

Towards Semantic-Preserving Obfuscation for Analysis-Resistant EVM Bytecode

DOI: <https://doi.org/10.54654/isj.v2i28.6410>

Nguyen Dinh Dai*, Lai Minh Tuan

Abstract— With the transparency of the Ethereum platform, deployed smart contracts remain permanently public, exposing their virtual machine code to risks such as reverse engineering, control-flow analysis and malicious behavior identification. Although several obfuscation approaches for the EVM have been proposed, existing solutions often suffer from limited resistance against advanced analysis techniques, insufficient structural transformation capability or excessive execution overhead. In this work, we propose a novel obfuscation framework for EVM bytecode that enhances security by combining semantic-aware transformations with control-flow perturbation techniques. The proposed framework significantly increases structural complexity, hinders Control Flow Graph (CFG) recovery and alters discriminative virtual machine code characteristics while preserving the semantic correctness of smart contracts. Experimental results demonstrate that the proposed framework achieves a 100% obfuscation success rate with an average cyclomatic complexity of 90.80. Across all 15 evaluated transformation combinations, the framework introduces an overall mean virtual machine code size increase of 15.99% and a mean gas overhead of 7.04%. Notably, the complete multi-layer pipeline (T1+T2+T3+T4) exhibits overheads of 29.73% for virtual machine code size and 12.58% for gas, which remain acceptable considering the achieved robust resistance against reverse engineering and automated static analysis.

Tóm tắt— Với tính minh bạch của nền tảng Ethereum, các hợp đồng thông minh sau khi triển khai luôn tồn tại công khai khiến mã máy đối mặt với nguy cơ bị phân tích dịch ngược, phân tích luồng điều khiển và nhận diện hành vi độc hại. Mặc dù các phương pháp làm rối cho EVM đã được

ngiên cứu, những giải pháp hiện có thường gặp hạn chế về khả năng chống phân tích, mức độ biến đổi cấu trúc hoặc chi phí thực thi quá lớn. Trong nghiên cứu này, chúng tôi đề xuất một giải pháp làm rối mới cho EVM bytecode nhằm tối ưu hóa khả năng bảo mật thông qua việc kết hợp các kỹ thuật biến đổi ngữ nghĩa và gây nhiễu luồng điều khiển. Giải pháp này gia tăng đáng kể độ phức tạp cấu trúc gây khó khăn quá trình khôi phục đồ thị luồng điều khiển (Control Flow Graph – CFG) và làm thay đổi các đặc trưng nhận dạng của mã máy trong khi vẫn bảo toàn tính toàn vẹn về logic của hợp đồng. Kết quả thực nghiệm cho thấy giải pháp đề xuất đạt tỉ lệ làm rối thành công tuyệt đối 100% với độ phức tạp vòng (cyclomatic complexity) trung bình đạt 90,80. Đối với 15 tổ hợp phép biến đổi được khảo sát, giải pháp đề xuất làm tăng trung bình 15,99% kích thước EVM bytecode và 7,04% mức tiêu thụ gas. Trong đó, tổ hợp đầy đủ các biến đổi (T1+T2+T3+T4) tạo ra mức tăng 29,73% về kích thước EVM bytecode và 12,58% về chi phí gas. Mặc dù vậy, các chi phí phát sinh này vẫn nằm trong ngưỡng chấp nhận được, xét đến khả năng cung cấp mức độ bảo vệ mạnh mẽ trước các kỹ thuật dịch ngược và các công cụ phân tích tĩnh tự động.

Keywords— Ethereum, smart contract, EVM Bytecode obfuscation.

Từ khóa— Ethereum, hợp đồng thông minh, làm rối EVM Bytecode.

I. INTRODUCTION

The rapid growth of decentralized ecosystems, particularly those built on the Ethereum platform, has made smart contracts a fundamental component of numerous critical applications including decentralized finance (DeFi), digital asset management and electronic identity systems [1 - 2]. Unlike traditional software, deployed smart contract code is permanently exposed on the blockchain and can be accessed by any participant. As a result, smart contracts containing valuable

This manuscript was received on June 11, 2026. It was reviewed on July 28, 2026, revised on August 8, 2026 and accepted on August 20, 2026.

* Corresponding author

business logic or managing high-value assets have become primary targets for reverse engineering techniques [3]. Although Solidity source code may remain unpublished, attackers can still leverage decompilation and static analysis tools to recover control flow graphs (CFGs), function selectors, jump destinations and high-level pseudo-code from Ethereum Virtual Machine (EVM) bytecode [4]. By combining symbolic execution and pattern-matching techniques, modern analysis frameworks are capable of reconstructing execution semantics with high accuracy, thereby facilitating vulnerability discovery, exploit generation and unauthorized replication of proprietary contract logic [5]. Consequently, protecting EVM bytecode against reverse engineering and static analysis has become an increasingly important challenge in blockchain security.

However, developing effective obfuscation techniques for EVM bytecode remains a challenging task. First, the EVM operates as a stack-based execution environment where even minor modifications to stack states or jump destinations may alter execution semantics or trigger runtime exceptions. Second, smart contracts are strictly constrained by gas consumption, making many conventional obfuscation techniques impractical due to excessive execution overhead [6]. Finally, lightweight protection methods such as instruction substitution or naive junk code insertion can often be eliminated by modern CFG recovery and symbolic execution frameworks.

To address these challenges, this paper proposes a specialized obfuscation framework for EVM bytecode that combines semantic-aware transformations with control-flow perturbation techniques. The proposed framework employs contract-specific entropy keys to generate diversified obfuscation variants and integrates context-sensitive semantic transformation, stack-aware junk insertion, CFG confusion and jump destination encoding to significantly degrade the effectiveness of static analysis and CFG recovery while preserving semantic equivalence. Experimental results on 7,000 smart contracts demonstrate that the proposed

framework achieves strong obfuscation capability with low execution overhead. Overall, the obfuscation pipeline introduces an average gas overhead of only 12.58% while substantially disrupting CFG reconstruction performed by existing analysis tools. In particular, the semantic transformation technique incurs only 0.90% additional gas cost whereas the CFG confusion technique increases gas consumption and code size by merely 1.33% and 2.81% while providing significant resistance against CFG recovery. Furthermore, no contract failures or semantic inconsistencies were observed throughout the evaluation process, demonstrating the stability and practical deployability of the proposed approach.

The main contributions of this paper are summarized as follows:

1. We propose a multi-layer obfuscation framework for EVM bytecode that combines semantic-aware transformation and control-flow perturbation to enhance resistance against reverse engineering while maintaining low gas overhead.
2. We develop a large-scale experimental evaluation on 7,000 smart contracts to comprehensively assess the proposed framework in terms of gas overhead, semantic equivalence and resistance to CFG recovery, thereby demonstrating the effectiveness and practicality of the proposed solution.

The remainder of this paper is organized as follows. Section II presents background knowledge on Ethereum smart contracts, reverse engineering and code obfuscation. Section III reviews related work. Section IV describes the proposed framework and obfuscation techniques. Section V presents the experimental setup and evaluation results. Finally, Section VI concludes the paper and discusses future research directions.

II. BACKGROUND

2.1. Ethereum Virtual Machine and Smart Contract

Smart contracts on Ethereum are executed by the EVM which provides a quasi-Turing-complete execution environment for blockchain

systems [6]. Unlike traditional computing systems based on the Von Neumann architecture, the EVM does not allow instructions to be stored and manipulated as ordinary data in memory. The runtime code of a smart contract is stored in a read-only code region (i.e., a virtual ROM) and cannot be modified during execution [7]. This characteristic introduces fundamental differences for anti-reverse-engineering techniques. In conventional systems, protection mechanisms such as self-modifying code, runtime unpacking or encrypted code execution can be employed to conceal execution logic [8]. However, such approaches are generally infeasible in the EVM because deployed code cannot dynamically decrypt or modify itself at runtime.

In addition, the EVM adopts a stack-based architecture rather than a register-based architecture such as x86 or ARM. Control-flow instructions including JUMP and JUMPI, obtain target addresses directly from the execution stack, making CFG recovery heavily dependent on stack-state analysis and symbolic execution [9]. Furthermore, Ethereum employs a gas mechanism to constrain computational resources, requiring any obfuscation technique to carefully control both gas overhead and code size [6]. Consequently, characteristics such as the non-Von Neumann execution model, immutable code, dynamic jump behavior and strict gas constraints make many traditional anti-reverse-engineering techniques difficult to apply directly to the EVM, thereby motivating the need for specialized obfuscation mechanisms for smart contracts.

2.2. Reverse Engineering

Reverse engineering is the process of analyzing code to recover the structure and execution logic of a program [10]. In the Ethereum ecosystem, this process typically involves disassembly, CFG recovery, symbolic execution and decompilation. Modern analysis tools are capable of identifying basic blocks, function selectors and jump destinations, as well as reconstructing high-level pseudo-code directly from EVM bytecode.

Among these techniques, CFG recovery plays a critical role in inferring the execution flow and

behavioral semantics of smart contracts [11]. Although the EVM adopts a stack-based architecture and supports dynamic jump operations, existing reverse engineering frameworks can still reconstruct program logic with high accuracy by leveraging symbolic execution and pattern-matching techniques. As a result, smart contracts face significant risks of business logic exposure, vulnerability discovery, exploit development and unauthorized contract replication.

2.3. EVM Bytecode Obfuscation

EVM bytecode obfuscation refers to the process of transforming program code to increase the difficulty of reverse engineering while preserving the original program semantics. The primary objective of obfuscation is not to completely prevent reverse engineering, but rather to increase the analysis cost and complexity beyond the potential benefits obtainable by an attacker [12]. From a theoretical perspective, Barak et al. demonstrated that general-purpose obfuscation capable of fully protecting arbitrary programs is impossible in systems without hardware-assisted protection mechanisms [13]. Nevertheless, in practice, obfuscation remains an effective defensive approach if it can substantially reduce the effectiveness of modern reverse engineering tools.

In the context of the EVM, virtual machine code obfuscation must simultaneously satisfy three essential requirements: (1) preserving semantic equivalence; (2) effectively disrupting CFG recovery, symbolic execution, decompilation; (3) minimizing overhead in terms of gas consumption and code size [12]. Existing obfuscation techniques generally fall into three major categories: control-flow obfuscation, which aims to complicate CFG structures; instruction-level obfuscation, which modifies opcodes or inserts junk instructions; semantic obfuscation, which conceals execution logic through semantically equivalent transformations.

However, due to the stack-based architecture, dynamic jump behavior and strict gas constraints of the EVM, many traditional

obfuscation techniques are difficult to apply directly or introduce excessive execution overhead. Therefore, designing specialized obfuscation mechanisms for smart contracts that effectively balance reverse-engineering resistance and execution efficiency remains an important research challenge.

III. RELATED WORK

In recent years, numerous studies have investigated obfuscation techniques for EVM bytecode to improve resistance against reverse engineering and decompilation of Ethereum smart contracts. Existing approaches primarily exploit the stack-based architecture and dynamic jump behavior of the EVM to disrupt CFG recovery, symbolic execution and static analysis through instruction-level and control-flow-level transformations.

EVeilM is one of the representative obfuscation frameworks operating directly on EVM runtime bytecode. The framework employs multiple techniques including fake JUMPDEST insertion, fake branches and pseudo-path generation, dynamic jump target transformation and function dispatcher obfuscation to distort CFG reconstruction and decompilation [14]. In contrast, HermHD extends this direction by incorporating control-flow flattening, memory indirection and jump address obfuscation to further resist symbolic execution and static analysis [13]. BOSC adopts a structural and semantic perturbation approach, utilizing malformed instructions, false branches, instruction reordering and junk instructions to interfere with virtual machine code level disassembly and decompilation [15].

Experimental results reported in prior studies [12] indicate that these approaches can substantially increase CFG complexity and reduce the effectiveness of modern analysis tools. EVeilM achieves a Graph Edit Distance (GED) of approximately 0.93–0.99 while maintaining a valid code ratio of around 84–100%, whereas BOSC reports a GED of approximately 0.87–0.95 and significantly increases the failure rate of existing decompilers. Nevertheless, EVeilM remains highly dependent on Solidity compiler virtual machine code

patterns and only approximately 15–28% of evaluated samples preserve semantic equivalence after obfuscation. BOSC exhibits limited stability with only around 18–39% of obfuscated virtual machine code samples remaining valid while several configurations increase deployment gas consumption by more than 7,000 - 12,000%. Meanwhile, HermHD frequently introduces semantic inconsistencies and runtime errors due to its complex control-flow and arithmetic transformations.

Overall, existing studies primarily focus on syntactic or CFG-level perturbation techniques to disrupt disassembly and decompilation, whereas resistance against semantic-aware analysis and symbolic execution remains limited. Furthermore, the trade-off between obfuscation effectiveness, semantic correctness, execution stability and gas overhead has not yet been comprehensively addressed. This limitation motivates the research presented in this paper, which proposes a semantic-aware and context-sensitive obfuscation framework designed to enhance reverse-engineering resistance while preserving semantic equivalence and maintaining practical deployability on Ethereum.

IV. PROPOSED APPROACH

The framework consists of three main modules. First, the EVM Bytecode Preprocessing Module separates the creation code from the runtime code, ensuring that obfuscation is applied only to the executable logic while preserving deployment integrity. Next, the EVM Bytecode Obfuscation Module performs the core transformations using four complementary techniques: Context-Sensitive Equivalence Transformations (T1), Stack-Aware Junk Insertion (T2), Control Flow Graph Confusion (T3) and Pattern-Based Transformation (T4). These techniques collectively perturb semantics, stack behavior and control-flow structures to enhance resistance against reverse engineering and automated analysis. Finally, the EVM Bytecode Reconstruction Module reintegrates the obfuscated runtime code into the original creation code by updating parameters such as

CODECOPY offsets and lengths, thereby generating a consistent and deployable obfuscated smart contract.

4.1. EVM Bytecode Preprocessing Module

The primary objective of this module is to accurately isolate the runtime code while removing non-essential components, namely the deployment and auxiliary (auxdata) segments of a smart contract. This step is critical because it extracts the core executable logic of the contract, thereby improving both the precision and efficiency of subsequent obfuscation processes.

A typical bytecode file (e.g., with a .hex suffix) can be logically divided into three distinct parts: deployment code, runtime code and auxdata. Since most decompilers and analysis tools operate exclusively on the runtime portion, we focus its obfuscation only on this segment while treating auxdata as appended metadata primarily used for verification purposes. To precisely identify these segments, we first disassemble the contract creation code into EVM opcodes while preserving the byte offset of each instruction. We then perform a backward scan to locate the final 0xF3 (RETURN) instruction. Immediately preceding this instruction, Solidity-generated bytecode typically follows a characteristic copy–return pattern where 0x39 (CODECOPY) is used to copy the runtime code into memory. This operation is accompanied by two constants loaded via PUSH instructions, representing the starting offset and length of the runtime segment.

By extracting these two constants, we can determine the exact boundaries of the runtime code in a deterministic manner. Specifically, bytes before the identified start offset correspond to deployment code, bytes within the extracted range constitute the runtime code and the remaining suffix is classified as auxdata.

4.2. EVM Bytecode Obfuscation Module

After the preprocessing stage, the runtime code is passed to the EVM Bytecode Obfuscation Module to perform a series of transformations aimed at enhancing resistance against analysis in the EVM environment. This module serves as the core component of the proposed framework,

responsible for applying both semantic-level and structural-level obfuscation techniques to alter code characteristics while preserving the original execution semantics of the smart contract.

Before applying the obfuscation techniques, the framework first generates a contract-specific entropy key to ensure that each smart contract is transformed within a distinct transformation space. This design increases polymorphism and reduces the effectiveness of pattern-matching and cross-contract correlation analysis. Specifically, the framework extracts runtime code and applies the SHA3-256 hash function to generate a global entropy key K :

$$K = \text{SHA3-256}(\text{runtime_code}) \quad (1)$$

where K is a unique 32-byte entropy key associated with the target contract. Even a minor modification in the code results in a completely different entropy key, thereby increasing transformation diversity and reducing structural similarity among obfuscated contracts. After generating the global key K , the framework derives multiple sub-keys dedicated to different obfuscation components, including Context-Sensitive Equivalence Transformations, Stack-Aware Junk Insertion, Control Flow Graph Confusion and Pattern-Based Transformation.

4.2.1. Context-Sensitive Equivalence Transformations (T1)

Context-Sensitive Equivalence Transformations is a semantic-aware virtual machine code obfuscation technique designed to replace easily recognizable opcode patterns with more complex instruction sequences while preserving the original execution semantics. Instead of applying transformations mechanically, the technique analyzes the functional role of each code region to ensure that the obfuscation process does not compromise contract correctness or introduce excessive gas overhead.

Specifically, the runtime code is first partitioned into Basic Blocks (BBs), which are subsequently classified according to their functional roles including function dispatching (SELECTOR_CHECK), storage access (STORAGE_READ), arithmetic and logical computation (ARITHMETIC_BODY),

sensitive regions (SKIP_ROLES). Sensitive blocks are intentionally excluded from transformation to preserve execution safety and semantic correctness.

The transformation process consists of two stages: (i) pre-identifying candidate transformation locations without modifying the original code; (ii) applying opcode substitutions in descending address order to prevent offset inconsistencies caused by instruction insertion. Based on this strategy, the framework performs several major semantic-equivalent transformations as follows:

4.2.1.1. Function Dispatcher Obfuscation

In Solidity-generated bytecode, the function dispatcher typically follows a highly recognizable pattern:

DUP1 | PUSH4(sig) | EQ | PUSH(dest) | JUMPI

This structure directly exposes the function selector and can be easily identified by static analysis and pattern-matching techniques. To mitigate this weakness, the proposed framework replaces the original dispatcher sequence with semantically equivalent but structurally more complex expressions that are significantly harder to infer and analyze.

TABLE I. COMPARISON BETWEEN ORIGINAL AND OBFUSCATED FUNCTION DISPATCHER OBFUSCATION

Original	Obfuscated
$PUSH4(sig) EQ$	$PUSH4(K_i) XOR PUSH4(sig \oplus K_i) EQ$ <i>K_i: Sub-key of K</i>

4.2.1.2. Storage Access Obfuscation

In Solidity-generated bytecode, storage access operations commonly follow the direct pattern $PUSH_n(slot) | SLOAD$, which exposes storage slot addresses and makes them easily recoverable through static analysis. To conceal this information, the proposed framework replaces the original sequence with semantically equivalent expressions based on intermediate encoding transformations, thereby obscuring the actual storage slot values while preserving execution semantics.

TABLE II. COMPARISON BETWEEN ORIGINAL AND OBFUSCATED STORAGE ACCESS OBFUSCATION

Original	Obfuscated
$PUSH4(sig) EQ$	$PUSH(slot \oplus mask) PUSH(mask) XOR SLOAD$ mask: Sub-key of K (1)
	$PUSH(slot + \delta) PUSH(\delta) SUB SLOAD$

4.2.1.3. Logical Operator Obfuscation

Logical operators such as EQ and ISZERO frequently appear in EVM bytecode, forming highly recognizable patterns for static analysis and symbolic execution frameworks. To reduce the effectiveness of such analysis, the proposed framework replaces these operators with semantically equivalent but structurally diversified expressions, thereby increasing pattern complexity while preserving the original program behavior.

TABLE III. COMPARISON BETWEEN ORIGINAL AND OBFUSCATED LOGICAL OPERATOR OBFUSCATION

Original	Obfuscated
EQ	XOR ISZERO
EQ	SUB ISZERO
ISZERO	$PUSH1(0) EQ$
ISZERO	$PUSH1(1) LT$

The correctness of T1 is based on algebraic and logical equivalence under the EVM's 256-bit execution semantics. Each replacement sequence produces the same stack output as the original opcode while preserving operand order and subsequent execution behavior. Moreover, storage-slot transformations employ reversible masking operations, ensuring that the original storage address is reconstructed before executing SLOAD. Consequently, although opcode patterns are modified, the observable behavior and persistent contract state remain unchanged.

4.2.2. Stack-Aware Junk Insertion (T2)

Stack-Aware Junk Insertion is designed to increase the structural complexity and size of EVM bytecode through the insertion of junk opcode sequences. Unlike conventional approaches that rely on fixed insertion patterns, this technique adopts a stack-aware strategy in which junk insertion depends on the stack depth at each candidate location. As a result, all transformations preserve execution semantics while completely avoiding stack underflow and overflow conditions. To identify valid insertion locations, the framework simulates stack states within each basic block. Two primary categories of insertion points are considered: (i) entry points, located immediately after JUMPDEST instructions where insertion is inherently safe; (ii) mid-block points, corresponding to positions inside sufficiently large basic blocks. The simulation process tracks the stack effect of each opcode, where instructions such as PUSH and DUP increase stack depth, operations such as POP, ADD and SUB decrease it and SWAP instructions preserve the current stack depth. Based on this analysis, the framework selects only insertion locations that satisfy all stack consistency constraints.

To preserve Program Counter (PC) consistency, the framework employs a two-pass architecture. In the first pass, all candidate insertion points and their associated stack states are collected from the entire set of basic blocks, excluding sensitive regions without modifying the original EVM bytecode. In the second pass, junk instruction sequences are inserted in descending address order. This reverse-order insertion strategy prevents code size changes from invalidating previously collected offsets and addresses. Finally, a single global update step is performed to adjust all affected PC values and maintain program integrity. The inserted junk instructions are organized into a stack-depth-aware library parameterized by a minimum required stack depth D_{min} . Each junk sequence is carefully constructed such that its net stack effect remains zero, thereby ensuring that the execution state before and after insertion remains semantically equivalent. The following

table summarizes the primary categories of junk instruction sequences employed in the T2 transformation module.

TABLE IV. STACK-AWARE JUNK OPCODE INSERTION RULES

Condition D	Opcode
$D \geq 2$	DUP1 DUP2 XOR POP
	DUP1 DUP2 AND POP
	DUP2 DUP2 SUB POP
$D \geq 1$	DUP1 PUSH1 k XOR POP
	DUP1 PUSH1 k AND POP
$D \geq 0$	PUSH1 k POP

The constant k (in Table IV) is generated using a pseudo-random number generator seeded by a sub-key derived from K , ensuring that junk instruction sequences differ across contracts and are therefore difficult to identify using fingerprinting or cross-contract pattern analysis techniques. Furthermore, the proposed technique does not inject junk instructions into every possible insertion point. Instead, the insertion process is regulated by an obfuscation intensity parameter I which is dynamically derived from a sub-key of K . The value of I ranges from 0.4 to 1.0 and determines the probability that a candidate location will receive a junk insertion. Through this mechanism, the framework can flexibly control obfuscation density: contracts requiring stronger protection are assigned a higher density of junk instructions while still maintaining manageable gas overhead.

All inserted junk sequences are designed to have a zero net stack effect and contain no state-modifying instructions such as SSTORE, CALL, or CREATE. Candidate insertion points are accepted only after stack simulation confirms that sufficient stack depth is available, thereby preventing stack underflow or overflow. As a result, the stack state immediately after executing a junk sequence is identical to that of the original program, and only the bytecode structure and execution cost are affected.

4.2.3. Control Flow Graph Confusion (T3)

Control Flow Graph Confusion is designed to distort the CFG of smart contracts, thereby reducing the effectiveness of static analysis and decompilation frameworks. In the EVM environment, CFG construction is typically performed by identifying graph nodes through JUMPDEST instructions and control-flow edges through JUMP and JUMPI operations. Consequently, if these components are systematically manipulated, the resulting CFG reconstruction may become inaccurate or unreliable.

To achieve this objective, the proposed technique employs two complementary transformation phases: (i) JUMPDEST Injection (Fake Node Expansion); (ii) Opaque Predicate-Based Fake Branching; (iii) Full-Width Control-Flow Masking. The combination of these mechanisms produces a path explosion effect, significantly enlarging the analysis space explored by static analysis and symbolic execution tools.

4.2.3.1. JUMPDEST Injection (Fake Node Expansion)

In the first phase, the framework increases CFG complexity by inserting additional fake JUMPDEST instructions. However, arbitrary insertion of such opcodes may corrupt the code structure, particularly when inserted inside the operand region of PUSH instructions. To preserve EVM bytecode correctness, the framework adopts a block-boundary safe insertion strategy where fake JUMPDEST instructions are inserted only at the boundaries between basic blocks. This approach guarantees that opcode structures remain intact and that the resulting EVM bytecode preserves syntactic validity.

4.2.3.2. Opaque Predicate-Based Fake Branching

After generating isolated fake nodes, the framework further connects them through fake control-flow edges constructed using opaque predicates. These predicates are designed such that they always evaluate to False at runtime while remaining difficult for static analysis frameworks to infer precisely. Instead of employing simple conditional expressions, the proposed approach

utilizes more complex arithmetic predicates derived from entropy-based constants generated from the global entropy key K . The use of arithmetic operations such as MUL and ADD with dynamically varying constants substantially increases the complexity of constraint solving during symbolic execution.

TABLE V. ARITHMETIC-BASED CONSTRUCTION OF OPAQUE PREDICATES

Logical expression	Formula of A
$a*b+c \neq A$	$A=(a*b+c+1) \bmod 2^{16}$
$(a+b)*c \neq A$	$A=((a+b)*c+1) \bmod 2^{16}$
$a*b*c \neq A$	$A=(a*b*c+1) \bmod 2^{16}$

For each pair of fake jump points, the framework inserts a conditional verification sequence before executing the JUMPI instruction. Although the predicate always evaluates to False and the corresponding branch is never executed at runtime, a control-flow edge is still constructed during static analysis, thereby distorting the reconstructed CFG structure.

4.2.3.3. Full-Width Control-Flow Masking

In addition to generating fake control-flow edges, the proposed framework further obfuscates the control-flow dispatch mechanism itself. Conventional static analysis tools typically recover CFGs by identifying simple jump patterns of the following form: $PUSH\ dest_pc \mid JUMPI$. Using this pattern, the destination address of a branch can be directly inferred from the EVM bytecode. To invalidate this analysis strategy, the framework replaces static jump destinations with runtime-computed expressions:

$$Runtime_Dest = k \oplus (dest_pc \oplus k)$$

where k is a sub-key derived from the global entropy key K ensuring that every byte of the target address is masked during obfuscation. As a result, jump destinations can no longer be directly inferred from static EVM bytecode analysis, forcing reverse engineering frameworks to rely on more expensive execution simulation or symbolic execution techniques to recover control-flow information.

TABLE VI. COMPARISON BETWEEN ORIGINAL AND OBFUSCATED CONTROL FLOW GRAPH CONFUSION

Original	Obfuscated
	JUMPDEST ; <i>fake_A</i>
	PUSH1 0x03
	PUSH1 0x05
	MUL
	PUSH1 0x02
	ADD ; <i>= 17</i>
PUSH1 0x01	PUSH2 0x0012 ; <i>wrong = 18</i>
PUSH1 0x02	EQ ; <i>FALSE</i>
LT	PUSH2 0x0020 ; <i>fake_B</i>
PUSH2 0x0010	JUMPI ; <i>never taken</i>
JUMPI	PUSH1 0x01
PUSH1 0x00	PUSH1 0x02
JUMPDEST	LT
	PUSH2 0x00A7 ; <i>masked value</i>
	PUSH2 "K"
	XOR
	JUMPI ; <i>real branch</i>
	JUMPDEST ; <i>fake_B</i>

Although T3 introduces additional CFG nodes and edges, the injected branches are guarded by opaque predicates that always evaluate to false during concrete execution. Therefore, fake execution paths are never traversed at runtime. Furthermore, masked jump destinations are reconstructed using reversible XOR operations before each actual jump, and all affected jump offsets are updated after code insertion. Consequently, only the statically reconstructed CFG is modified, whereas the executable control flow remains equivalent to that of the original contract.

4.2.4. Pattern-Based Transformation (T4)

Pattern-Based Transformation focuses on semantically obfuscating arithmetic operations in EVM bytecode, particularly the ADD operator. Since ADD appears with very high frequency in most smart contracts, transforming this operator can substantially reduce the effectiveness of pattern-based reverse engineering techniques. However, indiscriminately replacing all ADD instructions would introduce significant gas overhead. Therefore, the proposed technique follows a gas-aware design principle, applying transformations only at carefully selected locations to balance obfuscation effectiveness and execution efficiency.

Before applying the transformations, the framework reanalyzes the EVM bytecode after the preceding obfuscation phases to identify gas-critical regions also referred to as hot paths. These regions typically include basic blocks containing high-cost or frequently executed operations such as SSTORE, CALL, DELEGATECALL or access-control verification logic. Such blocks are intentionally excluded from transformation to avoid excessive execution overhead or unintended runtime side effects. This reclassification step is necessary because the opcode structure may already have been modified by earlier obfuscation stages.

After excluding hot paths, the remaining ADD instructions are replaced with semantically equivalent arithmetic expressions. These transformations are constructed based on arithmetic identities within the modulo 2^{256} execution domain of the EVM, thereby guaranteeing semantic correctness under all execution conditions.

TABLE VII. EQUIVALENT ARITHMETIC TRANSFORMATIONS FOR OPCODE SUBSTITUTION

Formula	Opcode
$a+b = \neg(\neg a - b)$	NOT SWAP1 SUB NOT
$a+b = a - (\neg b + 1)$	SWAP1 NOT PUSH1 1 ADD SUB

TABLE VIII. EXAMPLES OF PATTERN-BASED TRANSFORMATION

Original	Obfuscated 1	Obfuscated 2
PUSH1 0x05	PUSH1 0x05	PUSH1 0x05
PUSH1 0x03	PUSH1 0x03	PUSH1 0x03
ADD	NOT SWAP1 SUB NOT	SWAP1 NOT PUSH1 0x01 ADD SUB

The replacement patterns used in T4 are derived from arithmetic identities that remain valid under the EVM's 256-bit modular arithmetic. Each transformed instruction sequence preserves the original operand order and produces the same arithmetic result at the top of the execution stack. Since no additional storage operations, external calls, or memory modifications are introduced, the transformation

preserves the original execution semantics while modifying recognizable opcode patterns.

4.3. EVM Bytecode Reconstruction Module

The EVM Bytecode Reconstruction Module is responsible for producing a complete and deployable bytecode by consistently reintegrating the obfuscated runtime into the original structure of the contract. This phase ensures that all dependencies introduced by obfuscation are properly resolved at both the metadata and deployment levels. First, the auxdata is updated to maintain consistency with the modified runtime code. Since auxdata encodes the Swarm hash, which is deterministically derived from the contract metadata and runtime, any transformation applied during obfuscation invalidates the original value. To address this, the contract metadata is reconstructed using the original compilation parameters and a new Swarm hash is computed and encoded into the updated auxdata segment. Next, the creation (deployment) code is recalibrated. As this code governs how the runtime code is copied and returned during contract deployment, any change in the runtime's size or layout leads to inconsistencies in embedded parameters such as offset and length. Therefore, this module identifies LENGTH_OFFSET patterns and constructor-related structures and updates the corresponding arguments of the CODECOPY instruction to ensure that the entire obfuscated runtime code is correctly copied into memory during initialization. Finally, the updated auxdata is appended to the obfuscated runtime code and this modified runtime segment is embedded back into the adjusted creation code. The result is a semantically correct, fully deployable code that remains compatible with standard verification mechanisms.

V. EXPERIMENTAL EVALUATION

In this section, we conduct a comprehensive evaluation of the proposed obfuscation framework to analyze both the effectiveness of individual transformation techniques and their overall impact on EVM bytecode. Specifically, the experiments are performed on a dataset of 7,000 smart contracts using multiple combinations of the proposed obfuscation techniques including Context-Sensitive Equivalence Transformations (T1), Stack-Aware

Junk Insertion (T2), Control Flow Graph Confusion (T3) and Pattern-Based Transformation (T4). Through these experiments, we evaluate the obfuscation success rate, semantic preservation capability and the overhead introduced in terms of EVM bytecode size and gas consumption after obfuscation. In addition, we analyze structural changes in the program by examining the resulting CFG, thereby assessing the ability of the proposed framework to hinder disassembly, decompilation and control-flow recovery processes. Furthermore, to evaluate the capability of the framework to conceal malicious behaviors, we construct machine learning models for malicious smart contract detection using EVM bytecode samples collected before and after obfuscation. This evaluation enables us to analyze the degradation in detection performance caused by each obfuscation technique. Finally, the proposed framework is directly compared with two state-of-the-art EVM obfuscation frameworks, BOSC and EVeilM, in order to demonstrate its effectiveness, resistance against reverse engineering and practical applicability.

5.1 Experimental Setup

All experiments were conducted on an Ubuntu 20.04 environment equipped with an Intel Core i7-9700 CPU and 16 GB of RAM. To evaluate the effectiveness of the proposed obfuscation framework, we utilized the PhishingHook dataset [16] consisting of 7,000 EVM bytecode contracts including 3,458 malicious contracts and 3,542 benign contracts. During the bytecode analysis process, Heimdall [17] was employed for disassembly and CFG extraction to evaluate structural modifications introduced by the obfuscation process.

The effectiveness of the proposed framework was assessed using multiple evaluation metrics including obfuscation success rate, EVM bytecode size overhead, gas consumption overhead, obfuscate time, scalability, CFG extraction success rate, cyclomatic complexity, Shannon entropy, unique opcode ratio, Graph Edit Distance (GED) and evasion rate, which collectively measure the ability of the framework to conceal malicious behaviors from automated analysis systems.

In addition, to evaluate malicious contract detection performance before and after obfuscation, we constructed several machine learning models using the Scikit-learn framework including Random Forest, Gradient Boosting, LinearSVC combined with CalibratedClassifierCV and Logistic Regression. Model performance was evaluated using standard metrics including Accuracy, Precision, Recall, F1-score and Area Under the Curve (AUC).

To comprehensively evaluate the contribution of individual obfuscation techniques and their interactions, each evaluated framework was executed under all possible non-empty combinations of its four constituent techniques. Consequently, fifteen configurations were generated for each framework including four individual techniques, six pairwise combinations, four three-technique combinations, and one complete four-technique pipeline. The results of these configurations are analyzed in Sections 5.2–5.5, while Section 5.6 reports the arithmetic mean across all fifteen configurations for framework-level comparison.

5.2. Impact of Obfuscation Techniques on EVM Bytecode and Gas Consumption

To evaluate the impact of each obfuscation technique, we conducted experiments on the entire dataset of 7,000 smart contracts using multiple transformation configurations including individual techniques (T1, T2, T3 and T4), combinations of two techniques, combinations of three techniques and the full transformation pipeline. The evaluation primarily focuses on two aspects: the increase in EVM bytecode size and the gas overhead introduced after obfuscation.

TABLE IX. IMPACT OF OBFUSCATION TECHNIQUES ON EVM BYTECODE SIZE AND GAS CONSUMPTION

Combo	Obfuscated (%)	avg_delta_bytes	avg_pct_bytes (%)	avg_delta_gas	avg_pct_gas (%)
T1	100	146.07	2.55	253.85	0.90
T2	100	602.55	10.51	1154.19	4.08
T3	100	161.24	2.81	376.53	1.33
T4	100	810.08	14.14	2469.07	8.72
T1,T2	100	765.92	13.36	1427.03	5.04
T1,T3	100	318.42	5.56	625.16	2.21
T1,T4	100	911.52	15.91	2035.30	7.19
T2,T3	100	774.75	13.52	1525.44	5.39
T2,T4	100	1423.41	24.84	3195.51	11.29
T3,T4	100	982.42	17.14	2383.08	8.42

T1,T2,T3	100	937.89	16.37	1798.76	6.35
T1,T2,T4	100	1531.04	26.72	3197.28	11.29
T1,T3,T4	100	1084.35	18.92	2392.96	8.45
T2,T3,T4	100	1595.56	27.84	3490.32	12.33
T1,T2,T3,T4	100	1703.65	29.73	3560.22	12.58

The experimental results show that T1 and T3 introduce the lowest overhead among all individual techniques. Specifically, the average EVM bytecode size increases by only 2.55% and 2.81%, while the corresponding gas overhead reaches merely 0.90% and 1.33%, respectively. These results indicate that both techniques primarily affect execution semantics and control-flow structures without significantly increasing the number of inserted opcodes.

In contrast, T2 and particularly T4 introduce higher overhead due to the insertion of additional junk instructions and structural transformations. Among the individual techniques, T4 produces the largest impact, increasing EVM bytecode size by 14.14% and gas consumption by 8.72%.

For combined configurations, the overhead gradually increases with the number of integrated techniques. Combinations involving T4 consistently produce substantially higher overhead compared with other configurations. Notably, the full transformation pipeline T1+T2+T3+T4 results in the largest overall modification, with average increases of 29.73% in EVM bytecode size and 12.58% in gas consumption. Nevertheless, this overhead remains acceptable considering the significant improvement in structural diversification and resistance against reverse engineering achieved by the proposed obfuscation framework.

5.3 Obfuscation Time and Scalability Analysis

In addition to bytecode-size and gas overhead, we evaluate the computational cost of the obfuscation framework itself. Obfuscation time is measured as the processing time required to transform the original runtime bytecode into the corresponding obfuscated bytecode. The evaluation considers all 15 transformation configurations over the complete dataset of 7,000 contracts. We report the average, median, and maximum processing times, together with throughput in contracts per second. To further

evaluate scalability, contracts are grouped into six ranges according to their original runtime bytecode size.

TABLE X. COMPUTATIONAL COST AND THROUGHPUT OF THE PROPOSED OBFUSCATION FRAMEWORK

Combo	Avg. (ms)	Median (ms)	Max (ms)	Throughput (contract/s)
T1	11.84	7.62	95.08	84.4
T2	11.03	7.04	109.08	90.7
T3	15.01	9.83	116.83	66.6
T4	13.38	8.58	124.99	74.7
T1,T2	18.43	11.81	143.69	54.3
T1,T3	22.57	14.71	155.35	44.3
T1,T4	20.70	13.32	153.08	48.3
T2,T3	22.85	14.74	147.07	43.8
T2,T4	20.76	13.33	153.80	48.2
T3,T4	24.03	15.72	154.76	41.6
T1,T2,T3	30.59	19.89	186.94	32.7
T1,T2,T4	28.28	18.23	188.24	35.4
T1,T3,T4	31.67	20.68	183.32	31.6
T2,T3,T4	32.81	21.34	203.97	30.5
T1,T2,T3,T4	40.78	26.37	238.44	24.5

As shown in Table X, the computational cost increases with the number and complexity of the enabled transformations. Individual techniques require approximately 11–15 ms per contract on average, with throughput ranging from 66.6 to 90.7 contracts/s. Combining multiple techniques increases the processing cost, with the complete T1+T2+T3+T4 pipeline requiring 40.78 ms per contract on average and achieving a throughput of 24.5 contracts/s. This increase is expected because the complete pipeline sequentially performs multiple bytecode analysis and transformation stages.

TABLE XI. SCALABILITY OF OBFUSCATION TIME WITH CONTRACT SIZE

Size	N (Contracts)	T1 (ms)	T2 (ms)	T3 (ms)	T4 (ms)
<500B	822	0.55	0.52	0.95	0.57
500B – 1KB	194	1.88	1.72	2.68	2.08
1- 2.5KB	1707	4.59	4.26	6.04	5.15
2.5- 5KB	1394	7.71	7.12	9.87	8.61
5-10KB	1545	14.91	13.73	18.73	16.67
10- 25KB	1338	30.24	28.44	37.96	34.56

Table XI shows a clear increase in processing time as the input bytecode size grows. For the complete pipeline, the average obfuscation time increases from 1.98 ms for

contracts smaller than 500 bytes to 104.64 ms for contracts in the 10–25 KB range. A similar trend is observed for all individual techniques, indicating that the computational cost is primarily influenced by both contract size and transformation complexity. Despite this increase, the complete pipeline processes the largest evaluated contracts in approximately 105 ms on average, demonstrating practical scalability for offline smart-contract obfuscation.

Together with the bytecode-size and gas-overhead results, these measurements provide a more complete assessment of the practical cost of the proposed framework by considering both the offline cost of applying obfuscation and the runtime cost of executing the resulting contracts.

5.4 Structural Complexity and CFG Recovery Analysis

To evaluate the degree of structural transformation introduced by the obfuscation process, we analyze the effectiveness of CFG recovery on obfuscated EVM bytecode under multiple transformation configurations. The evaluation focuses on four primary metrics: CFG extraction success rate, opcode entropy, cyclomatic complexity and unique opcode ratio. These metrics enable quantitative assessment of control-flow diversification, structural complexity and the ability of the proposed framework to disrupt disassembly and decompilation tools.

TABLE XII. CFG RECOVERY AND COMPLEXITY ANALYSIS OF OBFUSCATION TECHNIQUES

Combo	CFG (%)	Entropy _mean	Cyclomatic _mean	Unique _ratio _mean	GED _mean
T1	100.0	4.71	85.70	0.0844	1.88
T1,T2	100.0	4.59	85.70	0.0793	0.64
T1,T2,T3	98.47	4.56	95.37	0.0618	46.76
T1,T2,T3,T4	98.71	4.58	95.34	0.0577	49.94
T1,T2,T4	99.94	4.60	85.67	0.0732	3.76
T1,T3	98.47	4.67	95.37	0.0657	46.03
T1,T3,T4	98.79	4.66	95.36	0.0606	49.36
T1,T4	99.9	4.69	85.67	0.0771	4.48
T2	100.0	4.59	85.70	0.0800	0.78
T2,T3	98.47	4.56	95.37	0.0624	46.25
T2,T3,T4	98.86	4.59	95.15	0.0581	50.32
T2,T4	99.74	4.60	85.49	0.0738	5.27
T3	98.47	4.67	95.37	0.0659	46.07
T3,T4	99.27	4.68	95.20	0.0612	47.85
T4	99.47	4.70	85.50	0.0779	7.61

The experimental results show that T1 and T2 have little impact on CFG recovery, maintaining a 100% CFG extraction rate and a cyclomatic complexity of approximately 85.7, indicating that these transformations mainly modify opcode semantics without significantly changing the control-flow structure. In contrast, T3 substantially increases cyclomatic complexity (above 95) while reducing the CFG extraction rate to approximately 98.47%, demonstrating its effectiveness in complicating CFG reconstruction. T4 primarily enhances opcode diversity, maintaining a CFG extraction rate above 99% while reducing the unique opcode ratio, thereby increasing analysis difficulty without major structural disruption.

The complete transformation pipeline (T1+T2+T3+T4) achieves the strongest structural obfuscation, with the highest cyclomatic complexity (95.34) and the lowest unique opcode ratio (0.0577). To further quantify structural changes, Graph Edit Distance (GED) is introduced as an additional metric. As shown in Table X, configurations containing T3 consistently produce the largest GED values, reaching 49.94 for the full pipeline, confirming that the proposed CFG confusion technique significantly alters the graph topology while preserving program semantics. These results complement the CFG recovery analysis and further demonstrate the effectiveness of the proposed framework against graph-based reverse engineering.

5.5 Malicious Behavior Concealment Evaluation

To evaluate the capability of the proposed obfuscation framework to conceal malicious behaviors, we constructed and trained multiple machine learning models on a labeled dataset consisting of 7,000 smart contracts categorized as malicious or benign. The dataset was divided using an 80:20 train-test split to ensure objective and unbiased evaluation. The evaluated models include Random Forest, Gradient Boosting, LinearSVC and Logistic Regression.

Before training the machine learning models, each smart contract bytecode is first converted into an opcode sequence using the `bytecode_to_opcode_str()` function. The opcode sequence is then encoded into a numerical

feature vector using a TF-IDF representation. Specifically, we employ `TfidfVectorizer` with word-level opcode tokens, an n-gram range of 1–3, sublinear term-frequency weighting and a maximum vocabulary size of 10,000 features. The resulting TF-IDF vectors serve as the input features for all evaluated machine learning classifiers. Since the proposed obfuscation techniques preserve execution semantics while modifying opcode sequences and their contextual patterns, they alter the corresponding TF-IDF feature representation, thereby degrading the effectiveness of machine-learning-based malicious contract detection.

TABLE XIII. CLASSIFICATION PERFORMANCE OF MACHINE LEARNING MODELS

Model	Accuracy	Precision	Recall	F1	AUC
Random Forest	93.29	94.49	91.76	93.11	97.29
Gradient Boosting	93.57	94.66	92.2	93.41	97.43
LinearSVC	91.50	90.87	92.05	91.46	96.90
Logistic Regression	90.71	90.26	91.04	90.65	95.77

Experimental results indicate that the Gradient Boosting model achieves the best overall performance, obtaining an Accuracy of 93.57%, an F1-score of 93.41% and an AUC of 97.43%. These results demonstrate that the model provides more effective and stable discrimination between malicious and benign smart contracts compared with the other evaluated models.

After selecting the best-performing classifier, we further evaluated the evasion capability of the proposed obfuscation techniques by applying obfuscation transformations to malicious contracts before feeding them into the detection model. To provide a more comprehensive evaluation, the performance of the best-performing classifier (Gradient Boosting) on the obfuscated contracts is reported using Accuracy, Precision, Recall, F1-score, and Evasion Rate. While the evasion rate serves as the primary indicator of malware concealment effectiveness, the additional classification metrics provide further insight into how different obfuscation configurations affect the detection capability of the classifier.

TABLE XIV. EVASION EFFECTIVENESS OF INDIVIDUAL AND COMBINED OBFUSCATION TECHNIQUES

Combo	Acc	Prec	Rec	F1	Evasion
T1	0.952	1.00	0.952	0.975	4.8%
T2	0.903	1.00	0.903	0.949	9.7%
T3	0.960	1.00	0.960	0.979	4.0%
T4	0.885	1.00	0.885	0.939	11.5%
T1,T2	0.900	1.00	0.900	0.947	10.0%
T1,T3	0.949	1.00	0.949	0.974	5.1%
T1,T4	0.888	1.00	0.888	0.941	11.2%
T2, T3	0.899	1.00	0.899	0.947	10.1%
T2,T4	0.811	1.00	0.811	0.896	18.9%
T3,T4	0.883	1.00	0.883	0.938	11.7%
T1,T2, T3	0.895	1.00	0.895	0.945	10.5%
T1,T2, T4	0.809	1.00	0.809	0.895	19.1%
T1,T3, T4	0.885	1.00	0.885	0.939	11.5%
T2,T3, T4	0.808	1.00	0.808	0.894	19.2%
T1,T2, T3,T4	0.802	1.00	0.802	0.890	19.8%
Mean	0.882	1.00	0.882	0.937	11.80%

The experimental results show that T1 and T3 have a limited impact on the classifier, maintaining accuracies above 95% with relatively low evasion rates of 4.8% and 4.0%, respectively. In contrast, T2 and especially T4 degrade detection performance more significantly, reducing the accuracy to 90.3% and 88.5%, while increasing the evasion rates to 9.7% and 11.5%. These results indicate that junk instruction insertion and pattern-based transformation are more effective at disrupting the TF-IDF opcode features used by the classifier.

As additional obfuscation techniques are combined, the classifier performance decreases consistently while the evasion rate increases. Configurations containing both T2 and T4 achieve the strongest concealment effect, with evasion rates approaching 19%. Notably, the complete pipeline (T1+T2+T3+T4) achieves the highest evasion rate of 19.8%, while reducing the classification accuracy and F1-score to 80.2% and 0.890, respectively, demonstrating the effectiveness of combining complementary obfuscation techniques.

5.6 Comparative Evaluation with Existing Obfuscators

To comprehensively evaluate the effectiveness of the proposed framework, we perform a direct comparison with two representative EVM obfuscation frameworks, BOSC and EVeilM, using the same dataset and experimental environment. The comparison is conducted across multiple aspects including obfuscation success capability, bytecode size and gas consumption overhead, the degree of CFG structural transformation, and the ability to conceal malicious behaviors from automated detection systems.

To ensure a fair and representative comparison, all three obfuscation frameworks (BOSC, EVeilM, and the proposed framework) were evaluated using the same PhishingHook dataset and identical evaluation metrics under the experimental environment described in Section 5.1. Since each framework comprises four obfuscation techniques, the evaluation was conducted using the same fifteen transformation configurations including four individual techniques, six pairwise combinations, four three-technique combinations and the complete four-technique pipeline. Unless otherwise stated, the results presented in Tables XV and XVI correspond to the arithmetic mean across all fifteen configurations for each framework. Consequently, the reported values reflect the overall effectiveness of each obfuscation framework rather than the performance of a particular transformation configuration, thereby providing a fairer and more robust basis for comparison.

TABLE XV. COMPARISON OF BYTECODE OVERHEAD, GAS CONSUMPTION AND SKIP RATE

Tool	avg_pct_bytes_mean	avg_pct_gas_mean	skip_rate_pct_mean
BOSC	4.11	2378.68	2.66
EEveilM	8.51	3.36	4.34
Our solution	15.99	7.04	0.0

(Note: Unless otherwise specified, all reported values correspond to the arithmetic mean over the fifteen evaluated transformation configurations (four individual techniques, six

pairwise combinations, four three-technique combinations and one complete four-technique configuration). This averaging strategy provides a framework-level evaluation and avoids bias toward any specific obfuscation configuration)

Experimental results demonstrate that the proposed framework introduces substantially greater bytecode transformation compared with BOSC and EVeilM, achieving an average bytecode overhead of 15.99%, whereas BOSC and EVeilM produce only 4.11% and 8.51%, respectively. These results indicate that the proposed approach generates significantly stronger structural perturbations, thereby increasing resistance against disassembly, decompilation and CFG recovery frameworks.

In terms of gas consumption overhead, EVeilM maintains the lowest execution cost with an average overhead of 3.36%, while the proposed framework introduces an average overhead of 7.04%. Although this overhead is higher than that of EVeilM, it remains within an acceptable range considering the substantially stronger obfuscation capability and higher degree of structural transformation achieved by the proposed approach. In contrast, BOSC produces extremely high gas overhead, averaging 2378.68% indicating that the framework significantly affects execution cost and is therefore less suitable for practical deployment in blockchain environments where gas efficiency is critical.

Notably, the proposed framework achieves a skip rate of 0%, whereas BOSC and EVeilM exhibit skip rates of 2.66% and 4.34%, respectively. These results demonstrate that the proposed approach provides superior stability and compatibility on large-scale datasets ensuring that all smart contracts are successfully obfuscated without skipped samples or processing failures.

To further evaluate the degree of structural transformation introduced after obfuscation, we additionally compare the proposed framework with BOSC and EVeilM using multiple structural analysis metrics including Shannon entropy, cyclomatic complexity and unique opcode ratio.

TABLE XVI. STRUCTURAL COMPLEXITY COMPARISON OF OBFUSCATION TOOLS

Tool	entropy_ mean	cyclomatic _mean	unique _ratio_ _mean	GED _mean
BOSC	5.79	28.21	0.097	332.88
EveilM	4.61	87.96	0.064	40.08
Our solution	4.63	90.80	0.069	27.13

Table XVI compares the structural characteristics of the proposed framework with BOSC and EVeilM. BOSC achieves the highest entropy (5.79) and Graph Edit Distance (332.88), indicating the most aggressive structural transformations. However, as discussed in the previous section, this comes at the cost of substantially higher runtime overhead. In contrast, EVeilM and the proposed framework exhibit comparable entropy (4.61 vs. 4.63) and cyclomatic complexity (87.96 vs. 90.80), demonstrating similar levels of control-flow diversification. Although the proposed framework produces a lower GED (27.13) than EVeilM (40.08), it maintains comparable structural complexity while introducing more controlled graph modifications. This result suggests that the proposed framework achieves a balanced trade-off between structural diversification and practical deployment overhead rather than maximizing graph-edit distance alone.

In addition, we compare the evasion capability of the proposed framework with EVeilM to assess its effectiveness in concealing malicious behaviors from automated detection systems. The evasion rate is defined as the proportion of malicious contracts incorrectly classified as benign by machine learning-based detectors after obfuscation. Experimental results show that the proposed framework achieves an average evasion rate of 11.80%, slightly higher than EVeilM at 11.23%. Although the difference is modest, the results demonstrate that the proposed framework is more effective in disrupting bytecode features leveraged by automated detection models, thereby providing stronger resistance against machine learning-based malicious contract analysis.

Although the performance gap is relatively small, these results demonstrate that the proposed

framework is more effective at degrading the performance of ML-based detection systems. This improvement primarily stems from the combined use of multiple structural perturbation and semantic transformation techniques, which diversify bytecode characteristics more extensively and thereby create greater challenges for malicious contract classification models.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed an obfuscation framework for EVM bytecode aimed at enhancing analysis resistance and concealing the behaviors of Ethereum smart contracts. Experimental results on the evaluation dataset demonstrate that the proposed framework maintains stable obfuscation capability with a high successful processing rate while introducing substantial structural transformations through increased control-flow complexity and reduced CFG recovery effectiveness for automated analysis tools. Although the proposed framework introduces additional bytecode and gas overhead, the resulting execution cost remains within an acceptable range for practical deployment in EVM environments. Furthermore, evaluation results on machine learning models indicate that the framework is capable of more effectively concealing malicious behaviors, thereby reducing the detection capability of automated analysis systems. These findings demonstrate that the proposed framework has strong practical potential for protecting execution logic and improving reverse-engineering resistance in blockchain-based smart contracts.

Despite the encouraging results, the proposed framework has several limitations. First, the current evaluation is conducted using the PhishingHook dataset and comparisons with representative EVM obfuscation frameworks. Although this provides a consistent experimental basis, further evaluation on more diverse smart-contract datasets and application scenarios would improve the generalizability of the results. Second, this work primarily evaluates resistance against existing static analysis, CFG recovery, and machine-learning-based detection techniques. The effectiveness of the proposed framework against more advanced reverse engineering approaches

including next-generation decompilation, symbolic execution, and AI-assisted program analysis, remains an important direction for further investigation. Another limitation of the current evaluation is that the concealment capability has been assessed using conventional machine-learning classifiers. Although these models provide a reproducible baseline for evaluating changes in bytecode characteristics, further investigation against more advanced deep-learning and LLM-based smart contract analysis frameworks is needed to comprehensively assess the robustness of the proposed obfuscation framework.

Based on these limitations, future work will focus on optimizing execution overhead, improve resistance against modern symbolic execution and decompilation frameworks, and investigate adaptive obfuscation mechanisms capable of automatically selecting suitable obfuscation strategies based on the characteristics of individual smart contracts.

ACKNOWLEDGMENT

This work was supported by the national science and technology project under Grant No. 03/2025/KCM.

REFERENCES

- [1] Z. Sheng, T. K. Quang, S. Wang, S. Duan, K. Li and Y. Duan, "Understanding and Characterizing Obfuscated Funds Transfers in Ethereum Smart Contracts", *Cryptography and Security*, 2026. DOI: 10.48550/arXiv.2505.11320.
- [2] P. V. Huong, N. D. Tuyen, D. H. Long, T. Q. Toanh and N. N. Tuyen, "A Novel Model of Comprehensive Data Encryption for the Website Using Blockchain", *Journal of Science and Technology on Information Security*, vol 1, no. 27, pp. 87-98, 2026. DOI: 10.54654/isj.v1i27.1252.
- [3] Y. Liu, J. He, X. Li, J. Chen, X. Liu, S. Peng, H. Cao and Y. Wang, "An overview of blockchain smart contract execution mechanism", *Journal of Industrial Information Integration*, vol. 41, 2024. DOI: 10.1016/j.jii.2024.100674.
- [4] David, L. Zhou, D. Song, A. Gervais and K. Qin, "Decompiling Smart Contracts with a Large Language Model", *Computer Science*, 2025. DOI: 10.48550/arXiv.2506.19624.
- [5] Github, "Mythril", Access time: 18/5/2026, <https://github.com/ConsenSysDiligence/mythril>.

- [6] Ethereum, "Ethereum.org" (2026), Access time: 10/5/2026, <https://ethereum.org/developers/docs/evm/>.
- [7] A. Olaoye, "HackMD" (12/01/2026), Access time: 05/5/2026, <https://hackmd.io/@ayoola/SJ5jMdGrZg>.
- [8] Hannah, "Medium", Access time: 30/4/2026, <https://medium.com/@hannah.scherz.23/the-basic-structure-of-computer-systems-von-neumann-architecture-18c1cc546ab2>.
- [9] M. Ogawa, "Malware Analysis: A Perspective from Dynamic Symbolic Execution of Binary Code," *Journal of Science and Technology on Information Security*, vol. 2, no. 25, pp. 5-20, 2025. DOI: 10.54654/isj.v2i25.1093
- [10] Z. Zhong, R. Wu, J. Wan, M. Zou and D. Tian, "Hardening Deep Neural Network Binaries against Reverse Engineering Attacks", *CCS '25: Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pp 201 – 215, 2025. DOI: 10.1145/3719027.3765144.
- [11] D. Wang, J. He, Y. Yang, L. Wu, R. Chang and Y. Zhou, "Building Reuse-Sensitive Control Flow Graphs (CFGs) for EVM Bytecode", *Computer Science*, 2025.
- [12] P. D. Rosa, P. Felber and V. Schiavoni, "Seeing Through EVM Bytecode Obfuscation", *IEEE*, vol 14, pp. 25515 - 25536, 2026. DOI: 10.1109/ACCESS.2026.3662238.
- [13] Z. Hou, C. Dong and Y. Shang, "HermHD: Enhancing smart contract security based on code obfuscation", *ICIT '23: Proceedings of the 2023 11th International Conference on Information Technology: IoT and Smart City*, 2024. DOI: 10.1145/3638985.3639001.
- [14] T. Forissier, "EVeilM: EVM Bytecode Obfuscation" (2024), Access time: 10/5/2026, <https://github.com/tit0uanf/EVeilM>.
- [15] P. Zhang, X. Wang, Y. Tong, H. Dong, Y. Xiao and S. Ji, "Bytecode Obfuscation for Smart Contracts via Structural and Semantic Perturbation", *ACM Transactions on Software Engineering and Methodology*, 2026. DOI: 10.1145/3809489.
- [16] P. D. Rosa, S. Queyrut, Y.D. Bromberg, P. Felber and V. Schiavoni, "PhishingHook: Catching Phishing Ethereum Smart Contracts leveraging EVM Opcodes", *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2025. DOI: 10.1109/DSN64029.2025.00033.
- [17] Jon-Becker, "Github", Access time: 10/5/2026, <https://github.com/Jon-Becker/heimdall-rs>.

ABOUT THE AUTHOR

Nguyen Dinh Dai



Workplace: Institute of Cryptographic Science and Technology, Vietnam Government Information Security Commission

Email: nguyendinh dai01@gmail.com

Education: Graduated with an Engineering degree from Military Technical Academy in 2022.

Recent research direction: Cybersecurity, software security and malware analysis.

Tên tác giả: **Nguyễn Đình Đại**

Cơ quan công tác: Viện Khoa học - Công nghệ mật mã, Ban Cơ yếu Chính phủ, Việt Nam.

Email: nguyendinh dai01@gmail.com

Quá trình đào tạo: Tốt nghiệp Kỹ sư năm 2022 tại Học viện Kỹ thuật Quân sự.

Hướng nghiên cứu hiện nay: An ninh mạng, bảo mật phần mềm và mã độc.

Lai Minh Tuan



Workplace: Information security department, Academy of Cryptography Techniques, Vietnam.

Email: tuanlm@actvn.edu.vn

Education: From 2008 – 2016: student, postgraduate student (PhD) majoring in Information Security at Academy of the Federal Security Service of Russia. From 2017 – present: lecturer in Information Security Faculty, Academy of Cryptography Techniques.

Recent research direction: Software security, information system assurance.

Tên tác giả: **Lại Minh Tuấn**

Cơ quan công tác: Học viện Kỹ thuật mật mã, Việt Nam.

Email: tuanlm@actvn.edu.vn

Quá trình đào tạo: Từ năm 2008 đến năm 2016: Sinh viên, nghiên cứu sinh (Tiến sĩ) chuyên ngành An toàn thông tin tại Học viện Cơ quan An ninh Liên bang Nga (FSB). Từ năm 2017 đến nay: Giảng viên Khoa An toàn thông tin, Học viện Kỹ thuật mật mã.

Hướng nghiên cứu hiện nay: An toàn phần mềm, bảo đảm an toàn hệ thống thông tin.