Smart Contract Vulnerability Detection Using Deep Learning

Mark Kane [#1]

[#] *Dublin City University*
Dublin, Ireland
[1] mark.kane8@mail.dcu.ie

***Abstract*— Blockchain technology has revolutionized various industries by enabling decentralized and secure value transfer through smart contracts—self-executing programs that enforce contract terms when specific conditions are met. However, this integration of financial transactions and complex agreements has attracted financially motivated bad actors, leading to real-world attacks that underscore the critical need for smart contract security. Past incidents, such as the DAO and Parity wallet hacks, have resulted in significant financial losses and emphasized the intricate challenges posed by smart contract vulnerabilities. The detection of vulnerabilities in smart contracts is of paramount importance to ensure the security and reliability of decentralized applications built on blockchain platforms. In this paper, we explore the application of deep learning techniques for smart contract vulnerability detection. We begin by replicating ESCORT, a state-of-the-art multi-output Deep Neural Network (DNN) that processes smart contract bytecodes and leverages Gated Recurrent Units (GRUs) to extract features from opcode sequences. Building upon this baseline, we draw inspiration from research on more general software vulnerability detection to investigate the use of abstract syntax tree (AST) representations for vulnerability detection in Solidity smart contract source code. By transforming the ASTs into sequential data suitable for RNNs, we analyze whether semantic and syntactic information from the source code can enhance detection results. We find very similar performance by both approaches, demonstrating that the limitations in detection capabilities of deep learning models in this domain are more likely a result of DNN architecture design and dataset quality.**

I. Introduction

Blockchain technology is a decentralized and distributed digital ledger that records transactions on multiple computers, providing an immutable and transparent record of data. The original proof of concept for blockchain technology was the peer-to-peer cash network Bitcoin introduced in 2008, and over time many other cryptocurrencies and alternative applications of blockchain have evolved. One such blockchain is Ethereum, which differs to Bitcoin in a few ways. One key differentiator is that Ethereum supports smart contracts. Smart contracts are self-executing computer programs that automatically enforce the terms of a contract when specific conditions are met. They are built on top of blockchain technology, allowing for secure execution of complex agreements between two untrusting parties. Smart contracts can be used to automate a wide range of processes, from simple payment transactions to more complex supply chain management and legal agreements. The combination of blockchain and smart contracts has the potential to revolutionize many industries by increasing efficiency, reducing costs, and improving transparency and accountability.

Blockchain has fostered the exchange of value between two peers in a decentralized manner across the internet which was never possible before its existence. Any technology which allows for the transfer of value will attract financially motivated bad actors. Real world attacks have already demonstrated the need for smart contract security. For example, the DAO (Decentralized Autonomous Organization) attack in 2016 resulted in the loss of over $50 million worth of Ethereum due to a vulnerability in the smart contract code [5]. In this case, an attacker was able to exploit a security bug in the smart contract's code (called reentrancy) to drain funds from the DAO's account. Similarly, the Parity wallet hack in 2017 resulted in the loss of over $30 million worth of Ethereum due to a bug in the smart contract code. In this case, a vulnerability in the code allowed an attacker to take control of the Parity wallet and steal the funds held within it [5]. In 2021, 680 million US dollars' worth of digital assets managed by smart contracts were stolen/hacked through the exploitation of security vulnerabilities [6]. These real-world examples highlight the importance of smart contract security. The security problems caused by smart contracts are more complicated and arguably higher consequence than traditional software programs, and analysing smart contracts to detect these vulnerabilities can also be more difficult [2]. It is clear that failure to adequately secure smart contracts can lead to significant financial loss and damage to the reputation of the blockchain ecosystem as a whole.

With the proliferation in smart contract development over recent years, security has now become a prioritised concern by researchers and developers alike. However, up until the study carried out by [6] it was unclear how security was approached by developers when writing smart contracts and deploying them to the blockchain. The qualitative study carried out by [6] with smart contract developers of varied levels of experience finds a wide variety of perceptions and practices used by the participants with respect to the vulnerability detection tools and resources they used. 83% of participants did not even mention security as a priority to their smart contract development process. The study found that the more junior developers had a far lower success rate at identifying security vulnerabilities in a code review task (15%) as compared to their more experienced counterparts (55%). It was found that

many developers rely on external auditing of code to ensure the security of their smart contracts and when they do assess the security themselves, they find a lack of tools and resources to do so and do it manually. It is clear from that the tools and resources in this domain are not yet adequate to support developers in deploying secure smart contracts to the blockchain. Here lies the motivation for this study.

After a review of the literature around smart contract vulnerabilities and different techniques employed by the various vulnerability detection tools available, this work pursues a novel deep learning approach to vulnerability detection using natural language processing (NLP) on smart contract source code represented as abstract syntax trees (ASTs). The contributions made by this study are summarized below:

1. After careful review of the literature on smart contract vulnerability detection tools, this study appoints ESCORT [10], an Ethereum smart contract vulnerability detection tool using deep learning, as the current state-of-the-art deep learning solution to smart contract vulnerability detection. We replicate the design outlined by [10], training a model on a publicly available labelled dataset of Solidity bytecodes [9] and then evaluating its performance. Doing this allows us to obtain an objective evaluation of the methodology designed by [10] (as ESCORT was not made open source) as an initial contribution. This also provides us with a baseline approach that can act as a benchmark to compare further experimentation of smart contract vulnerability detection using deep learning.
2. After performing an objective evaluation of the efficacy of the ESCORT model as described by [10], and establishing a performance baseline, drawing inspiration from [1], this study then proposes a novel approach of transforming the modality of the smart contract source code to their abstract syntax tree (AST) representation, serializing the AST data into sequences before applying NLP techniques to the data. This experimental approach is then evaluated through comparison with the baseline approach.

II. Background

The below sections provide some background information on smart contracts, their vulnerabilities, tools to detect them, along with some background on deep learning.

A. Smart Contracts

Smart contracts are self-executing contracts with predefined terms and conditions encoded in code. They operate on blockchain platforms, enabling decentralized and trust less execution without the need for intermediaries. Among various blockchain platforms, Ethereum stands out as a prominent choice for deploying smart contracts. Ethereum is a decentralized blockchain platform that employs a consensus mechanism called Proof of Stake (PoS), ensuring security and immutability of the blockchain. To write smart contracts on the Ethereum platform, developers primarily use the programming language Solidity. Solidity is a high-level, statically typed language inspired by JavaScript, Python, and C++. It targets the Ethereum Virtual Machine (EVM), the runtime environment where smart contracts are executed. In the process of writing a smart contract, developers define the contract's functionality and logic using Solidity's syntax, incorporating functions, variables, and data structures to represent the terms and conditions of the agreement. State variables are utilized to hold data that persists across contract invocations.

Once the smart contract is written in Solidity, it needs to be compiled into bytecode, which can be understood by the EVM. This compilation process converts the high-level Solidity code into low-level EVM bytecode. Solidity compilers such as Solc (Solidity compiler) and Remix (web-based IDE and compiler) are commonly used for this purpose. Once deployed on the Ethereum blockchain, the smart contract becomes a part of the global network. Users can interact with the contract by sending transactions to its address. Ethereum nodes execute the contract code in the EVM, and the results are recorded on the blockchain. The execution of smart contracts is deterministic, meaning the same input will consistently produce the same output.

Despite the potential benefits of smart contracts, they are not without vulnerabilities. As already alluded to in the introduction, bugs in the code, logical flaws, and security oversights can lead to severe consequences, including the locking or theft of funds. Thus, effective vulnerability detection techniques are essential to ensuring the reliability and security of smart contracts. This study addresses this crucial challenge in the adoption and usage of smart contracts and contributes to enhancing the overall security and trustworthiness of decentralized applications built on the Ethereum blockchain.

B. Smart Contract Vulnerabilities

One of the earliest studies on smart contract vulnerabilities in Ethereum was carried out by [7] in 2017. [7] collects and analyses the vulnerabilities seen in smart contracts deployed to the Ethereum blockchain, giving solidity code examples of each and illustrating some attacks which exploit these vulnerabilities, many of which are inspired by real world attacks. [7] gives an original taxonomy of the vulnerabilities which can be exploited in smart contracts written for EVM-based blockchains, separating the vulnerabilities into three different levels: Solidity vulnerability, EVM vulnerability and Blockchain vulnerability. This original taxonomy was applied to 12 vulnerabilities which were found by [7], but as time has gone on more and more vulnerabilities have been discovered. Most subsequent studies on vulnerabilities in Ethereum smart contracts use this same taxonomy when classifying vulnerabilities.

C. Smart Contract Vulnerability Detection Methods

Smart contract vulnerability detection encompasses various approaches aimed at securing the decentralized and immutable nature of blockchain-based smart contracts. [2] classifies smart contract vulnerability tools into one of the following

five classes: formal verification, symbolic execution, fuzzing tests, intermediate representation and deep learning. *Formal verification* utilizes formal languages to transform the contract program into a smart contract model, enabling rigorous logic and proof to verify correctness and safety [2]. *Symbolic execution* reasons about the program symbolically rather than executing the code, generating test cases for various contract execution paths [2]. *Fuzzing tests* involve feeding a large number of test inputs into the program to detect abnormal behaviour that may indicate security vulnerabilities [2]. *Intermediate representation* converts smart contract source code into a semantically rich form for vulnerability analysis [2].

Deep learning has emerged as a promising method for smart contract vulnerability detection, leveraging artificial neural networks and large datasets to make decisions based on input data [2]. ESCORT, a state-of-the-art Deep Neural Network (DNN)-based framework, introduces transfer learning for new unseen vulnerabilities, allowing extensibility and generalizability [10]. The model includes a feature extractor shared among vulnerability branches, learning fundamental features from smart contract bytecode, and an RNN to process the data's temporal sequence [10]. ESCORT achieves strong performance, detecting six pre-trained vulnerabilities with an average f1-score of 95% and an impressive detection time of 0.02 seconds per contract [10]. Its capability for transfer learning yields an average f1-score of 93% for new vulnerability types, with reduced training time compared to training from scratch [10]. Despite its promising results, ESCORT's lack of publicly available source code limits formal evaluation [10].

Deep learning as a vulnerability detection approach offers advantages in its ability to handle complex and evolving smart contract security threats. Its flexibility and potential for generalization to new vulnerabilities make it a promising area of research for enhancing the security and reliability of blockchain-based smart contracts [2]. The combination of advanced deep learning techniques and the unique challenges posed by smart contract security present a rich and rewarding avenue for further exploration in this critical field of study.

D. Deep learning

In this study we operate on either the bytecode or serialized abstract syntax tree (AST) representation of smart contracts. Both can be considered a special case of text data. Here we introduce some concepts of deep learning and natural language processing that are relevant to this study:

- *Text Representation:* In machine learning algorithms, text data is transformed into numerical vectors for processing. This conversion can be achieved through various methods, such as a bag of words, n-gram language model, or embedding layer. These numerical vectors are then directly used as input for deep learning (DL) models [10].
- *Recurrent Neural Network (RNN):* RNN is a type of deep neural network in which the connections between nodes (neurons) form a direct computational graph along the sequential input data, such as text documents [10]. One important characteristic of RNN is its ability to use internal states as memory cells to retain information from prior inputs, capturing the contextual information in the sequential data [10]. The hidden states from previous inputs influence the output in the current time step. RNNs utilize a "parameter sharing" mechanism, where weight matrices are shared across different time steps, allowing them to handle sequences of varying lengths [10].
- *Multi-label vs. Multi-class classification:* Smart contract vulnerability detection can be approached through two classification paradigms. In multi-class classification, the task involves more than two mutually exclusive classes, and each sample is assigned only one class label. On the other hand, multi-label classification also deals with multiple classes, but each data sample can have more than one associated label. In multi-label tasks, the classes describe non-exclusive attributes of the input, such as colour and length, allowing for a more flexible and nuanced classification [10].

III. BASELINE APPROACH

The first step of this practicum is to replicate the model design of ESCORT as described in [10] to establish a baseline approach to smart contract vulnerability detection using deep learning. After reviewing the literature, ESCORT [10] is considered to be the strongest deep learning-based vulnerability detector to base this study off for a few reasons:

1) It supports the detection of multiple vulnerabilities – most deep learning-based solutions are single-class classifiers trained to detect one vulnerability type. However, we know that the list of potential smart contract vulnerabilities is long and that new vulnerabilities can arise over time. Therefore, a detector should be equipped to detect multiple vulnerability types to be useful for developers.
2) The architecture design of ESCORT supports transfer learning, meaning that a pre-trained model can quickly adapt to a new vulnerability type by extending the model with a branch of layers to be trained to detect the new vulnerability exclusively. Not having to train a model from scratch allows for faster adaptation of new vulnerabilities.
3) The test accuracy and f1-scores achieved by the model are purported by [10] to be very high across all vulnerability types the model was trained to detect.

ESCORT [10] designs an extensible DNN detector that outputs a probability that a smart contract contains certain vulnerabilities and is capable of classifying multiple vulnerability types with a single model. The key innovation of ESCORT's design is the decomposition of the vulnerability detection process into two subtasks: (1) Learning the general smart contract bytecode features agnostic to attack type; (2) Learning to detect each particular vulnerability class [10]. The ESCORT architecture is hence divided into two tiers to achieve these subtasks as illustrated by figure 1. These tiers are described below:

1) *Feature Extractor:* A feature extractor is a stack of layers that learn general fundamental features in the input data which are useful across different attributes. In this context, the feature extractor is trained to learn semantic and syntactic information in the smart contract's bytecode [10]. The output of the feature extractor acts as the input to each of the individual vulnerability branches, and so can be thought of as the 'stem' that is shared by all branches. In ESCORT's design, the feature extractor consists of an input layer, an embedding layer and a GRU/LSTM layer:

- *Embedding Layer:* An embedding layer is a fundamental component in NLP and machine learning, used to transform categorical data or discrete elements into continuous vector representations, often referred to as embeddings. In the context of smart contract bytecode data, an embedding layer can be employed to convert the low-level EVM bytecode, represented as a sequence of hexadecimal numbers, into a dense vector representation with continuous values. This embedding process enables the utilization of machine learning and deep learning techniques to analyse and understand the patterns and features present in the bytecode data. By representing the bytecode as embeddings, complex relationships and similarities among different parts of the bytecode can be captured, facilitating the development of powerful models for tasks like smart contract vulnerability detection. The embedding layer acts as a bridge between the raw, discrete bytecode data and the mathematical operations of machine learning algorithms, thereby enhancing the effectiveness and accuracy of analysing smart contract bytecode.
- *GRU/LSTM Layer:* Recurrent Neural Networks (RNNs), including specific variants like Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) layers, are powerful architectures designed to process sequential data with temporal dependencies. In the context of analysing smart contract bytecode data, RNNs are well-suited for capturing the contextual information and relationships among the opcodes and instructions present in the bytecode. By taking the output of the embedding layer as input, RNNs can effectively leverage the continuous vector representations (embeddings) of the discrete bytecode elements. The embeddings provide a dense and continuous representation of the bytecode, allowing the RNNs to learn meaningful patterns, dependencies, and long-term dependencies between different opcodes or instructions. This enables the RNNs to capture the sequential nature of the bytecode data, recognize patterns in the execution flow, and learn from the historical context of opcode sequences. By employing RNNs with GRU or LSTM layers, we can build a sophisticated model capable of analysing and understanding the intricacies of smart contract bytecode.

2) *Vulnerability Branches:* The second component of ESCORT's multi-output DNN architecture involves the ensembling of multiple vulnerability branches [10]. Each branch consists of a stack of layers specifically trained to identify patterns and hidden representations associated with different vulnerability classes. Although the branches have no direct interdependence, they all share the same feature extractor, meaning they receive the same input. This design is feasible because the input, which also serves as the feature extractor's output, captures the semantic information present in the contract's bytecode. This shared input contains common and general data that is relevant for detecting various vulnerabilities [10]. In each vulnerability branch, the last layer is a Dense layer with just one neuron. The activation value of this neuron undergoes sigmoid evaluation, resulting in the probability that the input contract contains the specific vulnerability. Consequently, ESCORT generates detection results with enhanced interpretability by providing confidence scores for its diagnoses, rather than merely offering binary decisions regarding vulnerability presence. This approach allows for a more informative and nuanced assessment of the contract's security status [10].

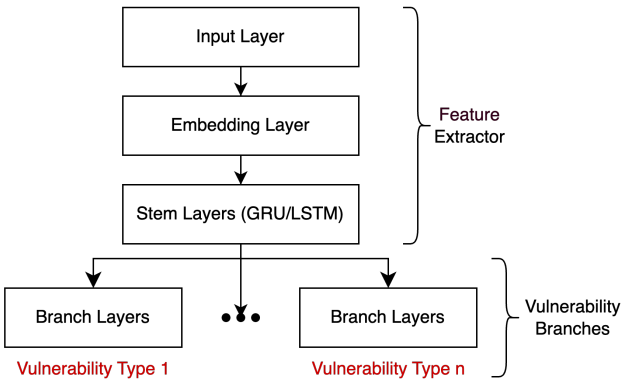

Figure 1: General architecture of baseline approach based on ESCORT [10] model.

IV. DATASET

Before being able to move forward, a suitable labelled dataset of smart contract EVM bytecode must be acquired. Because this area of research is still in its infancy, open-source, labelled Solidity smart contracts are scarce. The two most commonly known datasets are Smartbugs-Wild [4] and ScrawlD [8], but these datasets are smaller than those typically needed for training DNNs. In the last year, however, a labelled dataset composed of more than 100,000 smart contracts labelled using the Slither static analyzer [3] was published on the HuggingFace hub [9]. This dataset provides the address, source code, and bytecode for real-world Solidity smart contracts that have been verified on Etherscan.io, along with a classification of their vulnerabilities obtained from the Slither static analysis framework as a list of labels [9]. This dataset is most suitable for this study as it is sufficiently large for the training of a DNN, and it contains both the bytecode and Solidity source code for each smart contract which is necessary to support a comparative analysis of the baseline approach using bytecodes and experimentation using Solidity source code pursued by this study [9]. During construction of the dataset, each smart contract was classified by the Slither

tool which passes the smart contract through 38 rule-based detectors and returns a JSON file detailing where those detectors found a vulnerability. Any vulnerability detected are then mapped to one of the following five vulnerability classes:

1) *Access Control:* Listed number 5 on the OWASP [12] top 10, access control issues are a security concern across all programs, not just smart contracts [c]. Typically, a contract's functionality is accessed via its public or external functions. However, vulnerabilities can arise from insecure visibility settings that provide attackers with direct access to a contract's private values or logic. Besides these obvious weaknesses, access control bypasses can also be more subtle. These vulnerabilities may emerge when contracts rely on outdated practices like using the deprecated tx.origin to authenticate callers, handle extensive authorization logic with lengthy require statements and make reckless use of delegatecall in proxy libraries or proxy contracts [c].

2) *Arithmetic:* Also known as integer overflow and integer underflow, this type of vulnerability is not a new class unique to smart contracts. It is, however, particularly dangerous in smart contracts where the use of unsigned integers is more prevalent, and most developers are more used to using simple signed integer types. Seemingly benign code paths can become vectors for theft or denial of service attacks [c].

3) *Reentrancy:* Probably the most famous Ethereum vulnerability as it was first unveiled in the 2016 multi-million-dollar DAO heist which led to a hard fork of Ethereum, this is a recursive call vulnerability where an attacker can make multiple calls to a contracts function before the initial call is terminated. If the internal contract state is not securely updated, the attacker can drain Ether from the contract through recursive calls of the function [10].

4) *Unchecked-calls:* Solidity supports low level functions such as call(), callcode(), delegatecall() and send(). These functions handle errors differently to other typical Solidity functions as they will not propagate and lead to a full reversion of the contract execution. These functions will instead simply return a Boolean value set to false, and the code will continue executing [c]. If the return value of such calls are not asserted, this can lead to unexpected and unwanted outcomes [c].

5) *Others:* The class of vulnerability groups together all other relevant Slither detectors that were not included in the previous four class, for example, 'incorrect-equality', which detects if a contract enforces strict equalities to determine if an account has enough Ether/tokens.

Figure 2 provides an illustration of the distribution of these vulnerability types in the dataset.

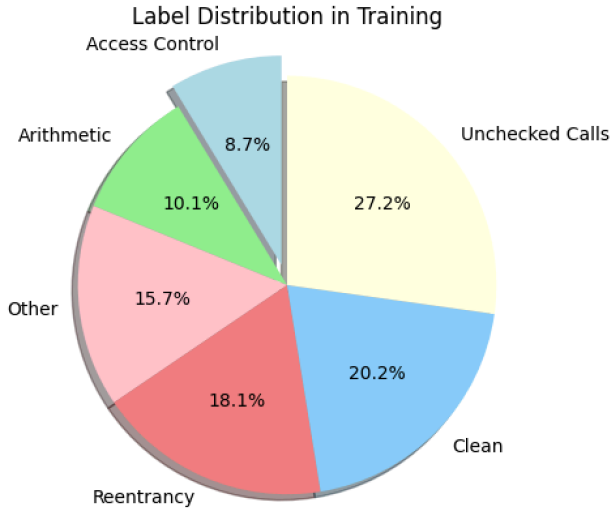

Figure 2: Vulnerability distribution of dataset

V. BASELINE IMPLEMENTATION

A. Data Pre-Processing

The ESCORT model is designed to extract features from the smart contract bytecode, which is essentially a long string of hexadecimal numbers. This data is not suitable for a DNN to process as input, so the data must be transformed into a DNN-friendly representation of the bytecode. The ESCORT architecture is designed to use NLP techniques to extract features from the input data. The smart contract bytecode consists of a series of double-digit hexadecimal numbers, each representing a particular opcode which gets executed by the EVM. The bytecode pre-processing step therefore transforms each raw bytecode into its corresponding sequence of opcodes, each divided by a whitespace. These mappings were retrieved directly from the Ethereum documentation [9]. Furthermore, any operations with the same functionality are merged into one opcode. For example, the *PUSH1-PUSH32* commands represented by bytes *0x60-0x7f* are all mapped to the operation *PUSH*. This aids in keeping the corpus size of the pre-processed bytecodes smaller. Any hexadecimal digits which did not map to any opcode were considered invalid and replaced with the word "XX".

Following this transformation, we now have each smart contract represented as a sequence of opcodes. This, however, is still string data which cannot be fed to the input layer of a DNN. Before being passed to the input layer, this sequence data needs to be vectorized by a *tokenizer* to transform the sequences into numeric vectors. It is these numeric vectors which get passed to the input layer of the model. Each smart contract bytecode, and hence its pre-processed opcode sequence, can be a unique length. Therefore, the numeric vectors will also vary in length and so need to be either padded or truncated to a fixed length defined by a hyperparameter *MAX_SEQUENCE_LENGTH (MSL)*. Sequences shorter than this are post-padded with zeros and sequences longer are post-truncated to fit this length. We did this tokenization as part of the pre-processing step ahead of

model training, storing the tokenized vectors in a HDF5 file. The corpus size of the bytecode data after tokenization is 82 (i.e., the number of unique opcodes present in the dataset).

We also studied the distribution of opcode sequence lengths in the dataset before deciding on a value for the *MSL* hyperparameter. [10] cites that in their dataset setting this length value to 4100 ensures that 98.5% of the contracts in their dataset are not truncated. The same was not found for the dataset used by this study, where the *MSL* must be set to 16,750 to ensure that 90% of the bytecodes are not truncated. This is illustrated by Figure 3, which shows the distribution of smart contract sequence lengths in the training set (similar distribution found in validation and test sets). This is a notable discrepancy which is noted in the results section when comparing this study's test results to those recorded by [10].

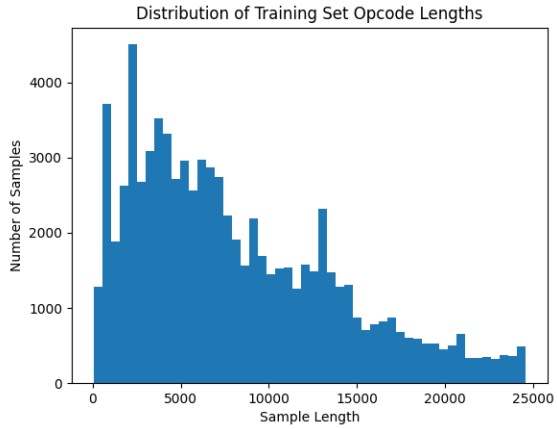

Figure 3: Distribution of opcode sequence lengths in dataset

B. DNN Implementation

The ESCORT DNN model was built, trained, and evaluated using the *tensorflow.keras* [e] package. As already mentioned, each smart contract in the dataset is labelled as being either safe or containing vulnerabilities from five different classes of vulnerability. A smart contract can also contain more than one vulnerability. Therefore, this is a multi-label, multi-class classification task. To realise our baseline model based on the multi output DNN architecture of ESCORT [10], we instantiate the Multi-Output-Layer (MOL) DNN model illustrated in Figure 4 with the hyperparameters listed in Table 1.

Because we are processing sequential data, a recurrent neural network is constructed using stacked embedding, GRU, dropout and dense layers to process sequential bytecode inputs. The feature extractor made up of the embedding and GRU layers learns generic features about the sequences before being concatenated by multiple branches consisting of stacked dense and dropout layers with the output of each branch computing the binary cross-entropy loss for exclusively one vulnerability. This enables us to construct a single model comprised of multiple detectors trained to detect exclusively one vulnerability class in smart contract bytecode. This is also what makes this model extensible: we can load the pre-trained feature extractor, append a new branch for a new vulnerability type and train the branch layers to detect the new vulnerability type. The output of each branch is fed through a sigmoid activation function to output a probability of the smart contract containing the given vulnerability. This is superior to a binary output as it provides a confidence score about the presence of a vulnerability in the code.

TABLE I
HYPERPARAMETERS

Parameter	**Value**
MSL	16,750
Embedding Layer Dimensions	Input_dim=82, Output_dim=5, Input_length=MSL
Hidden Units	GRU: 64, Dense: [128, 64, 1]
Optimizer	Adam
Loss Function	Binary Cross-Entropy
Learning Rate	0.001
Activation Function	Sigmoid
Dropout	0.2
Batch Size	32
# Local Epochs	1
#Global Epochs	1
Chunk Size	1024

C. Data Chunking

The size of the dataset adds a layer of complexity to the training process. Loading all the training data into memory at once can exceed the memory constraints of the resources available. In Section IV it is mentioned that for this implementation done by this study, the pre-processed vector representations are stored in a HDF5 file. This storage format is used to enable efficient data chunking during the training process. Data chunking is simply the process of loading chunks of training data into memory to train the model on. The chunk size used in this implementation is 1024 samples of data. Storing the training set in a HDF5 file in persistent storage allows us to create a reference to this file in memory and then iteratively retrieve chunks of 1024 rows at a time, so as to not overwhelm the memory resources and get an out-of-memory (OOM) error. We also ensure that each data chunk carries a relatively even distribution of vulnerability classes for optimal training.

Defined in Table 1 are two hyperparameters: local epochs and global epochs. Each data chunk can be thought of as a mini training set which we train the model with. 'Local epochs' refers to the number of times each data chunk is processed by the model. 'Global epochs' refers to the number of times the entire training set (all chunks) is processed by the model. During the training process, the model is not evaluated on the validation set until after each global epoch.

The hyperparameters listed in Table 1, aside from *MSL* as already mentioned, are the same as those listed by [10]. Any

experimentation related to the tuning of these parameters carried out by this study are documented in section VII.

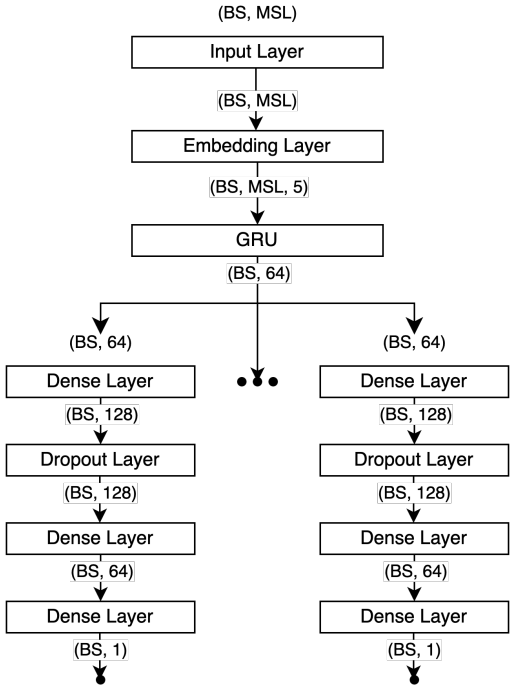

Figure 4: Baseline ESCORT architecture

VI. EXPERIMENTATION WITH ABSTRACT SYNTAX TREES

After successfully implementing a baseline approach to using DNNs for vulnerability detection in bytecode data, this study then investigates if using similar NLP techniques on an alternative modality of the smart contract data can yield stronger results. One critique of using the compiled smart contract bytecode to detect vulnerabilities is the loss in semantic and syntactic information present in the Solidity source code. In light of this, we draw inspiration from [e], an DNN approach to vulnerability detection in C++ code using abstract syntax tree (AST) representations of the source code. ASTs are hierarchical data structures used to represent the syntactic structure of a program or code snippet in a way that is easily accessible and analyzable to computer programs. ASTs capture the structure and arrangement of tokens in code like keywords, operators, variables, functions etc. in a hierarchical tree-like structure. Each node in the tree represents a specific construct of the programming language such as an expression, statement or function definition, and the nested and hierarchical nature of the code is reflected in the relationships between nodes. The abstraction of certain unnecessary details in actual source code like parentheses allows more focus on the programs essential structure. ASTs are useful for the analysis of source code as they conform all programs into a consistent, formalized representation. This quality is capitalized on by [e] to train a recurrent DNN to detect vulnerabilities in C++ code, with promising results. We adopt this same approach in the blockchain domain for Solidity source code and document the results in Section VII.

A. Data Pre-Processing Design

The primary difference between this approach and the baseline approach is the representation of the smart contract. Section V discusses the pre-processing needed to transform the raw EVM bytecode into a numerical vector representation that can be fed to a DNN. The same transformation needs to be made in this approach to the Solidity source code before it can be passed as input to an embedding layer. As illustrated in Figure 5, this transformation requires three steps.

1) *Code Parsing:* First, the smart contract source code must be parsed into an AST. As all the implementation of this study is carried out using Python, we use the Python Solidity parser, an open-source Solidity parser for Python built on top of a robust ANTLR4 grammer. When parsed, each node of an AST contains a 'type' indicating the type of node, e.g., 'ContractDefinition', 'FunctionDefinition', 'StateVariableDefinition', etc. Each node is described through a series of key-value pairs which appear in consistent order. Nodes are nested in values of other nodes to represent the hierarchical nature of source code.

2) *AST Standardization:* The AST representation of source code can become quite long and convoluted to the casual observer. It must be decided what information provided by the AST, if not all, is suitable to an NLP task. Processing the entire AST has the potential of obscuring the relevant patterns leading to less effective feature extraction. Extracting too few of the nodes may cause loss of information pertinent to vulnerability detection. After some deliberation around where most Solidity smart contract vulnerabilities reside, with particular attention to the vulnerability classes present in the dataset used by this study, we decide that the function level is the optimal level to capture vulnerability patterns without causing a reduction pattern recognition performance.

In addition to this, a decision also must be made about how to handle user-defined names for variables, functions etc. The approach taken by [e] is to replace user-defined names with numbered-fixed names like 'var0', 'var1', 'func0', 'func1' in an attempt to map functions with the same structure to the same patterns. The problem with this approach is that different smart contracts have different numbers of variables and functions, and so this does not seem like an optimal abstraction of user-defined names because the DNN will be no more capable of recognizing patterns between 'func0' and 'func1' as it will between two functions with user-defined names. Therefore, this study takes a slightly different approach to handling user-defined names, replacing all with the same fixed string 'XX'. This is affordable as the AST nodes generated by the Python parser are already rich in information. This way, all nodes with the same 'type' will map to the same patterns.

We also treat string literals and number literals found in the AST in a similar way. This is done to keep the corpus size as small as possible when tokenization is done later in the process. Remember that the embedding layer contains an embedding for each word in the corpus. Therefore, without this step, the resultant corpus size, and hence the input

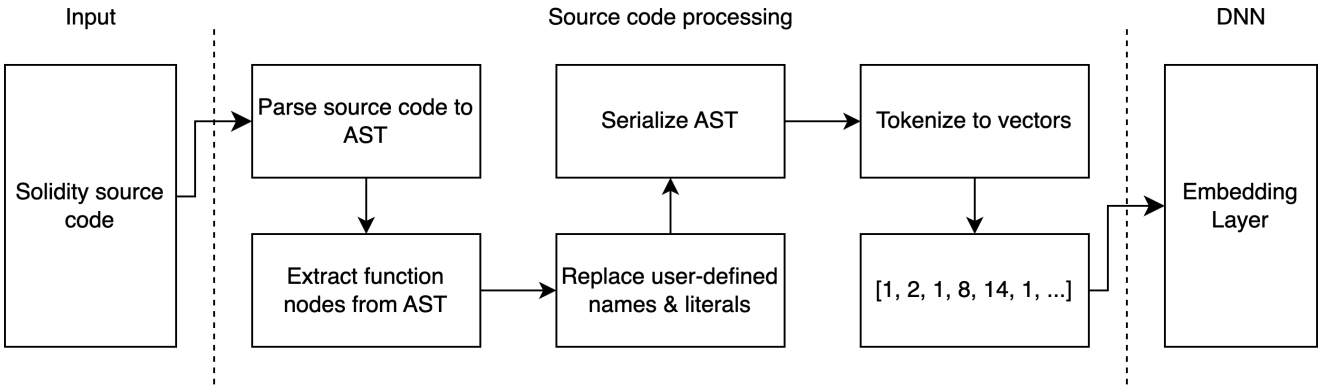

Figure 5: Data pre-processing for experimental approach

dimension of the embedding layer, will be extraneously large due to all the different string literals and number literals defined across ~100,000 smart contracts.

After this step, the functionality of each function in each smart contract should be represented in a consistent, standardized format without the variability of user-defined variables and string/number literals.

3) *AST Serialization:* After obtaining standardized ASTs at the function level, we cannot pass this tree-like structured data as input to a DNN. The ASTs must be transformed into a sequence structure and then converted to vectors before being passed to a DNN. To serialize the AST into a sequence structure, we must traverse the tree to collect all elements. In this implementation, we perform a depth-first search (DFS) tree traversal to serialize the AST whist preserving the order of elements of the AST. As already mentioned, the nodes in the ASTs generated by the Solidity parser consist of key-value pairs describing all details of the function. We serialize the AST into a sequence of key-value pairs by performing a DFS tree traversal.

B. Data Pre-Processing Implementation

Here, we have divided the data pre-processing steps into three separate steps. In the actual implementation, steps 2 and 3 are achieved in unison, modifying any user-defined names or literals as we traverse through the AST serializing the key-value pairs. After serialization of the ASTs, we tokenize the sequences into padded vectors as was done in the baseline approach. These vectors are then passed to the DNN for model training.

The computational complexity of the data pre-processing tasks for this approach is far greater than what was needed for the baseline approach. Substantial time is spent parsing the source code to ASTs. To pre-process a dataset of ~80,000 smart contracts sequentially, it can take 40-50 hours. To reduce this time constraint, we open the maximum number of sessions allowed in Google Colab (5) and pre-process chunks of the dataset in parallel. This reduces the data pre-processing time by a factor of 5 to about 8 hours. It should also be noted that the Python Solidity parser is an experimental open-source project, and so there were a few smart contracts which threw exceptions when being parsed. The size of the training set reduced from ~78,000 to ~72,000 after data pre-processing because of this.

C. DNN Implementation

To be able to evaluate the effectiveness of using ASTs for vulnerability detection in deep learning as compared to smart contract bytecode data, we use the same MOL-DNN architecture as was used in for the baseline approach. This allows us to isolate the evaluation of both approaches to the modality of the smart contract used. We set the *MSL* hyperparameter to 21,500 to ensure that 95% of smart contract sequences do not get truncated.

VII. RESULTS AND DISCUSSION

In sections III to V, we detail the design and implementation of a baseline approach to vulnerability detection using DNNs using NLP techniques on smart contract bytecode data. In section VI we then detail the design and implementation of an experimental approach which performs similar NLP techniques to parsed abstract syntax tree representation of Solidity smart contract source code. The goal of this experiment is to investigate whether the semantic and syntactic information in smart contract source code, which gets lost when compiled to bytecode, provides a DNN with more information for vulnerability detection than that which is present in bytecode data. Both approaches, however, provide concurrent detection of multiple vulnerability classes.

A. Evaluation Metrics

We evaluate the performance of both the baseline and experimental approach with prediction accuracy, F1-score, precision and recall. To calculate the latter three metrics, the following base values must be used: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). The true values denote the number of correctly predicted results, i.e., either a correct positive prediction or a correct negative prediction. The false values represent incorrect predictions output by the model.

- *Precision & Recall:* Precision gives the ratio of correct positive predictions to all positive predictions. This indicates the reliability of the model's positive predictions. Recall shows the proportion of true

positives which are classified correctly. These two metrics have the following formulas:

$$Precision = \frac{TP}{TP + FP}, Recall = \frac{TP}{TP + FN}$$

- *F1 Score:* This metric uses the precision and recall to quantify the overall decision accuracy. Also referred to as the harmonic mean between precision and recall, the F1 score is calculated as follows:

$$F1_{score} = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

The performance for each vulnerability class is evaluated separately.

B. Evaluation Results

For both the baseline and experimental approaches, the model performance for each vulnerability class is given in table 2. The classes are numbered as follows:

- *Class 1:* Access Control
- *Class 2:* Arithmetic
- *Class 3:* Other
- *Class 4:* Reentrancy
- *Class 5:* Unchecked calls

As can be seen in table 2, the results of the baseline approach and experimental approach are very similar across all vulnerability classes. The models from both approaches have good detection accuracy for access control and arithmetic vulnerabilities. The detection capabilities are slightly reduced for the class of "other" vulnerabilities (described in section IV) and the reentrancy vulnerability. Detection is quite weak for the unchecked calls vulnerability.

We also compare these results to those documented by ESCORT [10], the study we replicate to establish a baseline approach. [10] cites F1 scores of 0.9 or above for all vulnerability classes in their dataset. There are a few things which can account for the slight discrepancy in performance between ESCORT [10] and the baseline implemented by this study.

1) Firstly, [10] curate their own dataset of Solidity smart contracts to test their model on. This comes with the advantage of taking control of the labelling process. [10] use several different static and dynamic analyzers to label their dataset, decreasing the probability of miss labelling smart contracts. In this study, we use an open-source dataset [9]. Using open-source data has the advantage of allowing other studies to verify the results achieved here. Lack of standardized data for studies to evaluate new deep learning methods to smart contract vulnerability detection stunts progress in this area as there is no benchmark for researchers to compare methods against. Hence why we use an open-source dataset in this study. The dataset used [9], however, is labelled by only one tool: Slither [3]. Although quite a strong analysis tool, the models trained in this study are limited by the Slither tool's ability accurately detect vulnerabilities with minimal false positive and true negatives.
2) Another difference between this study's baseline approach and ESCORT [10] is the fact that the vulnerabilities present are different within the different datasets used. The dataset used in ESCORT contains eight vulnerabilities of which only one is shared with the dataset used by this study [9]. Therefore, we cannot directly compare performances of the model for different vulnerability types.
3) As mentioned in section 5.A, and illustrated in figure 3, the opcode sequence lengths of the smart contracts in the dataset used by this study are on average much longer than that of the dataset used by [10]. [10] scrape and label 1.2 million real world smart contracts and then from that bank of smart contracts extract a dataset of similar size to the one used in this study: ~100,000. The final dataset used by [10] contains only smart contracts with relatively small bytecode lengths. We know that NLP techniques are usually used on sentences which usually contain 10-30 words. Therefore, the dataset used by [10] is more tailored to the NLP based DNN of ESCORT [10] than the open-source dataset used by this study [9]. This is likely another reason for the discrepancy between the results of this study and [10].

TABLE 2
PERFORMANCE

Model	Metrics	Vuln Class				
		cl. 1	cl. 2	cl. 3	cl. 4	cl. 5
Baseline	Loss	0.43	0.46	0.59	0.67	0.74
	Accuracy	0.85	0.83	0.74	0.7	0.54
	Precision	0.85	0.83	0.75	0.72	0.58
	Recall	1	1	0.98	0.97	0.96
	F1 score	0.92	0.91	0.85	0.83	0.72
Experiment	Loss	0.44	0.48	0.65	0.69	0.81
	Accuracy	0.85	0.83	0.75	0.7	0.58
	Precision	0.85	0.83	0.75	0.7	0.57
	Recall	1	0.98	0.98	0.98	0.97
	F1 score	0.91	0.85	0.85	0.82	0.72

It must also be noted that during implementation of both the baseline and experimental approaches, some time was spent tuning hyperparameters and experimenting with variations of the ESCORT architecture. It was found that any modifications made generally did not produce notable improvements in performance. The results documented here are based on the design and implementations outlined in this paper using the hyperparameters listed in table 1.

Comparing the performance of both approaches, it is clear that the experimental approach using serialized AST representations of the smart contract source code does not yield superior detection capabilities than the baseline approach. It appears that the performance results are more limited to the design decision around what variation of DNN to use and the dataset used to train with.

VIII. Conclusion

In this study, we investigate smart contract vulnerability detection using deep learning. After review of the literature, we attempt to replicate the state-of-the-art deep learning

solution to smart contract vulnerability detection, ESCORT [10]. ESCORT [10], a multi-output-layer DNN, processes smart contract bytecodes converted to their corresponding opcode sequences and extracts features from these opcode sequences by creating word embeddings for each opcode and using a Gated Recurrent Unit (GRU, an RNN more suitable for long sequences than other prevalent RNN variants).

After establishing this baseline implementation, we adopt the vulnerability detection method based on abstract syntax tree representations of source code implemented by [1] (for vulnerability detection in C++ source code) to investigate if the semantic and syntactic information present in source code can help yield stronger detection results when processed by a DNN. Serializing the AST to be used as sequence data for an RNN as done by [1] has not, to the author's knowledge, been investigated for Solidity smart contract vulnerability detection. This study carries out this investigation using an open-source dataset [9] containing five different vulnerability classes.

This study uses prediction accuracy, precision, recall and F1 score to evaluate the performance of both the baseline and experimental approaches. We find that both approaches produce very similar performance across all metrics for each vulnerability class. Therefore, we conclude that any limitations in the detection capabilities of a model which uses NLP techniques on the smart contract data are not a result of lost information (e.g., semantic and syntactic information in source code) in how the smart contract is presented to the model. Any limitations observed in detection capabilities are more likely to be a result of limitations of the DNN model used and/or limitations of the smart contract data used for training.

Regarding which of the two approaches tested by this study are optimal, we conclude that the added computational and time cost of parsing source code to an AST representation does not get outweighed by a substantial increase in performance.

Regarding this field of study as a whole, we conclude that the extensibility of DNNs for concurrent detection of vulnerabilities has potential but needs a more standardized method of testing for further progress to me made. There would be great benefit in curating a smart contract dataset large enough to be suitable for deep learning that has been labelled by multiple tools but has also been reviewed for vulnerabilities by experienced Solidity developers. This would allow all methods of smart contract vulnerability detection to be ranked against each other.

ACKNOWLEDGMENT

Many thanks to Dr Geoffrey Hamilton for his support and guidance on this research, it is greatly appreciated.

REFERENCES

[1] H. Feng, X. Fu, H. Sun, H. Wang and Y. Zhang, "Efficient Vulnerability Detection based on abstract syntax tree and Deep Learning," IEEE INFOCOM 2020 - *IEEE Conference on Computer Communications Workshops* (INFOCOM WKSHPS), Toronto, ON, Canada, 2020, pp. 722-727, doi: 10.1109/INFOCOMWKSHPS50562.2020.9163061.

[2] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian and X. Wang, "Smart Contract Vulnerability Detection Technique: A Survey," *Journal of Software*, vol. 33, no. 8, pp. 3059-3085, 2022.

[3] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts", Proceedings - 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB 2019, bll 8–15,2019.doi:10.1109/WETSEB.2019.00008

[4] SolidiFI Benchmark, March. 2021, [online] Available:https://github.com/smartbugs/SolidiFIbenchmark

[5] Dingman, Wesley & Cohen, Aviel & Ferrara, Nick & Lynch, Adam & Jasinski, Patrick & Black, Paul & Deng, Lin. (2019). Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework. *International Journal of Networked and Distributed Computing.* 7. 10.2991/ijndc.k.190710.003.

[6] Sharma, Tanusree & Zhou, Zhixuan & Miller, Andrew & Wang, Yang. (2022). Exploring Security Practices of Smart Contract Developers.

[7] Atzei, Nicola & Bartoletti, Massimo & Cimoli, Tiziana. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). 164-186. 10.1007/978-3-662-54455-6_8.

[8] S. Y. Chavhan, S. Kumar, and A. Karkare, "ScrawlD: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities," arXiv preprint arXiv:2202.11409, 2022.

[9] M. Rossini, "Slither Audited Smart Contracts Dataset," 2022.

[10] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "Escort: Ethereum Smart Contracts Vulnerability Detection Using Deep Neural Network and Transfer Learning," arXiv preprint arXiv:2103.12607, 2021.