The Journal of Systems and Software 231 (2026) 112593

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Examining the effectiveness of transformer-based smart contract vulnerability scan[☆]

Emre Balci [a], Timucin Aydede [b], Gorkem Yilmaz [a], Ece Gelal Soyak [a],[*]

[a] Bahcesehir University Computer Engineering, Istanbul, Turkey
[b] Bahcesehir University Artificial Intelligence Engineering, Istanbul, Turkey

ARTICLE INFO

Dataset link: https://github.com/emrbalc/VASCOT/

Keywords:
Smart contracts
Vulnerability detection
Transformers
Bytecode analysis
Ethereum Virtual Machine (EVM)
Blockchain security
Deep learning
Decentralized Applications (dApps)

ABSTRACT

Smart contracts can be used for various scenarios, from automating transactions in decentralized finance to managing supply chain logistics, or ensuring the integrity of digital assets. However, smart contracts may expose vulnerabilities that may be exploited, which can lead to financial losses and disruptions in decentralized applications. These vulnerabilities involve decision logic, branching, sequencing, and interaction with other Ethereum addresses, and therefore are challenging to detect. In this work, we study the effectiveness of smart contract vulnerability detection using deep learning. We propose VASCOT, a Vulnerability Analyzer for Smart COntracts using Transformers, which performs sequential analysis of Ethereum Virtual Machine (EVM) bytecode and incorporates a sliding window mechanism to overcome input length constraints. To assess VASCOT's detection efficacy, we construct a dataset of 16,469 verified Ethereum contracts deployed in 2022, and annotate it using trace analysis with concrete validation to mitigate false positives. VASCOT's performance is then compared against a state-of-the-art LSTM-based vulnerability detection model on both our dataset and an older public dataset. Our findings highlight the strengths and limitations of each model, providing insights into their detection capabilities and generalizability.

1. Introduction

Smart contract technology is a fundamental innovation in blockchain, which enables self-executing contracts with the terms of agreement directly written into code. These decentralized programs eliminate the need for intermediaries in transactions, therefore enabling trustless and automated operations. Thanks to being deployed on blockchains such as Ethereum, these contracts are transparent and immutable, which facilitates a wide array of use cases across industries, including decentralized finance (DeFi), supply chains, Internet of Things (IoT) ecosystems, decentralized applications (dApps), and even digital assets such as non-fungible tokens (NFTs) that redefine the ownership of digital art.

Despite their benefits, smart contracts may sometimes contain errors, or the way they are invoked by numerous independent parties can trigger certain errors, rendering them vulnerable for exploitation by malicious entities (Zhang et al., 2023b). Such exploits may compromise the integrity of the contract's execution, leading to financial losses, data breaches, and disruptions to decentralized applications. Several incidents highlighted the impact of these vulnerabilities. To review examples, the most well-known vulnerability occurred in 2016 with the DAO hack, which resulted in a loss of approximately 60 million dollars worth of Ether due to a reentrancy vulnerability (Wikipedia, 2024). Similarly, in 2017, hackers exploiting the vulnerability in Parity Multisignature Wallet stole Ethereum tokens worth 30 million dollars (Palladino, 2017). The bZx protocol suffered from a flash loan attack in 2020, leading to substantial losses (bZx Protocol Exploit Detailed Analysis, 2022). More recently, in Q1-Q2 2023, the DeFi industry lost about 735 million dollars due to exploits (Pearson, 2023). These incidents highlight the continuous threat of financial damage that can occur when attackers exploit vulnerabilities in smart contracts.

Due to the way blockchain works, once smart contracts are deployed, they cannot be updated (i.e., bug fixes cannot be deployed). Therefore, it is crucial to catch any vulnerabilities prior to deployment. Many research studies focused on this endeavor. Static and dynamic analysis techniques have been proposed to identify vulnerabilities such as reentrancy, integer overflow/underflow, and timestamp dependencies (Tikhomirov et al., 2018; Akca et al., 2019); these techniques may overlook certain vulnerabilities. Because some vulnerabilities arise only during specific sequences of contract invocations, symbolic analysis techniques have been proposed (Luu et al., 2016; Nikolić et al.,

[☆] Editor: Professor Tao Zhang.
[*] Corresponding author.
E-mail address: ece.gelalsoyak@bau.edu.tr (E.G. Soyak).

https://doi.org/10.1016/j.jss.2025.112593
Received 9 November 2024; Received in revised form 30 July 2025; Accepted 1 August 2025
Available online 14 August 2025

2018); these methods may be time consuming as the exploration of all executable paths is required. Among the various approaches to vulnerability scanning, machine learning-based sequential analysis of smart contract opcodes offers advantages in quickly detecting vulnerabilities that manifest during contract execution.

In this work, we investigate the efficiency of detecting smart contract vulnerabilities by applying Transformer-based models to EVM opcode sequences. To this end, we propose Vulnerability Analysis of Smart COntracts using Transformers (VASCOT) and to analyze its detection performance, we collect recent contracts from the Ethereum network and label them using trace analysis. We evaluate and compare the performance of VASCOT with the LSTM sequential analysis approach, on both our newly constructed dataset and an older public dataset of smart contracts. This work extends our previous conference paper (Balcı et al., 2023), in which we initially introduced the VASCOT architecture. In this journal version, we significantly enhance the model design, expand the experimental evaluation, and include performance comparisons on two datasets with another model, to demonstrate the robustness and generalizability of our approach. The contributions of our work can be summarized as follows:

- We explain the design of VASCOT, a vulnerability scanner that utilizes the Transformer model. As Transformers work with limited-size input sequences, we implement a sliding window mechanism to ensure VASCOT can properly analyze all input sequences.
- We construct a data set comprising 16,469 verified Ethereum smart contracts deployed in 2022. Having a recent dataset is important to capture the current state of contracts with the most recent language and compiler improvements.
- We compare the performance of VASCOT with that using LSTM on both our new data set, and on an older public data set. We evaluate precision, generalizability, computational requirements. We discuss our observations, and identify the scenarios where VASCOT may or may not be ideal. We highlight the challenges and the mechanisms that can be implemented to improve the generalizability of the AI-based sequential vulnerability scanners.

We release the source code of VASCOT and our collected dataset via GitHub for reproducibility and as reference for future works (https://github.com/emrbalc/VASCOT).

2. Related work

Different techniques have been proposed for detecting vulnerabilities in smart contracts prior to deployment, to prevent incorrectly functioning smart contracts from being created. Earlier **static analysis** approaches examined the source code of a smart contract without executing it. Smartcheck (Tikhomirov et al., 2018) performed lexical and syntactical analysis on Solidity source code by translating it into an XML-based intermediate representation and checking against XML path patterns to identify vulnerabilities. ASGVulDetector (Zhang and Liu, 2022) created an abstract semantic graph to capture the syntactic and semantic features of smart contracts. While it enables studying all possible execution paths, static analysis may not fully capture the interactions between contracts in runtime and thus, may miss vulnerabilities that occur under specific conditions. With **dynamic analysis**, functions are executed to identify vulnerabilities; however, execution has its own limitations, *e.g.,* when test cases do not account for all possible inputs and scenarios. As a result, several studies combined both approaches to improve the accuracy of detection. SolAnalyser (Akca et al., 2019) statically analyzed contract source code to assess locations prone to vulnerabilities, then dynamically executed all functions and transactions with different inputs. The framework proposed in Samreen and Alalfi (2020) combined static and dynamic analysis to detect reentrancy vulnerabilities in Ethereum smart contracts. More recently, VULDEFF (Zhao et al., 2023) proposed vulnerability detection based on function fingerprints and code differences.

Symbolic execution scans EVM *bytecode* to detect vulnerabilities. It works by treating inputs and variables as symbols rather than actual values, and enables testing multiple execution paths at the same time rather than running the program on a specific concrete input. Oyente (Luu et al., 2016) used symbolic execution to capture traces that match the characteristics of defined vulnerability categories. Mythril (A framework for Bug Hunting on the Ethereum Blockchain, 2017) and Manticore (Mossberg et al., 2019) also used symbolic execution; the main drawback with these two tools was their time complexity. Zeus (Kalra et al., 2018) used abstract interpretation and symbolic model checking to analyze Solidity contracts and operated in less time compared to its predecessors.

Many of these early symbolic analyzers modeled the behavior for a single invocation of a contract and were not designed to capture so-called **trace vulnerabilities**, *i.e.,* bugs that arise upon a sequence of calls to a contract. To address this open question, Securify (Tsankov et al., 2018) was designed to decompile EVM bytecode into a stackless dependency graph representation and scan this graph for violation patterns. MAIAN (Nikolić et al., 2018) proposed inter-procedural symbolic analysis on EVM bytecode, to find contracts that violate specific properties of traces. The tool flagged vulnerable contracts with three categories; the contracts that lock funds indefinitely (*greedy*), contracts that leak funds to arbitrary users (*prodigal*), and contracts that can be killed by any user (*suicidal*). The shortcoming of this approach is that, in practice there can be a large number of parallel invocations of a contract's public functions, and it may not be possible to model all possible traces for analysis in limited time. To maintain low levels of analysis duration, the tool has to maintain low invocation depth. To address this limitation, machine learning (ML) and deep learning (DL) based smart contract vulnerability detection techniques have been proposed.

The use of **machine learning** for detecting vulnerabilities was proposed in Tann et al. (2018). The study performed sequence learning on EVM opcode using the Long Short Term Memory (LSTM) model. The model trained and tested on a public data set yielded 99.57% accuracy and F1 score of 86.04%. In our work, we show that these results are not viable on a *verified* data set, and that VASCOT outperforms LSTM in sequential analysis.

Other studies also utilized **deep learning** for smart contract vulnerability detection in various ways. BLSTM-ATT (Qian et al., 2020) transformed EVM source code to snippet representations capturing semantic information and control flow dependencies, then applied bidirectional LSTM with attention mechanism. AWD-LSTM (Gogineni et al., 2020) used average stochastic gradient descent weight-dropped LSTM; to reduce the class imbalance between normal and vulnerable contracts they considered only normal contracts with distinct opcode combinations. ContractWard (Wang et al., 2021) extracted bigram features from opcodes, applied SMOTE and undersampling, and compared different supervised learning models on this data. These approaches had limited efficiency due to requiring initial processing, as features would need to be re-evaluated as contract languages and capabilities evolve. A different approach in Zhang et al. (2022) studied smart contract vulnerability detection as an image classification problem, using a multi-objective detection neural network (MODNN) to convert the operation sequence into explicit features, and constructed co-occurrence matrices using opcodes as implicit features. Differently, some studies considered the syntactic and semantic program information rather than opcodes. Blass (Ren et al., 2023) framework constructed program slices with complete semantic structure information and utilized an attention mechanism to capture the key features of vulnerabilities. Similarly, ESCORT (Sendner et al., 2023) proposed extracting features that capture the semantics of vulnerability classes and trained a multi-output neural network architecture for detecting these types; this work also proposed transfer learning for scaling to new vulnerability types. EA-RGCN (Chen et al., 2023) constructed a semantic graph for each function and trained residual graph convolutional network using an edge attention module. These tools only work on source code.

More recent studies have explored the use of **large language models** (LLMs) for smart contract analysis, initiated by the inaugural work in David et al. (2023). ASSBert (Sun et al., 2023) used transformers but tackled a different problem, i.e., the scarcity of labeled smart contract data for real-world vulnerability detection tasks. To reduce the need for extensive labeled data, semi-supervised BERT network was used to enhance manual labeling and active learning in constructing training set for vulnerability detection. GPTScan (Sun et al., 2024) combined GPT with static analysis for detecting logic vulnerabilities in smart contracts; it utilized GPT for understanding code to break down different logic vulnerability types into code-level scenarios and properties. Similarly, LLM-SmartAudit (Wei et al., 2024) introduced a multi-agent conversational approach, where LLM agents with role-specific instructions collaborate to audit smart contracts and identify vulnerabilities. iAudit (Ma et al., 2024) proposed an iterative approach utilizing LLMs tasked with identifying potential vulnerabilities, analyzing the potential causes for vulnerabilities, then followed by debate-based refinement of causes via additional LLM-based agents. While demonstrating promising results, these tools rely heavily on pre-trained models and prompt engineering strategies, and primarily target source code. In contrast, VASCOT applies a lightweight Transformer architecture on EVM bytecode sequences.

3. Smart contract vulnerabilities

We provide a brief overview of how smart contracts are executed on the Ethereum Virtual Machine (EVM) environment, and how their compiled code may contain patterns that may indicate vulnerabilities.

3.1. Smart contracts and the Ethereum Virtual Machine (EVM)

Smart contracts are stored on a blockchain, and are executed automatically and in a decentralized manner when pre-determined terms and conditions are met. Ethereum Virtual Machine (EVM) is the run-time environment for executing smart contracts on one of the largest and most influential blockchains, the Ethereum network. Smart contracts on the Ethereum network are typically written in Solidity or Vyper, and are compiled to bytecode that can be executed on the EVM. When a smart contract is deployed, its bytecode is stored in a specific address; users interact with smart contracts by sending transactions (*i.e.,* data or Ether) to the contract's address.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract HelloWorld {
    string message;
    constructor(string memory initMessage) {
        message = initMessage;
    }
    function setMessage (string memory newMessage) public {
        message = newMessage;
    }
    function getMessage () public view returns (string memory) {
        return message;
    }
}
```

Algorithm 1: Source code of Hello World contract.

Algorithm 1 shows a contract named *HelloWorld* written in Solidity.[1] The contract contains a single string variable called *message* that is initialized with a value passed to the constructor function. The *setMessage* function allows external parties to set the value of the message variable, and *getMessage* allows external parties to read its current value. The Solidity compiler *solc* converts each function and variable in the source code to its corresponding bytecode representation according to the rules of the EVM. After compilation, the source code may or may not be publicly available on Etherscan, but the EVM bytecode is stored on-chain.

```
6080604052348015620001157600080fd5b5060405162000bf23803
```

Fig. 1. Part of the Hello World contract converted to EVM bytecode.

EVM bytecode is a low-level representation of the program, comprising numeric codes, constants, address references that are generated as the compiler parses and performs a semantic analysis of type, scope, and nesting depths of program objects. Fig. 1 represents part of the bytecode of the Hello World smart contract in Algorithm 1.

The instructions of the hexadecimal bytecode representation are often mapped into a human readable form which is referred to as *opcode* (Bistarelli et al., 2020). Each byte in the bytecode represents a different EVM instruction (opcode) from the Ethereum Yellow Paper (Wood et al., 2014), for operations such as arithmetic (*e.g.,* ADD, SUB, MOD), stack/memory access (*e.g.,* SLOAD, MLOAD, MSTORE), control flow (*e.g.,* JUMP, STOP, SELFDESTRUCT), or other tasks (*e.g.,* ORIGIN, CALLER, GASLIMIT). Each opcode has a unique value; for example, the opcode for pushing a value onto the stack is PUSH1, with hexadecimal representation of 0×60. When the EVM encounters the 0×60 opcode, it reads the next byte as the value to be pushed onto the stack. Similarly, when the EVM encounters the hexadecimal 0×52, representing the MSTORE opcode, it writes the unsigned integer in memory. The bytecode 6080604052 (From Fig. 1) represents the opcode sequence "PUSH1 0×80 PUSH1 0×40 MSTORE", i.e., PUSH the decimal value corresponding to the hexadecimal 0×80 (*i.e.,* 128) onto the stack, then PUSH the decimal value represented by 0×40 onto the stack, then allocate 128 bytes of memory space and move the pointer to the beginning of the 64th byte. EVM documentation (Ethereum Virtual Machine Opcodes, 2017) and the Ethereum yellow paper (Wood et al., 2014) present the list of EVM opcodes and their respective hexadecimal values.

3.2. Smart contract trace vulnerabilities

Smart contract vulnerabilities are flaws in their code that can compromise the security of the contract, render them susceptible to exploitation, potentially leading to financial loss. A list of smart contract vulnerabilities is provided in Muhs (2020). These bugs or vulnerabilities can be categorized in terms of how they are generated (Zhang et al., 2020), following the IEEE Standard Classification for Software Anomalies (IEEE Standard Classification for Software Anomalies, 2010).

Smart contracts can invoke other smart contracts through calls or other message-passing mechanisms. This invocation generally involves multiple transactions, which potentially modify the state of the contract. This system functions similarly to the concept of shared-memory concurrency in computers, and the errors commonly seen in concurrent programming, such as atomicity, synchronization problems, or resource ownership conflicts, are also seen in the transactions on EVM (Sergey and Hobor, 2017).

Consider a smart contract containing the function in Algorithm 2 that allows users to transfer tokens between accounts. Data flow analysis (e.g., tracking the use of *_to* and *_value* values) would help to capture how values flow through the code, e.g. the order in which the statements are executed, or whether certain statements can be executed conditionally based on the values of certain variables.

```
function transfer(address \_to , uint256 \_value) public returns (
    bool) {
    require(_value <= balances[msg.sender]);
    balances[msg.sender] -= _value;
    balances[_to] += _value;
```

[1] In this work, without loss of generality, we focus on the smart contracts written in Solidity and executed on EVM.

```
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
```

Algorithm 2: Function allowing transfer of tokens between funds.

A *trace* is a sequence of invocations of a contract recorded on the blockchain. Trace vulnerabilities in contract source code refer to weaknesses that can be exploited by observing *i.e., tracing* the sequence of operations and the respective state of the contract on the blockchain, during the execution of a smart contract. Such trace-based attacks can lead to vulnerabilities like *front-running*, where attackers anticipate and preempt valid transactions based on their traces, or *reentrancy* attacks that exploit the sequence of function calls and states.

According to their representative characteristics, *trace vulnerabilities* have been categorized into three main groups (Nikolić et al., 2018); those that permanently lock cash, those that leak funds to random users, and those that have the potential to be terminated by anyone. In this work, we focus on these categories of trace vulnerabilities that are explained next.

Greedy smart contracts are contract types where the party to the contract can no longer send funds. An example is the *Race Condition* vulnerability, where two or more actions can be executed concurrently or in an unpredictable order, leading to unexpected or undesirable outcomes. To prevent race condition vulnerabilities, locking mechanisms and atomic operations must be used to ensure concurrent operations do not interfere with each other. Another example is the *Out of Gas* vulnerability. On Ethereum, every transaction is associated with an upper bound on the amount of gas that can be spent, i.e., the amount of computation allowed. Any transaction will fail if the gas spent exceeds this limit. This vulnerability may be exploited in Denial-of-service (DoS) attacks, where the contract may become unavailable or unusable by being overwhelmed with requests or by consuming excessive amounts of resources.

Prodigal smart contracts are those that give away Ether to an arbitrary address that is not the owner or has never deposited Ether to the contract. Vulnerabilities arise that cause funds to be sent from the contract to a malicious user. An example is the *Unchecked Send* vulnerability, which happens due to lack of proper checking during transfer of ownership, paving the way for attackers to manipulate the send function.

Suicidal smart contracts are those where the source code contains the SUICIDE instruction and it can be triggered by a message sender that does not appear in the contract's state at the moment of receiving the message. With these contracts, the ownership can be seized and the contract may be killed, thereby locking the funds in the contract forever.

4. Transformer-based vulnerability scanner system design

Vulnerability Analyzer for Smart COntracts using Transformers (VASCOT) utilizes the transformer architecture to detect the relations between the semantic representations in the smart contract opcode sequence. The VASCOT system accepts compiled smart contract bytecode as input, extracts opcode sequences, and predicts whether the contract exhibits a known vulnerable behavior. This section details the full design of the system, including input processing, model handling, and output interpretation.

4.1. Input and preprocessing pipeline

VASCOT operates on verified smart contracts retrieved from the Ethereum blockchain. The bytecodes of these contracts are obtained using the Etherscan API, as described in detail in Section 5. The raw bytecode is parsed into an opcode sequence by decoding each byte (or byte pair for multi-byte opcodes) into its corresponding symbolic instruction. Opcode sequences are then tokenized using a fixed vocabulary derived from the Ethereum Yellow Paper (Wood et al., 2014) and extended to include opcodes introduced in newer Solidity versions (Solidity Documentation, 2025). Each opcode (e.g., PUSH1, CALL, MSTORE) is mapped to a unique integer token; unsupported opcodes (e.g., opcodes introduced in newer compiler versions) are mapped to a special token. This mapping ensures all contracts can be processed by the model. As a result, a numerical representation of the contract instruction flow is obtained.

Table 1
Transformer model configuration.

Layer	Input shape	Output shape	Parameters
Embedding	(batch size, 2048)	(batch size, 2048, 128)	19,200
Transformer Encoder	(batch size, 2048, 128)	(batch size, 2048, 128)	592,256
Attention Pooling	(batch size, 2048, 128)	(batch size, 128)	49,536
Linear	(batch size, 128)	(batch size, 2)	258

4.2. Transformer architecture

The architecture of the transformer model that is at the core of VASCOT is shown in Fig. 2. For each smart contract sample, the corresponding token sequence is passed through an embedding layer, which transforms each token into a 64-dimensional vector representation. This process allows the model to represent similar opcodes with vectors that are close in the embedding space.

In opcode analysis, where the relative ordering of instructions directly influences contract logic, keeping track of the position of each token within the sequence is essential for understanding control flow and execution context. To achieve this, positional encodings are added to the embedded vectors. These encodings explicitly encode the position of each token within the sequence, allowing VASCOT to distinguish between identical opcodes appearing in different locations.

The position-aware embeddings are then passed to the Transformer block. This block leverages a self-attention mechanism, which enables VASCOT to dynamically focus on different parts of the opcode sequence based on contextual relevance. This is achieved by computing attention scores between all token pairs in the sequence; thus, the model can capture relationships between opcodes at various positions of the sequence. VASCOT uses the attention scores to weigh the importance of each token in understanding the overall structure within the contract. It must be noted that prior to attention computation, positional encodings are added to the embeddings to ensure that VASCOT can distinguish between identical opcodes occurring in different positions.

The output of the Transformer block then undergoes an Attention Pooling layer, which dynamically learns to focus on the most important parts of the sequence while reducing it to a 1D format while retaining crucial features. Specifically, this attention mechanism computes importance scores using a tanh-activated linear projection followed by a softmax normalization, and forms a weighted sum of token representations to capture the most informative subsequences. We then apply two dropout layers with a rate of 0.2 to prevent overfitting. Between these, we include a dense layer with a ReLU activation function to add non-linearity and further refine the learned features. This combination of dropout, ReLU activation, and subsequent dropout layers contributes to the model's robustness while preserving meaningful representations before the classification step. The model configuration is shown in Table 1.

4.3. Sliding window implementation

The Transformer model has a limit on the sequence length it can handle in one pass; this is due to the self-attention mechanism, which scales quadratically with the sequence length. This limit is controlled by the *max_length* parameter, which is set to restrict the size of the input sequence, for computational efficiency. Our data set contains longer sequences than the *max_length* value. While we could address

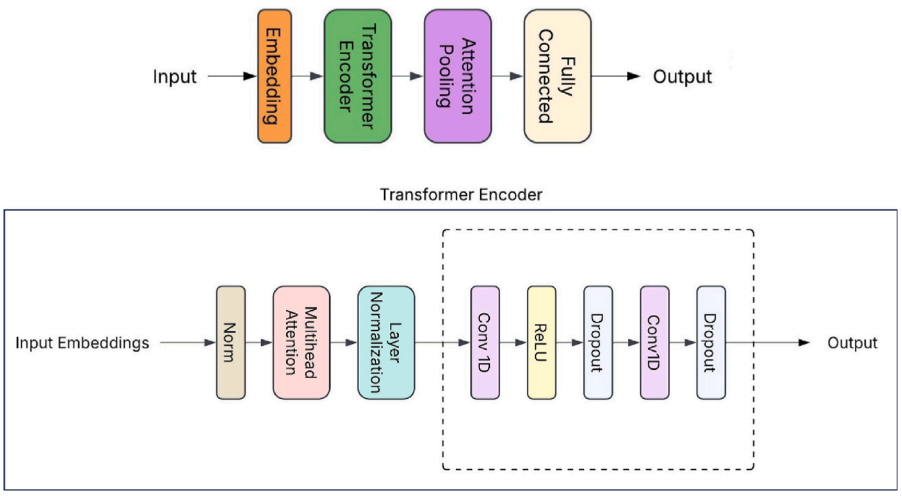


Fig. 2. VASCOT transformer architecture.

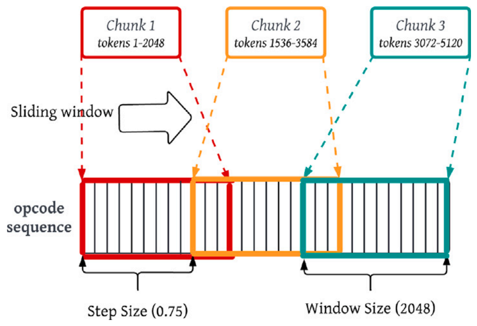


Fig. 3. Sliding window implementation on VASCOT for handling contracts larger than 2048 opcodes.

this issue by *truncating* the opcode sequences to *max_length*, this would potentially manipulate the data in a way where vulnerability traces did not exist in the feature set. Another alternative approach would be *chunking*, where the sequence is split into non-overlapping chunks smaller than the *max_length*; however, this approach could conceal important information if vulnerability features are scattered across separate chunks.

Since all of the input sequences are important in our dataset, we proceeded with another solution, and we process the input sequence in a *sliding window* approach. Any input sequence that is larger than *max_length* is divided into overlapping chunks of tokens. These chunks are fed into the model sequentially, and the results from each chunk are aggregated by taking the mean of their predicted probabilities to produce a final contract-level prediction. This approach provides a balanced view over the entire opcode sequence, avoids overfitting to any single window, and ensures consistency with the training-time assumption that each chunk contributes independently to the prediction. In our implementation, we have included the window size and the step size parameters in hyperparameter optimization, and the optimization yields a window size of 2048 tokens and a step size of 0.75, yielding an overlap of 0.25 across windows. The sliding window approach is demonstrated in Fig. 3.

5. Data sets

The experimental study in this work has been carried out on two separate datasets. We collected the smart contracts in the first dataset; its construction is explained in detail in this section. The second dataset is a public dataset that contains more samples but dates back several years prior to the first dataset.

5.1. Dataset comprising verified contracts

As Solidity language and solc compiler are enhanced through the years (Pinna et al., 2019), the vulnerabilities present in smart contracts written/compiled in different versions of the language/compiler may change. To obtain a data set of recent contracts, we construct a labeled data set comprising the *verified* smart contracts deployed in 2022. Verified smart contracts have their source code (as well as exact compiler version and settings used) publicly available, such that a third-party platform such as Etherscan can recompile the provided source code and check if the resulting bytecode exactly matches the bytecode deployed on the Ethereum network. If the bytecodes match, the contract is marked as “verified”.

5.1.1. Accessing smart contracts

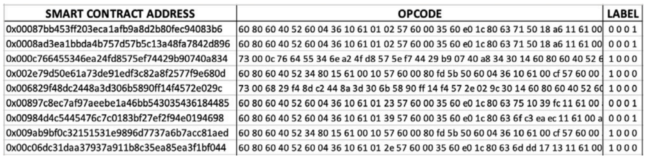
We used Etherscan (Etherscan Export, 2024) to access verified smart contracts. First, we created a developer key to interact with the Etherscan API via the Etherscan.Net.lib library. Using this API, we obtained a list of verified contract addresses that have open-source licenses. For each address in the .csv file, the API was invoked to retrieve the corresponding source code from the specified endpoint. Retrived contracts were saved using the ContractAddress_ContractName.sol naming convention, which helped to identify any duplicates.

While saving the contract source codes returned from the Etherscan API, we noticed cases where more than one .sol file was returned. In order to parse the source code correctly, we compared the .sol values in the *sourceCode* field of the JSON-formatted response. We extracted and saved only the contracts that contained a single “.sol” value.

The Etherscan API provides access to a random set of contracts typically ranging from 100 to 600 on a daily basis. Using this API, we were able to obtain the data of 16,363 *verified* contracts over an interval of four months between January 2022 and April 2022.

The JSON response returned over the Etherscan API also contains the bytecode, in addition to source code. To convert the EVM bytecode to EVM opcode hex, each byte in the bytecode is parsed into its respective hexadecimal representation, but without the leading $0x$’s. For example, in the bytecode in Fig. 1, the first opcode is PUSH1 0×80, which can be represented by the hexadecimal representation of 0×60 0×80. The next opcode is PUSH1 0×40, which is represented as 0×60 0×40. The respective opcodes for Fig. 1 (without the leading 0x’s) are 60 80 60 40 52 34 80 15 62 00 00 11 57 60 00 and so on.

The opcode sequence corresponding to each contract address is recorded, along with the corresponding label. This sequence constitutes the feature set for VASCOT, using which the vulnerability patterns will be identified.



SMART CONTRACT ADDRESS	OPCODE	LABEL
0x00087bb453ff203eca1afb9a8d2b80fec94083b6	60 80 60 40 52 60 04 36 10 61 01 02 57 60 00 35 60 e0 1c 80 63 71 50 18 a6 11 61 00	0 0 0 1
0x0008ad3ea1bbda4b757d57b5c13a48fa7842d896	60 80 60 40 52 60 04 36 10 61 01 02 57 60 00 35 60 e0 1c 80 63 71 50 18 a6 11 61 00	0 0 0 1
0x000c766455346ea24fd8575ef74429b90740a834	73 00 0c 76 64 55 34 6e a2 4f d8 57 5e f7 44 29 b9 07 40 a8 34 30 14 60 80 60 40 52 6	1 0 0 0
0x002e79d50e61a73de91edf3c82a8f2577f9e680d	60 80 60 40 52 34 80 15 61 00 10 57 60 00 80 fd 5b 50 60 04 36 10 61 00 cf 57 60 00	1 0 0 0
0x006829f48dc2448a3d306b5890ff14f4572e029c	73 00 68 29 f4 8d c2 44 8a 3d 30 6b 58 90 ff 14 f4 57 2e 02 9c 30 14 60 80 60 40 52 60	1 0 0 0
0x00897c8ec7af97aeebe1a46bb543035436184485	60 80 60 40 52 60 04 36 10 61 01 23 57 60 00 35 60 e0 1c 80 63 75 10 39 fc 11 61 00	0 0 0 1
0x00984d4c5445476c7c0183bf27ef2f94e0194698	60 80 60 40 52 60 04 36 10 61 01 39 57 60 00 35 60 e0 1c 80 63 6f c3 ea ec 11 61 00 a	0 0 0 1
0x009ab9bf0c32151531e9896d7737a6b7acc81aed	60 80 60 40 52 34 80 15 61 00 10 57 60 00 80 fd 5b 50 60 04 36 10 61 00 cf 57 60 00	1 0 0 0
0x00c06dc31daa37937a911b8c35ea85ea3f1bf044	60 80 60 40 52 60 04 36 10 61 01 2e 57 60 00 35 60 e0 1c 80 63 6d dd 17 13 11 61 00	1 0 0 0

Fig. 4. Part of the final data set.

5.1.2. *Labeling the data*

To label the smart contracts, for each retrieved smart contract source code, we ran the analysis using MAIAN (Gritz, 2022) to categorize the data into four classes representing smart contracts with no vulnerabilities, as well as those that demonstrate suicidal, greedy, and prodigal smart contract characteristics. In our trials, we encountered errors caused by MAIAN's incompatibility with the more recent smart contracts' compiler versions. For the collective solution to this problem in the new contract pool we create, we overwrote the compiler version to the range between Solidity v0.4.22 and v0.9.0 to work correctly with MAIAN.

MAIAN formulates the erroneous behaviors in terms of predicates of observed traces that are created during execution. For example, a transition label of the form *call(id, m)* captures the fact that a currently running contract has transferred control to another contract *id*, by sending it a message *m*. MAIAN checks if there is an execution trace that violates ETHERLITE rules. The predicates allow capturing repeating patterns in contract life cycle (*greedy*), the conditions indicating vulnerability to unauthorized access to a contract's funds (*prodigal*), or a contract's suicide functionality (*suicidal*).

In our output dictionary, the labels are one-hot encoded. Thus, the string "1 0 0 0" represents no vulnerabilities, "0 1 0 0" represents suicidal, "0 0 1 0" represents prodigal, and "0 0 0 1" represents greedy vulnerabilities. A .csv file containing the addresses of all smart contracts is formed; the final data set contains contract address, the opcode, and the respective label. A sample of contracts in our final data set are shown in Fig. 4.

5.1.3. *Preprocessing*

The Etherscan API returns a random set of contracts upon each query, and our observations revealed many duplicates among the 16,363 collected contracts in the data set. The contracts that were flagged as vulnerable contained 8559 contracts with the suicidal flag set to 1 and 7803 contracts with greedy flag set to 1. We observed that our data set contained only 1 contract with the prodigal flag set to 1 . We attribute this to the fact that the obtained data set comprises only contracts that have been verified via an audit process, and that the prodigal behavior is explicit and therefore can be caught and prevented. The single prodigal contract, as well as the duplicate contracts that were reported to be both suicidal and prodigal, were removed from the data set. We then removed duplicate contracts that were marked both suicidal and greedy. We found 733 duplicates in Suicidal contracts and 572 duplicates in Greedy contracts and deleted these from the dataset, obtaining 7826 suicidal and 7231 greedy contracts. At the end of this phase, there were 1202 contracts which were not marked as any of the three vulnerable types.

MAIAN facilitates symbolic execution of the contract bytecode on a private fork of Ethereum blockchain using concrete transaction values, to validate the contract exhibits vulnerable behavior that can be exploited in concrete execution. Contracts that do not exhibit specified traces are identified as false positives. Upon concrete validation on a private fork of Ethereum, it was observed that the greedy and suicidal contracts in our dataset contained false positives. Among the false positive suicidal contracts, some of them were greedy true positives, and vice versa. For an accurate classification into the sub-classes, we wanted to eliminate any obfuscation; thus, we removed such entries

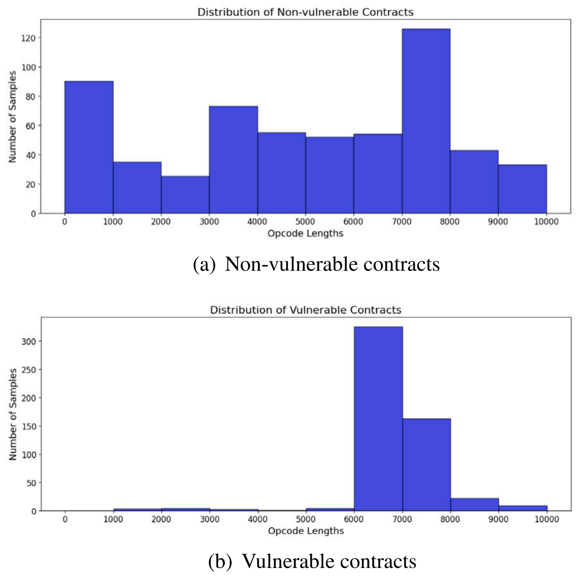


Fig. 5. Distribution of the number of vulnerable (greedy) and non-vulnerable smart contracts in our collected data set.

from the data set. After removing such entries, our dataset contained contracts that were each flagged in a single vulnerability category. Specifically, **1915** total unique contracts remained in the data set, containing **1202** non-vulnerable contracts and **713** greedy contracts.

Fig. 5 demonstrates the distribution of contract length (number of opcodes) for all verified smart contracts in our data set. We observe that the length of contracts in these figures, particularly those that are flagged as vulnerable, are longer (average length of ∼6000) compared to the distribution of Google BigQuery data set explained in Section 5.2, where the average contract length is ∼1500 opcodes. Verified contracts that are active in Ethereum as of 2022 have longer opcode lengths.

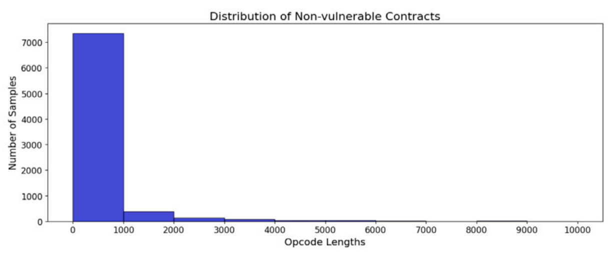
5.2. *Dataset comprising unverified contracts*

In order to measure the generalizability of VASCOT and compare it with that of LSTM, we also used a large public data set, the Google BigQuery data set (Google BigQuery, 2018). This dataset contains a total of 891,162 contracts, including all contracts from the first block of Ethereum, up until block 4,799,998, which was the last block mined on December 26, 2017. It has been used in previous studies (Nikolić et al., 2018; Tann et al., 2018).

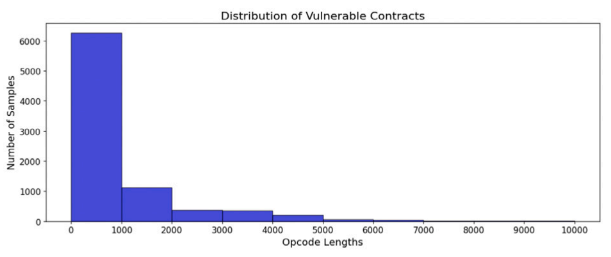
In Tann et al. (2018), this dataset was preprocessed, and 8469 distinct contracts were identified as vulnerable, with **1207** of them flagged as suicidal, **1461** of them prodigal, and **5801** of them greedy. We used these vulnerable contracts in our experiments. After removing invalid opcode instructions or duplicates, the public dataset contained 416,944 unflagged contracts. From this set, we selected a subset of 8000 unflagged contracts and thus constructed a balanced data set comprising 16,469 contracts.

Fig. 6 demonstrates the histograms of the length of smart contracts in terms of the number of opcodes in the compiled opcode, for vulnerable and non-vulnerable contracts. On these figures we have two main observations; *(i)* the vulnerable contracts are slightly longer in length, and *(ii)* the smart contracts in this data set have at most 7000 opcodes and most of the smart contracts contain less than 1500 opcodes.

Next, we also generated a data set comprising only the type of vulnerabilities that exist in the 2022 dataset. Thus, we included only the **5801** greedy vulnerable contracts and **8000** non-vulnerable contracts and combined them into a dataset of size 13,801. We plot the distribution of length for vulnerable and non-vulnerable smart contracts in Fig. 7. The distribution of compiled contract length for vulnerable

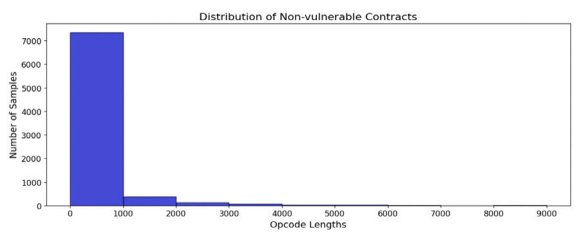


(a) Non-vulnerable contracts

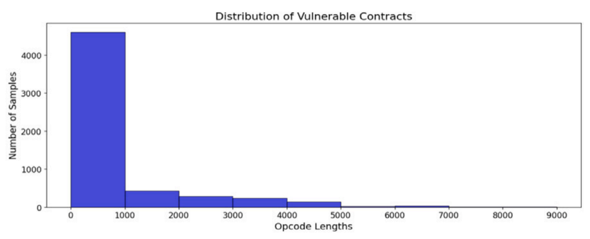


(b) Vulnerable contracts

Fig. 6. Distribution of the number of vulnerable (greedy, prodigal, suicidal) and non-vulnerable smart contracts in the Unverified contract data set containing 16,469 contracts.



(a) Non-vulnerable contracts



(b) Vulnerable contracts

Fig. 7. Distribution of the number of greedy and non-vulnerable smart contracts in the Unverified data set containing 13,801 contracts.

and non-vulnerable contracts for this data set closely match those in the complete unverified data set (containing all three vulnerability types) but strictly differs from the respective distributions in the newly constructed 2022 data set.

6. Experiments

In our experiments, we evaluate the performance of VASCOT on the different smart contract data sets explained in Section 5, and compare it with the LSTM model in terms of accuracy, precision and recall for identifying vulnerabilities. The datasets are summarized in Table 2. In the following, we first provide model implementation details, then explain each setup and our observations therein.

Table 2
Smart contract data sets.

	Date	Normal count	Vulnerable count	Vuln.	Trans-parency
Dataset I	2022	1202	713	G	Verified
Dataset II	2017	8000	8469	S,P,G	Unverified
Dataset II-r	2017	8000	5801	G	Unverified

Table 3
Optimized model parameters.

Parameter	VASCOT	LSTM
num_words	150	150
max_length	2048	2048
embedding_dim	128	128
batch_size	64	64
epochs	50	50
learning_rate	0.0005	0.0005
dropout	0.2	0.2
head_size	32	–
hidden_dim	–	256
num_heads	4	–
ff_dim	4*256	–

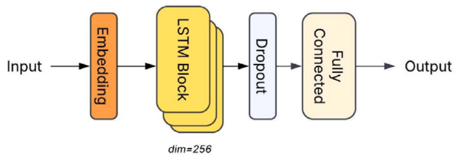


Fig. 8. The LSTM architecture used in the experiments.

6.1. Implementation details

The experiments are carried out on NVIDIA TESLA P100 GPU. The datasets are divided into train, validation, and test partitions at 68%, 15%, and 17% proportions, respectively. We study both the binary classification problem of identifying vulnerable contracts, as well as multi-class classification of individual vulnerability types.

Our model implementation is based on PyTorch. The model architectures for VASCOT and LSTM are demonstrated in Fig. 2 and Fig. 8, respectively. We selected the LSTM-based model as our baseline due to its prior use in opcode-level vulnerability detection (Tann et al., 2018), ensuring a fair comparison under identical input and preprocessing conditions.

We have performed hyperparameter optimization for both models, and the optimized Transformer and LSTM parameters are shown in Table 3. We observed that VASCOT performed better when the model was simpler, *e.g.,* with fewer attention heads and smaller hidden layers. The number of distinct input tokens, i.e., *num_words* for the Transformer is based on the number of distinct opcodes in EVM (Wood et al., 2014). We used Adam optimizer and binary cross-entropy loss function in both LSTM and Transformer models. For multi-class classification into specific vulnerability types, we used categorical crossentropy, as the target labels are one-hot encoded (Fig. 4).

6.2. Experiments on Dataset I: Verified contracts

First, we comparatively study the performances of LSTM and VASCOT on Dataset I. This dataset contains only greedy vulnerabilities and non-vulnerable contracts. The relative performances of the two models in this setup is of particular importance, as it more realistically captures the success in capturing vulnerabilities in more recent and verified smart contracts.

The training loss for LSTM and VASCOT models are 0.19 and 0.20, respectively, while the validation loss values are 0.30 and 0.19 for these models. We observe that LSTM slightly overfits while VASCOT

Table 4
Binary classification performance on verified contracts.

	Precision		Recall		f1-Score		Support
	L	V	L	V	L	V	
0	0.89	0.95	0.90	0.96	0.90	0.95	205
1	0.83	0.93	0.81	0.91	0.82	0.92	121

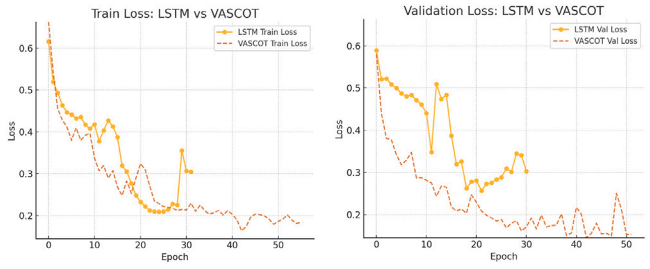
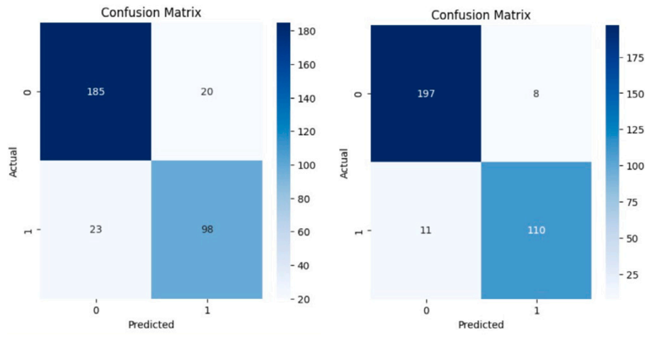


Fig. 9. Training and validation loss curves for LSTM vs. VASCOT.



(a) LSTM (b) VASCOT

Fig. 10. Confusion matrices for LSTM vs. VASCOT on verified contracts.

can generalize better to new sequences. Fig. 9 demonstrates the slight overfit with LSTM on this dataset.

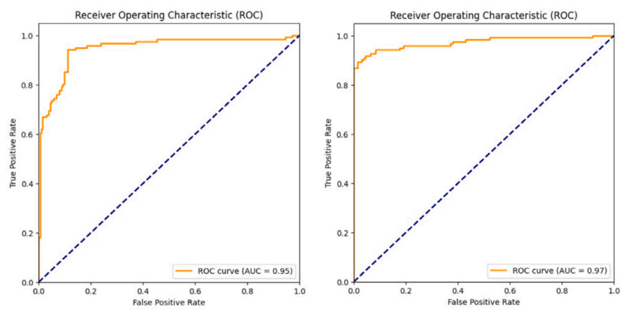
Table 4 shows the test performance of both models on this data set, where **L** is for LSTM and **V** is for VASCOT. Testing has been performed on 326 samples, LSTM obtained an accuracy of 90% and VASCOT's accuracy was 93%. VASCOT outperformed LSTM in detecting both non-vulnerable and greedy contracts, despite the small training set size. As the problem at hand aims to detect vulnerability, it is particularly valuable that VASCOT attains 10% higher precision and 10% higher recall than LSTM in detecting vulnerable samples (class "1").

Fig. 10 displays the confusion matrices for both models when trained and tested on verified contracts. VASCOT significantly reduces both false positives and false negatives compared to LSTM. Thanks to its self-attention mechanism, transformer scales well with long sequences, whereas LSTMs have lower performance in capturing dependencies in longer sequences due to their sequential nature. The vulnerable contracts in this dataset are longer on average compared to non-vulnerable contracts (Fig. 5); LSTM's lower performance on vulnerable contracts may be due to a limitation of handling long sequences. The ROC curves in Fig. 11 also strengthens the argument that VASCOT performs better on recent, verified contracts.

6.3. Experiments on Dataset II: Unverified contracts

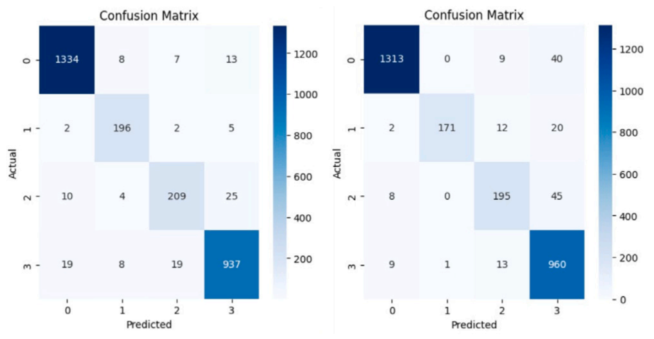
Next, the two models are compared on Dataset-II. This set comprises all three types of vulnerabilities as shown in Table 2. We evaluate the model's success in identifying any type of vulnerable contracts.

On this dataset, a slightly higher validation loss compared to training loss is observed with both LSTM and VASCOT. Fig. 12 shows the confusion matrices of both models, where 0, 1, 2, 3 correspond to non-vulnerable, suicidal, prodigal and greedy contracts, respectively. We observe that among all classes, VASCOT performs poorest in detecting prodigal and suicidal vulnerabilities. This lower performance can be insignificant in assessing the applicability and contribution of VASCOT today, considering the suicidal and prodigal behaviors have significantly been eliminated in recent contracts (hence they are unseen in Dataset I), both due to improvements in Solidity language (discussed under Section 7) and the use of auditing practices prior to deployment. It must also be highlighted that the contracts in this dataset were *unverified*, suggesting this dataset may not be suitable for benchmarking. We also observe that LSTM outperforms VASCOT in identifying non-vulnerable contracts. This result is in support of the previous work (Tann et al., 2018), where the reported accuracy was primarily due to correctly detecting the large number of (specifically, 200,000) non-vulnerable samples. On the other hand, we observe VASCOT can detect greedy contracts with better precision.



(a) LSTM (b) VASCOT

Fig. 11. ROC curves for LSTM vs. VASCOT on verified contracts.



(a) LSTM (b) VASCOT

Fig. 12. Confusion matrices for LSTM and VASCOT on Dataset II.

6.4. Experiments on Dataset II-r: Unverified filtered contracts

Dataset II contains vulnerability classes that have not been found in the recent contracts. To provide a fair comparison, we removed the suicidal and prodigal contracts and constructed Dataset II-r (representing reduced Dataset II) comprising 5801 greedy contracts and 8000 non-vulnerable contracts.

Fig. 13 demonstrates the confusion matrices on Dataset II-r. Though the models demonstrated comparable performance, considering the false negatives are more critical, VASCOT outperforms LSTM in identifying vulnerable patterns with 6 false negatives compared to 24 from LSTM, out of 983 contracts.

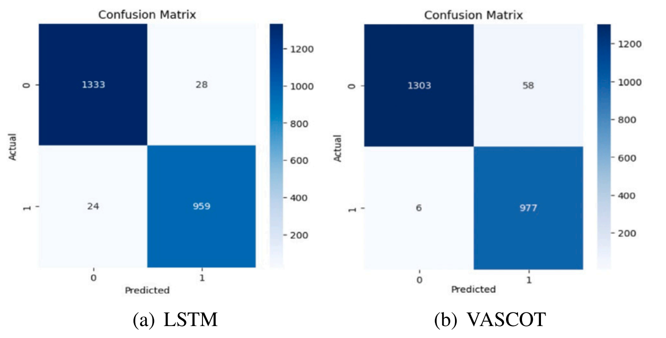


Fig. 13. Confusion matrices for LSTM and VASCOT on Dataset II-r.

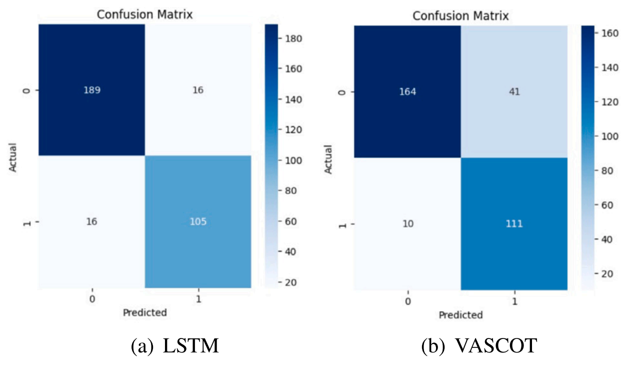


Fig. 14. Confusion matrices for the unverified–verified cross-dataset analysis.

6.5. Cross-dataset analysis

To examine the generalizability of the two models, we also perform cross-dataset analysis. Specifically, we train the models on unverified contracts and test on verified contracts, and vice versa. In these experiments, Dataset II-r is used for the unverified contracts, so that models train to recognize the classes that are present in both datasets, *i.e.,* non-vulnerable and greedy.

6.5.1. Train on unverified, test on verified

First, we train the model on all of the (unverified) contracts in Dataset II-r. We fine-tuned the model on a subset of the verified contracts (Dataset I), and tested on the remaining subset.

Fig. 14 demonstrates the confusion matrices corresponding to the binary classification using both models on 326 contracts, 205 of which are non-vulnerable and 121 greedy. Similar to other results, VASCOT has poorer performance on non-vulnerable contracts, but better performance on detecting vulnerabilities. LSTM's poor performance in detecting the vulnerable contracts in the recent dataset may again be due to the average contract length being larger in the recent contracts. A comparison of Figs. 5(a) and 7(a) demonstrates that the length of non-vulnerable contracts differs between the two datasets, but the difference is even more highlighted for the vulnerable contracts as observed in Figs. 5(b) and 7(b). Fig. 16 demonstrates that in addition to the length of contracts, the opcodes in them also differ in these two datasets. It must be noted that the bytecode-to-opcode translation has been done through the yellow paper (Wood et al., 2014), which does not include many opcodes added later (labeled as "UNKNOWN" in table). These changes may be due to updates in the Solidity language, changes to the EVM compiler, or new Ethereum protocol upgrades, which are discussed next.

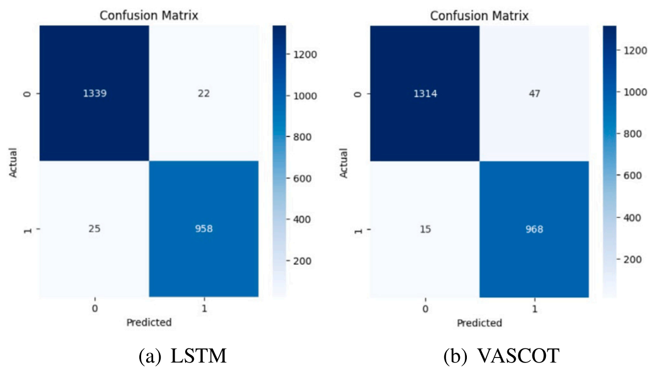


Fig. 15. Confusion matrices for the verified–unverified cross-dataset analysis.

6.5.2. Train on verified, test on unverified

Our final experiment investigates the performance of the models when trained on the verified contracts (Dataset I, 1915 contracts) and tested on the unverified contracts (Dataset II-r). The confusion matrices resulting from the binary classification using LSTM and VASCOT are presented in Fig. 15. Although LSTM more correctly classifies non-vulnerable contracts, it fails to detect 2.5% (25 out of 983) greedy contracts where VASCOT fails to detect only 1.5%.

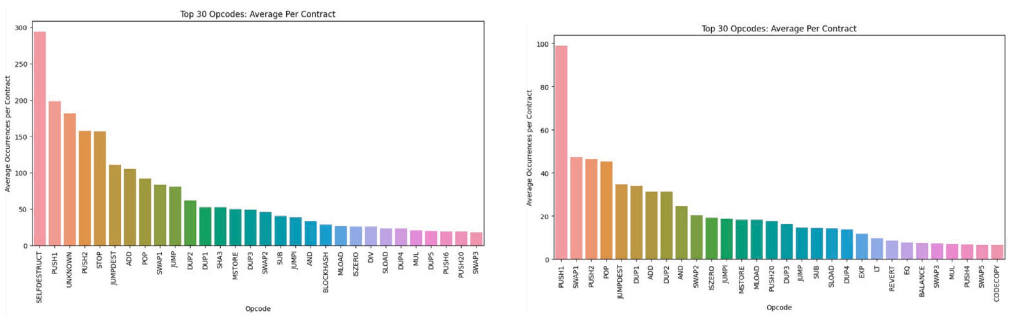
This scenario offers both the challenge of cross-dataset training and testing, and the fact that the training set is small. In summary, both cross-dataset experiments demonstrated challenges in the generalizability of the two models. The two datasets were collected four years apart, and they significantly differed as seen from the contract lengths in Figs. 5 and 6.

7. Challenges and future research directions

Towards identifying smart contract vulnerabilities, VASCOT has the advantage of not requiring contract source code or any preprocessing of the bytecode. VASCOT could be used as an additional layer of analysis upon contract verification on Ethereum. VASCOT could also be used alongside real-time monitoring or automation tools, such that an alert or an automated action could be triggered as soon as vulnerability detected. However, there are future research directions that need to be explored prior to deploying VASCOT or other AI-based scanners reliably in practice; we discuss them briefly in this section (see Fig. 15).

Labeling datasets: One shortcoming of this approach is the reliance on a particular tool for labeling the data in the training set. For VASCOT, a well-known symbolic analysis tool has been used (Nikolić et al., 2018). One area for improvement in future work could be to label contracts via multiple tools (*e.g.,* multiple symbolic analyzers) or multiple types of tools (*e.g.,* static and dynamic analyzers). Additionally, active learning or semi-supervised learning methods can be utilized, to manually label a subset of the data, or to train the model on both labeled and unlabeled data, which would help provide reliable datasets to train models on.

Adaptation to changing programming language and compiler: The bytecodes (opcodes) of smart contracts can change between different Solidity versions especially due to updates in the Solidity compiler. As an example, Solidity syntax evolved significantly between v0.4.x (2017) and 0.8.x (2022) (Solidity Documentation, 2025), covering the introduction of *(i)* visibility modifiers (*e.g.,* public/private, internal/external), affecting how functions are translated into bytecode, *(ii)* inheritance modifiers (*e.g.,* virtual/override) affecting layout and logic in the compiled bytecode, *(iii)* SafeMath library to prevent integer overflows and underflows, leading to different bytecode representations for arithmetic operations, and *(iv)* custom error handling (try/catch) instead of passing error messages to *require*() or *revert*() functions, changing the bytecode for revert operations. Additionally, Ethereum



(a) Verified, 2022   (b) Unverified, 2015-2017

Fig. 16. 30 most common opcodes in the unverified and verified datasets. For each opcode, the total count was averaged over the number of contracts that contained this opcode in the given dataset.

Table 5
Computational complexity with VASCOT compared to LSTM on different datasets.

	Dataset I					Dataset II					Dataset II-r				
	Train. (s)	Infer. (ms)	CPU (%)	RAM (GB)	GPU (GB)	Train. (s)	Infer. (ms)	CPU (%)	RAM (GB)	GPU (GB)	Train. (s)	Infer. (ms)	CPU (%)	RAM (GB)	GPU (GB)
LSTM	107.3	1.35	26.2	2.82	9.77	572.7	0.97	20.9	3.15	9.76	436.5	0.93	20.7	3.16	9.76
VASCOT	286.1	1.59	28.2	2.99	16.55	1281.0	1.45	19	2.95	16.54	1170.4	1.71	22.4	2.83	16.54

periodically introduces new opcodes or changes existing opcodes to make the EVM more efficient. The impact of such changes on the opcodes that are most commonly used in contracts in the two datasets considered in this work are shown in Fig. 16.

VASCOT must be able to adapt to identifying new vulnerabilities that arise in more recent contracts. To achieve this, continual learning approaches can help to adapt to new input sequences while maintaining the ability to detect the vulnerabilities in the older contracts. Combining transfer learning and online learning could also be used to fine-tune the model on the new dataset.

Computational complexity: We have measured training and inference speeds and hardware requirements for the two models being compared; the results are summarized in Table 5. Naturally, the training cost for both models increase with dataset size. On each dataset, LSTM completes training in approximately half the time compared to VASCOT. However, once trained, the inference times are comparable. The average inference time for the contracts in Dataset I are slightly longer for both models, as the average contract length in this set is higher (Fig. 5). With VASCOT, inference remains within reasonable bounds (*e.g.,* ∼1.5 ms); it may be efficiently integrated into auditing pipelines where contracts are scanned in batches. Memory requirements of the two models are similar, though VASCOT is slightly more efficient. CPU requirements are also similar; however, VASCOT requires GPU and has a bigger footprint on the GPU.

Robustness against adversarial attacks: Like many systems that rely on AI models for detection (of a human, or a malware/anomaly), transformer-based vulnerability detection, too, may be the target of adversarial ML attacks. Since Transformers rely on token relationships, VASCOT learns vulnerable patterns from the input sequence. One attack approach for such models is obfuscation, which would allow adversarial contracts to evade detection (Wu et al., 2024). Since LSTM relies on local sequential relationships, obfuscation via rearranging opcode sequences, insertion of redundant opcodes, or replacing with semantically similar opcode subsequences may reduce its detection performance. As Transformers better capture long-range dependencies among opcodes, the impact on VASCOT would be lower; nevertheless, added or shuffled tokens (opcodes) can still disrupt learned embeddings and cause misclassification. Adversarial training, *i.e.,* including obfuscated samples in the training set may help to improve VASCOT's generalizability and resilient against such attacks. Such samples may be generated by source obfuscation (Zhang et al., 2023a) or data augmentation (Hwang et al., 2024) techniques.

8. Conclusions

In this work, we studied vulnerability detection in smart contracts using deep learning tools that sequentially scan the contract bytecode/opcode. We proposed a smart contract vulnerability scanner, VASCOT, that utilizes the Transformer model. We also collected a dataset containing the opcodes of 16,469 verified Ethereum smart contracts from 2022. We evaluated the performance of VASCOT in comparison with a prior sequential opcode scanner based on LSTM; the models were analyzed on both the constructed data set and a public data set collected between 2015–2017, the choice of which was driven from its prior adoption by literature. We conducted controlled experiments, whose results showed that VASCOT outperforms LSTM when scanning newer and longer contracts, and consistently better identifies greedy vulnerabilities in both datasets and in cross-dataset experiments. In addition, once trained, the inference time per contract is very small, ∼1.5 ms, permitting the integration of VASCOT into auditing pipelines for increased utility with minimal time cost. However, the improvement in vulnerability detection comes at the cost of higher GPU resource requirement and some false positives. To improve trust and human interpretability, integrating attention-based explainability (*e.g.,* highlighting opcode spans that triggered false alerts) must be pursued as part of future work.

CRediT authorship contribution statement

Emre Balci: Conceptualization, Investigation, Methodology, Data curation, Visualization, Writing – original draft. **Timucin Aydede:** Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **Gorkem Yilmaz:** Conceptualization, Methodology, Software, Visualization. **Ece Gelal Soyak:** Conceptualization, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data is available at https://github.com/emrbalc/VASCOT/.

References

2017. A framework for bug hunting on the ethereum blockchain. URL https://github.com/ConsenSys/mythril.

Akca, S., Rajan, A., Peng, C., 2019. SolAnalyser: A framework for analysing and testing smart contracts. In: IEEE 26th Asia-Pacific Software Engineering Conference. APSEC, pp. 482–489.

Balcı, E., Yılmaz, G., Uzunoğlu, A., Soyak, E.G., 2023. Accelerating smart contract vulnerability scan using transformers. In: 2023 IEEE Asia-Pacific Conference on Computer Science and Data Engineering. CSDE, pp. 1–6.

Bistarelli, S., Mazzante, G., Micheletti, M., Mostarda, L., Tiezzi, F., 2020. Analysis of ethereum smart contracts and opcodes. In: Proc. Int.Conf. on Advanced Information Networking and Applications (AINA-2019) 33. Springer, pp. 546–558.

2022. BZx protocol exploit detailed analysis. https://www.immunebytes.com/blog/bzx-protocol-exploit-sep-14-2020-detailed-analysis/.

Chen, D., Feng, L., Fan, Y., Shang, S., Wei, Z., 2023. Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention. J. Syst. Softw. 202, 111705.

David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., Gervais, A., 2023. Do you still need a manual smart contract audit?. arXiv preprint arXiv:2306.12338.

2017. Ethereum virtual machine opcodes. URL https://www.ethervm.io/.

2024. Etherscan export CSV data - list of verified contract addresses with an open source license. URL https://etherscan.io/exportData?type=open-source-contract-codes.

Gogineni, A.K., Swayamjyoti, S., Sahoo, D., Sahu, K.K., Kishore, R., 2020. Multi-class classification of vulnerabilities in smart contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing. IOP SciNotes 1 (3), 035002.

2018. Google BigQuery. URL https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.

Gritz, A., 2022. MaianUpdated. URL https://github.com/alegritz/MAIANUpdated/tree/main/MAIAN-master.

Hwang, S.-J., Ho Ju, S., Choi, Y.-H., 2024. CGGNet: Compiler-guided generation network for smart contract data augmentation. IEEE Access.

2010. IEEE Standard Classification for Software Anomalies. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), pp. 1–23.

Kalra, S., Goel, S., Dhawan, M., Sharma, S., 2018. Zeus: analyzing safety of smart contracts. In: Ndss. pp. 1–12.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proc. ACM SIGSAC Conference on Computer and Communications Security. CCS, pp. 254–269.

Ma, W., Wu, D., Sun, Y., Wang, T., Liu, S., Zhang, J., Xue, Y., Liu, Y., 2024. Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications. arXiv preprint arXiv:2403.16073.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A., 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: IEEE/ACM Int. Conf. on Automated Software Engineering. ASE, pp. 1186–1189.

Muhs, D., 2020. Smart contract weakness classification (SWC) registry. URL https://github.com/SmartContractSecurity/SWC-registry.

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A., 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proc. Annual Computer Security Applications Conference. pp. 653–663.

Palladino, S., 2017. The parity wallet hack explained. URL https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

Pearson, T., 2023. Top 10 crypto hacks of 2023. URL https://www.dlnews.com/articles/defi/top-10-crypto-hacks-of-2023-ranked-as-stakecom-is-fifth/.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., Marchesi, M., 2019. A massive analysis of ethereum smart contracts empirical study and code metrics. IEEE Access 7, 78194–78213.

Qian, P., Liu, Z., He, Q., Zimmermann, R., Wang, X., 2020. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access 8, 19685–19695. http://dx.doi.org/10.1109/ACCESS.2020.2969429.

Ren, X., Wu, Y., Li, J., Hao, D., Alam, M., 2023. Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network. Comput. Electr. Eng. 109.

Samreen, N.F., Alalfi, M.H., 2020. Reentrancy vulnerability identification in ethereum smart contracts. In: IEEE International Workshop on Blockchain Oriented Software Engineering. IWBOSE, pp. 22–29.

Sendner, C., Chen, H., Fereidooni, H., Petzi, L., König, J., Stang, J., Dmitrienko, A., Sadeghi, A.-R., Koushanfar, F., 2023. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In: NDSS.

Sergey, I., Hobor, A., 2017. A concurrent perspective on smart contracts. In: Financial Cryptography and Data Security: FC 2017 International Workshops. Springer, pp. 478–493.

2025. Solidity documentation. URL https://docs.soliditylang.org.

Sun, X., Tu, L., Zhang, J., Cai, J., Li, B., Wang, Y., 2023. ASSBert: Active and semi-supervised bert for smart contract vulnerability detection. J. Inf. Secur. Appl. 73, 103423.

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y., 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13.

Tann, W.J.-W., Han, X.J., Gupta, S.S., Ong, Y.-S., 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. arXiv preprint arXiv:1811.06632.

Tikhomirov, S., et al., 2018. Smartcheck: Static analysis of ethereum smart contracts. In: IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. WETSEB, pp. 9–16.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M., 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.

Wang, W., Song, J., Xu, G., Li, Y., Wang, H., Su, C., 2021. ContractWard: Automated vulnerability detection models for ethereum smart contracts. IEEE Trans. Netw. Sci. Eng. 8 (2), 1133–1144.

Wei, Z., Sun, J., Zhang, Z., Zhang, X., Li, M., Hou, Z., 2024. Llm-smartaudit: Advanced smart contract vulnerability detection. arXiv preprint arXiv:2410.09381.

Wikipedia, 2024. The Free Encyclopedia. The DAO, [Online]. https://en.wikipedia.org/w/index.php?title=The_DAO&oldid=1240033956.

Wood, G., et al., 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap. 151 (2014), 1–32.

Wu, Y., Xie, X., Peng, C., Liu, D., Wu, H., Fan, M., Liu, T., Wang, H., 2024. Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using LLM and static analysis. In: Proc. IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 1019–1031.

Zhang, Y., Liu, D., 2022. Toward vulnerability detection for ethereum smart contracts using graph-matching network. Futur. Internet 14 (11).

Zhang, L., Wang, J., Wang, W., Jin, Z., Su, Y., Chen, H., 2022. Smart contract vulnerability detection combined with multi-objective detection. Comput. Netw. 217, 109289.

Zhang, P., Xiao, F., Luo, X., 2020. A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 139–150.

Zhang, P., Yu, Q., Xiao, Y., Dong, H., Luo, X., Wang, X., Zhang, M., 2023a. Bian: Smart contract source code obfuscation. IEEE Trans. Softw. Eng. 49 (9), 4456–4476.

Zhang, Z., Zhang, B., Xu, W., Lin, Z., 2023b. Demystifying exploitable bugs in smart contracts. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 615–627.

Zhao, Q., Huang, C., Dai, L., 2023. VULDEFF: Vulnerability detection method based on function fingerprints and code differences. Knowl.-Based Syst. 260, 110139.