Understanding and Characterizing Obfuscated Funds Transfers in Ethereum Smart Contracts

PAPER #75, 32 PAGES

Scam contracts on Ethereum have rapidly evolved alongside the rise of DeFi and NFT ecosystems, utilizing increasingly complex code obfuscation techniques to avoid early detection. This paper systematically examines how obfuscation exacerbates the financial risks associated with fraudulent contracts and undermines existing auditing tools. We propose a transfer-centric obfuscation taxonomy, distilling seven key features, and design ObfProbe, a framework that performs bytecode-level smart contract analysis to uncover obfuscation techniques and quantify the level of obfuscation complexity via Z-score ranking. In a large-scale study of 1.04 million Ethereum contracts, we isolate over 3,000 highly-obfuscated contracts and, in manual case studies, identify four patterns: MEV bots, Ponzi schemes, fake decentralization, and extreme centralization that are deeply coupled with various obfuscation maneuvers, including assembly usage, dead code, and deep function splitting. We further reveal that obfuscation substantially increases the scale of financial damage and the time required for evasion. Finally, we evaluate SourceP, a state-of-the-art Ponzi detection tool, on both obfuscated and non-obfuscated samples, observing its accuracy to drastically fall from 79% (non-obfuscated) to 12% (obfuscated) in real-world scenarios. These findings underscore the urgent need for enhanced "anti-obfuscation" analysis techniques and broader community collaboration to mitigate the proliferation of scam contracts in the expanding DeFi ecosystem.

1 INTRODUCTION

Smart contracts, which are self-executing programs deployed on blockchains, have been widely adopted recently. It has enabled various significant emerging applications, such as decentralized finance [8, 13, 70] and digital art trading [52]. Along with it, security issues are also on the rise. Recent reports [27, 74] show that malicious smart contracts, including scams and MEV bots, have become increasingly prevalent, resulting in significant financial losses. To address these significant threats, researchers have proposed various techniques to detect [54, 55, 66] and analyze [24, 60] these emerging security issues. Many of the techniques rely on static program analysis [28] and rule-based matching [3], which have been proven highly effective and efficient in detecting early malicious smart contracts, typically straightforward and easily identifiable, such as transfers to externally owned private account addresses.

As this arms race continues, malicious smart contracts are gradually replaced by more covert, complex, and obfuscated contract logic [1]. Recent studies [75, 79] confirm that obfuscation has become the primary means by which attackers conceal malicious transfer or backdoor control logic, which includes the use of assembly code, splitting functions, redundant instructions. When attackers employ such obfuscation techniques in their smart contracts, the static analysis and matching rules adopted by traditional detection tools are often disrupted, leading to high detection inaccuracies [59, 80] and further exacerbating financial losses [25, 71, 73, 81]. Moreover, obfuscation techniques are not only found in malicious contracts but also in benign contracts for various reasons (e.g., to protect proprietary business logic and deter copycat attacks), which also hinders security analysis and prevents users from better understanding the behaviors of the contracts.

Although obfuscation techniques are increasingly employed in smart contracts and have been linked to substantial financial losses, no comprehensive study has yet been conducted to systematically assess their impact in real-world scenarios — an understanding that is critical for developing effective defense mechanisms and mitigating associated security risks. In this paper, we aim to conduct the first systematic study on the obfuscation of funds transfer operations, which are the most essential and security-critical activities of a smart contract [47]. In particular, we aim to

thoroughly understand the status quo of obfuscated funds transfer operations in Ethereum smart contracts by answering the following four essential research questions:

- **RQ1 Definition:** What code obfuscation techniques are used for funds transfers in smart contracts, and how can they be defined and quantified?
- **RQ2 Prevalence:** How prevalent are obfuscated funds transfers in the real world, and what is the current trend regarding the use of obfuscation techniques?
- **RQ3 Financial Impact:** What are the consequences of using these techniques in malicious smart contracts with respect to economic impact?
- **RQ4 Impact on Malware Analysis:** Can state-of-the-art malicious contract detection tools maintain the same level of effectiveness when faced with heavily obfuscated funds transfers?

To answer these research questions, we develop a taxonomy to define and characterize different obfuscation techniques on funds transfer operations. Specifically, we propose seven robust features by systematically dissecting and examining the formal definition of funds transfer operations within the Ethereum virtual machine to ensure the comprehensiveness and the representativeness of the feature list. We quantify the obfuscation complexity of funds transfer operations in smart contracts by a Z-score representation model. We then propose ObfProbe, an EVM bytecode analysis framework that can accurately uncover different obfuscation techniques employed in real-world smart contracts. With the aid of this analysis framework, we conduct a series of studies on real-world smart contracts, for both malicious and benign usages of obfuscation techniques in funds transfer operations. Below, we highlight some interesting discoveries:

(1) By analyzing the top 3000 highly obfuscated contracts detected by ObfProbe, we found 463 contracts that exhibit substantial security risks, placing funds totaling ≈$100 million at risk.
(2) Compared to non-obfuscated scam contracts, obfuscated scam contracts demonstrate a significantly larger financial impact, with their highest recorded inbound funds being ≈2.4X higher and clear periods of intensified victimization occurring between 2019 and 2023.
(3) The evaluation of a state-of-the-art Ponzi detector on obfuscated scam contracts shows its accuracy dropped from 79% to 12%, implying that obfuscations can significantly undermine the performance of existing detection tools.

Contributions. The contributions of this paper are summarized as follows:

- We developed ObfProbe, the first EVM bytecode obfuscation analyzer that leverages seven bytecode-level features and a Z-score representation model to automatically detect obfuscated transfer logic from smart contracts.
- We conducted a large-scale measurement on more than 1.04 million Ethereum smart contracts, revealing the common obfuscation patterns in the top 3000 contracts, which include four types of contracts: MEV bots, Ponzi schemes, fake decentralization, and extreme centralization.
- We quantified the impact of obfuscation on financial damage and detection: obfuscated contracts can extract up to 201.74 ETH and trigger significant victim outbreaks, while state-of-the-art detectors suffer a decrease in accuracy from 79% to below 12% under deep obfuscation.
- We open-sourced ObfProbe and the collected data/artifacts to facilitate future research[1].

2 BACKGROUND

2.1 Blockchain and Smart Contracts

Blockchain, a technology with a decentralized distributed ledger at its core, ensures data security and immutability through cryptographic methods, making it a foundational infrastructure for

[1]https://github.com/nonname-byte/Obfuscation_Tool

various data transactions [4]. Its core features—decentralization, transparency, immutability, and security—position it as a transformative technology across a wide range of industries [67].

Smart contracts are programs built on top of blockchains that autonomously execute contractual terms when predefined conditions are satisfied, eliminating the need for intermediaries [51]. Smart contracts offer several key advantages, including reduced transaction costs, enhanced efficiency, and a lower risk of human error. By permanently recording execution results and data on the blockchain, they eliminate the possibility of unilateral alteration, thereby ensuring fairness and transparency in contract execution [53]. Smart contracts have been widely applied in various fields, including financial transactions, identity verification, supply chain management, and insurance [61]. In the financial sector, smart contracts enable the automatic settlement and payment on the blockchain, significantly enhancing transaction efficiency and transparency while reducing the need for intermediaries and human intervention. As blockchain technology evolves, smart contracts will play an increasingly important role across multiple industries [67].

2.2 Code Obfuscations

Code obfuscation transforms a program into a semantics-preserving but harder-to-analyze form, used both for software protection and for malware evasion to raise reverse-engineering cost. A standard taxonomy groups techniques into control-flow, data, layout (lexical), and instruction-substitution transformations [11]. Control-flow obfuscation perturbs the CFG (e.g., opaque predicates, bogus branches, flattening) and even virtualization to hinder static/dynamic analyses [35, 57]. Data obfuscation hides computations via variable encoding and mixed Boolean–arithmetic (MBA) rewriting; layout changes rename/reorder identifiers or inject dead/NOP code; instruction substitution replaces code with equivalent forms (e.g., shifts/adds for multiply), which diversifies byte patterns and weakens signature-based detection [11, 38, 57].

3 TAXONOMY OF OBFUSCATED FUNDS TRANSFERS

Our study focuses on funds transfer operations, which are the most essential and security-critical activities of a smart contract [47]. To answer our research questions, we develop a taxonomy to define and characterize different obfuscation techniques on funds transfer operations, based on how each component of funds transfer operations can be hidden, derived from the formal definition of funds transfer operations within the Ethereum virtual machine to ensure the comprehensiveness and the representativeness of our taxonomy.

We start by dissecting every element of the funds transfer operation. The standard way to realize such an operation is through a *transfer* API CALL in Solidity [64], which is implemented as a *CALL* EVM opcode in a given smart contract bytecode. Specifically, we define a *CALL* EVM opcode that implements a funds transfer operation as follows:

Definition 1. A *CALL* EVM opcode that implements a funds transfer operation can be defined as

$$T = (addr, value, context, log), where : \tag{1}$$

• *addr* determines the recipient address of the fund. This is the target address specified in the `CALL` instruction (a 20-byte Ethereum address).

• *value* is the non-zero value in wei transferred to the `addr`. This value represents the amount of native Ethereum tokens sent in the transfer.

• *context* is the execution context of the funds transfer operation, which includes the storage state of the contract, the remaining instructions, and control/data flow inside the function where the transfer is located. This information together determines whether and how the `CALL` instruction can be executed.

• *log* refers to the collection of events generated by the CALL operation during the transaction execution. This includes event signatures (topics) and data fields (data), which provide semantic information regarding the transfer and are useful for auditing and monitoring on-chain activities.

Based on our definition of a funds transfer operation, we investigate each element and derive seven obfuscation features. By exhaustively mapping each feature to one or more elements of a funds transfer operation, we ensure our taxonomy is comprehensive and covers all fundamental methods of hiding transfer operations in Ethereum smart contracts. Please note that our analysis is conducted at the bytecode level, and the source code listings below are provided for illustrative purposes.

3.1 Obfuscation of addr

The addr element is essentially a string that represents a 20-byte Ethereum address, indicating the recipient address of a funds transfer operation. We consider four obfuscation methods that stem from traditional string obfuscation techniques [50].

T1. Multi-step Address Generation. The address is derived through a sequence of external reads, arithmetic/bitwise operations, or import from another contract, preventing straightforward identification of the actual 20-byte recipient.

```
// Step 1: derive seed from block data
bytes32 seed = keccak256(...);
// Step 2: extract intermediate bytes
bytes20 part = bytes20(seed);
// Step n......
// compute recipient address
address rec = address(...(uint256(part)));
// core transfer
rec.transfer{value,...}("");
```

Listing 1. T1 Multi-step Address Generation

T2. Complex String Operations. The address is split into multiple substrings or byte segments stored separately. These segments are then concatenated at runtime to reconstruct the true addr, concealing it from static parsers.

```
// split address string into parts
string memory s1 = "0x";
string memory s2 = "a1b2c3";
string memory s3 = "d4e5f6";
// concatenate at runtime
string memory full = string(s1, s2, s3);
// parse back to address
address rec = parseAddr(full);
rec.transfer{value,...}("");
```

Listing 2. T2 Complex String Operations

T3. External Contract Calls. Instead of local computation, the contract with this obfuscation technique may choose to query a "router" or "delegate" contract to fetch addr, hiding the true recipient behind an external CALL.

```
interface AddrPro {
    function getAddr() external returns (address);
}
```

```
// fetch hidden address from another contract
address provider = 0x1234...;
address rec = AddrPro(provider).getAddr();
rec.transfer{value: value}("");
```

Listing 3. T3 External Contract Calls

T4. Control-flow complexity. The branch selection is dependent on run-time conditions (e.g., `block.timestamp`). Some branches are dummy and never execute, and different branches point to different `addr`s. This hides the real recipient because the true branch is known only at execution time, which hinders static rule matching.

```
address rec;
if (block.timestamp % 3 == 0) {
    for (uint i = 0; i < 2; i++) {
        if (i == 1) {
            if (msg.sender == owner)
                rec = addrA;
            else
                rec = addrB;
        }
    }
} else
    rec = addrC;
rec.transfer{value: value}("");
```

Listing 4. T4 Control-flow Complexity

3.2 Obfuscation of `value`

Since `value` is the amount transferred in wei, i.e., a 256-bit unsigned integer. Therefore, two obfuscation strategies from `addr` can also be applied.

T3. External Contract Calls. Similar to `addr`, the transfer amount can also be fetched from an external contract, rather than stored locally, complicating static quantity analysis.

T4. Control-flow complexity. `value` is determined by conditional logic or loops, introducing multiple potential numbers and obscuring the true transfer amount.

3.3 Obfuscation of `context`

The `context` element comprises (i) the contract's storage state, (ii) the internal instructions and control/data flow of the function in which the funds transfer operation resides. Smart contract developers can choose to obfuscate `context` to cloak the real intent of funds transfer operations. We outline two unique techniques below to obfuscate `context`.

T5. Camouflage Instructions. By injecting large numbers of meaningless loops, arithmetic operations, and `NOP`s into the transfer-related function body, the core `CALL` is camouflaged with many irrelevant instructions. Although the transfer is still executed correctly, the altered control and data flow within the function make it hard for static analyzers to isolate the actual context.

```
// meaningless loop
for (uint i = 0; i < 5; i++)
    uint tmp = i * 42;
// no-op arithmetic
uint x = (1 + 2) - 3;
```

```
// core transfer hidden among noise
address rec = 0xAbCd...;
rec.transfer{value,...}("");
```

Listing 5. T5 Camouflage Instructions

T6. Replicated Transfer Logic. Smart contract developers can duplicate identical or highly similar transfer logics across multiple functions that only differ in names (e.g., `withdraw` and `start`) or trivial execution paths. Hence, the contract selector randomly dispatches the CALL at runtime. This multiplies potential entry points and confuses analysis tools regarding which function's context truly carries out the transfer.

```
function withdraw(uint value) public {
    _doTransfer(value);
}
function start(uint value) public {
    _doTransfer(value);
}
function _doTransfer(uint value) internal {
    // same transfer code reused
    address rec = 0xAbCd...;
    rec.transfer{value,...}("");
}
```

Listing 6. T6 Replicated Transfer Logic

3.4 Obfuscation of log

Finally, `log` keeps a record and provides a semantic-level understanding of what has happened during the execution. Developers can choose to obfuscate the semantic signals in `Log` with the following technique.

T7. Irrelevant Log Events. One can emit misleading or unrelated events (e.g., logging a transfer to a "legitimate" address while sending funds elsewhere), diverting auditors' and tools' attention, and concealing the real log data that corresponds to the transfer.

```
event Info(string msg);
// misleading log before transfer
emit Info("Sending to safe address");
address rec = 0xAbCd...;//unsafe address
rec.transfer{value,...}("");
```

Listing 7. T7 Irrelevant Log Events

Answer to RQ1. We answer RQ1 by formalizing the EVM transfer path, deriving seven obfuscation *patterns* (T1–T7) and mapping each to measurable bytecode features (F1–F7). This taxonomy operationalizes transfer obfuscation and provides the quantitative basis used by our Z-score model in Section 4.

4 SYSTEM DESIGN AND IMPLEMENTATION

Building on the taxonomy, we design and implement an EVM bytecode analysis tool named ObfProbe to uncover how different obfuscation techniques manifest in real-world smart contracts, thereby answering RQ2.

4.1 System Overview

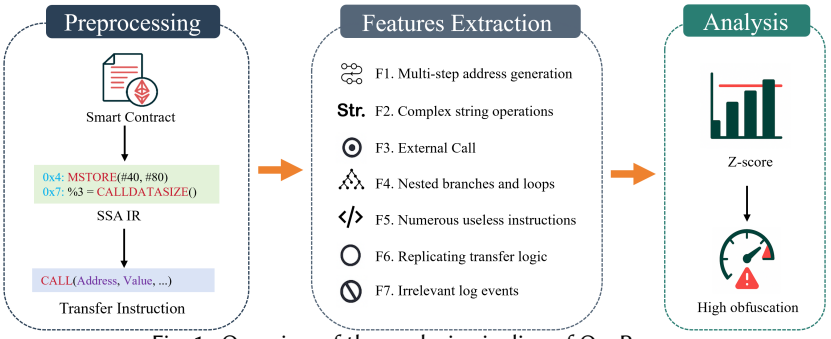


Fig. 1. Overview of the analysis pipeline of ObfProbe.

Figure 1 shows an overview of ObfProbe. At a high level, it converts a given smart contract bytecode to the static single assignment intermediate representation (SSA IR) using an existing tool named Rattle [16]. After that, it scans the IR to detect all fund transfer operations. For each transfer, our system extracts seven pre-defined obfuscation features. Finally, it applies a Z-score representation model [17], which is a numerical value that represents a data point's distance from the mean in terms of standard deviations, to convert the extracted features into an obfuscation score that indicates the degree of complexity in the obfuscation applied to the transfer operation.

Table 1. Summary of transfer-related obfuscation features.

Obfuscation strategies	Extracted features
T1. Multi-step address generation	F1. Number of steps in the `addr` generation.
T2. Complex string operations	F2. Number of string operations in the `addr` generation.
T3. External contract calls	F3. Presence of an external contract call in the `addr`/`value` derivation.
T4. Control-flow complexity	F4. Max branch/loop nesting depth along `addr`/`value` derivation.
T5. Camouflage instructions	F5. Transfer-related Instruction Ratio (TIR).
T6. Replicated transfer logic	F6. Inter-function similarity among transfer-containing functions.
T7. Irrelevant log events	F7. Semantic relevance between log events and the transfer operation.

4.2 Definition and Extraction of Obfuscation Features

For each of the aforementioned seven obfuscation strategies, we define a corresponding obfuscation feature that can be extracted by ObfProbe, as summarized in Table 1.

F1. Number of steps in address generation. This numerical feature represents the number of steps required to obtain `addr` in a transfer operation. Starting from each `CALL` operation in the contract's SSA IR, we perform backward dataflow analysis on the parameter `address` to trace its generation process. Each arithmetic operation, hash function invocation, bitwise manipulation, and external call is counted as one distinct step. Finally, we consolidate linear operations within each basic block to avoid overcounting trivial operations and count the number of steps for address generation as a numerical feature F1.

F2. Number of string operations. String operations (e.g., concatenation, hashing, slicing, and encoding) contribute to obfuscation. To quantify the level of complexity in string operations involved in the address generation process, we reanalyze the data flow of the parameter `addr` to count all instructions that involve string manipulations, including both built-in string operations

and hash operations. We then count the number of string operations for address generation as a numerical feature F2.

F3. Presence of external call. To determine if the generations of `addr` and `value` involve any external call, we inspect the dataflow of both and check if there exists any external call (i.e.,`CALL`), we set a binary feature F3 to `TRUE`, otherwise, to `FALSE`.

F4. Height of the branch tree. We use the height of the branch tree as a numerical feature to represent the complexity of the control flow associated with the transfer operation by analyzing conditional branch structures (`JUMPI` instructions). Starting from the transfer operation, we backtrace to traverse all conditional branches. The height of the branch tree is calculated as the maximum nesting depth traversed.

F5. Transfer-related instruction ratio (TIR). This feature quantifies the ratio of effective instructions contributing to a transfer operation in the residing function. To this end, we define *Transfer-related Instruction Ratio* (TIR) as: $\text{TIR} = \frac{|U|}{|\text{T}|}$, where, $|U|$ is the number of *effective* instructions contributed to transfers and critical state updates, and $|\text{T}|$ is the total number of instructions in the transfer-residing function. Specifically, U is computed as the union of critical transfer instructions C and state modifications S.

We obtain C from the function's program dependence graph (PDG) by considering all instructions directly related to the transfer logic (e.g., `CALL`, parameter calculations, state validation). S is from parsing the function to detect `SSTORE` operations and retaining only those updating parameters related to transfers (e.g., `address`, `value`). Consequently, a low TIR implies that a large portion of the code in a transfer-residing function does not affect how the transfer operation is performed, indicating significant obfuscation designed to dilute and hide the contract's true transfer logic.

F6. Transfer operation similarity. This feature quantifies whether a contract replicates the same transfer logic across multiple functions. To calculate this, we construct a PDG that includes information from both control and data dependencies for each function containing a transfer operation. Then, we embed the nodes of each PDG using a Word2Vec model and further employ a Relational Graph Convolutional Network (R-GCN) [56] to aggregate the node embeddings and edge relations into a single vector representation for each function's subgraph. Finally, we compute pairwise cosine similarity between these vector representations across all transfer-containing functions and use this similarity score as a numerical feature F6. A larger similarity score indicates that two transfer-containing functions implement more similar transfer logic. We treat this feature as a continuous indicator rather than a binary decision, allowing for some noise in the functions.

F7. Relevance of log events. We detect misleading log events near transfers in two steps. First, in the CFG we locate `emit` instructions within k=2 hops of the transfer node. Second, we execute the function with Foundry [44] to collect concrete logs and ask GPT-4o [2] for a binary judgment under a strict rubric: the log is *relevant* only if the recipient, token, and action type match the observed transfer; otherwise it is *irrelevant*. We set F7 to `TRUE` for relevant and `FALSE` for irrelevant.

4.3 Obfuscation Z-score Model

Upon extracting the seven obfuscation features, we customize the standard Z-score [17] and calculate an *Obfuscation Z-score* as a quantitative metric to represent the degree of obfuscation applied to transfer operations. In particular, we calculate the cumulative distance between each feature's standard deviation and mean, and further compute sum of standardized features to obtain the obfuscation Z-score for each contract:

$$Z_{\text{score}} = \sum_{i=1}^{7} \frac{x_i - \mu_i}{\sigma_i}, \tag{2}$$

where x_i is the i-th feature value for a given contract, and μ_i and σ_i are the mean and standard deviation of the corresponding feature across the entire sample set. This Z-score represents the cumulative distance between the values of all seven features and their means in standard deviations, indicating the degree of complexity for obfuscation applied to the fund transfer operations.

Why Z-score. Our seven features reside on heterogeneous scales (e.g., counts, ratios, graph-similarity scores), so summing raw values would be disproportionately influenced by the feature with the largest numeric range. Standardizing each feature using its corpus-level mean and standard deviation renders them unitless and comparable, ensuring that the aggregate score reflects joint extremeness rather than scale artifacts. The resulting Z-score quantifies how many standard deviations a contract's combined behavior deviates from the corpus mean in the current snapshot, enabling corpus-agnostic tail selection (e.g., "top three-sigma"). Operationally, re-estimating (μ_i, σ_i) for each snapshot allows the score to adapt to ecosystem or compiler drift without manual, per-feature threshold tuning. In sum, Z-score normalization promotes fairness across features, interpretability across datasets, and stability over time.

4.4 Evaluation of ObfProbe

We evaluate the performance of ObfProbe on a dataset of 453 Ponzi scam contracts with respect to its effectiveness. Table 2 summarizes the data sources and our manual classification results.

Table 2. Real-world Ponzi Scam Dataset

Category	Item	Count
Data source	CRGB [36]	137
	SourceP [40]	316
	Total	**453**
Classification	Obfuscated	92
	Non-Obfuscated	361

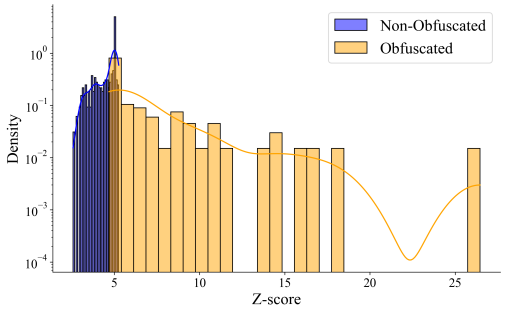
Table 3. Z-score statistics on the labeled dataset.

Obfuscation	Count	Mean	Std	Min	Max
Non-Obfuscated	361	4.571	0.641	2.581	5.261
Obfuscated	92	6.888	3.587	4.688	26.456
Welch's t-test: $t = -6.172$, $p < 10^{-6}$					

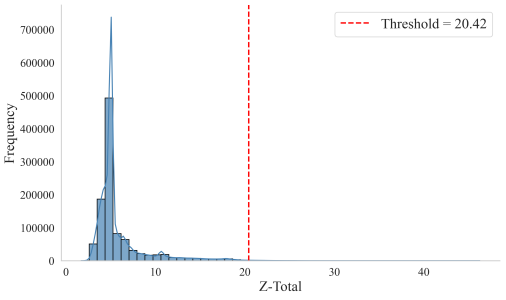
To evaluate the effectiveness of ObfProbe, we compile these 453 contracts to obtain their bytecode and apply ObfProbe to get the values of the seven obfuscation features as well as the Z-scores. We manually confirm that the values of the seven obfuscation features are all correct, indicating 100% accuracy of our bytecode analysis over the Ponzi dataset. Furthermore, we examine the distribution of the Z-scores for the two groups (obfuscated vs. non-obfuscated), which is presented in Figure 2a. As shown, the mean Z-score of the obfuscated group is significantly higher than that of the non-obfuscated group, and its standard deviation is also larger. This indicates that obfuscated contracts exhibit greater diversity and concealment. Additionally, we conduct a more detailed statistical investigation, as shown in Table 3. The results show that the difference between the Z-score distributions of the two groups is statistically significant (t = -6.172, $p \approx 0$), confirming that ObfProbe can effectively distinguish obfuscated contracts from non-obfuscated contracts by using Z-scores. In summary, the evaluation results show that ObfProbe can accurately extract the predefined obfuscation features. In addition, it also shows that our Z-score representation model can effectively differentiate obfuscated from non-obfuscated smart contracts.

5 SMART CONTRACT OBFUSCATIONS IN THE WILD

In this section, we aim to answer RQ2 by studying the prevalence of obfuscation techniques in real world smart contracts.



a Z-score Distribution on Ponzi Scam Dataset



b Z-score Distribution on Mainnet (red dashed = top 3000 cutoff)

Fig. 2. Z-score distributions on different datasets.

Table 4. Z-score statistics.

Metric	Value
Count (contracts)	1,042,923
Mean	5.867
Std	2.910
Min	1.698
Max	46.264

Table 5. Prevalence of obfuscated smart contracts.

Category	Count	Percentage
Above threshold (> 4.637)	739,763	70.93%
Below threshold (< 4.637)	303,160	29.07%

5.1 Prevalence Study

We collect 1,042,923 unique smart contract bytecodes that were *active* on Ethereum mainnet between Jun 2022 and Oct 2024 (i.e., at least one on-chain interaction in this window), *regardless of their original deployment dates.*

Distribution of Z-Score. Figure 2b shows that the distribution of the obfuscation score (`Z-score` is strongly right-skewed. Most contracts fall in the range $3 \lesssim Z \lesssim 8$, with a peak near $Z \approx 6$. For $Z > 10$, the bar heights decline steadily, forming a long tail that extends to the highest scores. This visual pattern indicates that while most smart contracts exhibit a moderate obfuscation level, there exists a small subset employing an extreme higher level of obfuscations. In Table 4, we summarize the distribution of Z-score across the 1,042,923 contracts. To examine the prevalence of obfuscated contracts, we use a Z-score threshold of 4.637 derived from real-world data in Table 2 to represent the 95% confidence interval (CI) upper bound, calculated as

$$4.571 \; + \; t_{0.975,360} \times \frac{0.641}{\sqrt{361}} \; \approx \; 4.637. \tag{3}$$

Applying this cutoff, Table 5 reveals that 70.93% of on-chain contracts deploy obfuscation at or above what would be considered a "normal" level. More than two-thirds of deployed contracts exceed the non-obfuscated CI ceiling, demonstrating that obfuscation has become a routine practice in smart contract development. This pervasive adoption underscores the need for more robust analysis tools and transparency mechanisms to manage obfuscation in the Ethereum ecosystem.

Answer to RQ2. Applying ObfProbe to real-world Ethereum smart contracts reveals that most real-world smart contracts exhibit a moderate level obfuscation, and there exists a small subset employing an extreme higher level of obfuscations;

Table 6. Top 3000 obfuscated contracts.

Metric	Value (Share)
z_{score} min / median / max	20.419 / 21.922 / 46.264
features ≥ 2	3,000 (100.0%)
features ≥ 3	2,943 (98.1%)
With verified source code	1,380 (46.0%)

Table 7. Distribution of z_{score} in top 3000 contracts.

z_{score} range	Count (Share)
20–25	2,644 (88.1%)
25–30	335 (11.1%)
30–35	16 (0.5%)
35–40	2 (0.1%)
40–45	2 (0.1%)
45–50	1 (0.0%)

5.2 Analysis of the 3000 highly-obfuscated Contracts

To better understand the impact of obfuscation techniques used in real-world smart contracts, we select the top 3000 contracts in terms of Z-scores as highly suspicious targets for in-depth analysis. We focus our analysis on top 3000 contracts based on two considerations. First, in statistics, the rightmost 3000 samples (roughly 0.3% of the total) in a normal distribution typically indicate extreme outliers [15, 32, 72] (i.e., values exceeding approximately three standard deviations from the mean). Second, focusing on the top 3,000 samples can significantly reduce our manual efforts while still adequately covering the typical values of the distribution. We hence limit our *manual analysis and case studies* to the top 3,000 contracts to strike a balance between feasibility and coverage.

Overview of the top 3000 contracts: Table 6 presents an overview of the top 3000 contracts. It can be seen that the top 3000 contracts' Z-score is *heavy-tailed*, evidenced by a median of 21.922, a minimum of 20.419, and a maximum of 46.264. In addition, all the 3000 contracts (100%) have exhibited at least two obfuscation features, and 98.1% exhibited at least three obfuscation features. Among the 3000 contracts, 46.0% of them have the source code published on Etherscan. Table 7 shows the distribution of the top 3000 contracts' Z-score. It can be seen that the majority of the contracts' Z-score fall into the range of 20–25 (88.1%). 11.1% of them fall into the range of 25 - 30. Less than 1% of them fall into the range of 30 - 50.

A close look at the top 3000 contracts: To facilitate further analysis, we rank the top 3000 contracts by the total amount of funds from highest to lowest and closely examine each contract. Our manual investigation reveals that all contracts employ obfuscation techniques, with 463 contracts posing very *high risks* due to their concealment of four potentially malicious and suspicious behaviors through extensive obfuscation. Such 463 contracts can be divided into four categories: MEV bots, Ponzi schemes, fake decentralization, and extreme centralization. The total amount of funds absorbed by these contracts reaches approximately $100 million USD in Ether. Table 8 summarizes the distribution of the 463 contracts among the four categories. In general, MEV bots account for the most significant portion (50.1%), followed by extreme centralization (33.0%). Ponzi/Scam and Fake Decentralization are less frequent but salient for risk analysis. Below, we conduct a detailed case study on the four types of contracts by discussing their typical obfuscation patterns, their associated financial impact, and the life span of their transaction activity.

5.3 Case I: MEV Bots

Due to different exchange rates across multiple decentralized exchanges (DEXs), various arbitrage opportunities exist. MEV bot contracts are thus developed and deployed to exploit the opportunities and gain profits with techniques such as front-running [77], back-running [42], and sandwich attacks [43]. Our study finds that some MEV bot contracts leverage *heavily obfuscated* to hide their profit-making logic and thwart analysis. Our analysis of the highly obfuscated MEV bot contracts

Table 8. Distribution of the contracts in the four case studies.

Case-study pattern	Count	Percentage
MEV bots	232	50.1%
Extreme Centralization	153	33.0%
Ponzi/Scam	48	10.4%
Fake Decentralization (renounce)	30	6.5%
Total instances (multi-label)	463	100%

reveals four unique patterns, which are MEV-specific obfuscation patterns that combine multiple obfuscation techniques from our taxonomy.

(1) **Fallback only.** In this pattern, MEV bot contracts only implement the fallback functions to parse the calldata, which then jump to the corresponding location to continue the execution. Eliminating the 4-byte function selectors poses additional challenges for static analysis, such as function identification and call graph generation. The code typically features numerous SWAP and JUMPI instructions. This strategy is highly relevant to F3 and F4 features in Table 1 because the fallback function typically relies on external calls to handle `calldata` and uses conditional branches to increase the complexity. A representative MEV bot employing this obfuscation pattern can be found at address `0x6b75d8af000000E20b7a7ddf000Ba900b4009A80`.

(2) **ABI distortion.** Some MEV bots manipulate function selectors by shortening or relocating them within `calldata`, making it difficult to identify entry points of the contract. This pattern is relevant to features F1, F2, and F4 because (1) it results in complex address generation processes, involving multiple steps; (2) manipulating ABI elements involves string operations like concatenation or hashing; and (3) it may introduce additional control flow complexity by introducing branches. A representative MEV bot employing this obfuscation pattern can be found at address `0x1F2f10d1C40777AE1da742455c65828fF36df387`.

(3) **Address obfuscation.** This pattern uses operations like `PUSH4`, `PUSH4`, and `XOR` to reconstruct the beneficiary address, and requires precisely-length `calldata` inputs, immediately reverting on mismatch. It is related to features F1 and F2 in that it introduces multiple steps to dynamically construct the transfer address and sometimes involves string manipulations. Moreover, our observation shows that it often introduces many irrelevant instructions, reducing the proportion of transfer-related instructions. Hence, it is directly captured by F1. A representative MEV bot employing this obfuscation pattern can be found at address `0xA69babEF1Ca67a37fFAf7a485DfFf3382056E78c`.

(4) **Runtime constraints.** This pattern introduces conditional branches based on chain-specific variables (e.g., block.coinbase), preventing frontrunning in the public mempool by directing different logic flows depending on the block builder. We find that this strategy is related to features F4 and F7 since it will introduce additional conditional branches in the control flow and often emits irrelevant or misleading logs. A representative MEV bot employing this obfuscation pattern can be found at address `0x51C72848C68A965F66fA7A88855F9F7784502a7F`.

Arbitrage transactions of MEV bots. To gain more insights into the impact of different obfuscation patterns adopted by MEV bots, we select the representative MEV bots from each pattern to analyze their transaction activities. We draw several observations from a time series analysis of arbitrage transactions submitted to each MEV bot, which is presented in Figure 3. Notably, the fallback-only MEV bot spikes to peak throughput ($\approx$ 100 000 transactions) early in the period before abruptly dropping to zero, indicating a narrow exploitation window. In contrast, the ABI

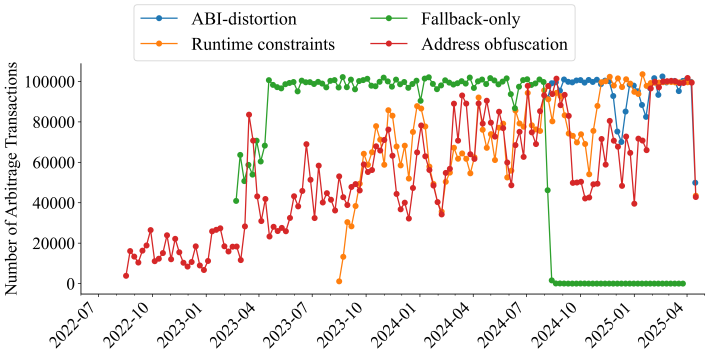


Fig. 3. Time Series Analysis of transaction Volumes

distortion contract ramps up gradually and sustains high volumes, while runtime constraint and address obfuscation patterns show slower growth and greater variability in transaction counts. *We hypothesize three drivers.* First, fallback-only bots exploit short-lived mispricings and accept almost any calldata, which makes them simple to template for copycats and easy for builders/routers to fingerprint; once competition and filtering rise, their opportunity window closes and traffic collapses. Second, ABI distortion exposes stable but custom entry points that are tightly coupled with off-chain solvers and private relays, enabling gradual adoption and sustained order flow over time. Third, runtime-constraint and address-obfuscation designs depend on block-level predicates (e.g., timestamp, basefee) and rotating proxies/recipients, which lower fill rates, increase operational churn, and yield bursty, more variable transaction counts.

5.4 Case II: Ponzi Schemes

Ponzi contracts encourage users to deposit funds or purchase specific tokens by claiming high yield returns through automatic buybacks and burns, compounded mining, or cross-platform arbitrage. Then, they force participants to hold the tokens, implement multi-level commission systems, and rely on new funds from subsequent investors to support returns for earlier participants [22], exhibiting classic Ponzi characteristics. When additional funds fall short, the project creator dumps tokens to reap enormous profits, triggering a collapse of the system and causing losses for participants [41]. We investigate obfuscated contracts and identify some representative Ponzi-specific behaviors (e.g., multi-level deduction, referral/downlines).

Obfuscation patterns. We exemplify how the contract obfuscates the logic of forcing token holding, implementing buyback and burn mechanisms, and using multi-layered function wrappers.

(1) **Multi-level deduction and address generation.** When users withdraw tokens, a cumbersome fee deduction process is involved, which requires parameters such as devTreasury, refBonus, and buyNBurn. ObfProbe performs backward slicing from transfer operations and detects an obfuscated computational process. Furthermore, we see that the owner controls these parameter configurations and can adjust them at will.

(2) **Layered Logic Based on External Inputs (Referral/Downlines).** We identify a recruitment mechanism in the contract, which is a multi-level data structure. This indicates that user returns do not come from the contract's own operations, but are instead distributed from the funds of new investors.

(3) **Abundant "Buyback and Holding Check" Strings/Events.** By analyzing event names or string constants, ObfProbe detects terms like "Burn", "MLMReward", or "RetirementYeld". These words are typically associated with forced token holding, token burning, or multi-level commissions, typical keywords of Ponzi/pyramid schemes.

A representative contract: Deployed at `0x25cb947ebef1c56e14d5386a80262829739dbdbe` on the Ethereum mainnet, the contract exhibits the above obfuscation patterns. Over *1,460* days (2020-07-29–2024-07-29), the contract involves *27,303* interactions that all transferred ETH (with 0 ERC-20 token transfer), totaling 1,433.21 ETH. This contract is found matching the following obfuscation patterns: (i) **Multi-level deduction.** The withdrawal path computes layered splits into `devTreasury`, `refBonus`, and `buyNBurn` sinks; these ratios are owner-configurable and can be adjusted at will, producing opaque fee chains. (ii) **Referral/downlines.** The call data encodes a referrer and users are positioned in a matrix-like structure; payouts to earlier participants are funded by subsequent deposits rather than any productive on-chain activity. (iii) **Buyback/holding narrative.** The code and logs contain terms such as "Burn", "MLMReward", and misspelled variants (e.g., "RetirementYeld"), alongside frequent "register/upgrade/reinvest/missed-earnings" events that suggest compounding and forced holding while simply redistributing funds to uplines. Through this example, we show that Ponzi contracts typically leverage heavy obfuscation techniques to hide their business logic to avoid detection. To our best knowledge, our work is the first to report this contract as a Ponzi scheme.

5.5 Case III: Fake Decentralization

This type of contract claims that the control of the contract is decentralized to attract participants, while maintaining obfuscated backdoor functions that allow owners to control the contract. Our study shows that two components are used to implement its malicious logic. The first one is called `fake renouncement of ownership`. Specifically, the contract claims that executing the `renounceOwnership()` can remove the centralized ownership. However, the function merely transfers the ownership to another address under the project owner's control or simply does nothing except emit a seemingly correct log to deceive participants. The second component is malicious backdoors, which are obfuscated functions (e.g., `Failsafe` or `Emergency`). They are claimed to handle system crashes, but in fact allow the project owner to withdraw assets at any time, which could potentially result in financial losses to participants.

Obfuscation patterns. Through our detection and investigation, we found this type of contracts adopt three obfuscation patterns. First, the contract duplicates ownership transferring logic in multiple functions (feature F6), which only differ in parameter names or variable names, to increase code complexity and hinder static analysis and manual auditing. Second, the contract often hides its core logic by inserting many empty and useless code segments (feature F5), making it difficult for auditors to quickly pinpoint the core backdoor. Third, the contract publicly claims that "control has been relinquished," while the `onlyOwner` modifier remains effective. Alternatively, it may contain a function (e.g., `_transferOwnership(addr)`), where `addr` is an address controlled by the owner, then the actual control is still maintained.

A representative contract: The above patterns are instantiated in a contract deployed at `0xb194A96AADC7e99a2462EF1669eB38E6B541DF79`. It involves *11* transactions over *3* days (2020-10-06 to 2020-10-10). During this period, it receives 0 ETH and handles one ERC-20 token, *YELD*, with 5,000 in and 5,000 out (gross 10,000, net 0). The contract is found matching the following patterns: (i) Fake renouncement: the contract claims ownership renouncement while preserving effective owner control via aliased transfer/renounce paths; (ii) Misleading logs: the contract emits "relinquished control" log events. However, the `onlyOwner` modifier still remains effective; (iii) Code padding/noise: the contract uses redundant stubs and near-duplicate permission paths without changing semantics. In Appendix A (§A.1),we provide additional code-level analysis of this contract.

5.6 Case IV: Extreme Centralization

Contracts in this category share a common design principle: all critical operations, from permission management to fund extraction, are ultimately controlled by a single address, despite superficial "decentralized" interfaces or multi-role declarations. Three typical manifestations are:

- **Centralized permission control:** Roles such as `admin`, `liquidateAdmin` (or `manager`, `_super`, etc.) are all initialized to the deployer's address, allowing unilateral modification of oracles, collateral ratios, fee structures, and forced liquidations.
- **Arbitrarily adjustable fee/tax:** They impose exorbitant buy/sell/withdrawal fees (often 10%–50%, up to 99%) via an `onlyOwner`-protected function, with all collected fees routed to one address.
- **Lack of funding lock.** Some contracts such as "staking" or "farm" claim to have emergency withdrawals. However, there often exist functions (e.g., emergencyWithdraw(), requestWithdraw(), or claimTokens()) under the owner's control, which can enable instant drainage.

Obfuscation patterns. We investigate the typical obfuscation patterns adopted by these contracts and found that they often use redundant functions, events, and misleading names to obscure the centralized control, as revealed by features F6 and F7. Here, we list a few very representative ones.

(1) **Role masquerading.** They tend to create multiple roles that point to the same address.

```
constructor() {
    admin = msg.sender;
    liquidateAdmin = msg.sender;
}
```

(2) **Redundant permission checks.** They introduce identical checks repetitively to inflate code complexity without adding any real safety.

```
require (msg.sender == admin);
require(msg.sender == liquidateAdmin);
```

(3) **Dynamic fee adjustment:** This function lets the owner unilaterally change both buy and sell tax rates at any time, enabling arbitrary fee hikes that can extract maximum revenue from users without prior notice.

```
function setTaxes(uint256 buyTax, uint256 sellTax) external onlyOwner {
    taxForBuy = buyTax;
    taxForSell = sellTax;
}
```

(4) **Backdoored withdrawals.** Although labeled as an emergency rescue, this owner-only method allows immediate token transfers from the contract to the owner's EOA, effectively serving as a hidden backdoor to drain all funds.

```
function emergencyWithdraw(uint256 amount) external onlyOwner {
    token.safeTransfer(msg.sender, amount);
}
```

(5) **Redundant event and function:** The contract may contain hundreds of emit calls (e.g., FeeEvent, UserUnlocked) and dozens of near-duplicate functions (e.g., _swapTokens, _addLiquidity, _withdrawFromBank, fulfillDeposited) interleaved to generate noise for auditing.

A representative contract: The above patterns are exemplified in a contract deployed at `0xd7caa679aa6e39c3891bd7a63b058bb8a269da52`. Our analysis shows that it involves *439* transfer transactions within *81* days (2023-06-14 to 2023-09-03). The total received ETH is 0.2006 ETH. The received ERC-20 tokens include 10 *USDT* and 19.074240298 *APE*. The contract is found matching the following patterns: (i) Role masquerading: multiple roles are ultimately pointing to the same address,

resulting in centralized control; (ii)Backdoored withdrawals: There are owner-controlled functions such as `withdraw`/`withdrawErc20`, which enable immediate extraction of funds in the contract. *More details.* Extended examples, code snippets, and detection cues for extremely centralized designs are deferred to Appendix A (§A.1, §A.2, §A.3).

Summary. With the above four case studies, we show that it is prevalent for real-world contracts to employ various obfuscation techniques to hide their malicious behaviors such as MEV bots, Ponzi schemes, and fake decentralization, posing security risks to users.

6 FINANCIAL IMPACT OF SMART CONTRACT OBFUSCATION

Following the detailed behavioral analysis, we further investigate the financial impact of obfuscation in the real world to answer RQ3. To achieve this, we collect a representative dataset of scam smart contracts, utilize ObfProbe to detect the presence of obfuscation, and quantitatively compare the financial impact between obfuscated and non-obfuscated scam smart contracts to gain a deeper understanding of how obfuscation techniques affect the scams' financial gain.

6.1 Dataset

We leverage the dataset from a prior study [34], which conducts a large-scale study on scams in the wild and reports approximately 13K scam arbitrage bot contracts. To obtain the ground truth on code obfuscation, we manually examine the dataset and label each contract to obtain two groups of contracts: with-obfuscation and no-obfuscation, containing 9,197 and 3,826 unique contracts, respectively.

6.2 Overview of Financial Loss

After obtaining the two groups of contracts, we first analyze the inbound funds (fund inflows) of each group and the involved victims on the Ethereum mainnet to compare the financial loss (focus on ETH only) and then quantify the involved victims by counting unique Externally Owned Addresses (EOA) that directly send ETH to the contracts. Table 9 shows the statistics of inbound funds of each contract group. It can be seen that the average inbound funds between two groups are very close (0.3403 ETH vs. 0.3455 ETH), and the median value are also at the same order (0.06 vs. 0.10 ETH). However, the maximum inbound funds of obfuscated contracts (201.74 ETH) is about 2.41× that of non-obfuscated contracts (83.62 ETH), which indicates that obfuscated contracts have a higher likelihood to cause a more severe financial damage to victims.

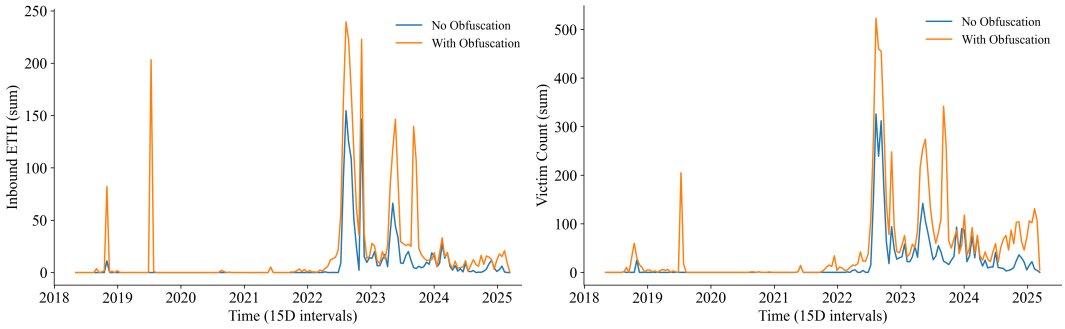
Table 9. Inbound funds statistics.

Contract Group	Count	Inbound Funds (ETH)			
		Sum	Median	Mean	Max
No-Obfuscation	3,826	1,321.82	0.10	0.3455	83.62
With-Obfuscation	9,197	3,129.997	0.06	0.3403	201.74

6.3 Timeline Trend of Financial Loss

To better illustrate the financial impact of obfuscation, we also analyze the timeline trend of victims and the inbound funds caused by the two contract groups, from 2018 to 2025 in a 15-day interval.

Inbound Funds: Figure 4a shows the inbound funds of each contract group aggregated in the 15-day interval. It can be seen that non-obfuscated contracts (blue line) remain relatively small throughout the period from 2018 to 2025, with occasional spikes. In contrast, the inbound funds of obfuscated contracts (orange line) exhibit multiple peaks, mostly centered between 2022 and 2024. Particularly in June 2019, the victims lost more than 200 ETH. Later in July 2022, victims lost more



a Inbound funds aggregated by a 15-day interval. b Victim EOA aggregated by a 15-day interval.

Fig. 4. Time Trend Analysis

than 250 ETH, which also represents the maximum loss of 201.74 ETH caused by a single contract, as shown in Table 9. Then, in three different months from June 2022 to August 2023, the victims lost more than 100 ETH each.

Victim Count. Figure 4b presents the aggregated victim addresses of each contract group in the 15-day interval. It is evident that the number of victims from obfuscated contracts (orange line) consistently exceeds that of non-obfuscated contracts (blue line) throughout the period from 2022 to 2025. Notably, there are sharp increases during specific months (e.g., from the latter half of 2022 to 2023), with the total number of victims within a single 15-day period reaching several hundred. In contrast, the number of victim addresses of non-obfuscated contracts remains much lower during the same period. Such results further highlight that obfuscated contracts can increase the likelihood of deceiving a much larger group of victims.

Answer to RQ3. Our analysis reveals that obfuscated scam contracts are more active than non-obfuscated ones and have resulted in higher financial losses and a larger number of victims. This finding aligns with our hypothesis: Obfuscation techniques can enable scam contracts to operate more covertly during their initial stages, allowing them to gain a larger amount of profits by deceiving more victims.

7 IMPACT ON EXISTING SCAM DETECTION TOOLS

Finally, to answer RQ4, we examine how obfuscation affects the effectiveness of existing malware analysis tools. We run SourceP [40], a state-of-the-art Ponzi detector, on a *single* dataset: the contract set released with the SourceP paper (summarized in Table 2). On this dataset, we **manually inspect and assign** obfuscation labels according to our taxonomy (T1–T7): a contract is labeled *obfuscated* if at least one of T1–T7 is present; otherwise, it is labeled *non-obfuscated*. Ambiguous cases are excluded to avoid label noise. We then compute accuracy, recall, and F1 to quantitatively compare SourceP's effectiveness on obfuscated vs. non-obfuscated samples.

Table 10. Effectiveness of SourceP for obfuscated and non-obfuscated Ponzi samples.

Class	Total	TP	FN	Accuracy	F1
Non-obfuscated	361	287	74	0.79	0.88
Obfuscated	92	11	81	0.12	0.21

Experimental Results. Overall, SourceP achieves an accuracy of 65.41% for all 453 samples. This contrasts with the higher precision reported in the original paper, suggesting that in real-world scenarios (especially for contracts with deeper obfuscation), many false negatives and false positives exist. A closer look at the results, shown in Table 10, reveals that the effectiveness difference between obfuscated and non-obfuscated samples is very significant. SourceP achieves an F-1 score of 0.88 for non-obfuscated samples, which roughly aligns with the detection capability claimed in the original paper. However, for obfuscated contracts, both accuracy and recall drop sharply, with an F1 score of only around 0.21, indicating that obfuscation techniques in real-world Ponzi samples can remarkably affect the performance of state-of-the-art malware detection tools.

The pronounced performance disparity between obfuscated and non-obfuscated samples can be attributed to the ways in which obfuscation disrupts SourceP's static analysis pipeline. First, externalized transfer paths (T4) relocate the payable sink beyond the intra-contract scope that SourceP primarily analyzes, resulting in under-approximated money-flow graphs and the omission of critical Ponzi indicators. Second, opaque predicates and deep nesting (T2) complicate the control-flow graph, leading to path pruning or premature cut-offs during data and control flow reconstruction, thereby suppressing essential features upon which SourceP relies, such as cyclic payouts and balance-dependent branches. Third, multi-step address synthesis (T1), achieved through hashing, bit-masking, or nonces, disrupts constant propagation and complicates recipient attribution, thereby weakening heuristics that depend on recognizable payout targets. Fourth, dispatcher flattening and function cloning (T5) obscure function boundaries and hinder the reuse of summaries, which degrades feature aggregation at the function level. Fifth, log interference (T6) introduces spurious events and alters the order of informative logs, potentially misleading log-aware heuristics or subsequent sanity checks. Finally, junk code inflation (T7) increases the size of the bytecode and the noise within the intermediate representation, thereby heightening the likelihood of timeouts or the application of conservative defaults. In contrast, non-obfuscated contracts reveal canonical money-flow structures that are consistent with SourceP's feature design and training distribution, resulting in significantly enhanced effectiveness.

We qualitatively inspect representative misclassified obfuscated contracts and find that errors primarily occur in cases involving externalized transfer paths (T4) and path explosion (T2). Additionally, captions and logs often reveal partial analyses, such as truncated call-graph slices. This suggests that the performance gap between obfuscated samples and non-obfuscated ones is unlikely due to noisy labels or dataset differences, but rather stems from the reduced observability of Ponzi-specific signals under obfuscation. It is essential to note that our evaluation employs an ETH-only victim/flow definition, excluding intermediaries such as routers, centralized exchanges, and miner extractable value (MEV). This means that the observed gap reflects limitations in our analysis rather than metric contamination.

Answer to RQ4. Our evaluation results demonstrate that obfuscation has a significant detrimental impact on existing scam detection tools, causing a substantial drop in both accuracy and recall. At a practical level, this observation highlights the importance of developing effective obfuscation analysis techniques, which are crucial for mitigating emerging security threats.

8 DISCUSSION

8.1 Limitations and Threats to Validity

Our taxonomy (T1–T7) and features (F1–F7) focus on *transfer-path* obfuscation. Within this formal scope, the taxonomy aims for comprehensive coverage. Nonetheless, our work still has the following five limitations: 1) We do not cover ERC-20/721 token flows, bridge/mixer interactions,

or metamorphic designs via CREATE2 or code replacement. These are outside our transfer-path scope and may require additional primitives. 2) The analysis is bytecode/IR–only (no dynamic execution or cross-transaction traces). Obfuscation that resolves purely at runtime (e.g., oracle-fed recipients, environment-gated dispatch) can reduce observability and yield false negatives. 3) When the payable path is fully delegated via DELEGATECALL/CALL to external contracts, F4 flags externalization but cannot recover the callee's logic; downstream features (e.g., F1–F7) may thus remain unset. 4) Z-score and absolute guards rely on baseline distributions; different snapshots can shift thresholds slightly (we release seeds to recompute them). 5) Parts of the benchmarks (e.g., the SourceP comparison) use single-annotator manual labels; we exclude ambiguous cases and will release address lists, but residual noise may remain. Overall, we regard F1–F7 as a *minimal, extensible core* tailored to the transfer-path scope; extending to non-transfer flows or adding orthogonal runtime evidence is left to future work.

8.2 Challenges and Future Work

Obfuscation techniques in smart contracts present significant challenges for auditing and regulatory practices, particularly in scam contracts, MEV bots, and highly centralized systems. These challenges include poor code readability, failure of conventional static detection tools, and delays in regulatory response. Several improvements are necessary to address these issues. Static analysis tools must be enhanced to track deeper control and data flows, while de-obfuscation preprocessing can simplify bytecode for more efficient audits. Dynamic analysis, such as runtime tracing or fuzzing, can bypass superficial obfuscation and validate fund flows during contract execution. Additionally, techniques from traditional software security, such as CFG flattening and semantic normalization, can be applied to EVM bytecode to identify critical logic, such as transfers and permission checks, that obfuscation attempts to conceal. Eventually, establishing collaborative platforms (e.g., a "contract blacklist" or a "high-risk obfuscation" repository) would enable researchers, auditors, and the public to tag suspicious contracts, thereby improving transparency and enhancing collective oversight of the DeFi ecosystem.

8.3 Obfuscation signals vs. malicious intent

Our taxonomy (T1–T7) and features (F1–F7) capture *how observable the transfer logic is*, not whether that logic is benign or harmful. The same primitives often appear in legitimate designs. For instance, T4 can be used in upgradeable proxies and routers, while T5 in flattened dispatchers for gas efficiency. Hence, we treat F1–F7 as *signals* that warrant scrutiny, not verdicts. When we discuss scams, the goal is to *illustrate correlation*, not to claim causation. Disambiguating intent typically requires orthogonal evidence (economic semantics, longitudinal flows, victim-side signals), which is outside the scope of our obfuscation quantification.

9 RELATED WORK

Classical Obfuscation Techniques. A large SE literature has studied source/IR-level obfuscation for native/managed code. Collberg et al. offer a widely used taxonomy (layout, control-, data-, and preventive transformations) with potency/resilience/stealth metrics [12]. Representative techniques include opaque predicates and bogus control-flow (control-flow obfuscation), anti-disassembly/anti-decompilation (preventive), instruction substitution and data encoding, and control-flow flattening; industrial-strength implementations such as Obfuscator-LLVM operationalize several of these passes [30, 33, 37]. Surveys and monographs synthesize two decades of progress and limitations [10, 58].

Smart Contract Security Analysis. Some research employs static analysis to enhance the security and efficiency of smart contracts. USCHUNT [6] explores the balance between adaptability and security in upgradeable contracts. Madmax [23] targets vulnerabilities to prevent execution failures, while Slither [62] and Smartcheck [68] automatically detect flaws in Solidity contracts. Symbolic execution is also used to improve security; Mythril [14] analyzes EVM bytecode, EthBMC [21] combines symbolic execution with concrete validation, and Reguard [39] and Manticore [46] identify reentrancy and other bugs. Smartian [9] integrates fuzzing with static and dynamic analysis, while Confuzzius [20] leverages data dependency insights for fuzzing. CRYSOL [78] applies fuzzing to detect cryptographic defects in contracts. ContractFuzzer [29] and Sfuzz [45] apply fuzzing to uncover security issues. Research highlights various formal verification methods to enhance the security of smart contracts. Sailfish [7] improves state inconsistency detection, while VetSC.[19] extends DApp verification. Zeus[31] and Verx [48] focus on contract safety and condition verification. Smartpulse [65] analyzes time-based properties, Securify [69] identifies security breaches, and Verismart [63] ensures contract safety.

Advanced Anti-Auditing Techniques. Recent analyses expose "fake" ownership renunciation: after invoking `renounceOwnership()`, some contracts zero public fields (e.g., `owner`, `getOwner`) yet retain control via concealed state. Shiaeles and Li (2024) document a live case where a benign-looking variable (e.g., `isTokenReceiver`) stores the deployer's address, leaving an address-bound backdoor [26]. More broadly, innocuous names such as `isTokenReceiver`, `failsafe()`, or `emergency()` can mask administrator-only operations (e.g., privilege restoration or fund extraction).

MEV Bot Obfuscation Techniques. To protect MEV bots from being front-run by generalized mimicking scripts in the public mempool, practitioners and researchers have developed various obfuscation and privacy-preserving techniques. The most common method is using private relays (e.g., Flashbots) to submit bundles directly to block builders, bypassing the public mempool and preventing adversaries from copying transactions [49, 76]. Intent-based protocols like CoW Swap perform off-chain batch matching of user intents, publishing only the final settlement on-chain to eliminate front-running risks [77]. Gas camouflage techniques, such as locking transactions to specific `tx.gasprice` values or adding dummy computations, confuse adversarial repricing strategies [76]. Multi-hop contract calls, often paired with flash loans and non-standard swap paths, increase attackers' simulation overhead [18, 77]. Bytecode-level obfuscations, like inserting `JUMPI` pseudo-branches or splitting constants via arithmetic, hinder static and dynamic analysis [77]. Recent work has also explored threshold encryption, delayed reveal schemes, and protocol-level MEV "tax" mechanisms to internalize ordering profits [5, 18]. Despite these advancements, systematic research on obfuscation techniques for MEV bots at the smart contract level remains scarce.

10 CONCLUSION

In this paper, we systematically investigate obfuscation techniques in Ethereum smart contracts, providing comprehensive definitions, quantitative methods, and empirical analysis. We introduce seven key quantifiable obfuscation features based on the detailed analysis of transfer instructions. Using a robust Z-score representation model, we analyze the prevalence of obfuscation techniques employed in over 1.04 million Ethereum contracts and conduct an in-depth analysis of 3,000 highly suspicious contracts, revealing four types of malicious contracts. Our further analysis shows that obfuscated scam contracts have a higher financial extraction capability than non-obfuscated scam contracts. We also demonstrate that obfuscation significantly undermines the effectiveness of existing detection tools. Overall, our findings underscore the security risks posed by obfuscation and highlight the urgent need for advanced analytical and detection methodologies to address this evolving threat, enhancing blockchain security and fostering transparency.

REFERENCES

[1] 2020. An analysis of crypto scams during the Covid-19 pandemic: 2020–2022. ResearchGate.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[3] Rachit Agarwal, Tanmay Thapliyal, and Sandeep Kumar Shukla. 2022. Vulnerability and transaction behavior based detection of malicious smart contracts. In *Cyberspace Safety and Security: 13th International Symposium, CSS 2021, Virtual Event, November 9–11, 2021, Proceedings 13*. Springer, 79–96.

[4] Jahangeer Ali and S Sofi. 2021. Ensuring security and transparency in distributed communication in iot ecosystems using blockchain technology: Protocols, applications and challenges. *Int J Com Dig Sys* 11, 1 (2021), 1–20.

[5] Mustafa Ibrahim Alnajjar, Mehmet Sabir Kiraz, Ali Al-Bayatti, and Suleyman Kardas. 2024. Mitigating MEV attacks with a two-tiered architecture utilizing verifiable decryption. *EURASIP Journal on Wireless Communications and Networking* 2024, 1 (2024), 62.

[6] William E Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1829–1846.

[7] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.

[8] Burgerswap. 2025. Burgerswap. https://burgerswap.org/trade/swap/.

[9] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing smart contract fuzzing with static and dynamic data-flow analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.

[10] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, Upper Saddle River, NJ.

[11] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations.

[12] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report 148. Department of Computer Science, The University of Auckland, Auckland, New Zealand. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html

[13] Compound. 2025. Compound.Finance. https://compound.finance/.

[14] ConsenSys. 2022. Mythril: security analysis tool for EVM bytecode. https://github.com/ConsenSys/mythril/.

[15] Denis Cousineau and Sylvain Chartier. 2010. Outliers detection and treatment: a review. *International journal of psychological research* 3, 1 (2010), 58–67.

[16] Crytic. 2025. Rattle: EVM Binary Static Analysis. https://github.com/crytic/rattle. Accessed: April 15, 2025.

[17] Alexander E Curtis, Tanya A Smith, Bulat A Ziganshin, and John A Elefteriades. 2016. The mystery of the Z-score. *Aorta* 4, 04 (2016), 124–130.

[18] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2019. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:1904.05234* (2019).

[19] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. 2022. Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications.. In *Proceedings of the 29nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[20] Christof Ferreira Torres, Antonio Ken Iannillo, and Arthur Gervais. 2021. CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *European Symposium on Security and Privacy, Vienna 7-11 September 2021*.

[21] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. 2757–2774.

[22] Sangita F Gazi. 2024. In Code We Trust: Blockchain's Decentralization Paradox. *VAND. J. ENT. & TECH. L* 27, 1 (2024), 59.

[23] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[24] Rajesh Gupta, Mohil Maheshkumar Patel, Arpit Shukla, and Sudeep Tanwar. 2022. Deep learning-based malicious smart contract detection scheme for internet of things environment. *Computers & Electrical Engineering* 97 (2022), 107583.

[25] Y Gupta, J Kumar, and A Reifers. 2022. Identifying security risks in NFT platforms. *arXiv preprint arXiv:2204.01487* (2022).

[26] O J Hall, S Shiaeles, and F Li. 2024. A Study of Ethereum's Transition from Proof-of-Work to Proof-of-Stake in Preventing Smart Contracts Criminal Activities. *Network* 4, 1 (2024), 33–47.

[27] Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. 2023. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal* 10, 14 (2023), 12178–12185.

[28] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2023. Security threat mitigation for smart contracts: A comprehensive survey. *Comput. Surveys* 55, 14s (2023), 1–37.

[29] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[30] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO'15)*, Brecht Wyseur (Ed.). IEEE, Florence, Italy, 3–9. https://doi.org/10.1109/SPRO.2015.10

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[32] K Senthamarai Kannan, K Manoj, and S Arumugam. 2015. Labeling methods for identifying outliers. *International Journal of Statistics and Systems* 10, 2 (2015), 231–238.

[33] Tímea László and Ákos Kiss. 2009. Obfuscating C++ Programs via Control Flow Flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica* 30 (2009), 3–19. https://ac.inf.elte.hu/Vol_030_2009/doi/003_30.pdf

[34] K. Li, S. Guan, and D. Lee. 2023. Towards Understanding and Characterizing the Arbitrage Bot Scam In the Wild. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 3 (2023), 1–29. https://doi.org/10.1145/3626783

[35] Shijia Li, Chunfu Jia, Pengda Qiu, Qiyuan Chen, Jiang Ming, and Debin Gao. 2022. Chosen-instruction attack against commercial code virtualization obfuscators. In *In Proceedings of the 29th Network and Distributed System Security Symposium*.

[36] R Liang, J Chen, K He, et al. 2024. Ponziguard: Detecting ponzi schemes on ethereum with contract runtime behavior graph (CRBG). In *ICSE '24*. 1–12.

[37] Cullen Linn and Saumya K. Debray. 2003. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*. ACM, New York, NY, USA, 290–299. https://doi.org/10.1145/948109.948149

[38] R. Little and D. Xu. 2023. Inspecting Compiler Optimizations on Mixed Boolean Arithmetic Obfuscation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[39] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[40] P Lu, L Cai, and K Yin. 2024. SourceP: Detecting ponzi schemes on ethereum with source code. In *ICASSP '24*. 4465–4469.

[41] Wei Ma, Chenguang Zhu, Ye Liu, Xiaofei Xie, and Yi Li. 2023. A comprehensive study of governance issues in decentralized finance applications. *ACM Transactions on Software Engineering and Methodology* (2023).

[42] ANGEL JAVIER BLASCO MAINAR, JOSÉ MARÍA DE LA CRUZ, and SÁNCHEZ Y ALBERTO MORENO BRASERO. [n. d.]. MAXIMAL EXTRACTABLE VALUE (MEV). ([n. d.]).

[43] Bruno Mazorra, Michael Reynolds, and Vanesa Daza. 2022. Price of mev: towards a game theoretical approach to mev. In *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security*. 15–22.

[44] Alexandre Mota, Fei Yang, and Cristiano Teixeira. 2023. Formally Verifying a Real World Smart Contract. *arXiv preprint arXiv:2307.02325* (2023).

[45] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[46] Trail of Bits. 2024. Manticore: symbolic execution tool for smart contract. https://github.com/trailofbits/manticore/.

[47] Yu Pan, Zhichao Xu, Levi Taiji Li, Yunhe Yang, and Mu Zhang. 2023. Automated generation of security-centric descriptions for smart contract bytecode. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1244–1256.

[48] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.

[49] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–214.

[50] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security Symposium (USENIX Security 19)*. 479–496.

[51] Azam Rashid and Muhammad Jawaid Siddique. 2019. Smart contracts integration between blockchain and Internet of Things: Opportunities and challenges. In *2019 2nd International Conference on Advancements in Computational Sciences*

(ICACS). IEEE, 1–9.
[52] X Ruan. 2022. *Exploring Vulnerabilities and Anomalies in NFT Marketplaces*. Ph. D. Dissertation. University of Guelph.
[53] SM Nazmuz Sakib. 2024. Blockchain technology for smart contracts: enhancing trust, transparency, and efficiency in supply chain management. In *Achieving Secure and Transparent Supply Chains With Blockchain Technology*. IGI Global Scientific Publishing, 246–266.
[54] MSVPJ Sathvik and Hirak Mazumdar. 2024. Detection of malicious smart contracts by fine-tuning GPT-3. *Security and Privacy* 7, 6 (2024), e430.
[55] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *Ieee Access* 8 (2020), 24416–24427.
[56] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*. Springer, 593–607.
[57] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 3055–3073.
[58] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *Comput. Surveys* 49, 1 (April 2016), 4:1–4:37. https://doi.org/10.1145/2886012
[59] C. Sendner, H. Chen, H. Fereidooni, et al. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s263_paper.pdf
[60] Harshit Shah, Dhruvil Shah, Nilesh Kumar Jadav, Rajesh Gupta, Sudeep Tanwar, Osama Alfarraj, Amr Tolba, Maria Simona Raboaca, and Verdes Marina. 2023. Deep learning-based malicious smart contract and intrusion detection system for IoT environment. *Mathematics* 11, 2 (2023), 418.
[61] Sakshi Sharma and Natasha Dutta. 2018. Development of New Smart City Applications using Blockchain Technology and Cybersecurity Utilisation. *Development* 7, 11 (2018).
[62] Slither. 2024. Slither, the Solidity source analyzer. https://github.com/crytic/slither/.
[63] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1678–1694.
[64] Solidity. 2025. Common Patterns. https://docs.soliditylang.org/en/v0.8.30/common-patterns.html/.
[65] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: automated checking of temporal properties in smart contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 555–571.
[66] Wesley Joon-Wie Tann, Xing Jie Han, Sourav Sen Gupta, and Yew-Soon Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632* (2018).
[67] Usman Tariq, Atef Ibrahim, Tariq Ahmad, Yassine Bouteraa, and Ahmed Elmogy. 2019. Blockchain in internet-of-things: a necessity framework for security, reliability, transparency, immutability and liability. *IET Communications* 13, 19 (2019), 3187–3192.
[68] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.
[69] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.
[70] Uniswap. 2025. Uniswap Protocol. https://uniswap.org/.
[71] F Victor. 2022. *Uncovering fraudulent activities in ethereum-based cryptoassets with distributed ledger analytics*. Ph. D. Dissertation. Technische Universität Berlin, Berlin. 2023.
[72] Steven Walfish. 2006. A review of statistical outlier methods. *Pharmaceutical technology* 30, 11 (2006), 82.
[73] W. Wang, P. Zhang, R. Ji, W. Huang, and Z. Meng. 2024. JANUS: A Difference-Oriented Analyzer For Financial Centralization Risks. *arXiv preprint arXiv:2412.03938* (2024). https://arxiv.org/abs/2412.03938
[74] J Wu, D Lin, Q Fu, et al. 2023. Toward understanding asset flows in crypto money laundering through the lenses of Ethereum heists. *IEEE Transactions on Information Forensics and Security* 19 (2023), 1994–2009.
[75] S. Xia, S. Shao, T. Yu, and L. Song. 2025. SymGPT: Auditing Smart Contracts via Combining Symbolic Execution with Large Language Models. *arXiv preprint arXiv:2502.07644* (2025). https://arxiv.org/abs/2502.07644
[76] Sen Yang, Fan Zhang, Ken Huang, Xi Chen, Youwei Yang, and Feng Zhu. 2024. SoK: MEV Countermeasures. In *Proceedings of the Workshop on Decentralized Finance and Security*. 21–30.
[77] Deniz Yüksel. [n. d.]. A Retrospective Analysis of Public and Private Order Flow on the Ethereum Blockchain. ([n. d.]).

[78] Jiashuo Zhang, Yiming Shen, Jiachi Chen, Jianzhong Su, Yanlin Wang, Ting Chen, Jianbo Gao, and Zhong Chen. 2024. Demystifying and Detecting Cryptographic Defects in Ethereum Smart Contracts. In *IEEE/ACM International Conference on Software Engineering*.
[79] Z. Zhang, Z. Lin, M. Morales, and X. Zhang. 2023. Your exploit is mine: Instantly synthesizing counterattack smart contract. In *USENIX Security*. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhuo-exploit
[80] L. Zhou, L. Wang, and K. Qin. 2024. DeFiAligner: Leveraging Symbolic Analysis and LLMs for Inconsistency Detection in DeFi. In *AFT 2024*. https://drops.dagstuhl.de/opus/volltexte/2024/19803/pdf/LIPIcs-AFT-2024-7.pdf
[81] L. Zhou, X. Xiong, and J. Ernstberger. 2023. SoK: Decentralized Finance (DeFi) Attacks. *IEEE Symposium on Security and Privacy* (2023). https://arxiv.org/pdf/2208.13035.pdf

A EXTREMELY CENTRALIZED CONTRACTS

A.1 Centralized Permission Control

(1) Overall Scam Logic Overview

Such contracts often appear under the guise of "lending protocols", "collateral management", "liquidity safeguarding", etc., mimicking the interfaces and function names of well-known protocols like Compound or Uniswap. However, in reality, they are entirely controlled by a few roles—such as `admin` and `liquidateAdmin` (and sometimes more, e.g., `manager` or `_super`)—that manage all key operations.

- **Highly Centralized Permissions:** The deployer (Owner) assigns multiple administrative roles to themselves in the constructor, enabling them to modify the oracle, collateral ratios, fee structures, or even forcefully liquidate user assets at any time.
- **Pseudo-"Decentralization":** Although the contract outwardly features multiple roles and safeguard mechanisms, the actual execution authority remains concentrated in a single private key address, leaving users unable to prevent backdoor operations by the owner.

In practice, these "highly centralized" contracts typically use lengthy, repetitive code and a plethora of events (`emit`) to create complexity, making it difficult for external auditors to immediately discern their true nature.

(2) Analysis of Code-Level Obfuscation Techniques

Below, we analyze a real-world case of a contract named `AegisComptroller.sol` (a pseudonym) to illustrate how such contracts conceal their centralized permission design through role masquerading, numerous redundant functions, and excessive event logging.

- **Role Masquerading: Multiple Names, Layered Functions, but Controlled by the Same Address Dual Roles with the Same Private Key:**

```
constructor () public {
  admin = msg.sender;
  liquidateAdmin = msg.sender;
}
```

  In the constructor, both `admin` and `liquidateAdmin` are set to the same address, creating an illusion of multiple roles while, in fact, the same entity controls everything.
- **Redundant Permission Checks:** The contract repeatedly uses REQUIRE(msg.sender == admin, ...) and REQUIRE(msg.sender == liquidateAdmin, ...) in various locations. Since these checks are essentially equivalent, they further complicate code readability and give the false impression of a robust permission system.
- **Numerous "Administrative" Functions and Spurious Security Checks:**
  - **Seemingly Compliant Configuration Functions:**

```
function _setPriceOracle(PriceOracle,
_newOracle)
public returns (uint) {
```

```
    REQUIRE(msg.sender == admin,
    "SET_PRICE_ORACLE_OWNER_CHECK");
    oracle = _newOracle;
    ...
}
function _setCollateralFactor(
AToken_aToken,
uint _newCollateralFactorMantissa)
external returns (uint) {
    REQUIRE(msg.sender == admin,
    "SET_COLLATERAL
    _FACTOR_OWNER_CHECK");
    ...
}
```

These functions are named very similarly to those in Compound (e.g., "set price oracle" or "set collateral factor"), but they only REQUIRE an admin call and do not incorporate any multisignature or time delay mechanisms.

- **Redundant Role Assignments:**
  For example, functions such as _setMintGuardianPaused(), _setBorrowGuardianPaused(), and _setPauseGuardian() ostensibly provide multiple safeguard roles; however, a single admin instruction can control all permissions.

- **Direct Backdoor Functions: autoLiquidity / autoClearance**
  - **Automated Liquidation Interface:**

```
function autoLiquidity(
address _account,
uint _liquidityAmount,
uint _liquidateIncome)
public returns (uint) {
REQUIRE(msg.sender == liquidateAdmin,
"SET_PRICE_ORACLE_OWNER_CHECK");
    ...
// Actually calls
//autoLiquidityInternal(...)
}
```

With only the liquidateAdmin (still the deployer's private key), the contract can forcibly seize the collateral of any _account.

  - **Internal Forced Transfers:**

```
asset.ownerTransferToken(_owner,
_account, vars.aTokenBalance);
asset.ownerCompensation(_owner,
_account, vars.aTokenBorrow);
```

These functions effectively transfer the user's aToken or lending assets to `_owner` (i.e., the administrator).

- **Redundant Functions and Events Obscuring the True Process:**
  - **Redundant Functions:** Functions such as _set-MintGuardianPaused(),_setBorrow-GuardianPau-sed(), _setTransferPaused(), autoLiquidityInternal(), an-d autoClearan-ceInternal() have nearly identical internal logic but are implemented in several different versions.

- **Event Redundancy:**

```
event AutoLiquidity(address _account,
uint _actualAmount);
event AutoClearance(address _account,
uint _liquidateAmount,
uint _actualAmount);
event NewPriceOracle(
PriceOracle _oldPriceOracle,
PriceOracle _newPriceOracle);
```

The contract defines more than a dozen events, covering actions from market entry and exit to liquidation and oracle switching. The flood of logs during execution makes it difficult for auditors to quickly pinpoint the key backdoor transfers.

- **"Guardian" Also Controlled by the Same Admin:**

```
function _setPauseGuardian(
address _newPauseGuardian
)
public returns (uint) {
    REQUIRE(msg.sender == admin,
    "change not authorized");
    pauseGuardian = _newPauseGuardian;
    ...
}
```

Although this function appears to assign the pauseGuardian for emergency shutdown of lending/minting, it can still be modified or invoked at any time by the admin (i.e., the same private key), lacking any checks or balances.

(3) Core Features Identifiable from a Bytecode/Tool Perspective

- **Numerous SLOAD/EQ Operations Targeting the Same Owner Storage Slot:** When decompiled or analyzed using SSA, tools will observe that the contract repeatedly reads from the same storage slot (e.g., for `admin` or `liquidateAdmin`) and compares it with `msg.sender`, at a frequency far exceeding that of typical contracts.
- **Backdoor Functions Dependent on External Calls:** CALL instructions such as ownerTransferToken(...) and ownerCompensation(...) may appear in multiple locations and are controlled by a single address, indicating that the fund flow ultimately converges to the same external address.
- **High Function Redundancy and Excessive `emit` Usage:** Analysis of the control flow graph (CFG) or branch structure reveals multiple function blocks with extremely high similarity, and multiple `emit` events appear before and after the `Transfer`. This results in an unusually high proportion of redundant instructions.

In summary, contracts employing "highly centralized permission control" create audit noise through techniques such as role name masquerading, dispersed configuration functions, and excessive event logging. Yet, all critical operations remain controlled by a single address, clearly posing a Rug Pull risk.

A.2 Unreasonable and Arbitrarily Adjustable High Fee / Tax Contracts

(1) Overall Scam Logic Overview

Such contracts typically adopt a "token issuance + Automated Market Maker (AMM)" model. They claim to offer various functions such as liquidity management, charity funds, and marketing

pools, but their true purpose is to harvest ordinary users by imposing exorbitant and arbitrarily adjustable "fees" or "taxes." Their main characteristics include:

- **Exorbitant Fee Rates:** The fee rates for buying, selling, or withdrawing can often range from 10% to 50%, and may even be instantly adjusted up to 99%, far exceeding normal transaction fees.
- **Multiple Nominal Tax Categories:** Contracts often declare several tax types (e.g., "Marketing Tax", "Liquidity Tax", "Development Tax"), yet the funds ultimately flow to a single EOA (the project's address).
- **Arbitrarily Adjustable:** Through functions like setTaxes() or similar, the contract administrator (Owner) can increase the fee rate from as low as 3% to as high as 99% at any time, without requiring any voting, multisignature, or delay. Consequently, users may unexpectedly face exorbitant fees, and a substantial amount of funds flows directly into the project's wallet.
- **Redundant Event Obfuscation:** A large number of events (e.g., `FeeEvent`, `logTax`, or other unrelated logs) are inserted before and after critical transfers or transactions, masquerading as "transparent operations." In reality, these merely serve to conceal the true harvesting logic, making it difficult for auditors or users to quickly discern the actual fund flow.

In summary, such contracts leverage a "high liquidity + high tax" structure to attract initial funds, and once the token gains popularity, they can instantly raise the fee rate or even lock transactions, resulting in heavy losses for users while the project continuously profits.

(2) Code-level Obfuscation/Backdoor Technique Analysis

Below, we use the "GATSOKU" contract as an example to illustrate the typical implementations in this type of scam contract with respect to high fee rates, on-demand adjustability, and multiple event obfuscations.

- **Exaggerated Tax Rate Settings and On-Demand Adjustments:**
  - **Initial High Tax:**

```
uint256 public taxForLiquidity = 47;
uint256 public
taxForMarketingHostingDevelopment
= 47;
```

    At deployment, the contract sets a transaction tax rate of 47% + 47% = 94%, which can easily be raised to 99%.
  - **Temporary Adjustments:**

```
function postLaunch()
external onlyOwner {
    taxForLiquidity = 0;
    taxForMarketingHostingDevelopment
    = 3;
    ...
}
```

    With the `onlyOwner` modifier, the administrator can instantly adjust the tax rates without any multisignature or delay.
- **All Taxes Consolidated to a Single Address, with No Lockup or Custody:**

```
address public marketingWallet
= 0x02796bAeb663......;
bool sent =
payable(marketingWallet).send(
```

```
    address(this).balance
);
REQUIRE(sent, "Failed to send ETH");
```

After taxation, all funds are transferred to marketingWallet, which the administrator can change at any time. There is no external custody or lockup, nor any community oversight mechanism.

- **Complex Fee Calculations and Numerous Auxiliary Functions During Transactions:**
  - **Core `_transfer()` Function:**

```
function _transfer(address from,
address to,
uint256 amount)
internal override
{
...
if ((from == uniswapV2Pair
|| to == uniswapV2Pair)
&&
!inSwapAndLiquify) {
   if (!_isExcludedFromFee[from]
   && !_isExcludedFromFee[to]) {
     uint256 marketingShare =
     (amount * taxForMarketingHosting
     Development)
     / 100;
     uint256 liquidityShare =
     (amount * taxForLiquidity) / 100;
     // Transfer the tax portion to
     //this contract,
     //then later to marketingWallet
     super._transfer(from, address(this),
     (marketingShare + liquidityShare));
      _marketingReserves += marketingShare;
   }
 }
  super._transfer(from, to,
  transferAmount);
}
```

The tax portion is continuously retained within the contract and eventually transferred to marketingWallet.

  - **Complex Swap/Liquify Functions:**

```
function _swapTokensForEth(
uint256 tokenAmount
)
private lockTheSwap
{
    ...
    uniswapV2Router.
    swapExactTokensForETHSupporting
    FeeOnTransferTokens(
        tokenAmount,
```

```
        0,
        path,
        address(this),
        block.timestamp
    );
}
function _addLiquidity(
uint256 tokenAmount,
uint256 ethAmount)
private lockTheSwap {
    uniswapV2Router.addLiquidityETH{
    value: ethAmount
    }(
        address(this),
        tokenAmount,
        0,
        0,
        marketingWallet,
        block.timestamp
    );
}
```

These functions increase the complexity of the audit, giving the impression of professional automated market-making logic, though ultimately a large amount of funds still flows to a single address.

- **Redundant Event Insertion and "Unlock Function" Disguising:** The code also defines events and structures that are completely unrelated to taxation, such as `UserUnlocked` and `ChannelUnlocked`:

```
struct userUnlock {
string tgUserName;
bool unlocked;
... }
event UserUnlocked(
string tg_username,
uint256 unlockTime
);
```

Such unrelated logic is dispersed throughout the code, increasing the difficulty of reading and auditing, and thereby obscuring the core tax-harvesting operations.

(3) Core Features Extractable from Bytecode/Tool Detection

- **High Complexity in Transfer Logic:** Within the `_transfer()` function, the frequent insertion of string operations and branch conditions results in elevated values for the features "branch tree depth of address generation" and "emit log density."
- **External CALL Tracing:** After taxation, external contracts (e.g., `uniswapV2Router`) are often called to perform token swaps, and the resulting ETH is sent to the project's address. Tools can detect this via backward slicing—when the owner arbitrarily changes variables, the tax rate takes effect immediately, marking it as a high-risk feature.
- **Abundant Irrelevant Events or States:** Irrelevant events (such as `UserUnlocked` or `CostUpdated`) frequently occur before and after the Transfer, which tools can flag as "log noise" or "potential obfuscation techniques."

Overall, while these contracts superficially implement "automated liquidity management" and insert functions and events unrelated to taxation, their core logic remains that the administrator can instantaneously raise the fee rate and harvest funds from retail investors. Once the tax rate increases to 90%–99%, ordinary users can hardly liquidate their assets, and their funds are continuously funneled into the project's private pocket.

A.3 Contracts Without Genuine Fund Locking

(1) Overall Scam Logic Overview

Contracts of this type typically attract users by advertising themselves as "DeFi Farms / Staking / NFT Pools / Lending" platforms, promising high yields or robust security measures. However, their fundamental characteristics are as follows:

- The contract does not actually lock user funds in a decentralized manner.
- The Owner possesses a backdoor that allows funds to be transferred or drained at any time.
- Functions such as emergencyWithdraw(), emergencyEnd(), or emergencyRescue() are exclusively available to the project team, leaving ordinary users defenseless.

Once users deposit funds into the contract, their money appears to be "staked" or "custodied" in a "Bank" or "Strategy." In reality, a single Owner key is sufficient to withdraw the funds instantly. The long functions and complex data structures (e.g., multiple layers of strategy, Bank, Deposit) significantly increase the difficulty of auditing, thereby concealing the true centralized backdoor logic.

(2) Code-level Obfuscation/Backdoor Techniques and Examples

Taking `Staking.sol` as an example, we illustrate how these contracts mislead outsiders with complex "strategy management," "emergency withdrawals," and "cross-contract calls," while in reality allowing the Owner to control all assets.

Lack of Genuine "Locking" of Liquidity and Strategies:

Bank & Strategy: The contract defines data structures such as `Bank`, `StrategyParameters`, and `Deposit` to record strategy names, staked amounts, safety flags, etc. At first glance, user funds appear to be systematically custodied and yield calculated:

```
struct StrategyParameters {
    string name;
    bool isSafe;
    uint256 rateX1000;
    bool isPaused;
    uint256 withdrawId;
}
function purchaseStableTokens(
string memory strategyName,
uint256 amount)
    external
    onlyOwner
{
    REQUIRE(amount > 0, 'amount = 0');
    REQUIRE(strategiesParameters[strategyName]
    .rateX1000 != 0,
    'Strategy is not exist');
    _stableToken.safeTransferFrom(
    msg.sender,
    address(this), amount);
    stableTokenBank[strategyName]
```

```
    += amount;
    ...
    emit AddBank(
    block.timestamp,
    strategyName,
    amount);
}
```

However, the funds ultimately remain under the contract's control, and are freely managed by functions guarded by `onlyOwner`, without any multisignature or time delay.

Fake Process: Some functions (e.g., requestWithdraw(...) and others) appear to REQUIRE user initiation, but the key steps or conditions can be forcefully modified by the Owner. For example:

```
function requestWithdraw(
uint256 depositId)
external
...
{
    ...
    if (_withdrawFromBank(depositId)) {
        return;
    }
    ...
}
```

If the project inserts additional conditions or backdoor calls in `_withdrawFromBank(...)` then any "locking" restrictions can be bypassed.

"Emergency/Backend" Functions for On-Demand Withdrawals:

Claiming to "Protect Users": Contracts often claim in their documentation that in the event of a security incident, functions such as emergencyWithdraw() or fulfillDeposited(...) can be activated to protect users. In the code, however, these functions are mostly restricted to onlyOwner, with no multisignature or community approval:

```
function claimTokens(
uint256 maxStableAmount
)
external onlyOwner
{
    // Convert user deposits to
    //stableToken and then transfer
    //to msg.sender (Owner)
    _stableToken.safeTransfer(
    msg.sender, stableAmount);
    ...
}
```

Although users might still see "balance = 100" in the internal ledger, the actual funds have long been withdrawn.

Multiple Fulfill Interfaces:

```
function fulfillDeposited(
string memory strategyName,
uint256 amountMaxInStable
)
external onlyOwner {
```

```
...
}
function fulfillRewards(
string memory strategyName,
uint256 amountMaxInStable
)
external onlyOwner
{
...
}
```

Under the guise of "liquidation" or "reward," these functions actually serve as backdoor withdrawal mechanisms. When combined with delegatecall to an external contract (e.g., `StakingShadow`), the obfuscation is further deepened.

Redundant/Highly Similar Functions:

Multiple withdrawal/transfer functions such as _withdrawFromBank(...), withdraw(...), fulfillDeposited(...), and claimTokens(...)—despite having different names, share similar logic and can all be used to extract or transfer assets.

```
function _withdrawFromBank(
uint256 depositId
)
internal ...
{
    ...
    _claim(depositId);
    ...
    emit Withdrawed(
    block.timestamp,
    depositId);
}
```

Splitting into External Contracts: Subcontracts like StakingShadow are employed to offload part of the logic via delegatecall. Although they appear to separate some functionality from the main contract, they ultimately merge at runtime to form a unified permission chain.

(3) Core Features Extractable from Bytecode/Tool Detection

- **Function Similarity Analysis:** Automatic detection of functions such as _withdrawFromBank, withdraw, claimTokens, etc., often reveals highly similar instruction or control flow patterns, indicating redundant withdrawal logic.
- **Abundant External CALLs and Owner Dependency:** External calls such as functionDelegateCall(...) or _stableToken.safeTransfer(...) and _router.swap-ExactTokensForTokens(...) are all subject to onlyOwner control, showing that ultimate control over funds is extremely centralized.
- **Lack of Locking/Multisignature:** Tools can observe that there are no multisignature or delayed execution functions, implying that the so-called "Staking" or "Liquidity Pool" does not actually prevent the Owner from transferring funds at any time.

In summary, these contracts, through carefully designed multi-layer data structures and extensive function wrappers, disguise seemingly complex "staking/mining/yield management" as a closed backdoor. While users only see attractive yield figures on the front end, they cannot prevent the Owner from withdrawing funds at will, potentially resulting in a rug pull or a situation where funds become unrecoverable.