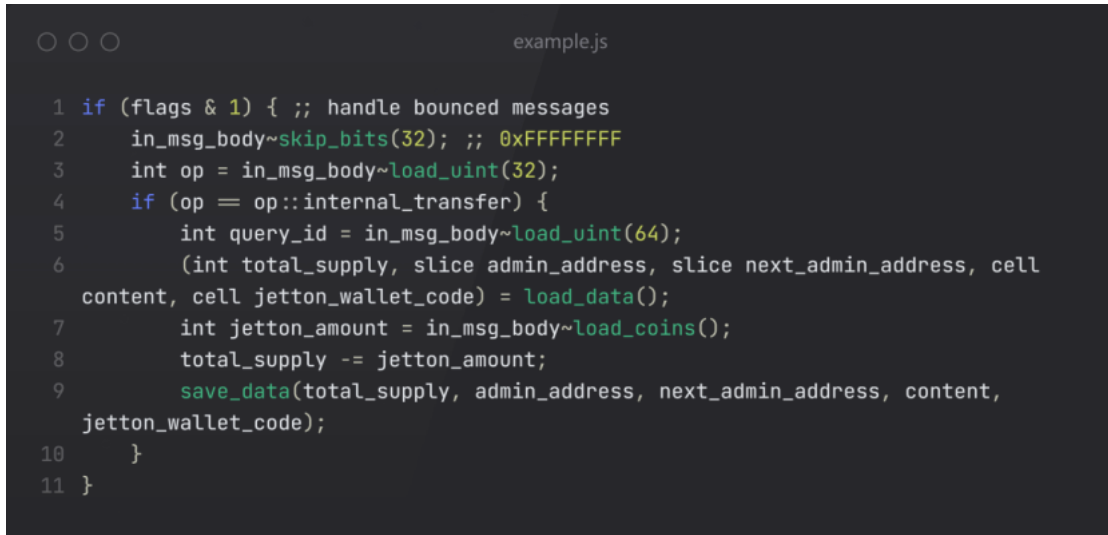
When minting tokens, the minted tokens are transferred to the sender via the *op::internal_transfer* message (**line 5**), which, however, may bounce if an error occurs. If an error occurs during token receipt, an *op::internal_transfer* bounced message will be triggered.



```
1 () mint_tokens(slice to_address, cell jetton_wallet_code, int amount) impure {
2   cell state_init = calculate_jetton_wallet_state_init(to_address, my_address(),
3     jetton_wallet_code);
4   slice to_wallet_address = calculate_jetton_wallet_address(state_init);
5   var master_msg = begin_cell()
6     .store_uint(op::internal_transfer, 32)
7     ;; query_id
8     .store_uint(0, 64)
9     ;; jetton_amount
10    .store_coins(amount)
11    ;; from_address = to_address
12    .store_slice(to_address)
13    ;; response_address
14    .store_slice(to_address)
15    ;; forward ton amount
16    .store_coins(0)
17    .end_cell();
18  var msg = begin_cell()
19    .store_uint(0x18, 6)
20    .store_slice(to_wallet_address)
21    .store_coins(gas_consumption() * 2)
22    .store_uint(4 + 2 + 1, 1 + 4 + 4 + 64 + 32 + 1 + 1 + 1)
23    .store_ref(state_init)
24    .store_ref(master_msg);
25  send_raw_message(msg.end_cell(), 1); ;; pay transfer fees separately, revert on
    errors
26 }
```

Figure 4: mint_tokens function

The contract must handle this to update the *total_supply*, otherwise the circulating tokens will not match the recorded *total_supply*. In **Figure 3**, the *if* branch (**line 8**) ignores all bounced messages. As shown in **Figure 5**, we should add the processing logic for *op::internal_transfer* bounce messages to ensure that *total_supply* is updated correctly.



```

1  if (flags & 1) { ;; handle bounced messages
2      in_msg_body~skip_bits(32); ;; 0xFFFFFFFF
3      int op = in_msg_body~load_uint(32);
4      if (op == op::internal_transfer) {
5          int query_id = in_msg_body~load_uint(64);
6          (int total_supply, slice admin_address, slice next_admin_address, cell
content, cell jetton_wallet_code) = load_data();
7          int jetton_amount = in_msg_body~load_coins();
8          total_supply -= jetton_amount;
9          save_data(total_supply, admin_address, next_admin_address, content,
jetton_wallet_code);
10     }
11 }

```

Figure 5: Code snippet for handling bounced message

Part 3. Card Sorting

We collaborate with a blockchain security company that specializes in smart contract audits and invite experts to join our data analysis effort. A total of four individuals participate in this work: two of the paper’s authors, each with two years of experience in smart contract development, and two professionals specializing in auditing smart contracts.

After security experts manually exclude data unrelated to *FunC* smart contract defects, we use an open card sorting approach[1] to analyze and categorize the official blogs and audit reports. Specifically, we create a card for each blog's tip and report, dividing the content into several sections: title, description, and recommendations (defect types). Two authors collaborate on the analysis and classification, which take place in two rounds. In the first round, we randomly select 40% of the cards. We first read the title and description of each card to understand the associated defects. Then, we review the recommendations to understand how to address the identified defects. Cards without root causes are ignored, and possible defects are categorized. The classification process

is supervised by security experts to ensure accuracy.

In the second round, we independently analyze and categorized the remaining 60% of the cards following the same steps as in the first round. Then, we compare the results and discuss the differences with security experts. These discussions lead to either unification or elimination of discrepancies. Finally, we categorize the defects into eight types.

Figure 6 illustrates an example of a card generated from a smart contract audit report [2]. The card comprises three parts: the finding name (title), description, and recommendation. As noted in the description, in the *vesting-lockup-wallet.fc* smart contract, the function *recv_function()* neither modifies the storage nor throws an exception, yet it is annotated with the *impure* modifier. The audit platform recommends removing the *impure* modifier. Therefore, we categorize this issue as *Improper Function Modifier*.

Improper Usage of impure Modifier				Title
Category	Severity	Location	Status	Description
Language Specific	● Informational	vesting/vesting-lockup-wallet.fc (base): 22-23	☑ Resolved	
The function <code>recv_internal()</code> is defined as <i>impure</i> , but it does not modify the storage or throw an exception.				
We recommend removing the impure modifier in the function definition.				Recommendation

Figure 6: Example of a card of audit reports

Figure 7 illustrates a card example generated from a TON official blog tip [3]. The card comprises three parts: the tip name (title), description, and recommendation. In the description, it is mentioned that the message flow may be constructed incorrectly, requiring the contract to handle bounced messages. For example, if the receiver's wallet contract is unable to accept tokens, the sender's wallet contract must handle the

bounced message. The recommendation similarly suggests processing bounced messages. Therefore, we categorize this issue as *Unchecked Bounced Message*.

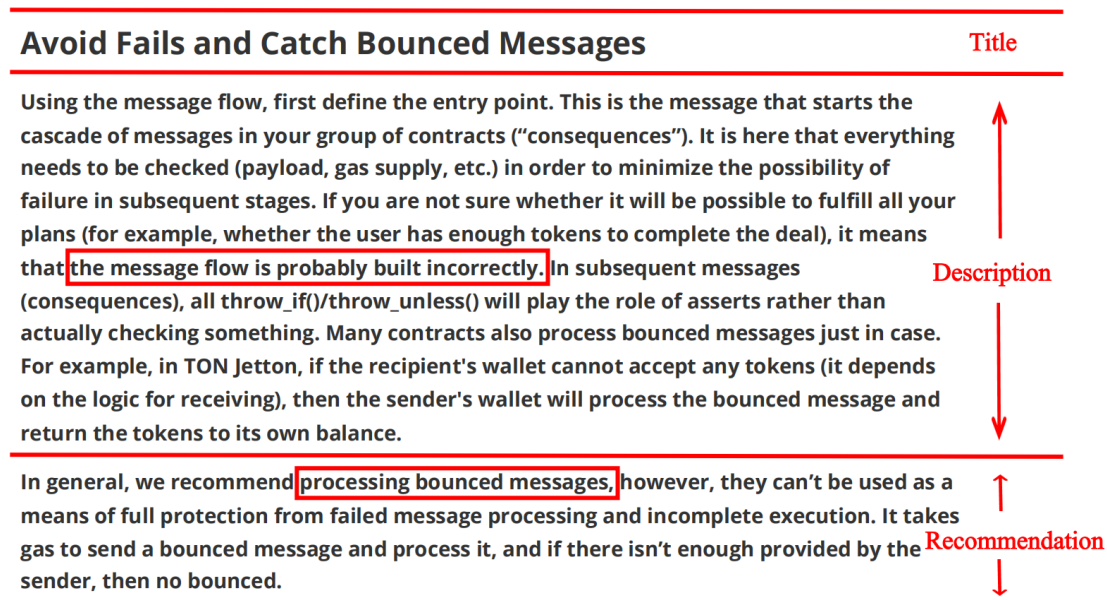


Figure 7: Example of a card of official blogs

[1] D. Spencer, Card Sorting: Designing Usable Categories. Rosenfeld Media, 2009.

[2] "TON Vesting," 2022. [Online]. Available: <https://skynet.certik.com/projects/the-open-network?auditId=TON%20Vesting#code-security>

[3] Certik, "Secure Smart Contract Programming in FunC: Top 10 Tips for TON Developers," 2023. [Online]. Available: <https://blog.ton.org/secure-smart-contract-programming-in-func>

Part 4. Data Comprehensiveness-List of audit firms

Security Assurance Providers(SAP)

! INFO

Test your software with the following quality assurance providers.

Primary TON Blockchain SAP

- certik.com
- quantstamp.com
- ton.tech
- trailofbits.com
- zellic.io

Figure 8: Security Assurance Providers(SAP)

- (1) Certik: <https://www.certik.com/>
- (2) Quantstamp: <https://quantstamp.com/>
- (3) TonTech: <https://ton.tech/>
- (4) Trail of Bits: <https://www.trailofbits.com/>
- (5) Zellic: <https://www.zellic.io/>

Part 5. Defects Differences

In addition to Ethereum, there are many other popular blockchains such as Polygon, BNB Chain, and Solana. Most of these blockchains are EVM-compatible, meaning smart contracts can be written in Solidity, and thus, the types of defects are largely shared. Although Solana introduces some design innovations, such as fee market segregation, and uses Rust for smart contract development, it can still be considered a transaction-synchronized state machine similar to Ethereum. The types of

defects in Solidity-written smart contracts can almost entirely cover these blockchains, making this the most well-researched area. To better illustrate the defects in Func, we will compare the differences between Func and Solidity, as shown in **Table 1**. We will also include relevant explanations to facilitate a better understanding.

Table 1: Differences between Func and Solidity

Defect Type	Defect abbreviation	Differences
Bad Randomness	BR	This defect is identical to SWC-120[1], where an insecure source of randomness is used, such as miner-controlled fields.
Precision Loss	PL	This defect is commonly found in smart contracts on other blockchain platforms[2].
Unchecked Return	UR	This defect is identical to SWC-104[3], where the return value of a message call is not checked.
Global Var Redefined	GVR	Since Func does not include pointers and references, information is passed through global variables. This defect arises as a result of this design choice. (Unique to Func) [4]
Improper Function Modifier	IFM	In Func, the role of function modifiers is the opposite of that in Solidity. Func requires adding a function modifier for stateful writes, while Solidity requires a function modifier for stateless writes. (some similarities as well) [5]
Unchecked Bounced Message	UBM	TON is designed as an asynchronous blockchain, where interactions between contracts are conducted through messages. This defect arises from this design choice. (Unique to Func) [6]
Inconsistent Data	ID	To make message transmission more efficient, messages are compressed using the cell data structure. This defect arises during the parsing of cells. (Unique to Func) [7]
Lack End Parse	LEP	This defect similarly arises from the cell-based design. (Unique to Func) [7]

As shown in **Table 1**, Bad Randomness (BR), Precision Loss (PL), and Unchecked Return (UR) are common defects found in other blockchains, while Global Var Redefined (GVR), Improper Function Modifier (IFM), Unchecked Bounced Message (UBM), Inconsistent Data (ID), and Lack End Parse (LEP) are unique to TON due to its design.

[1] "Weak Sources of Randomness from Chain Attributes," 2020. [Online]. Available:

<https://swcregistry.io/docs/SWC-120/>

[2] S. Song, J. Chen, T. Chen, X. Luo, T. Li, W. Yang, L. Wang, W. Zhang, F. Luo, Z. He, et al., "Empirical study of Move smart contract security: Introducing MoveScan for enhanced analysis," in Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1682-1694.

[3] "Weak Sources of Randomness from Chain Attributes," 2020. [Online]. Available: <https://swcregistry.io/docs/SWC-104/>

[4] SwiftAdviser, "Global variables," 2023. [Online]. Available: https://docs.ton.org/develop/func/global_variables

[5] AlexG, "Functions," 2023. [Online]. Available: <https://docs.ton.org/develop/func/functions>

[6] AlexG, "Sending messages," 2024. [Online]. Available: <https://docs.ton.org/develop/smart-contracts/messages>

[7] B. Mehdi, "Cells as data storage," 2023. [Online]. Available: <https://docs.ton.org/learn/overviews/cells>

Part 6. CDF Distribution

The CDF in **Figure 9** illustrates the distribution of defects detected

per contract. Approximately **75% of contracts** have fewer than 10 defects, while the remaining **25%** have more, with some exceeding 100 defects. Further analysis reveals that the majority of these higher defect counts are primarily due to *UR*.

Figure 10 also provides a detailed view of the defect distribution within specific intervals (e.g., the percentage of each defect type in contracts with 1-5 defects). **The data shows that *UR* and *LEP* constitute the largest proportions overall.** For contracts with fewer than 100 defects, as the defect count rises, the proportion of *UR* increases, while the proportion of *LEP* declines. This suggests that *UR* contributes significantly to the total number of defects and shows a rising trend, while other defect types remain relatively stable in number.

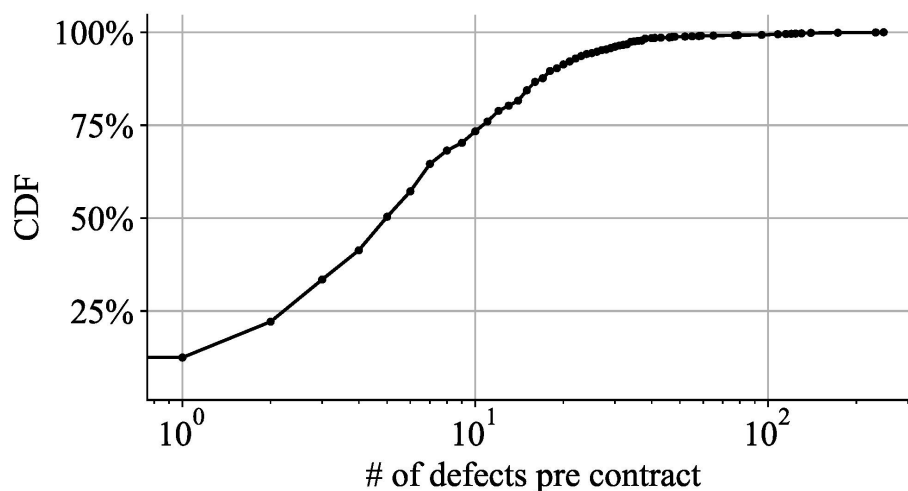


Figure 9: Number of defects pre contract

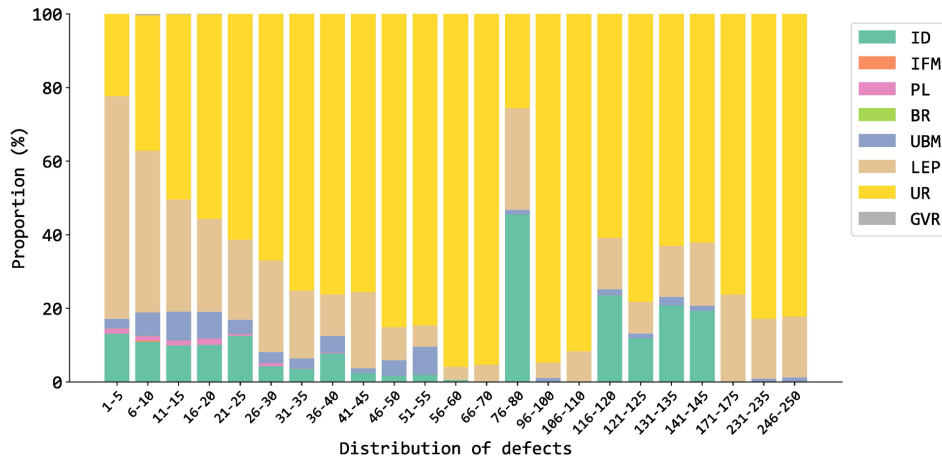


Figure 10: Defect distribution changes with the number of defects in each contract

Part 7. Case Studies of FP and FN Instance

To help clarify the limitations of the proposed tool, we conducted case studies on the FP instance of *UBM* and the FN instance of *LEP*.

The code snippet in **Figure 11** is taken from contract *EQCPXKOygMDIuWI6vnSBt498PwHAwaatg5yLYC7TitxLLQbr*. The main function of this code snippet is to extract the necessary information from the incoming message (such as the destination address, sender information, and message content), then construct and forward (proxy) the message. However, it is flagged by TONScanner as having a *UBM* defect, which is an FP.

The main reason for this issue is as follows. To distinguish different messages, different *op* codes are used, and the recipient performs different operations depending on the value of the *op* (**line 1**). When conducting *UBM* defect detection, it is necessary to identify the *op*. However, in this case, the condition *if(op == op::proxy_send)* imposes a constraint on the *op* within the if-branch. This limits the flexibility of the

implementation, as it can only send messages that match the *op* value.

In the case of FPs in *UBM* defect detection, to achieve the desired constraint functionality perfectly, it requires the use of symbolic execution techniques. But writing a symbolic execution engine for FunC from scratch would involve a significant amount of work. We will address this issue in future work.



```
1 if (op == op::proxy_send) {
2   slice to = in_msg_body~load_msg_addr();
3   cell body = in_msg_body~load_ref();
4   var msg = begin_cell()
5     .store_uint(0x18, 6)
6     .store_slice(to)
7     .store_coins(0)
8     .store_uint(0, 1 + 4 + 4 + 64 + 32 + 1 + 1)
9     .store_uint(op, 32)
10    .store_uint(query_id, 64)
11    .store_slice(sender_address)
12    .store_ref(body);
13  send_raw_message(msg.end_cell(), 64); ;; proxy all input coins
14  return ();
15 }
```

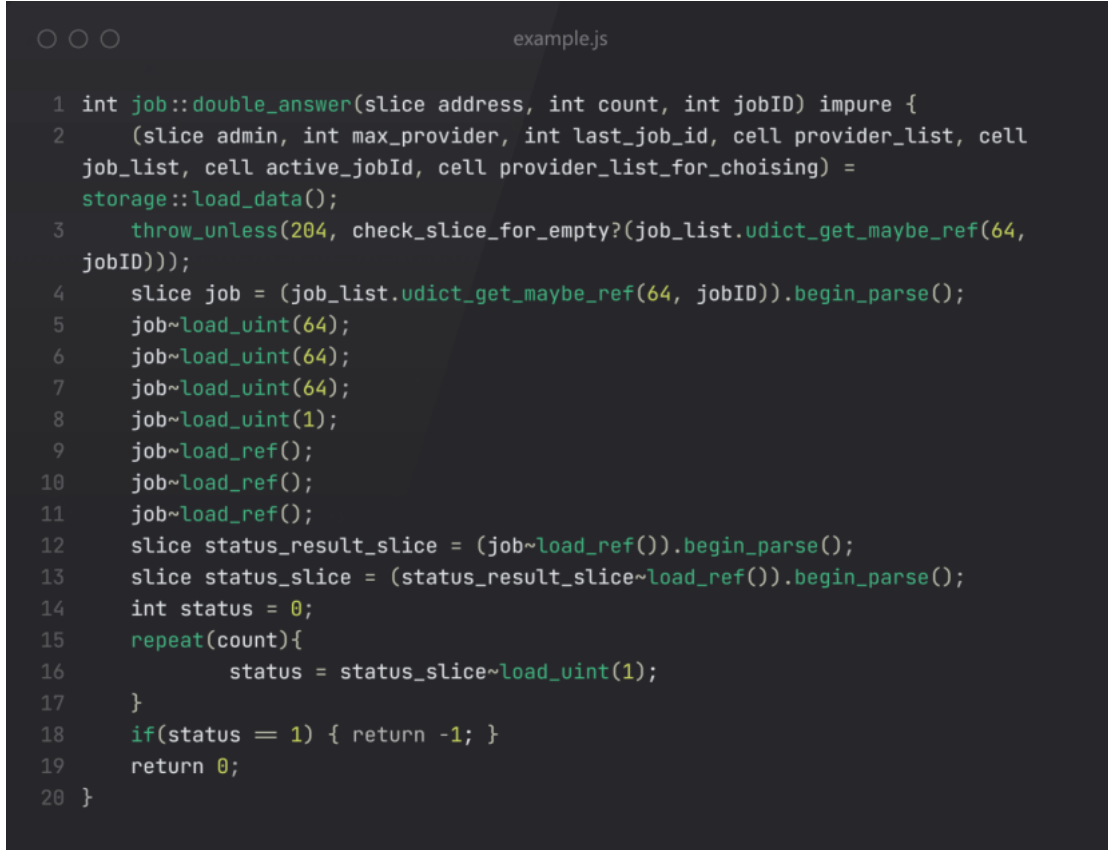
Figure 11: FP instance of UBM defect

The code snippet in **Figure 12** is taken from [repo/ton-link/ton-link-contract-v3/typescript/source/NativeOracle](https://github.com/ton-link/ton-link-contract-v3/typescript/source/NativeOracle). This *Func* code primarily performs the parsing of a job task and status checking. The *end_parse* function is missing, but TONScanner encountered an FN issue.

The missed *LEP* defects occur because when a cell is used within a loop, the number of iterations cannot be determined, leading to inaccurate cell analysis. As a result, it is impossible to identify the position of the *end_parse*.

In the case of FNs in *LEP* defect detection, the *status_slice* performs *loads_uint* within a *repeat* loop that appears in the CFG. Due to the

inability to pinpoint the exact location where *end_parse* should occur, an FN is generated.



```

1  int job::double_answer(slice address, int count, int jobID) impure {
2      (slice admin, int max_provider, int last_job_id, cell provider_list, cell
        job_list, cell active_jobId, cell provider_list_for_choising) =
        storage::load_data();
3      throw_unless(204, check_slice_for_empty?(job_list.udict_get_maybe_ref(64,
        jobID)));
4      slice job = (job_list.udict_get_maybe_ref(64, jobID)).begin_parse();
5      job~load_uint(64);
6      job~load_uint(64);
7      job~load_uint(64);
8      job~load_uint(1);
9      job~load_ref();
10     job~load_ref();
11     job~load_ref();
12     slice status_result_slice = (job~load_ref()).begin_parse();
13     slice status_slice = (status_result_slice~load_ref()).begin_parse();
14     int status = 0;
15     repeat(count){
16         status = status_slice~load_uint(1);
17     }
18     if(status == 1) { return -1; }
19     return 0;
20 }

```

Figure 12: FN instance of LEP defect

Part 8. Potential Solutions

To assist developers in writing secure TON smart contracts, we not only design TONScanner to detect potential defects but also provide possible defect solutions to help developers avoid the defined defects. **Table 2** outlines a brief overview of the possible solutions for each defect.

Table 2: Possible Solutions for the 8 Defects

Defect Type	Defect abbreviation	Possible Solution
Bad Randomness	BR	Use external sources of randomness via oracles.
Precision Loss	PL	Allow multiplication to be executed before division.

Unchecked Return	UR	Check return values every time.
Global Var Redefined	GVR	Avoid defining variables with the same names as global variables.
Improper Function Modifier	IFM	Add the <i>impure</i> modifier when a function has side effects; otherwise, remove it.
Unchecked Bounced Message	UBM	Catch and process bounced message.
Inconsistent Data	ID	Ensure that the order and type of read/write variables are consistent each time.
Lack End Parse	LEP	Include the <i>end_parse</i> function after each slice read.

For most defects, such as *UR*, *GVR*, *IFM*, *UBM*, *ID*, and *LEP*, developers can avoid them by paying close attention during the programming phase. We believe that, through our work, future developers will become more aware of the defined defects they may encounter during the development process.

For *PL*, developers can allow multiplication to be executed before division. For *BR*, TON officially recommends skipping a block before generating a random number, which alters the random seed in a less predictable way [1]. However, this approach still cannot eliminate the influence of miners. We suggest using an oracle to obtain the random seed for better unpredictability.

[1] TonSquare, "Random number generation," 2024. [Online]. Available: <https://docs.ton.org/mandarin/develop/smart-contracts/guidelines/random-number-generation>