

# Understanding and Characterizing Obfuscated Funds Transfers in Ethereum Smart Contracts

Zhang Sheng  
Hefei University of Technology  
China  
dcszhang@foxmail.com

TAN Kia Quang  
Singapore Management University  
Singapore  
kq.tan.2023@msc.smu.edu.sg

Shen Wang  
Singapore Management University  
Singapore  
shenwang918@gmail.com

Shengchen Duan  
Singapore Management University  
Singapore  
sc.duan.2024@phdcs.smu.edu.sg

Kai Li\*  
San Diego State University  
United States  
kli5@sdsu.edu

Yue Duan<sup>†</sup>  
Singapore Management University  
Singapore  
yueduan@smu.edu.sg

## ABSTRACT

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 investigates how obfuscation amplifies the financial risks of 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 1.03 million Ethereum contracts, we isolate over 3,000 highly obfuscated contracts and discover two scam types, three high-risk contract types, and MEV bots, that are deeply coupled with various obfuscation maneuvers such as assembly usage, dead code, and deep function splitting. We further reveal that obfuscation substantially increases the scale of financial damage and evasion time. 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 80% (non-obfuscated) to 12% (obfuscated) in real-world scenarios. These findings underscore an urgent need for enhanced “anti-obfuscation” analysis techniques and broader community collaboration to mitigate scam contract proliferation 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, 11, 68] and digital art trading [51]. Along with it, security issues are also on the rise. Recent reports [26, 73] 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 a variety of techniques to detect [53, 54, 64] and analyze [23, 58] these emerging security issues. These techniques rely mainly on static program analysis [28] and rule-based matching [3], which have been proven to be highly effective and efficient in detecting early malicious smart contracts, which are 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 [74, 79] confirm that obfuscation has become the primary means by which attackers conceal genuine malicious transfer or backdoor control logic, splitting functions, adding redundant instructions, using assembly code, and other obfuscation techniques. When attackers employ these obfuscation techniques in their malicious smart contracts, the static analysis and matching rules adopted by traditional detection tools are often disrupted, leading to high detection inaccuracies [57, 80] and further worsening financial losses [24, 70, 72, 81].

Although obfuscation techniques are witnessed to be increasingly employed in smart contracts to evade security analysis 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 the development of effective defense mechanisms and the mitigation of associated security risks. In this paper, we report the first systematic study on the obfuscation of funds transfer operations, which are the most essential and security-critical activities of a smart contract [46].

Specifically, 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:

\*Corresponding author

<sup>†</sup>Corresponding author

- **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 the aforementioned research questions, we systematically develop a taxonomy to define and characterize different obfuscation techniques on funds transfer operations and propose `OBFPROBE`, an EVM bytecode analysis tool that can accurately uncover how different obfuscation techniques manifest in real-world smart contracts. With the help of this analysis framework, we conduct a series of comprehensive studies on a huge real-world dataset of 1,042,928 smart contracts.

Here we highlight some interesting discoveries:

- (1) Our analysis framework `OBFPROBE` identified 3,128 highly obfuscated contracts in the wild. Within them, 153 contracts harbor substantial security risks, placing user funds totaling close to \$100 million at risk.
- (2) Compared to non-obfuscated samples, obfuscated scam contracts demonstrate a significantly larger financial impact, with their highest recorded inbound funds being roughly 2.4 times greater and clear periods of intensified victimization occurring between 2019 and 2023.
- (3) Deep obfuscation can seriously undermine existing detection mechanisms (e.g., the accuracy of a state-of-the-art Ponzi detector drops from 79% to 12% when applied to obfuscated code).

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

- We developed `OBFPROBE`, the first EVM bytecode obfuscation analyzer leveraging seven bytecode-level features and a Z-score representation model to detect obfuscated transfer logic automatically.
- We performed the first large-scale measurement study on more than 1.04 million Ethereum smart contracts, revealing that 0.3% exhibit complicated obfuscation and categorizing three major threat contexts: MEV bots, traditional scam tokens, and highly centralized permission wrappers.
- We quantified the impact of obfuscation on financial damage and detection: obfuscated contracts can extract up to

201.74 ETH per scam and trigger significant victim outbreaks, while state-of-the-art detectors suffer a decrease in accuracy from 79% to under 12% under deep obfuscation.

- We open-sourced the implementation of our prototype and all study data/artifacts to facilitate future research<sup>1</sup>.

## 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 various data transactions [4]. Its core features—decentralization, transparency, immutability, and security—position it as a transformative technological innovation across a wide range of industries [65].

Smart contracts are programs built on top of blockchains that autonomously execute contractual terms when predefined conditions are satisfied, eliminating the need for intermediaries [50]. 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 [52]. Smart contracts have been widely applied in various fields, including financial transactions, identity verification, supply chain management, and insurance [59]. In the financial sector, smart contracts facilitate automatic settlement and payment on the blockchain, significantly enhancing transaction efficiency and transparency while reducing the need for intermediaries and human involvement. As blockchain technology evolves, smart contracts will play an increasingly important role across multiple industries [65].

### 2.2 Code Obfuscations

Code obfuscation is the process of transforming a program into an equivalent version that preserves its semantics but significantly impedes human understanding and automated analysis. Originating in the late 1990s as a software-protection technique, obfuscation now finds applications both in legitimate software-protection (e.g. DRM, piracy resistance) and in malware-evasion (hindering static and dynamic analysis). While no obfuscation can guarantee unbreakable protection, carefully designed transformations raise the attacker’s cost—forcing reverse engineers to invest more time, specialized tools, or human effort to recover the original logic.

Research over the past years continues to refine a taxonomy of obfuscation techniques, typically grouped into control-flow obfuscation, data obfuscation, layout (lexical

<sup>1</sup>[https://github.com/dcszhang/Obfuscation\\_Tool](https://github.com/dcszhang/Obfuscation_Tool)

obfuscation, and instruction substitution. Control-flow obfuscation aims to alter the program’s control-flow graph (CFG) without changing semantics, employing opaque predicates, bogus branches, control-flow flattening (e.g. dispatcher loops) or virtualization-based transformations that compile sensitive code into custom bytecode (e.g. VMProtect) [34, 56]. Data obfuscation focuses on concealing computation logic by encoding variables, splitting values, or applying mixed Boolean-arithmetic (MBA) expressions that rewrite simple operations into complex algebraic equivalents—a method effective against pattern-based decompilers and symbolic engines but sometimes simplified by compiler optimizations [37]. Layout obfuscation modifies non-functional code aspects such as renaming identifiers, stripping or reordering symbols and functions, and inserting NOPs or dummy code; albeit low-cost, such lexical changes provide limited protection if used alone [10]. Instruction substitution replaces code sequences with semantically equivalent but syntactically distinct alternatives—for instance, unrolling multiplications into shift-add sequences or recasting boolean logic into arithmetic forms—to diversify byte patterns and defeat signature-based analysis and metamorphic malware detection [56].

### 3 TAXONOMY OF OBFUSCATED FUNDS TRANSFERS

In this paper, we primarily focus on funds transfer operations, which are the most essential and security-critical activities of a smart contract [46]. To answer RQ1 and further conduct our study, 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 our deep domain knowledge of Ethereum smart contracts.

Our taxonomy development starts 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 [62], 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

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

- *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. It 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 strategies. By exhaustively mapping each strategy to one or more elements of a funds transfer operation, we ensure our taxonomy is comprehensive and covers all fundamental ways to hide transfer operations in Ethereum smart contracts.

Please note that the source code listings below are for illustration purposes, while our analysis is done at the bytecode level.

#### 3.1 Obfuscation of *addr*

The *addr* element, which is essentially a string that represents a 20-byte Ethereum address, indicates the recipient address of a funds transfer operation. We consider four obfuscation methods, stemming from traditional string obfuscation techniques [49].

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

```

1 // Step 1: derive seed from block data
2 bytes32 seed = keccak256(...);
3 // Step 2: extract intermediate bytes
4 bytes20 part = bytes20(seed);
5 // Step n.....
6 // compute recipient address
7 address rec = address...(uint256(part));
8 // core transfer
9 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.

```

1 // split address string into parts
2 string memory s1 = "0x";
3 string memory s2 = "a1b2c3";
4 string memory s3 = "d4e5f6";
5 // concatenate at runtime
6 string memory full = string(s1, s2, s3);
7 // parse back to address

```

```

8 address rec = parseAddr(full);
9 rec.transfer{value, ...}("");

```

**Listing 2: T2. Complex String Operations**

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

```

1 interface AddrPro {
2   function getAddr() external returns (address);
3 }
4 // fetch hidden address from another contract
5 address provider = 0x1234...;
6 address rec = AddrPro(provider).getAddr();
7 rec.transfer{value: value}("");

```

**Listing 3: T3. External Contract Calls**

**T4. Control-flow complexity.** Dynamic conditions (e.g., `block.timestamp`) select among multiple branches—some dead—each leading to different addresses. This conditional complexity obscures which branch yields the actual `addr`.

```

1 address rec;
2 if (block.timestamp % 3 == 0) {
3   for (uint i = 0; i < 2; i++) {
4     if (i == 1) {
5       if (msg.sender == owner) {
6         rec = addrA;
7       } else {
8         rec = addrB;
9       }
10    }
11  }
12 } else {
13   rec = addrC;
14 }
15 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 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.

```

1 // meaningless loop
2 for (uint i = 0; i < 5; i++) {
3   uint tmp = i * 42;
4 }
5 // no-op arithmetic
6 uint x = (1 + 2) - 3;
7 // core transfer hidden among noise
8 address rec = 0xAbCd...;
9 rec.transfer{value, ...}("");

```

**Listing 5: T5. Camouflage Instructions**

**T5. Camouflage Instructions.** By injecting large numbers of meaningless loops, arithmetic operations, and No-ops 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 analysis tools to isolate the actual CALL context.

**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.

```

1 function withdraw(uint value) public {
2   _doTransfer(value);
3 }
4 function start(uint value) public {
5   _doTransfer(value);
6 }
7 function _doTransfer(uint value) internal {
8   // same transfer code reused
9   address rec = 0xAbCd...;
10  rec.transfer{value, ...}("");
11 }

```

**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, concealing the real log data that corresponds to the transfer.

```

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

```

Listing 7: T7. Irrelevant Log Events

## 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 so as to answer our RQ2. In this section, we elaborate on the design and implementation of our system.

### 4.1 System Overview

Figure 1 shows an overview of OBFPROBE. At a high level, it first converts a given smart contract bytecode to the static single assignment intermediate representation (SSA IR) using an existing tool named Rattle [14]. After that, it scans the IR to detect all funds transfer operations (stage 1). For each transfer, our system extracts seven pre-defined obfuscation features from the SSA IR (stage 2). Finally, it applies a Z-score representation model [15], 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 to an obfuscation score (stage 3), indicating the degree of obfuscation applied to 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, as summarized in Table 1.

**F1. Number of steps in address generation.** This feature represents the number of steps required to obtain `addr` in a transfer operation. Specifically, starting from each `CALL` operation in the contract’s SSA IR, we perform backward dataflow analysis on the variable `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 again analyze the data flow of the variable `address` to count all instructions involving 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`, `DELEGATECALL`, or `STATICCALL`). If so, we set a binary feature F3 to `TRUE`. Otherwise, it is set to `FALSE`.

**F4. Height of branch tree.** We use the height of the branch tree to represent the complexity of the control flow associated with the transfer operation by analyzing conditional branch structures (`JUMPI` instructions). Starting from the identified transfer operation, we trace backward 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 follows. Let  $|U|$  be the number of *effective* instructions contributed to transfers and critical state updates, and  $|T|$  be the total number of instructions in the transfer-residing function. Then, TIR is computed as:

$$\text{TIR} = \frac{|U|}{|T|}, \quad (2)$$

Specifically,  $U$  is the union of critical transfer instructions  $C$  and state modifications  $S$ .  $C$  is obtained 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, including 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) [55] to aggregate the node embeddings and

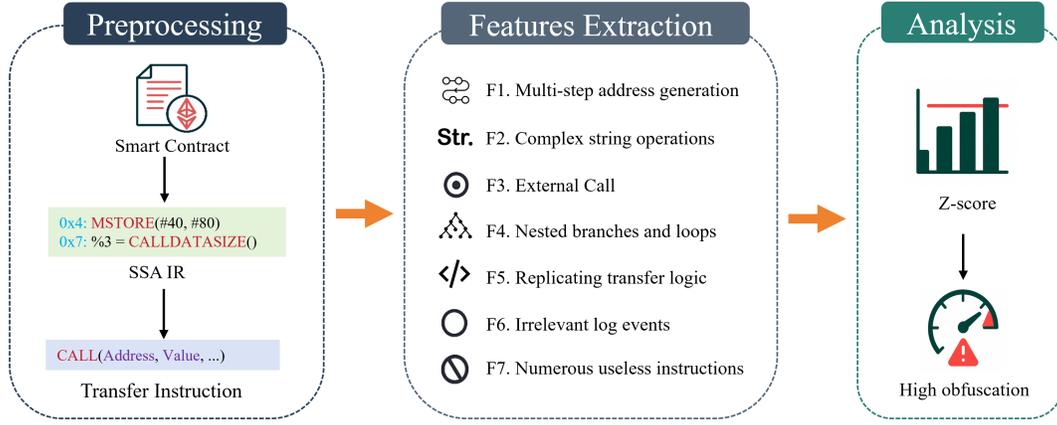


Figure 1: Overview of OBFPROBE.

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 generation
T4. Control-flow complexity	F4. Max branch/loop nesting depth in address/value derivation.
T5. Camouflage instructions	F5. Transfer-related instruction ratio (TIR)
T6. Replicated transfer logic	F6. Inter-function code similarity of transfer-containing functions
T7. Irrelevant log events	F7. Relevance between the semantics of the log event and the transfer operation

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 high similarity score indicates that there exist two transfer-containing functions that exhibit highly similar transfer logic.

**F7. Relevance of log events.** To detect misleading and irrelevant log events emitted during transfer operations, we identify emit operations within two hops in the CFG from the node representing the transfer operation. If an emit operation is identified, we employ the Foundry fuzzing framework [43] to execute the corresponding function and capture the emitted logs at runtime. After that, we perform an automated semantic analysis using GPT-4o [2] to determine whether the log messages are relevant to the transfer operation. If the relevance score is low (e.g., referencing a different recipient or transaction type), we set the feature to FALSE. Otherwise, the feature is set to TRUE.

Overall, we believe the above seven bytecode-level features can allow us to effectively identify obfuscated transfer-related functions in smart contracts.

### 4.3 Obfuscation Z-score Model

Upon extracting the seven obfuscation features, we 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 a weighted sum to obtain the obfuscation Z-score for each contract:

$$Z_{\text{score}} = \sum_{i=1}^7 \frac{x_i - \mu_i}{\sigma_i}, \quad (3)$$

where  $x_i$  is the  $i$ -th feature value for a given contract, and  $\mu_i$  and  $\sigma_i$  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 obfuscation features and their means in standard deviations, indicating the degree of obfuscation applied to the transfer operation for a given smart contract.

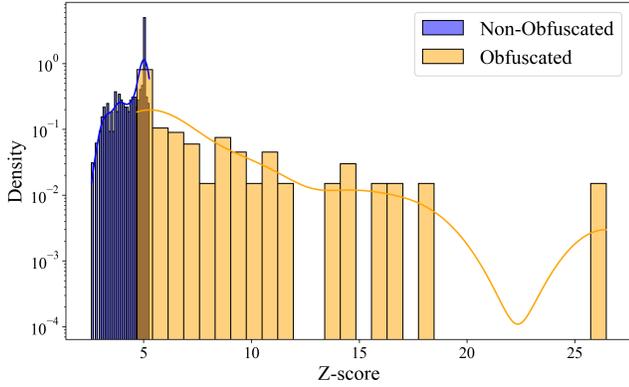
### 4.4 Evaluation of OBFPROBE

We evaluate the performance of OBFPROBE on a dataset of 453 Ponzi scam contracts. Table 2 summarizes the data sources and our manual classification.

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

**Table 2: Real-world Ponzi Scam Dataset**

Data Source	CRGB [35]	137
	SourceP [39]	316
<b>Total</b>		<b>453</b>
Classification	Obfuscated	92
	Non-Obfuscated	361



**Figure 2: Z-score Distribution**

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 two groups’ Z-scores, which is presented in Figure 2. 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.

In addition, we conduct a more detailed statistical investigation, 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.

**Table 3: Z-score Statistics**

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
<b>Welch’s t-test</b>	$t = -6.172, p = 0.000000$				

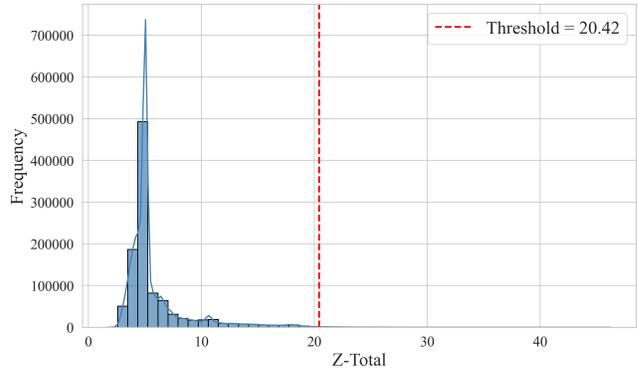
In summary, the evaluation result shows that OBFPROBE can accurately extract the pre-defined obfuscation features and calculate the Z-scores, which can effectively differentiate obfuscated and non-obfuscated smart contracts.

## 5 REAL-WORLD STUDY

In this section, we aim to answer RQ2 to study the prevalence of obfuscation techniques in smart contracts.

### 5.1 Prevalence Study

To conduct the study, we collect 1,042,923 smart contract bytecodes deployed on the Ethereum mainnet spanning from Jun 2022 to Oct 2024, and then leverage OBFPROBE to extract the seven obfuscation features from these contracts as well as to calculate their Z-scores. Next, we examine the top 0.3% most heavily obfuscated contracts to identify key hidden behaviors and their intrinsic obfuscation mechanisms. Finally, we present three case studies, namely MEV bot contracts, scam contracts, and extremely centralized contracts, to illustrate representative obfuscation strategies.



**Figure 3: The Z-score Distribution on the Ethereum Mainnet. The dashed red line marks the top 0.3% cutoff.**

**Distribution of Z-Score.** Figure 3 shows that the distribution of the aggregate obfuscation score  $z\_total$  is strongly right-skewed. Most contracts fall in the range  $3 \leq Z \leq 8$ , with a peak near  $Z \approx 6$ . For  $Z > 10$ , the bar heights decline steadily, forming a long tail that extends to the highest observed scores. This visual pattern indicates that while most contracts exhibit moderate obfuscation, a small subset reaches much higher levels.

**Table 4: Zscore Statistics**

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

Table 4 summarizes the distribution of the aggregate obfuscation score ( $z\_total$ ) across 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 and represents the 95% CI upper bound, calculated as

$$4.571 + t_{0.975,360} \times \frac{0.641}{\sqrt{361}} \approx 4.637. \quad (4)$$

**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 %

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.

## 5.2 Detailed Analysis

To better understand how obfuscation techniques are used in the wild, we select the top 0.3% (3,128 contracts) as highly suspicious targets for in-depth analysis. We empirically set a 0.3% threshold based on two considerations. First, in statistics, if a variable is approximately normally distributed, roughly the rightmost 0.3% of the samples typically correspond to extreme outliers [13, 31, 71] (i.e., values exceeding approximately three standard deviations from the mean). Therefore, the tail region corresponding to about  $3\sigma$  is commonly regarded as highly suspicious and requires priority auditing. Second, selecting the top 3,000+ samples out of 1.04 million contracts significantly reduces our manual efforts while still adequately covering the typical values of the distribution. Thus, choosing 0.3% satisfies both the need to focus on the most extreme suspicious cases and to maintain a reasonably feasible auditing scale. In summary, selecting 0.3% as the high-suspicion interval does not imply that we assume the data strictly follows a normal distribution; rather, it leverages a general empirical value from the extreme tail of a normal distribution to provide an intuitive and relatively reasonable cutoff for prioritizing contracts with extremely high obfuscation.

To this end, we sort these contracts by the total amount of influx funds and examine each contract from the top. Our study uncovers that all of them are indeed equipped with obfuscation techniques, and 153 contracts pose very *high risks* by hiding four potentially malicious and suspicious behaviors via heavy obfuscation, jeopardizing 100M USD

worth of Ethers. We present case studies for the four malicious behaviors hidden under obfuscation and elaborate on their typical hidden strategies.

## 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 [41], and sandwich attacks [42]. 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 reveals four unique strategies.

- (1) **Fallback only.** Under this strategy, 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 pseudo-branches. 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.
- (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 strategy is relevant to F1, F2, and F4 features 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 branching.
- (3) **Address obfuscation.** This method 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 also directly related to F6.
- (4) **Runtime constraints.** This strategy 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 F4 and F7 features since it will introduce additional

conditional branches in the control flow and often emits irrelevant or misleading logs.

**Arbitrage transactions of MEV bots.** To gain more insights into the impact of different obfuscation strategies adopted by MEV bots, we select one representative MEV bot contract for each strategy and analyze its transaction activity. We can draw several interesting observations from a time series analysis of arbitrage transactions submitted to each MEV bot, which is presented in Figure 4. Notably, the fallback-only 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 distortion contract ramps up gradually and sustains high volumes, while runtime constraint and address obfuscation strategies show slower growth and greater variability in 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 [21], 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 [40].

**Obfuscation strategies.** Here, we use yUSDC contract, an obfuscated Ponzi contract detected by OBFPROBE, to exemplify how the contract obfuscates the logic of forcing token holding, implementing buyback and burn mechanisms, and using multi-layered function wrappers.

- *Multi-level deduction and address generation.* When users withdraw tokens, there is a cumbersome fee deduction process involving 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.
- *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.
- *Abundant "Buyback and Holding Check" Strings/Events.* By analyzing event names or string constants, OBFPROBE detects terms like “Burn”, “MLMReward”, or “RetirementYield”. These words are typically associated with forced

token holding, token burning, or multi-level commissions, typical keywords of Ponzi/pyramid schemes.

Overall, we observe that Ponzi scam contracts leverage heavy obfuscation techniques to hide their core malicious logic to avoid detection.

## 5.5 Case III: Fake Decentralization

This type of contract claims that the control of the contract is decentralized to attract participation, 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, resulting in financial losses to participants.

**Obfuscation strategies.** Through our detection and investigation, we see the contracts adopt three obfuscation strategies. First, the contract duplicates ownership transferring logic in multiple functions, 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, 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.

## 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 private key, 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

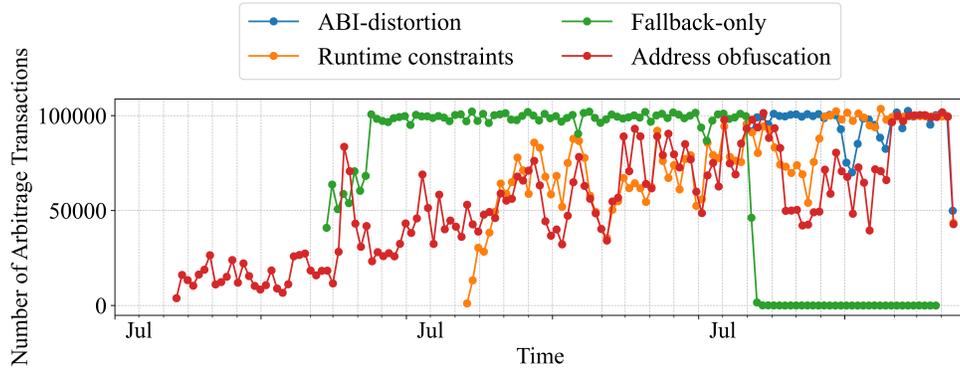


Figure 4: Time Series Analysis of transaction Volumes

an `onlyOwner`-protected function, with all collected fees routed to one EOA.

- **Lack of fund locking.** “Staking” or “farm” contracts advertise emergency withdrawals or multi-layer strategies, but there often exist functions (e.g., `emergencyWithdraw()`, `requestWithdraw()`, or `claimTokens()`) that are only accessible by the owner and can enable instant drainage.

**Obfuscation strategies.** We investigate the typical obfuscation strategies adopted by these contracts and see that they often use redundant functions, events, and misleading names to obscure their true centralization. Here, we list a few very representative ones. Please refer to Appendix A for a more comprehensive breakdown.

- *Role masquerading.* They tend to create multiple roles that point to the same address.

```
1 constructor() {
2   admin = msg.sender;
3   liquidateAdmin = msg.sender;
4 }
```

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

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

- *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.

```
1 function setTaxes(uint256 buyTax, uint256
2   sellTax) external onlyOwner {
3   taxForBuy = buyTax;
4   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.

```
1 function emergencyWithdraw(uint256 amount)
2   external onlyOwner {
3   token.safeTransfer(msg.sender, amount);
4 }
```

- *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.

**Summary.** Our study shows that it is prevalent for real-world contracts to employ various obfuscation techniques to hide their code logic, especially malicious behaviors such as MEV, Ponzi, and fake decentralization, posing huge security risks to end users.

## 6 FINANCIAL IMPACT ANALYSIS

After understanding the prevalence and real-world usages of obfuscation techniques in smart contracts, we investigate the financial impact of obfuscation, aiming to answer RQ3. To do so, we collect a representative dataset of scam smart contracts, use OBFPROBE to detect the existence of obfuscation, and qualitatively compare the financial impact between obfuscated and non-obfuscated scam smart contracts using several key metrics.

### 6.1 Dataset

We leverage the dataset from a prior research, Li et al. [33], which reports approximately 14K scam arbitrage bot contracts. To collect the ground truth information on obfuscation, we manually examine the dataset and label each contract as obfuscated or non-obfuscated. Eventually, we gain 9,197 unique obfuscated contracts and 3,826 unique contracts without obfuscation.

### 6.2 Comparison of Financial Loss

With the dataset, we conduct statistical analyses, visualize the inbound ETH (fund inflows) of these contracts, and calculate victim counts on the Ethereum mainnet to quantify the impact. The results indicate that obfuscated contracts exhibit a substantially more noticeable capacity for “money-grabbing” regarding economic damage and have significantly impacted users.

**Comparison of Statistical Indicators.** Table 6 summarizes the mean and maximum inbound ETH for both no-obfuscation and with-obfuscation groups.

**Table 6: Mean and Maximum Inbound ETH**

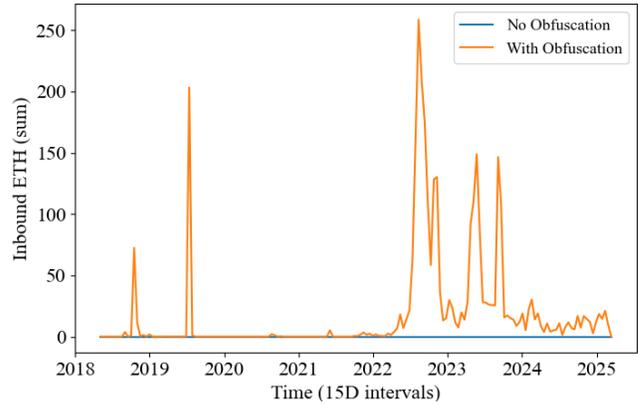
Contract Group	Mean Inbound	Maximum Inbound
No-Obfuscation	0.3403 ETH	83.62 ETH
With-Obfuscation	0.3454 ETH	201.74 ETH

As we can see, although the mean values are close (0.3403 ETH vs. 0.3454 ETH), the maximum inbound ETH for obfuscated contracts (201.74 ETH) is nearly 2.4× higher than that of non-obfuscated contracts (83.62 ETH). This indicates that obfuscated scams can secure significantly larger inflows, likely owing to their enhanced resistance to analysis and detection.

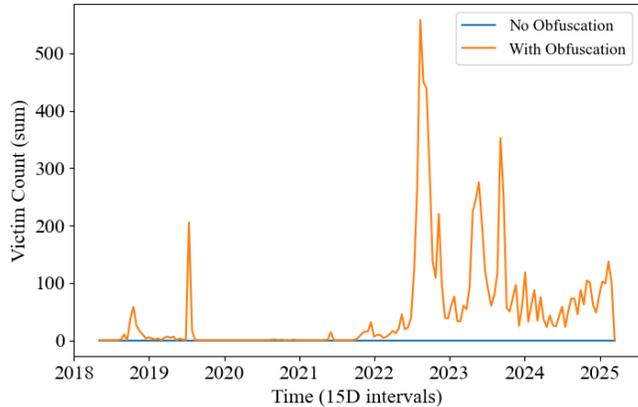
### 6.3 Time Series Analysis

To better illustrate the impact, we conduct a time series analysis on the victim counts and the aggregated inbound funds that represent the financial loss caused by the scam contracts, from 2018 to 2025, using a 15-day interval as the time unit.

**Inbound ETH.** As illustrated in Figure 5, the blue line representing the inbound funds of no-obfuscation scams remains near the zero axis throughout the period from 2018 to 2025, with only occasional minor increases. In contrast, the orange line, which represents obfuscation scams, displays multiple significant peaks, particularly in 2019, 2020, 2022, and 2023. These peaks highlight substantial financial losses for victims, with losses exceeding 200 ETH during the 2022-2023 period.



**Figure 5: Aggregated Inbound Ether Analysis**



**Figure 6: Temporal Analysis of Victim Counts**

**Victim Count.** Figure 6 presents our temporal analysis of victim counts. The orange line represents the total number of victims affected by obfuscated scams, while the blue line corresponds to those impacted by non-obfuscated scams. It is evident that the number of victims from obfuscated scams consistently exceeds that of non-obfuscated scams throughout the period from 2022 to 2025. Notably, there are sharp increases during specific intervals (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, victim counts for non-obfuscated scams remain exceptionally low during the same periods, with no significant outbreaks observed. These findings further highlight that obfuscated scams result in significantly greater harm and affect a much larger number of victims.

**Summary.** Our analysis reveals that obfuscated scam contracts are more active than non-obfuscated ones and have resulted in greater financial losses and a higher number of victims. This finding aligns with our hypothesis: *Obfuscation*

*techniques enable scam contracts to operate more covertly during their initial stages, allowing them to amass larger amounts of funds and impact a greater number of victims.*

## 7 IMPACT ON EXISTING TOOLS

Finally, we answer RQ4 by studying how obfuscation techniques can affect the effectiveness of existing malware analysis tools. To conduct this study, we run SourceP [39], a state-of-the-art tool for detecting Ponzi schemes in smart contracts, on the same dataset described in Table 2. Then, we calculate metrics, including accuracy, recall, and F1 Score, to quantitatively compare its effectiveness in handling obfuscated and non-obfuscated samples.

**Evaluation Results.** Overall, its accuracy is 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 positives exist. A closer look at the results, shown in Table 7, we can see 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.

**Summary.** Our study results demonstrate that obfuscation has a significant detrimental impact on existing detection tools (such as SourceP), causing a substantial drop in both accuracy and recall. In particular, for Ponzi schemes with obfuscation, the false negative rate can exceed 80%, rendering state-of-the-art detection tools useless. At a practical level, this observation underscores the importance of developing practical obfuscation analysis techniques, which is essential to mitigating the emerging security threats.

## 8 DISCUSSION

### 8.1 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.

**Enhanced Detection Techniques.** 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.

**De-Obfuscation Strategies.** Techniques from traditional software security, such as CFG flattening reversal and semantic normalization, can be adapted to EVM bytecode to identify critical logic like transfers and permission checks that obfuscation tries to hide.

**Community-Driven Auditing.** Establishing collaborative platforms, such as a "contract blacklist" or "high-risk obfuscation" repository, would allow researchers, auditors, and the public to tag suspicious contracts, improving transparency and enhancing collective oversight of the DeFi ecosystem.

### 8.2 Limitations

Our work has the following limitations. First, our findings are based on the Ethereum mainnet and do not cover other EVM-compatible blockchains like BSC, Polygon, or Tron. Hence, these blockchains may exhibit different obfuscation patterns. Second, our study only considers contracts deployed between 2021 and 2024. Thus, new compiler features or obfuscation techniques may emerge in future versions of Solidity or other blockchain platforms. Third, not all obfuscation strategies were covered. More advanced techniques, such as inline assembly abuse or internal EVM feature manipulation, require further investigation.

## 9 RELATED WORK

**Smart Contract Security Analysis.** Some research employs static analysis to enhance smart contract security and efficiency. USCHUNT [6] explores the balance between adaptability and security in upgradeable contracts. Madmax [22] targets vulnerabilities to prevent execution failures, while Slither [60] and Smartcheck [66] automatically detect flaws in Solidity contracts. Symbolic execution is also used to improve security; Mythril [12] analyzes EVM bytecode, EthBMC [19] combines symbolic execution with concrete validation, and Reguard [38] and Manticore [45] identify reentrancy and other bugs. Smartian [9] integrates fuzzing with static and dynamic analysis, while Confuzzius [18] leverages data dependency insights for fuzzing. CRYSQL [78] applies fuzzing to detect cryptographic defects in contracts. ContractFuzzer [29] and Sfuzz [44] apply fuzzing to uncover security issues.

Research highlights various formal verification methods to enhance smart contract security. Sailfish [7] improves state inconsistency detection, while VetSC.[17] extends DApp verification. Zeus[30] and Verx [47] focus on contract safety and condition verification. Smartpulse [63] analyzes time-based properties, Securify [67] identifies security breaches, and Verismart [61] ensures contract safety.

**Table 7: Effectiveness of SourceP for Obfuscated and Non-Obfuscated Ponzi Samples.**

Class	Total	TP	FN	Precision	Recall	F1
Non-obfuscated	361	287	74	0.79	0.79	0.88
Obfuscated	92	11	81	0.12	0.11	0.21

**Excessive Owner Control.** Centralization risk has been emphasized by Lamby et al. (2023), who reported that centralized backdoors were responsible for \$1.3 billion in DeFi losses in 2021 [32]. Lin et al. (2023) further identified centralization flaws, such as a single owner having the ability to modify transfer fees or update proxy contracts arbitrarily [36]. Similarly, Yu et al. (2025) highlighted hidden backdoors, including arbitrary fund transfers [76]. Another form of centralization, adjustable transaction fees, is frequently exploited by developers to impose excessive taxes on sales, as noted by Shiaeles and Li (2024) [25]. Liquidity scams also leverage centralization, with advanced schemes using disguised function names or dynamic withdrawal logic to evade detection [27, 36].

**Ponzi Schemes.** Ponzi schemes on the blockchain gained prominence with the advent of Ethereum, prompting significant research into their detection. Galletta et al. (2024) compiled thousands of Ponzi scheme contract samples and developed classifiers to identify these schemes, focusing on investment-dividend models where profits rely on funds from later participants [20]. While detection methods for traditional Ponzi schemes are well-established, research on emerging DeFi variants remains limited. Earlier studies primarily addressed simple fund-sharing contracts, but newer schemes, such as Forsage, incorporate multi-level referral reward systems and cyclic dependency token designs, which have drawn legal scrutiny [69].

**Advanced Anti-Auditing Techniques.** Recent studies and case analyses have revealed various deceptive tactics involving fake ownership renunciation. Normally, when developers invoke `renounceOwnership()`, the contract should no longer have an administrator. However, malicious actors employ advanced techniques to retain control. Shiaeles and Li (2024) documented a real-world token case where misleading variable names were used within the contract. For example, the public variables `owner` and `getOwner` were set to the zero address, making ownership appear to have been renounced. Yet, a hidden control variable, such as the seemingly harmless `isTokenReceiver`, still stored the original developer’s address [25]. As a result, although the contract appeared to have relinquished ownership, a fixed backdoor tied to a designated address remained. This type of fake renunciation is both highly covert and technically complex. Beyond ownership deception, many contracts use innocuous-sounding function names to disguise malicious logic. For instance, in the

forementioned case, the function name `isTokenReceiver` concealed administrative privileges. Similarly, other scam contracts may include functions with names like `failSafe()` or `emergency()`, which internally enable fund extraction or permission restoration, accessible only to the developer.

**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 [48, 75]. 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 [75]. Multi-hop contract calls, often paired with flash loans and non-standard swap paths, increase attackers’ simulation overhead [16, 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, 16]. 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 contract scams, providing comprehensive definitions, quantitative methods, and empirical analysis. We introduce a novel approach based on the detailed analysis of transfer instructions, identifying seven key quantifiable obfuscation features. Using a robust Z-score screening method, we analyze over 1.04 million Ethereum contracts, isolating approximately 3,000 (top 0.3%) highly suspicious contracts. Further quantitative financial impact analysis shows a pronounced disparity between obfuscated and non-obfuscated scam contracts: obfuscated contracts show an extreme financial extraction capability. We also demonstrate that obfuscation significantly undermines their effectiveness. Overall, our findings underscore the severe security risks posed by obfuscation and highlight the urgent

need for advanced analytical and detection methodologies to address these evolving threats, 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 Altschmidt, 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, Clark Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations.
- [11] Compound. 2025. Compound.Finance. <https://compound.finance/>.
- [12] ConsenSys. 2022. Mythril: security analysis tool for EVM bytecode. <https://github.com/ConsenSys/mythril/>.
- [13] Denis Cousineau and Sylvain Chartier. 2010. Outliers detection and treatment: a review. *International journal of psychological research* 3, 1 (2010), 58–67.
- [14] Crytic. 2025. Rattle: EVM Binary Static Analysis. <https://github.com/crytic/rattle>. Accessed: April 15, 2025.
- [15] Alexander E Curtis, Tanya A Smith, Bulat A Ziganshin, and John A Eleftheriades. 2016. The mystery of the Z-score. *Aorta* 4, 04 (2016), 124–130.
- [16] 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).
- [17] 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 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.
- [18] 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*.
- [19] 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.
- [20] L Galletta and F Pinelli. 2024. Explainable ponzi schemes detection on ethereum. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. 1014–1023.
- [21] Sangita F Gazi. 2024. In Code We Trust: Blockchain’s Decentralization Paradox. *VAND. J. ENT. & TECH. L* 27, 1 (2024), 59.
- [22] 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.
- [23] 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.
- [24] Y Gupta, J Kumar, and A Reifers. 2022. Identifying security risks in NFT platforms. *arXiv preprint arXiv:2204.01487* (2022).
- [25] 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.
- [26] 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.
- [27] P D Huynh, S H Dau, N Huppert, et al. 2024. Serial Scammers and Attack of the Clones: How Scammers Coordinate Multiple Rug Pulls on Decentralized Exchanges. *arXiv preprint arXiv:2412.10993* (2024).
- [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] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.
- [31] 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.
- [32] M Lamby, V Zieglmeier, and C Ziegler. 2023. Trusting a Smart Contract Means Trusting Its Owners: Understanding Centralization Risk. In *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 1–4.
- [33] 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>
- [34] 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*.
- [35] 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.
- [36] Z Lin, J Chen, J Wu, et al. 2024. Definition and Detection of Centralization Defects in Smart Contracts. *arXiv preprint arXiv:2411.10169* (2024).

- [37] 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)*.
- [38] 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.
- [39] P Lu, L Cai, and K Yin. 2024. SourceP: Detecting ponzi schemes on ethereum with source code. In *ICASSP '24*. 4465–4469.
- [40] 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)*.
- [41] ANGEL JAVIER BLASCO MAINAR, JOSÉ MARÍA DE LA CRUZ, and SÁNCHEZ Y ALBERTO MORENO BRASERO. [n. d.]. MAXIMAL EXTRACTABLE VALUE (MEV). ([n. d.]).
- [42] 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.
- [43] Alexandre Mota, Fei Yang, and Cristiano Teixeira. 2023. Formally Verifying a Real World Smart Contract. *arXiv preprint arXiv:2307.02325 (2023)*.
- [44] 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.
- [45] Trail of Bits. 2024. Manticore: symbolic execution tool for smart contract. <https://github.com/trailofbits/manticore/>.
- [46] 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.
- [47] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1661–1677.
- [48] 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.
- [49] 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.
- [50] 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.
- [51] X Ruan. 2022. *Exploring Vulnerabilities and Anomalies in NFT Marketplaces*. Ph. D. Dissertation. University of Guelph.
- [52] 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.
- [53] MSVPJ Sathvik and Hirak Mazumdar. 2024. Detection of malicious smart contracts by fine-tuning GPT-3. *Security and Privacy* 7, 6 (2024), e430.
- [54] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *Ieee Access* 8 (2020), 24416–24427.
- [55] 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.
- [56] 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.
- [57] 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](https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s263_paper.pdf)
- [58] 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.
- [59] Sakshi Sharma and Natasha Dutta. 2018. Development of New Smart City Applications using Blockchain Technology and Cybersecurity Utilisation. *Development* 7, 11 (2018).
- [60] Slither. 2024. Slither, the Solidity source analyzer. <https://github.com/crytic/slither/>.
- [61] 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.
- [62] Solidity. 2025. Common Patterns. <https://docs.soliditylang.org/en/v0.8.30/common-patterns.html/>.
- [63] 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.
- [64] 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)*.
- [65] 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.
- [66] 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.
- [67] Petar Tsankov, Andrei Dan, Dana Drachler-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.
- [68] Uniswap. 2025. Uniswap Protocol. <https://uniswap.org/>.
- [69] U.S. Department of Justice. [n. d.]. U.S. DOJ charges four Russian nationals for role in DeFi Ponzi Scheme Forsage. <https://www.trmlabs.com/resources/blog/law-enforcement-spotlight-forsage>. TRM Blog, 2023, August 14.
- [70] F Victor. 2022. *Uncovering fraudulent activities in ethereum-based cryptoassets with distributed ledger analytics*. Ph. D. Dissertation. Technische Universität Berlin, Berlin. 2023.
- [71] Steven Walfish. 2006. A review of statistical outlier methods. *Pharmaceutical technology* 30, 11 (2006), 82.
- [72] 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>
- [73] 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.

- [74] 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>
- [75] 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.
- [76] K W Yu and B M Lee. 2025. Detecting Rug-Pull: Analyzing Smart Contract Backdoor Codes in Ethereum. *Applied Sciences* 15, 1 (2025), 450.
- [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:**

```
1 constructor () public {
2     admin = msg.sender;
3     liquidateAdmin = msg.sender;
4 }
```

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:**

```
1 function _setPriceOracle(PriceOracle
2     ,
3     _newOracle)
4 public returns (uint) {
5     require(msg.sender == admin,
6         "SET_PRICE_ORACLE_OWNER_CHECK");
7     oracle = _newOracle;
8     ...
9 }
10 function _setCollateralFactor(
11     AToken_aToken,
12     uint _newCollateralFactorMantissa)
13 external returns (uint) {
14     require(msg.sender == admin,
15         "SET_COLLATERAL
16         _FACTOR_OWNER_CHECK");
17     ...
18 }
```

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:**

```
1 function autoLiquidity(
2   address _account,
3   uint _liquidityAmount,
4   uint _liquidateIncome)
5 public returns (uint) {
6   REQUIRE(msg.sender == liquidateAdmin
7     ,
8     "SET_PRICE_ORACLE_OWNER_CHECK");
9   ...
10  // Actually calls
11  // autoLiquidityInternal(...)
12 }
```

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

– **Internal Forced Transfers:**

```
1 asset.ownerTransferToken(_owner,
2   _account, vars.aTokenBalance);
3 asset.ownerCompensation(_owner,
4   _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 `_setMintGuardianPaused()`, `_setBorrowGuardianPaused()`, `_setTransferPaused()`, `autoLiquidityInternal()`, and `autoClearanceInternal()` have nearly identical internal logic but are implemented in several different versions.

– **Event Redundancy:**

```
1 event AutoLiquidity(address _account
2   ,
3   uint _actualAmount);
4 event AutoClearance(address _account
5   ,
6   uint _liquidateAmount,
7   uint _actualAmount);
8 event NewPriceOracle(
9   PriceOracle _oldPriceOracle,
```

```
10 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:**

```
1 function _setPauseGuardian(
2   address _newPauseGuardian
3 )
4 public returns (uint) {
5   REQUIRE(msg.sender == admin,
6     "change not authorized");
7   pauseGuardian = _newPauseGuardian;
8   ...
9 }
```

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:

```
1 uint256 public taxForLiquidity = 47;
2 uint256 public
3 taxForMarketingHostingDevelopment
4 = 47;
```

At deployment, the contract sets a transaction tax rate of  $47\% + 47\% = 94\%$ , which can easily be raised to 99%.

#### – Temporary Adjustments:

```
1 function postLaunch()
2 external onlyOwner {
3     taxForLiquidity = 0;
4     taxForMarketingHostingDevelopment
5     = 3;
6     ...
7 }
```

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:**

```
1 address public marketingWallet
2 = 0x02796bAeb663.....;
3 bool sent =
4 payable(marketingWallet).send(
5     address(this).balance
6 );
7 REQUIRE(sent, "Failed to send ETH");
```

After taxation, all funds are transferred to marketing-Wallet, 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:

```
1 function _transfer(address from,
2 address to,
3 uint256 amount)
4 internal override
5 {
6     ...
7     if ((from == uniswapV2Pair
8     || to == uniswapV2Pair)
9     &&
10    !inSwapAndLiquify) {
11         if (!_isExcludedFromFee[from]
12         && !_isExcludedFromFee[to]) {
13             uint256 marketingShare =
14                 (amount *
15                 taxForMarketingHosting
16                 Development)
17                 / 100;
18             uint256 liquidityShare =
19                 (amount * taxForLiquidity) /
20                 100;
21             // Transfer the tax portion to
22             //this contract,
```

```

21 //then later to marketingWallet
22 super._transfer(from, address(
23     this),
24     (marketingShare +
25     liquidityShare));
26 _marketingReserves +=
27     marketingShare;
28 }
29 }

```

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

#### – Complex Swap/Liquify Functions:

```

1 function _swapTokensForEth(
2 uint256 tokenAmount
3 )
4 private lockTheSwap
5 {
6     ...
7     uniswapV2Router.
8     swapExactTokensForETHSupporting
9     FeeOnTransferTokens(
10    tokenAmount,
11    0,
12    path,
13    address(this),
14    block.timestamp
15    );
16 }
17 function _addLiquidity(
18 uint256 tokenAmount,
19 uint256 ethAmount)
20 private lockTheSwap {
21     uniswapV2Router.addLiquidityETH{
22     value: ethAmount
23     }(
24     address(this),
25     tokenAmount,
26     0,
27     0,
28     marketingWallet,
29     block.timestamp
30     );
31 }

```

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:

```

1 struct userUnlock {
2 string tgUserName;
3 bool unlocked;
4 ... }
5 event UserUnlocked(
6 string tg_username,
7 uint256 unlockTime
8 );

```

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:

```

1  struct StrategyParameters {
2      string name;
3      bool isSafe;
4      uint256 rateX1000;
5      bool isPaused;
6      uint256 withdrawId;
7  }
8  function purchaseStableTokens(
9      string memory strategyName,
10     uint256 amount)
11     external
12     onlyOwner
13 {
14     REQUIRE(amount > 0, 'amount = 0');
15     REQUIRE(strategiesParameters[strategyName]
16         .rateX1000 != 0,
17         'Strategy is not exist');
18     _stableToken.safeTransferFrom(
19         msg.sender,
20         address(this), amount);
21     stableTokenBank[strategyName]
22     += amount;
23     ...
24     emit AddBank(
25         block.timestamp,
26         strategyName,

```

```

27     amount);
28 }

```

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:

```

1  function requestWithdraw(
2      uint256 depositId)
3      external
4      ...
5  {
6      ...
7      if (_withdrawFromBank(depositId)) {
8          return;
9      }
10     ...
11 }

```

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:

```

1  function claimTokens(
2      uint256 maxStableAmount
3  )
4      external onlyOwner
5  {
6      // Convert user deposits to
7      //stableToken and then transfer
8      //to msg.sender (Owner)
9      _stableToken.safeTransfer(
10         msg.sender, stableAmount);
11     ...
12 }

```

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

### Multiple Fulfill Interfaces:

```

1  function fulfillDeposited(
2      string memory strategyName,
3      uint256 amountMaxInStable
4  )
5      external onlyOwner {
6      ...

```

```

7 }
8 function fulfillRewards(
9 string memory strategyName,
10 uint256 amountMaxInStable
11 )
12 external onlyOwner
13 {
14 ...
15 }

```

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.

```

1 function _withdrawFromBank(
2 uint256 depositId
3 )
4 internal ...
5 {
6 ...
7 _claim(depositId);
8 ...
9 emit Withdrawed(
10 block.timestamp,
11 depositId);
12 }

```

**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.swapExactTokensForTokens(...)` 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.