Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning

Christoph Sendner*, Huili Chen‡, Hossein Fereidooni†, Lukas Petzi*, Jan König*, Jasper Stang*
Alexandra Dmitrienko*, Ahmad-Reza Sadeghi†, and Farinaz Koushanfar‡
*University of Würzburg, Germany
†Technical University of Darmstadt, Germany
‡University of California San Diego, USA

***Abstract*—Ethereum smart contracts are automated decentralized applications on the blockchain that describe the terms of the agreement between buyers and sellers, reducing the need for trusted intermediaries and arbitration. However, the deployment of smart contracts introduces new attack vectors into the cryptocurrency systems. In particular, programming flaws in smart contracts have been already exploited to lead to enormous financial loss. Hence, it is crucial to detect various vulnerability types in contracts effectively and efficiently. Existing vulnerability detection methods are limited in scope as they typically focus on one or a very limited set of vulnerabilities. Also, extending them to new vulnerability types requires costly re-design.**

In this work, we develop ESCORT, a deep learning-based vulnerability detection method that uses a common feature extractor to learn generic bytecode semantics of smart contracts and separate branches to learn the features of each vulnerability type. As a multi-label classifier, ESCORT can detect multiple vulnerabilities of the contract at once. Compared to prior detection methods, ESCORT can be easily extended to new vulnerability types with limited data via transfer learning. When a new vulnerability type emerges, ESCORT adds a new branch to the trained feature extractor and trains it with limited data. We evaluated ESCORT on a dataset of 3.61 million smart contracts and demonstrate that it achieves an average F1 score of $98\,\%$ on six vulnerability types in initial training and yields an average F1 score of $96\,\%$ in transfer learning phase on five additional vulnerability types. To the best of our knowledge, ESCORT is the first deep learning-based framework that utilizes transfer learning on new vulnerability types with minimal model modification and re-training overhead. Compared with existing non-ML tools, ESCORT can be applied to contracts of arbitrary complexity and ensures 100% contract coverage. In addition, we enable concurrent detection of multiple vulnerability types using a single unified framework, thus avoiding the efforts of setting up multiple tools and greatly reducing the detection time. We will open source our dataset and the data labeling toolchain to facilitate future research.

I. Introduction

The success of Bitcoin [85] fueled the interest in cryptocurrency platforms. As a result, next-generation blockchain-powered application platforms emerged, such as Ethereum [5] and Hyperledger [10]. These platforms provide smart contracts that are automated decentralized applications describing the terms of agreements and the transaction rules between the buyers and the sellers. Smart contracts have various benefits including accuracy, efficiency, trust and transparency, savings, and security [39].

Blockchains are append-only, distributed, and replicated databases with two key properties: immutability and tamper-resistance [51]. In smart contract systems, the above two properties enforce the "code is law" principle, meaning that conditions recorded in a smart contract are not to be modified since they have been written and published. However, these properties bear their own security risks and challenges: First, smart contracts are written in error-prone programming languages such as Solidity [29], and can contain exploitable programming errors/bugs that are often overlooked or detected only after deployment on the blockchain, and cannot be easily fixed. Second, Ethereum operates on open networks where everyone can join without trusted third parties while smart contracts are often in control of significant financial assets. Hence, smart contracts are attractive and easy attack targets for adversaries to gain financial profits [43].

The consequences of bug exploitation may have global effects on the entire underlying blockchain platform, far beyond the boundaries of individual contracts. For instance, vulnerabilities in a single smart contract, the DAO [67], affected the entire Ethereum network, when in June 2016 the attacker exploited a re-entrancy bug and had withdrawn most of its funds worth $60M US dollars [48], [98]. In the aftermath, the value of Ether, Ethereum's cryptocurrency, dropped dramatically [92], and the postulated "code is law" principle was undermined through the deployment of a hard fork – a manual intervention orchestrated by a notable minority, the team of Ethereum core developers. The Ethereum blockchain was thus split into two versions, Ethereum and Ethereum Classic, which are maintained in parallel since then. In another case, a critical bug accidentally triggered in 2017 resulted in freezing of more than $280M worth of Ether in the Parity multisig wallet [99].

To prevent economic losses caused by bug exploitation, various smart contracts vulnerability detection techniques have been proposed. For instance, researchers have adapted symbolic execution [81], [105], satisfiability modulo theories (SMT) solving [42], data flow analysis [55], runtime monitoring [53], and fuzzing [56] from conventional software security to detect smart contracts vulnerabilities. These techniques require human expertise and need a long detection time [23], [55]. Furthermore, they do not cover 100% of smart contracts: For instance, symbolic execution tools are not guaranteed

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23263
www.ndss-symposium.org

to finalize evaluation in a given time budget due to their complexity[1], while analysis by tools that rely on decompilation of smart contracts (such as Vandal [45]) may fail if the contract cannot be decompiled. Furthermore, these detection tools only check for specific vulnerabilities and are hard to extend on new vulnerability types since experts need to investigate the new security loopholes. As such, one needs to deploy multiple detection tools to cover diverse security bugs, which (i) makes testbed setup cumbersome for smart contract developers as well as contract users; (ii) increases the latency overhead of vulnerability scanning and thwarts runtime analysis.

Recently, Machine Learning (ML) has attracted the attention of security researchers due to its capability to automatically learn the hidden representation from the abundant data [41]. Prior works have shown effectiveness of ML techniques for detecting vulnerabilities in smart contacts [65], [101], [107]. However, as we elaborate in Section VIII, existing ML-based solutions suffer from the following shortcomings: (i) The tools [23], [114] require access to source code, which is only available for 1% of overall contracts [88]; (ii) They only distinguish between vulnerable and safe smart contracts (i.e., binary classification), without the ability to detect vulnerability types; (iii) They are inherently *unscalable* and *unextendible*, as the inclusion of any new vulnerability types would require training-from-scratch of new models;

Our goal and contributions. In this paper, we address the deficiencies of existing solutions and propose ESCORT, the first ML-based smart contract vulnerability detection framework that enables transfer learning and fast adaption to new vulnerability types with limited data. In comparison to existing vulnerability detection tools, ESCORT has the following advantages: (i) Operates on contract bytecodes that are more accessible than source code; (ii) Distinguishes safe and vulnerable contracts; (iii) Identifies the exact vulnerability types in unsafe contracts; (iv) Adapts fast and effective transfer learning to detect new vulnerability types with limited data; (v) Enables efficient detection of multiple vulnerabilities in a single scan; (vi) Evaluation can be conducted by smart contract developers and users by leveraging a convenient API or a mobile app and without an extensive labour of a testbed setup.

In particular, our technical contributions are as follows:

- ESCORT enables *efficient* and *scalable* multi-vulnerability detection of smart contracts. We propose a novel *multi-output* network architecture where the common feature extractor learns general contract semantics and each branch learns representation of a specific vulnerability type (Section IV-B). This enables the defender to utilize a single tool, ESCORT, for concurrent detection of multiple vulnerability types in a single scan instead of deploying multiple detection tools. Therefore, ESCORT greatly reduces the detection time and the effort of testbed setup for contract developers and users.
- ESCORT is the first DNN-based framework that supports *lightweight transfer learning* on new vulnerability types, thus is extensible and generalizable. More specifically, to perform transfer learning, the defender adds a new branch to the feature extractor and updates it to learn the new vulnerability type (Section IV-C). This *modular* update approach has two advantages: (i) ESCORT only needs a *small set of new data* to learn the new vulnerability since we only update parameters in the new branch. This suggests that defender can enable detection of the new security bug in a short time. (ii) ESCORT preserves the detection performance on existing vulnerability types. Our multi-output architecture resolves the catastrophic forgetting issue in transfer learning [50], [75] since the new branch layer is designated to learn the new task (i.e. new vulnerability type). The independence between different branches ensures that our transfer learning does not impact the performance on the old tasks (detection of existing vulnerability types).
- ESCORT is automated for inspecting vulnerabilities in smart contracts prior to their deployment and demonstrates superior vulnerability detection performance with low overhead (Section VI). We perform a comprehensive evaluation of ESCORT on a dataset of 3,640,153 EVM-compatible smart contracts to corroborate its effectiveness, efficiency, and extensibility (Section VII). Our framework can detect 11 vulnerability classes in $0.15\,\mathrm{s}$ per smart contract on average and yields an average F1 score of $97\,\%$ across all evaluated classes.
- We develop a toolchain named Demeter that automates bytecode acquisition, and smart contracts labeling from Ethereum blockchain (Section V). Demeter's modular structure enables one to easily integrate other tools for labeling. We will open-source Demeter and make our dataset of 3,640,153 labeled smart contracts available for research at https://github.com/sss-wue/smarter-contracts.
- To facilitate interacting with ESCORT, we have implemented a model serving API where a smart contract developer or blockchain user can query ESCORT for detected vulnerabilities (Section VI-B1). We also have embedded ESCORT into the Metamask Android application [14] where a user can privately and securely check the smart contract against vulnerabilities before sending funds (Section VI-B2).

To summarize, ESCORT provides an automated and transfer learning-friendly vulnerability detection for smart contracts. Our modular design of the multi-output architecture allows ESCORT to distill the knowledge of existing detection tools and quickly adapt to new vulnerability types with limited data using transfer learning. The transfer learning capability of ESCORT allows it to collect sufficient training data and achieve lifetime security maintenance. Compared to existing non-ML based detection methods that focus on very few vulnerability types, ESCORT is a unified framework that can identify diverse vulnerabilities in a quick single run. As such, ESCORT provides security support for contract developers at contract development time and also enables runtime analysis for contract users.

II. Background

We provide background information on smart contracts in this section. More detailed description of Ethereum and deep learning are given in Appendix.

[1]For example, while analysis by Oyente [81] takes 350 sec. on average, around 1,4% of analyzed contracts require more than 30 min.

A. Smart Contracts and Vulnerabilities

A smart contract is written in high-level programming languages such as Solidity [29] and is called by its address to run operations on the blockchain. Once compiled, the bytecode of the contract is generated and executed inside the Ethereum Virtual Machine (EVM). Since there is a one-to-one mapping between a blockchain operation and bytecode representation, it is feasible to analyze the control flow of a contract at the bytecode-level. When triggered, the execution of the smart contract is autonomous and enforceable for all participating parties [52]. The EVM itself is a stack-based machine with a word size of 256 bits and stack size of 1024 [108]. The memory applies a word-addressable model. Once a contract is deployed on the blockchain, it requires gas to function. Gas is the unit used to pay the computational cost of the miners running contracts or transactions and is paid in Ether.

Similar to any other software, smart contracts might suffer from vulnerabilities and programming bugs. The Smart Contract Weakness Classification (SWC) Registry [22] collects information about various vulnerabilities. We differentiate four categories of vulnerability types: External Calls, Programming Errors, Execution Cost, and Influence by Miners.

External Calls. Any public function of a smart contract can be called by any other contract. A malicious user can then exploit public availability to attack vulnerable functions of smart contracts. A prominent example is the so-called reentrancy bug (SWC-107 [22]). Here, an attacker can call a contract's function multiple times before the initial call is terminated. If the internal contract state is not securely updated, the attacker can drain Ether from the contract by recursively calling the function.

Programming Errors. Some programming errors in smart contracts are very similar to those found in traditional programs, such as missing input validation, typecast bugs, use of untrusted inputs in security operations, unhandled exception, exception disorder, and Integer overflow and underflow vulnerabilities. In another example, an *assert* function used in tests and not removed by the programmer in the release version may lead to its misuse by an attacker, which can result in exploitable error conditions (SWC-110 [22]). Other vulnerabilities can be specific to smart contracts. Examples are greedy contracts that lock Ether indefinitely, gasless send bug that does not provide sufficient gas to execute the code of the smart contract, Ether lost in transfer if sent to unknown recipients, etc. Further examples are *callstack depth* limit reached exception bug and unprotected *selfdestruct* instructions (see SWC-106 [22]), where an attacker can call a smart contract's public function containing a *selfdestruct* to terminate the contract, or he can fill up the stack to reach the stack size limit. Both attacks result in a Denial of Service (DoS) of vulnerable smart contracts.

Execution Cost. Every transaction on the Ethereum network costs gas. However, every block has a spendable gas limitation. An attacker can use this limit to induce a DoS of a vulnerable contract. For example, if the execution time of a function is dependent on input from the caller, a malicious caller can expand the execution time of the smart contract over the gas limit (SWC-128 [22]). Thereby, execution is terminated by exceeding the gas limit before it is finished. Another way an attacker can misuse the gas limit per block is to induce an error on a *send* call. If a programmer bundles *multiple sends* in one function of the smart contract, the attacker can then prevent the execution of other *send* calls in the function.

Influence by Miners. Miners are entities that actually execute transactions on the blockchain. They can decide which transactions to execute, in what order, and are also able to influence environment variables (e.g., timestamps). To illustrate the problem, let us assume a scenario where a smart contract is instructed to send Ether to the first user that solved the puzzle. If two users commit a transaction with the solution at the same time, a miner decides who will be first and therefore will be getting the Ether (SWC-114 [22]). This vulnerability type is generally referred to as Transaction Order Dependence (TOD).

III. Goal, Threat Model and Challenges

In this section, we first introduce our goal and then present our threat model and assumptions. We also discuss the challenges of developing an effective and scalable detection tool.

A. Goal

Our goal is to simplify the effort of security testing for both smart contract developers and users. On the one hand,the process of security testing by smart contract developers is cumbersome, since existing vulnerability detection tools have limited vulnerability coverage and are hardly extensible. Many of them also require significant amount of time for testing. Hence, sound testing requires non-trivial engineering effort for setting up individual testing environments for every vulnerability scanner, and may also result in time-consuming testing. This load is often unbearable for smart contract developers, who either opt to outsource security testing to professionals, which induces additional costs or bear the risks of publishing poorly tested smart contracts. On the other hand, runtime security testing on the contract user side is also critically important. The user typically has less expertise and computing power compared to the contract developer, which makes user-side testing even more challenging.

With ESCORT, one can integrate the knowledge of many security tools in one solution and enable high-coverage and highly efficient testing with only one framework. Overall, our approach has the potential to significantly simplify the effort of smart contract developers for security testing.

B. Threat Model and Assumptions

Assumptions. We assume the detection metrics (such as F1 score, false positive/negative rates) reported in the previous papers [59], [81], [84] as well as the open-sourced implementation of existing detection tools [4], [17], [23] are reliable (accurate). ESCORT uses these detection scores to guide the aggregation of detection decisions made by these tools when labeling smart contracts (detailed in Section V). This assumption is feasible since expert inspection has been performed to cross-validate the performance of proposed detection methods in the previous works.

Attacker Capability. The attacker is a malicious party that can obtain knowledge from any public data structure in the blockchain and can upload his contract code to the Ethereum system. The attacker also knows how to exploit software vulnerability in the Solidity source code. However, he cannot interfere with the detection or make further changes to the smart contracts after uploading it.

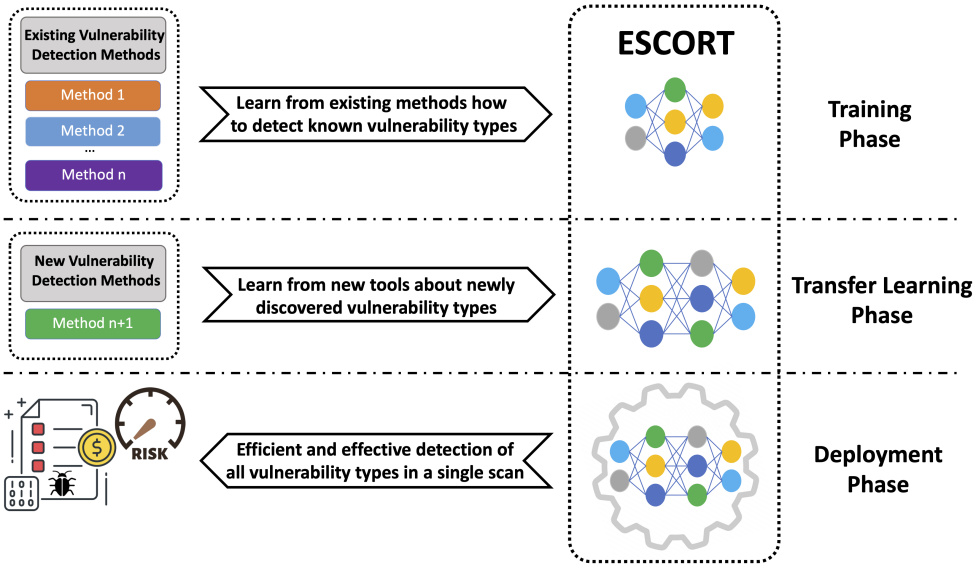


Fig. 1: Overview of ESCORT framework for Ethereum smart contract vulnerability detection. ESCORT has three stages: training, transfer learning, and deployment as shown in the top, middle, and bottom part of the figure.

ESCORT aims to provide the defenders with a holistic smart contract vulnerability detection solution. To this end, we formulate vulnerability inspection as a *supervised multi-label classification* problem where the input is the contract (can be represented in high-level language, opcodes, or bytecodes) and the output is the corresponding vulnerability types as introduced in Section VII-A.

C. Challenges

In this section, we present the challenges posed on developing an effective DNN for automated smart contract vulnerability detection.

(C1) Feature Extraction. Analyzing smart contracts based on the bytecode is a preferable option, since only 1% of all contracts are open-sourced [88]. Moreover, blockchain platforms typically host their smart contracts as long bytecodes. The first challenge concerns the problem of finding the proper feature representation of the smart contract programs with such long bytecode. On one hand, manual design of contract features is time-consuming and has limited efficacy, since the contract bytecode is long and hard to interpret by human developers. On the other hand, current implementations of automated feature extraction [101], [107], [114] (Section VIII) mainly apply traditional software testing techniques on the smart contract without exploring its domain-specific properties, thus the inspection time is long.

(C2) Dataset Imbalance. The second challenge is a class imbalance in the training set. Prior works have shown that the number of contracts with specific vulnerability types is much lower than the one of non-vulnerable contracts [107]. Learning the characteristics of vulnerable contracts with a DNN is challenging since the stochastic gradient descent (SGD) based learning of DNN models is inherently biased towards the majority class [76], while our objective is to recognize the minority class (i.e., vulnerable contracts). As such, it is crucial to provide the DNN model with sufficient vulnerable contracts to ensure a high true positive rate.

(C3) Efficiency. The third challenge is to ensure the efficiency of DNN training and inference for concurrent detection of multiple vulnerability types. Identifying diverse attacks with a single DNN detector is challenging since different vulnerabilities exploit distinct loopholes in the contract, which might be hard to capture with a conventional DNN. For non-ML based detection methods, the defender needs to deploy multiple tools in order to cover various vulnerability types, which increases the overhead of testing. Efficiency is important for practical development of the contract scanner since devising individual classifiers for each vulnerability class, as done, e.g., in [107], is unscalable and incurs large computation overhead.

(C4) Extensibility. Further, smart contract inspection should be capable of learning new vulnerability types quickly while preserving the knowledge of the known ones. We call this requirement 'extensibility'. This property is important for both, researchers, and practitioners since new attacks on smart contracts are emerging at a fast speed. Augmenting an existing contract detector to new attacks is non-trivial since the new attack exploits the unseen and unpredictable susceptibilities of smart contracts compared to the previously known attacks. Training a new DNN from scratch to accommodate the new vulnerabilities consumes extensive resources and incurs additional engineering costs.

(C5) Limited Data. Finally, the number of available samples per vulnerability might be limited, which hinders researchers and practitioners to apply ML techniques to more vulnerability types. Data augmentation may mitigate this issue. However, augmentation will result in bias in the model [66].

IV. ESCORT Design

We propose ESCORT, the first extensible and transfer learning-friendly DNN-based framework for vulnerability detection of Ethereum smart contracts. The key innovation of ESCORT is that we decompose the task of vulnerability detection into two subtasks: (T1) Learning the bytecode features of general contracts (*attack-agnostic*); (T2) Learning to identify each particular vulnerability class (*attack-specific*). To achieve (T1), we design a *feature extractor* that captures the semantic and syntactic information of contract bytecode regardless of its vulnerabilities. To perform (T2), we devise an individual *vulnerability class branch* to characterize susceptibility given

the bytecode features extracted in (T1). Our **divide-and-conquer** design is highly modular and flexible compared to previous ML-based detection techniques (cf. Section VII).

A. ESCORT Global Flow

Figure 1 shows the global flow of ESCORT. The three main stages are detailed below:

Training. The top part of Figure 1 shows ESCORT's training pipeline. In this stage, ESCORT distills the knowledge of multiple existing detection tools into an coherent architecture while improving the attack coverage. To enable supervised learning for vulnerability detection, the defender first constructs the smart contracts bytecode dataset with corresponding labels, as detailed in Section V. The defender specifies the system parameters, including the vulnerabilities of interests and the available hardware resources for ESCORT's multi-output DNN design. Finally, the devised model is trained on the collected contract data with their corresponding labels, resulting in a converged DNN detector.

Transfer Learning. The middle part of Figure 1 shows the transfer learning phase of ESCORT. Given a trained detector and the bytecode of smart contracts with new vulnerability labels, ESCORT extends the DNN architecture devised in the original training phase with new parallel branches. The layers in the expanded branch are then trained on the new vulnerability data with the associated labels to perform transfer learning. To the best of our knowledge, ESCORT is the first framework that supports transfer learning to accommodate new vulnerability types of the smart contracts.

Deployment. The bottom part of Figure 1 shows the workflow of ESCORT's deployment stage. After the training/-transfer learning phase completes, ESCORT returns a trained DNN classifier that can detect whether an unknown smart contract has any of the learned vulnerability types. Our detector provides diagnosis results for multiple vulnerabilities with high accuracy and short runtime. More specifically, for smart contract developers, we devise a REST API that hosts the learned ESCORT model and performs inference for vulnerability detection (Section VI-B1). For contract users, we integrate ESCORT into MetaMask Mobile [14], a popular crypto wallet app for Android and iOS devices. The integration of ESCORT ensures that the safety of a transaction is checked either on-device by retrieving the contract bytecodes and running ESCORT inference, or the user can outsource the security test to our remote server by sending the receiver address via a REST API web request (Section VI-B2).

Note that ESCORT is a smart-contract-specific classifier. In particular, the length of the bytecode is limited to 17.5k, and our preprocessing is specific to EVM bytecode. Both of these design choices are tailored for smart contracts and are not transferable to other domains. We comprehensively compare though alternative approaches in the domain of vulnerability detection in smart contracts using ML in Section VIII.

B. Neural Network Design

Prior ML-based detection techniques have tried various models for vulnerability detection/classification such as SVM, Decision Trees [107], CNN [65], LSTM [58], [101], and GNN [114]. However, all of these methods require one to train a new ML model from scratch given contracts with new vulnerabilities, resulting in slow and inefficient model adaptation. ESCORT aims to design an **extensible** DNN detector that: (i) provides the probability that the smart contract has certain vulnerabilities, instead of making a binary decision about contract security; (ii) classifies multiple vulnerabilities using a single DNN with the consideration of vulnerability type extension. To this end, we propose a **multi-output architecture** for **concurrent** detection of multiple vulnerability types. This neural network design step is shown by the 'Multi-output DNN Design' module in ESCORT's global flow (Figure 1).

Figure 2 shows the generic architecture of our DNN detector. The stem and branch layers are typical DNN layers such as the Dense layer, Dropout layer, and GRU layer. The multi-output model has two main components discussed below:

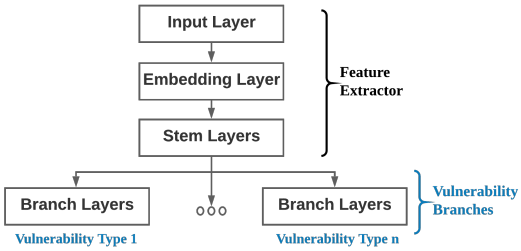


Fig. 2: General topology of ESCORT's multi-output model for concurrent detection of multiple vulnerability types.

(i) Feature Extractor. The first component of ESCORT's extensible DNN model is the common feature extractor (i.e., 'stem') shared by all the bottom branches. The feature extractor is a stack of layers that aim to learn the fundamental features in the input data that are general and useful across different attributes. In the context of smart contracts, the feature extractor is trained to learn the semantic and syntactic information from the contracts' bytecode. To this end, we incorporate several key layers in ESCORT's feature extractor:

- *Embedding Layer.* The bytecode of smart contracts are long hexadecimal numbers, while DNNs typically work with fractional numbers to achieve high accuracy. The embedding can solve this discrepancy since it stores the word embedding in the numerical space and retrieves them using indices [111]. The embedding layer provides two key benefits: (i) Compressing the input via a linear mapping, thus reducing the feature dimension; (ii) Learning bytecode in the embedding space (fractional numbers). This facilitates representation exploration and gathers similar bytecode in the vicinity of each other. ESCORT leverages the advantages of the embedding layer to capture the semantics in the input bytecode.
- *GRU/LSTM Layer.* The stem layers and branch layers in Figure 2 can include GRU/LSTM layers for processing sequential inputs. Gated Recurrent Units (GRU) and Long Short-Term Memory (LSTM) are two typical layers in recurrent neural networks that help to overcome the short-term memory constraint and vanishing gradient problem [70] using a *'gating'* mechanism. More specifically, both types of layers have internal gates that regulate the information flow along the time sequences and decide which data shall be kept/forgotten. We mainly use GRU layers in ESCORT's DNN design.

(ii) Vulnerability Branches. The second component of ESCORT's multi-output DNN architecture is the **ensembling** of multiple vulnerability branches. Each branch is a stack of layers that are trained to learn the patterns/hidden representation of the corresponding vulnerability class. While there is no direct dependency between different branches, they share the same feature extractor, i.e., the input to each branch is the same. This is feasible since the branch input (which is also the feature extractor's output) shall capture the semantics in the contract's bytecode, which is common/general information useful for different vulnerabilities. Note that the last layer of each vulnerability branch is a Dense layer with one neuron. The *sigmoid* evaluation of this neuron gives the *probability* that the input contract has the specific vulnerability. As such, ESCORT engenders detection results with better *interpretability* by providing the confidence score for its diagnosis instead of the binary decision about vulnerability existence.

ESCORT's 'stem-branches' architecture is similar to the *'mixture of experts'* paradigm where the problem space is divided into homogeneous regions and individual expert models (learners) are trained to tackle each sector [72]. The main difference between ESCORT's multi-output design and the mixture of experts model is that the latter one requires a trainable gating network to decide which expert shall be used for each input region, while our DNN model does not need such a gating mechanism since we aim to detect multiple vulnerability types of the input contract in parallel.

In summary, ESCORT's multi-output architecture solves the feature extraction, efficiency and extensibility challenges (C1, C3, C4) identified in Section III-C. In particular, ESCORT allows the defender to train a *single* DNN for detecting multiple vulnerability types instead of training an individual classifier for each attack, thus demonstrating superior efficiency compared to the prior ML-based techniques [58], [65], [101], [107], [114], as we elaborate in Section VIII. ESCORT design incurs minimal non-recurring engineering cost and is scalable as new vulnerabilities are identified.

The superior efficiency of ESCORT can be attributed to two factors: (i) We use a single multi-branch DNN architecture to achieve *concurrent, multi-output detection* instead of devising individual ML models for each vulnerability type. Therefore, our model has much fewer trainable parameters and requires smaller computation overhead for training. (ii) Fast *transfer learning* with new branches. We empirically show in Section VII-C that our transfer learning is successful with a small set of new data and a new branch (which implies the total computation amount is also small).

C. Transfer Learning

Malicious parties have a strong incentive to discover and exploit new vulnerabilities of smart contracts due to the associated prodigious profits. As such, the contract inspection technique shall be extensible to learn new vulnerabilities as they are identified. We propose *transfer learning* as the solution to the challenge (C4 and C5) in Section III-C. Transfer learning makes it possible to exploit the knowledge gained from existing vulnerabilities to improve generalization about new vulnerabilities with limited data. More specifically, we suggest to *expand* the pre-trained multi-output DNN model by adding new vulnerability branches for transfer learning. This process is demonstrated in the middle part of Figure 1. The transfer learning capability of ESCORT ensures that our detection framework can be upgraded with the minimal cost to defend against emerging attacks on smart contracts.

Our transfer learning stage has two goals:

(G1) Preserve Knowledge on Old Vulnerabilities. On the one hand, the DNN detector shall retain knowledge about the previous vulnerability types that are used in the initial training phase. This property is important since ESCORT aims to provide a holistic and extensible solution to concurrent detection of multiple vulnerabilities. As such, maintaining high classification accuracy on the known attacks is essential.

(G2) Learn New Vulnerabilities Quickly. On the other hand, transfer learning aims to adapt the pre-trained model to achieve high accuracy on the new task data (limited size) in an efficient way. This is also required by the extensibility challenge (C4) and limited data challenge (C5) in Section III-C. To achieve fast adaptation, transfer learning shall yield minimal runtime overhead. This requirement is crucial for practical deployment, since training a new DNN model from scratch for the new vulnerabilities with limited data is prohibitively expensive and hard to maintain.

ESCORT's transfer learning phase works as follows. When a new vulnerability is identified, the defender constructs a new training dataset accordingly and updates the converged DNN detector by adding a new vulnerability branch (i.e., the stack of layers). During transfer learning, the parameters of the common feature extractor and existing vulnerability branches are *fixed*. Only the parameters in the newly added branch are updated with the new vulnerability dataset. Freezing the feature extractor and the converged branches ensures that the updated DNN classifier preserves the detection accuracy on the old vulnerabilities (G1), and training a new branch enables the updated model to learn the new attack (G2).

Besides extensibility, ESCORT also enables lightweight and fast adaptation when *model drift* occurs. Smart contracts running on Ethereum are dynamic and change over time, which might lead to a decrease of ESCORT's performance. To alleviate the model drift concern, the contract developer can update the parameters of the vulnerability branches given a set of labeled new contracts while keeping the weights of the feature extractor fixed.

V. Dataset Construction Toolchain

In this section, we present Demeter, a toolchain we built to harvest smart contracts and construct the labeled dataset. We will open source our toolchain and the collected dataset to facilitate the development and comparison between emerging detection techniques.

A. Design Choices

To build a sufficiently large training set for our supervised DNN training, we make the design choice to work on the bytecode-level because not only the bytecode of smart contracts are publicly available but also easily attainable for vulnerability detection task running by users. The users are unlikely to have access to the source code of smart contracts (only 1% of smart contract source codes are public [88]), and even if the source codes are accessible, the users need a guarantee to ensure the bytecode they communicate with is an exact compiled version of the intended source code. Operating on bytecode-level also makes our approach agnostic

to the programming languages of smart contracts, since smart contracts that are written in different languages (e.g., Solidity [29], Viper [38], and Serpent [28]) get compiled to the same bytecode eventually. The EVM bytecodes are executed in a stack context, thus the control flow of a program at the bytecode level contains useful information for detection.

Demeter is designed to obtain the bytecode files of smart contracts from the Ethereum platform consisting of the blockchain Ethereum Mainnet and its testnets [20] (e.g., Goerli [31], Rinkeby [33], Ropsten [34], Kovan [32]), label them using available bytecode-level detection tool(s), and store the result in a database. Note, that we concentrate on Ethereum smart contracts for exemplary purposes in this paper. Generally, ESCORT can also be used for vulnerability detection in smart contracts of other cryptocurrency platforms that use Ethereum-compatible EVM, such as Quorum [25], Vechain [37], Rootstock [26], and Tron [36], to name a few. This is possible since the bytecode of smart contracts on these platforms is compatible with Ethereum EVM.

B. Demeter Architecture

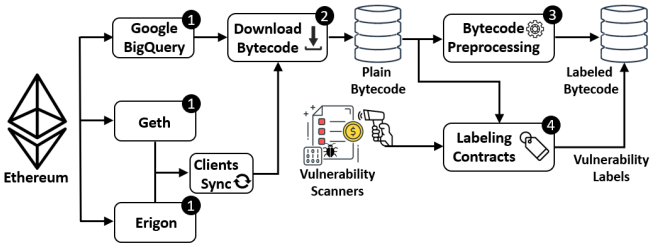


Fig. 3: Generic workflow of Demeter for smart contract acquisition and labeling.

We show the generic workflow of Demeter toolchain in Figure 3. In step (1), the addresses of contracts are retrieved from the blockchain. Step (2) involves downloading the bytecode from the Ethereum network by block and extracting information into a database. In step (3), the bytecode is preprocessed for input efficiency. The last step (4) outputs the vulnerability types of the contracts using the bytecode-level detection tools. It is worth noticing that any vulnerability detection techniques that take bytecode as input can be used by Demeter for contract labeling, including existing methods such as Oyente [23], Mythril [17], Vandal [45], Maian [88]. We show in Section VII that ESCORT outperforms existing tools in terms of both detection effectiveness and efficiency.

For a concrete instantiation of Demeter, we use the open-source clients *Geth* (shorthand for *Go Ethereum*, official implementation) [7] and *Erigon* (third-party client) [3], as well as the open dataset of Google BigQuery [8] to access the bytecode of contracts in step (1). Both of the Ethereum clients (i.e., *Geth* and *Erigon*) sync with the entire blockchain. We use Erigon to gain access to the testnets Goerli, Ropsten, and Rinkeby since it is faster and generates less data on disk. For Kovan, we use Geth since Erigon does not support Kovan yet. We download Ethereum Mainnet from a open dataset hosted on Google's BigQuery [8]. For step (2), we also utilize existing tools. Multiple Web APIs exist and allow us to download contract bytecodes by its address or block number. We opt for the Python API Web3 [11] for instantiating the bytecode downloading module of Demeter. For the instantiation of step (3), we develop an assistive Python module named *Contract Loader* to extract the information from smart contracts into a MySQL database [16]. In the last step (4), we use four tools for contract bytecode labeling: Oyente [23], Mythril [17], Vandal [45], Maian [88]. The modular structure of Demeter makes it easy to extend our toolchain with other vulnerability detection tools.

C. Demeter Workflow

1) Bytecode Acquisition: For dataset construction, Demeter first utilizes Python's Web3 library to extract smart contracts from each blockchain. We were able to extract in total 26,740,370 bytecodes from the five nets. Particularly, we obtained 22,789,100 bytecodes from Ethereum Mainnet, 101,998 from Goerli, 1,382,338 from Rinkeby, 1,831,168 from Ropsten, and 635,766 from Kovan. Since smart contracts can be deployed across all networks, we need to *deduplicate* the retrieved contracts. Deduplication also eliminates contracts that are identical in the same net due to contract factories and multiple deployments of the same contracts. In some cases, the download resulted in an empty bytecode 0x. The possible reasons could be: (i) The Ethereum node is not fully synced with the network, thus the bytecode is not available; (ii) An empty contract is deployed; (iii) The smart contract is self-destructed. At the end of the bytecode acquisition process, 4,062,844 smart contract bytecode files become available for ESCORT's vulnerability analysis.

2) Bytecode Preprocessing: The downloaded bytecodes consist of hexadecimal digits that represent particular operation sequences and parameters. In the preprocessing step, Demeter first transforms the collected raw bytecodes to sequences of operations divided by a unique separator, and removes input parameters from the bytecode to reduce the input size. Furthermore, it merges operations with the same functionality into one common operation. For instance, similar commands *PUSH1 - PUSH32* (represented by the bytes *0x60-0x7f*) are replaced with the *PUSH* operation (represented by *0x60*). Note that some hexadecimal digits in the crawled bytecode do not correspond to any operations defined in the Ethereum Yellow Paper [108]. These bytes are considered as invalid operations and substituted with the value *XX*. This operation merging step might map bytecodes to identical preprocessed bytecodes. We also deduplicate the dataset of preprocessed bytecodes, which results in 3,640,153 bytecodes in our dataset.

Note that we only model opcodes and ignore operands, which is sufficient to capture semantics. For instance, in the case of an integer overflow/underflow, any integer can be overflown if no boundary checks are added to the code. ESCORT can distinguish the presence or absence of those at the level of opcodes. Similarly, a reentrancy bug results in state change after a call command.

3) Labeling of Smart Contracts: A smart contract might have multiple vulnerabilities as introduced in Section II-A. Each of the vulnerability detection tools used by Demeter is specialized for detecting a specific set of vulnerability types. Note that besides the bytecode-level detection tools, Demeter also stores the performance metrics (e.g., F1 score) of each tool on each vulnerability type that they can detect. The performance characterizations are obtained from the previous publications [4], [17], [23] with experts' manual inspection to ensure the correctness. To determine if a given smart contract

has a specific vulnerability type, Demeter selects the detection tool that features the highest F1 score on this vulnerability among all available tools and uses it for contract labeling. Demeter repeats the above process for each contract and each vulnerability type. We develop Python modules to perform the above task. In the end, 3,640,153 smart contracts are labeled. It is worth repeating that since there are no dependencies among the vulnerability scanning tools, the set of vulnerability types can be easily extended with other available tools.

We would like to point out that we do not focus on dealing with the labeling bias of existing detection methods (Oyente [23], Mythril [17], Vandal [45], Maian [88]) since our goal is not to design a meta-detector that performs 'smart' ensembling/aggregation on top of the existing detectors. Therefore, we do not tackle the labeling bias of existing tools used by our toolchain Demeter. The main objective and contribution of ESCORT is to simplify security testing (replace multiple tools, provide high coverage, simple usage, fast detection, no access to source code required, extendable, the analysis never fails). No existing tools can provide this.

VI. Implementation

For our proof of concept, we instantiate design of ESCORT (cf. Section IV) on eleven vulnerability types and elaborate on the implementation details below.

A. ESCORT DNN Instantiation

1) Dataset imbalance: We construct the labeled bytecode dataset as explained in Section V for supervised learning of ESCORT. The collected contract data might have the class imbalance issue [107] (Challenge (C2) in Section III-C). In our work, we construct a balanced training set to ensure that each batch of data fed into ESCORT's DNN model has a comparable number of vulnerable and safe contracts. Particularly, we take an under-sampling approach and randomly removed examples from the majority class labels to avoid the class imbalance problem. Another possible selection strategy is to over-sample minority classes, which we do not opt for to avoid introducing additional bias. In addition, utilizing Transfer Learning, we solve the data imbalanced problem in the vulnerability type extension phase to a great extent (Section VII-C) if the vulnerability type is underrepresented. Transfer learning can augment learning when training examples are not sufficient and induce balance into skewed datasets. Details about our dataset balancing is given in Section VII-A.

2) Model Building and Training: We instantiate a Multi-Output-Layer (MOL) DNN based on our generic architecture in Section IV-B. Figure 4 shows the actual model used in our experiments. We build a MOL-DNN with six branches for main model training, and then extend it with five new branches for transfer learning. Our multi-output RNN enables concurrent detection of multiple vulnerability types as discussed in Section IV-B. ESCORT learns vulnerabilities in Demeter's labeled bytecode dataset via supervised learning in this stage.

We train the above instantiated RNN using the hyper-parameters shown in Table I. These hyper-parameters are found by grid-search. Before the data is passed to the model's input layer, the bytecode sequence needs to be vectorized. This is realized by a *tokenizer*, which transforms the hexadecimal data into numeric vectors. After tokenization, a hyper-parameter *MAX_SEQUENCE_LENGTH* is applied to the input vectors.

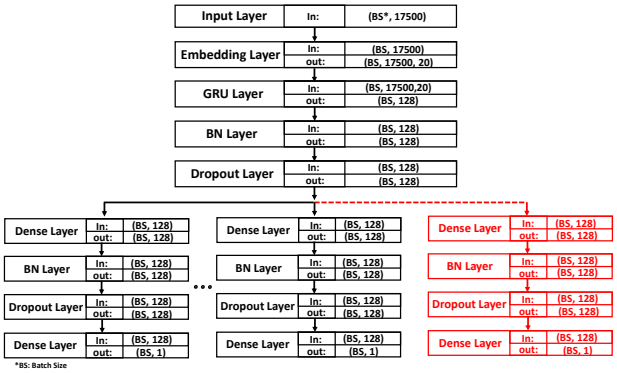


Fig. 4: Multi-output DNN architecture of ESCORT for concurrent detection of multiple vulnerability types. Here, BS is the batch size and BN is batch normalization, the rightmost red branch denotes adding a new branch for transfer learning on new vulnerabilities.

Sequences are zero-padded or truncated to this length. We empirically study the distribution of the bytecode length and show the results in Figure 5. The hyper-parameter *MAX_SEQUENCE_LENGTH* is set to 17,500 to ensure none of the contracts are truncated.[2]

Variable	Setting
#Hidden Units	GRU:128, Dense:[128 ,1]
Optimizer, Loss Function	Adam, BCE
Learning Rate	0.001
Batch Size	128
MAX Seq. Length	17500

TABLE I: Model Hyper-parameters.

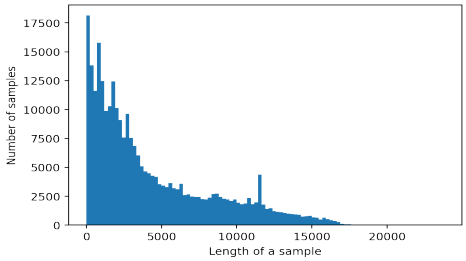


Fig. 5: Distribution of bytecodes length in the dataset.

The tokenized data are then passed to the RNN for training. In the training phase, we monitor the loss of our model for performance improvements. The training and validation results are stored with Tensorboard. In the testing phase, the remaining (unseen) data are passed to the model and we compute the detection metrics. At the end of model training, we save the converged MOL-DNN model, the deployed tokenizer, and the evaluation metrics files to wrap the model as an API service.

B. End-to-end deployment

We discuss how ESCORT provides automated, end-to-end security testing for two deployment cases below.

1) Deployment for smart contract developers: Our trained DNN model can detect pre-defined vulnerability types in smart contracts. We wrap the model within an API to ensure that

[2]Both excessive padding and truncation may impair the performance. For instance, truncated contracts might be mislabeled as benign if a vulnerability resides in the removed part of the contract.

we can serve predictions to end-users on the fly. We utilize Flask [6] and devise a Python module to provide a REST API endpoint for running model inference on bytecode files. The learned ESCORT model and associated configurations are passed to our API. Our API also performs automated bytecode transformation (input preprocessing) to remove the manual efforts for the defender. The Python module creates two different API endpoints. The first endpoint shows the configuration passed to the module and the second one triggers model inference. Listing 1 shows an example of the plain bytecode of a smart contract, which is passed to the second endpoint for vulnerability detection.

Listing 1: Sample request body when calling ESCORT's prediction endpoint.

```
{"smart_contract": "606060405236150100000..."}
```

In the second endpoint, the bytecode of the input smart contract is vectorized using the same tokenizer as the one used in ESCORT's model training step. The processed sequence is then fed as the input to the trained MOL-DNN model for vulnerability detection. In addition to the detected vulnerability types, the prediction time of ESCORT is tracked and shown to the user. An example response from ESCORT's API endpoint is shown in Listing 2.

Listing 2: ESCORT's response to the sample request when calling our prediction endpoint. The response includes the analysis results of multiple vulnerability types.

```
{"prediction": {
    "ASSERT_VIOLATION": 0.0001,
    "ACCESSIBLE_SELFDESTRUCT": 0.9998,
    "DoS (UNBOUNDED_OP)": 0.9996,
    "MULTIPLE_SENDS": 0.0012,
    "TAINTED_SELFDESTRUCT": 0.9998,
    "CALLSTACK": 0.9995,
    "MONEY_CONCURRENCY": 0.0013,
    "REENTRANCY": 0.0009},
 "prediction_time in_second": "0.02"}
```

2) Deployment for smart contract users: We also developed a Proof-of-Concept (PoC) implementation of ESCORT within *MetaMask Mobile* [14], a widely used crypto wallet app for Android and iOS devices that provides easy access to the Ethereum blockchain. It can also be used as a gateway to communication with decentralized apps and smart contracts. Adding vulnerability detection to MetaMask allows the users to check the smart contracts before interacting with them. This is especially important, since non-developer users of the blockchain generally do not have the knowledge and time to test smart contracts with several security tools. For instance, setting up the tooling for security testing is not trivial and demand proficient knowledge of the underlying technologies. Further, a deployment of those tools (i.e., Mythril, Oyente, Vandal) for Android is not available. With ESCORT deployed in MetaMask, users can easily check smart contracts for vulnerabilities before sending funds.

For this purpose, we modified the app's behavior in the following way. As soon as the user begins a new transaction, the app determines whether the receiver address belongs to a smart contract. If that is the case, the Android app can either send the address to our remote server using a REST-API web request (c.f. Section VI-B1), or directly downloads, pre-processes the smart-contract bytecode, and then run the inference on the device. The results (i.e., the probabilities of vulnerability classes) are displayed to the user to take an action: interrupting or resuming the transaction. The former way offers a better performance (c.f. Section VII-E) due to more powerful hardware (i.e., remote server) and the latter provides a better privacy due to on-device pre-processing and classification. Our PoC shows that ESCORT can easily be adopted by existing applications.

VII. Evaluation

We extensively evaluated ESCORT on the large-scale smart contract dataset built as described in Section V. We utilize the *tf.keras* package [30] for model building, training, and inference. All of our experiments, if not otherwise stated, are conducted on a machine with Arch Linux OS having AMD Ryzen 9 3950X and NVIDIA GeForce RTX 3090 GPU with 32 and 24 GB of RAM, respectively. The software versions are as follows: Tensorflow 2.7.0, CUDA 11.6, NVIDIA driver 510.54, cuDNN 8.2.1 and kernel 5.15.28. In this section, we explain our experimental setup and the evaluation metrics to characterize the performance of ESCORT's DNN model.

A. Dataset

To build our dataset, we collected 3,640,153 smart contracts from five Ethereum nets using Demeter and labelled them accordingly (Section V-C). We consider eleven vulnerability types in our evaluations, while ESCORT can be easily extended to detect new attacks (Section IV-C). We use the first six vulnerability types (cl. 1-6) for main model training and cl. 7-11 for transfer learning experiments. The mechanisms of these eleven vulnerability types are discussed below (general vulnerability categories are provided in Section II-A):

- **Callstack Depth [cl. 1] (Oyente)**: This vulnerability class belongs to the *Programming Error* category (Section II-A) and exploits the stack size limit issues of the EVM.
- **Money Concurrency [cl. 2] (Oyente)**: This vulnerability is also known as Transaction Ordering Dependence (TOD) and belongs to *Influence by Miners*.
- **Assert Violation [cl. 3] (Mythril)**: This *Programming Error* leads to a constant error state of the smart contract, which can be exploited by an attacker.
- **Reentrancy [cl. 4] (Vandal)**: Reentrancy bugs are caused by *External Calls* and allow an attacker to drain funds.
- **Unchecked Calls [cl. 5] (Vandal)**: With this *Programming Error*, function calls are not checked. In the worst case, the contract execution will continue despite a possible exception of the call.
- **Time Dependency [cl. 6] (Mythril)**: The contract uses block values as a proxy for time (*Programming Error*). However, block values are not exact and can lead to an unpredictable state.
- **DoS with Failed Call [cl. 7] (Mythril)**: This is similar to the Unchecked Calls class 5. Here, the error will induce a DoS if the calls are in a loop.
- **Integer Under/Overflow [cl. 8] (Mythril)**: This *Programming Error* is identical to the one in conventional programming languages. Integers may flip to zero/infinity when incremented over the feasible range.

- **Accessible Selfdestruct [cl. 9] (Mythril)**: This *Programming Error* can be exploited to terminate a contract such that the remaining funds are sent to a predefined address.
- **Arbitrary Jump with Function Type Variable [cl. 10] (Mythril)**: If a user alters a function type variable (i.e., pointer to a function), an attacker may abuse this *Programming Error* to jump to an arbitrary function.
- **Weak Sources of Randomness [cl. 11] (Mythril)**: Randomness is difficult to obtain in standard computer programs and this task is more challenging on the blockchain since the consensus protocol is based on determinism. Using blockchain attributes to generate randomness makes the contract vulnerable to an *Influence by Miners*.

For each vulnerability class in our main model (trained on cl. 1-6), we select around 60,000 samples with this specific vulnerability from our raw dataset 3,640,153 contracts) and concatenate them to construct the dataset of vulnerable contracts (thus it has equally-sized distribution). We empirically set the minimal sample number to 60,000 since this is the smallest size of the well-represented vulnerability types in our dataset. The dataset of vulnerable contracts was then extended with 60,000 safe smart contracts for which no vulnerabilities were detected by the tools used in Demeter (Section V-C).

Figure 6 shows the vulnerability class distribution of our labeled dataset. We group vulnerabilities into three groups, depending on the number of vulnerable samples per class. The extension dataset includes vulnerability classes with at least 20k samples per class, while underrepresented have at least 1k but far less than 20k.

We use transfer learning to tackle the challenge (C2) in Section III-C. Transfer learning uses vulnerability classes in main training to augment the learning process, when new vulnerability classes (shown as Extension dataset in Fig. 6) are not sufficient for training or are highly imbalanced towards the main vulnerability classes. We show that a baseline model (without transfer learning) does not converge when trained using the Underrepresented Dataset (not enough data). However, the model trained using Transfer Learning can converge successfully. Hence, Transfer Learning allows us to cover vulnerability classes for which only a small amount of data is available.

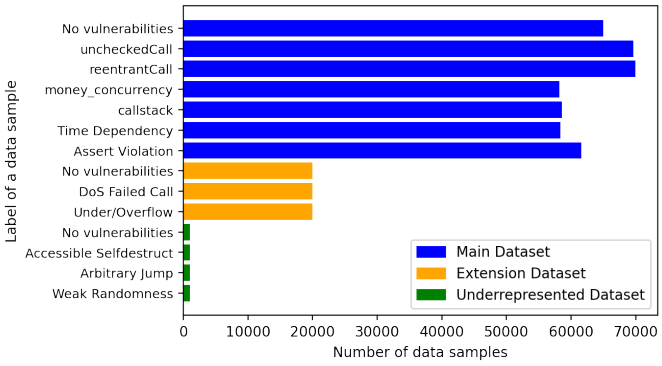
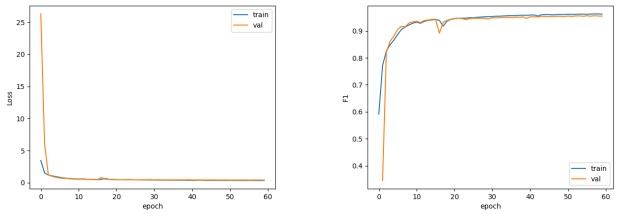


Fig. 6: Vulnerability types distribution.

It is worth mentioning that the actual size of our labeled main dataset is 279,726 samples instead of $60{,}000 \times (6+1)$ that one could expect. This is because ESCORT formulates vulnerability detection as *multi-label classification*, meaning that a contract might have multiple labels and repeatedly appear in the selection of several vulnerability types described above. Note that the labeled dataset is split into three sets of 80%, 10%, and 10% for training, validation, and testing, respectively. The definition of the evaluation metrics is provided in Appendix D.

B. Evaluation Results of Main Training

1) Classifier Learning: We train ESCORT's model on the training set with hyper-parameters listed in Table I and assess it on the test set (consisting of 27,973 contracts). To corroborate the detection effectiveness of ESCORT, we plot the learning curves of our multi-output DNN in Figure 7. The training and the validation curves demonstrate the *time-evolving* performance of ESCORT and the *generalization* capability of the model, respectively. The learning curves show that our MOL-DNN model can achieve an average F1 score higher than 97 % on both the training and validation set. This corroborates that ESCORT successfully learns the knowledge of multiple vulnerability types from the reference detection tools used in Demeter's data labelling.



(a) BCE loss. (b) F1 score.

Fig. 7: Learning process of ESCORT.

ESCORT is able to detect those vulnerabilities only by observing opcodes. For example, an integer overflow may be present, if the attacker controls a variable (i.e, function arguments are accessed via opcode) and the arithmetic operation has no bounds-checks (i.e., branching opcodes). For reentrancy, a state change may happen after a external call—both are visible as opcodes. Further, the issue with time dependency on bytecode level is the branching based on block values (i.e., timestamp), both of which are visible as opcodes.

2) Sensitivity to Training Configurations: ESCORT's training pipeline is described in Section VI-A. We use the MOL-DNN architecture in Figure 4 with six vulnerability branches in ESCORT's training phase. This model has a total number of 199,214 learnable parameters. We investigate whether the detection performance of ESCORT can be improved by increasing the number of layers, epochs, and learning rate. ESCORT's accuracy has no increase with a longer training time. Further, our model exploration did not yield any better performing models.

3) Class-wise Detection Performance: We describe ESCORT's overall performance on the vulnerability types introduced in Section VII-A. Here, we provide a fine-grained insight into ESCORT's capability on each vulnerability class. Table II shows the class-specific metrics obtained by our MOL-DNN model. ESCORT achieves an average of 94 % precision and 98 % F1 score across all six vulnerability types. In the case of the *Money Concurrency* vulnerability (cl. 2), ESCORT's

recall score is relatively lower than others. The low recall value indicates that ESCORT yields more false negatives on this vulnerability class according to Equation (1).

C. Extensibility Performance

We perform transfer learning on the trained main model by expanding branches as described in Section IV-C. Each new branch (for cl. 7-11) has 16,897 trainable parameters.

Transfer learning on well-represented data. For initial transfer learning, we use the same hyper-parameters listed in Table I and train the model on vulnerability types 7 and 8 simultaneously with 20,000 sample for each class. The detection result on the new vulnerability types are shown in the two columns cl. 7 and cl. 8 of Table II. It can be seen that ESCORT achieves 95 % and 94 % F1 score on the two new vulnerabilities cl. 7 and cl. 8, respectively. The empirical results corroborate that ESCORT is extensible to new attacks by enabling lightweight and effective transfer learning. We would like to point out that none of the existing detection techniques consider/solve the challenge of vulnerability extension, which is essential to ensure high attack coverage in the dynamic blockchain environments. ESCORT demonstrates fast and effective extension to new vulnerabilities for the first time.

Effect of transfer learning dataset size. To further evaluate ESCORT extensibility in a more practical scenario where new smart contract vulnerabilities are underrepresented, we gradually reduce the dataset size of transfer learning on cl. 7 and 8 by a certain percentage until we observe large performance degradation. Note that the extension dataset distribution in this new transfer learning experiment is shown in Figure 6 (orange color). Figure 8 shows how ESCORT performs in the transfer learning scenario when different amounts of new contracts with the new vulnerability type (cl.7, cl.8) are given to update our multi-output detector (detailed in Section IV-C). The y-axis is the F1 score of vulnerability detection. The vertical bars with different colors along the x-axis correspond to different transfer learning dataset sizes. One can observe that ESCORT can effectively identify new vulnerability types even when only 1,000 contracts (for cl. 7-8) are available for transfer learning. If only 500 samples are available per class, the performance of ESCORT degrades for cl.8.

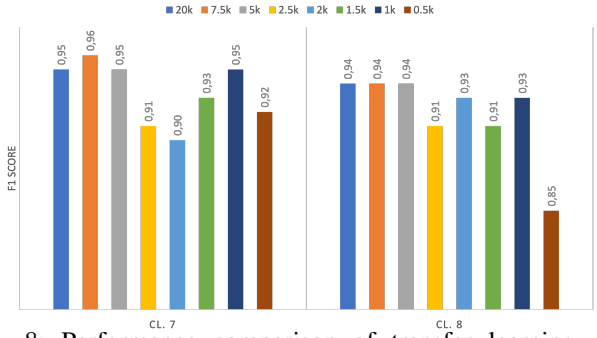


Fig. 8: Performance comparison of transfer learning with different dataset sizes.

We also demonstrate the superior *efficiency* of ESCORT's transfer learning compared to the existing ML-based detection techniques when extending to detect new vulnerabilities with limited samples. In this comparison experiment, we define a regular RNN consisting of the five-layered feature extractor and two four-layered branches shown in Figure 4 as the baseline model. This model is trained *from scratch* on vulnerability types cl.7 and cl.8 with limited data separately. For ESCORT, we expand the trained main model with two new branches and only update parameters in these two branches. We use 1,000 smart contracts in cl.7 and cl.8 in this experiment. This process takes only 1 h and 23 min, which is about half of the baseline model training time. We train the baseline model on cl. 7-8 with different hyper-parameters and observe that it does not converge on only 1,000 samples. This is because the limited data for new vulnerabilities (cl. 7-8) is not sufficient to learn all parameters in the baseline model.

Transfer learning on underrepresented data. To assess the superior *efficiency* on underrepresented data, we conducted further experiments on cl. 9-11. In Figure 6 we can see the underrepresented dataset, where the vulnerability classes have only around 1,000 samples (in green color). We applied our transfer learning approach again and added three new branches to our main model. As expected, we are able to achieve an average F1 score of 97 % on cl. 9-11. We similarly trained a baseline model with three branches. However, the baseline model also does not converge for cl. 9-11 with only 1,000 samples for each class.

D. Comparison with other ML approaches

For the comparison with other ML approaches, we use the labels of our dataset obtained by Maian [88] since the related work [58], [101] uses Maian to label their dataset. The dataset consists of 15k samples for each of the four classes labeled by Maian. We compare ESCORT against three papers: Color-inspired [65], Towards Sequential [101], and NLP-inspired [58].

Color-inspired. Since no open-source implementation or data is available for the color-inspired detection [65], we re-implemented the model and preprocessing based on the description in the paper. Here, we transform the bytecode of smart contract to RGB pictures and use a pre-trained CNN model (Inception-v3) to train on and classify contracts in our dataset. While the retraining of the classifier works, we achieved an F1 score of 40%. We hypothesize that the authors of [65] achieved a better performance due to a more homogeneous dataset.

Towards Sequential. We used our dataset to test the open-source model provided by Towards Sequential [101] and NLP-inspired [58] (both use the same dataset). First, we tested our dataset against Towards Sequential [101] and achieved an F1 score of 76% with their binary classifier. While this approach achieves better performance than Color-inspired [65] on the same test dataset, the performance of Towards Sequential is not as good as reported [101]. After investigating their training dataset, we found many duplicates were used in training and testing—resulting in poor performance.

NLP-inspired. Next, we test NLP-inspired [58]. While the authors use the same dataset as Towards Sequential [101], we achieved an F1 score of only 54%. We attribute the reduced performance to the over-sampling approach the authors took to generate more data for training. Consequently, the authors introduced bias impairs the general performance.

ESCORT. Last, we used the Maian-labeled dataset to train ESCORT. When we train from scratch, we achieve an average F1 score of 98%, showing superior performance.

Metrics	Initial vulnerability types						New Vuln. Classes				
	cl. 1	cl. 2	cl. 3	cl. 4	cl. 5	cl. 6	cl. 7	cl. 8	cl. 9	cl. 10	cl. 11
Precision	0.92	0.92	0.96	0.93	0.94	0.98	0.88	0.97	0.94	0.97	0.95
Recall	0.97	0.93	0.98	0.94	0.97	0.99	0.97	0.88	0.99	0.94	0.94
F1 score	0.98	0.97	0.99	0.97	0.98	0.99	0.94	0.95	0.98	0.97	0.96
FPR	0.02	0.02	0.01	0.02	0.02	0.01	0.07	0.02	0.03	0.01	0.02
FNR	0.03	0.07	0.02	0.06	0.03	0.01	0.03	0.12	0.00	0.06	0.06

TABLE II: Class-specific metrics (for unseen/test data) retrieved by our multi-output model for initial training phase (cl. 1-6) and after transfer learning phase (cl. 7-11).

When we apply our transfer learning, we achieve an average F1 score of 95%. While this is worse than training from scratch, we still see better performance than other works.

E. Runtime Performance

In this section, we evaluate the performance of our model serving API (c.f. Section VI-B1) and MetaMask Mobile (c.f. Section VI-B2). The API is assumed to run on a server, where a CPU and GPU may be available. We assume for the MetaMask Application a normal Android phone. We choose ten "random" contracts for PC and Android performance analysis. We use the median runtime as the efficiency metric.

API performance. Here, ESCORT is executed on CPU (with and without Tensorflow's Docker container) and a GPU. The performance on the CPU is different, whether the model is loaded via the host OS or the Tensorflow Docker container. Loading the model directly on the host yields an inference time of 1.5 s. When loading the model inside a docker container, the inference time is only a third with 0.5 s. The performance on the GPU is the fastest with 0.15 s.

MetaMask performance. Our MetaMask prototype is based on MetaMask v5.0.1 (877) and runs on a Nokia 7.1 with a Qualcomm Snapdragon 636 using Android 10. We ported ESCORT to Tensorflow Lite for Android (embedded in the MetaMask Application). The median runtime of our bytecode preprocessing and vulnerability type detection is 102.5 ms and 226.79 s, respectively. For time-sensitive classification tasks the user can switch to server-sided classification, which allows for drastic performance increases with the drawback of reduced privacy. Note that Tensorflow Lite currently does not support GPU acceleration of DNN inference.

Using *dumpsys*, a tool that runs on Android devices and provides information about the system, we measured the performance of our MetaMask prototype. We classified random contracts for about 9 minutes on the device as performance measurements. For a total runtime of 520.6 seconds, *dumpsys* measured a drain of 183.7 mAh (milliampere-hour). Considering the median classification time of a contract, we calculated an average battery drain per contract of around 80 mAh or 2.62% of the capacity. Furthermore, the RAM consumption was continuously at 0.6 GB and the CPU utilization remained at around 20 %. In consequence, we assume that the resource utilization, especially battery usage, of our prototype is within accepted bounds.

F. Ground Truth Analysis

In this section, we first explain our data acquisition, statistics of the labeling tools (i.e., Mythril, Vandal, and Oyente) and software aging of smart contracts. We then show the performance comparison between the labeling tools and ESCORT against ground truth.

Data acquisition. Ground truth data is notoriously hard to obtain as one need to invest a substantial amount of time for correct labeling. For this task, we base our analysis on professional security audits of smart contracts. More specifically, we use the audit repositories of Quantstamp [24], OpenZeppelin [21], Trail of Bits [35], ConsenSys [2], and CertiK [1]. Those audits are publicly available. To match the vulnerability classes in those audits with the ones in our setup, we went through the audits, and generated a ground truth dataset of 373 smart contract projects in source code for classes 2, 4, 5, 6, and 8. The inspected audits do not include any samples of the remaining vulnerability classes.

Compilation and analysis. Since ESCORT operates on contract bytecodes, we compiled the obtained source code of the smart contracts. We found that 156 smart contract projects from the audits cannot be compiled because of software aging (i.e., libraries or repositories are not available anymore). Among the rest of 217 projects (containing 268 smart contracts), Vandal fails to complete analysis for 104 and Oyente failed for 8 smart contracts. Mythril completed testing of all smart contracts. All 268 contracts from the ground-truth dataset are used for the analysis of all three tools. If Oyente and Vandal failed analysis, respective contracts are considered non-vulnerable (since they fail to detect any vulnerability).

Detection performance. We analyzed the performance of ESCORT and three detection tools used by Demeter on the compiled ground truth smart contracts and summarize the results in Table III. Recall that we show the mapping Vulnerability class $\rightarrow$ Tool (i.e., which detection tool was used for which vulnerability class) in Section VII-A. Table III summarizes the F1 score for each vulnerability of its respective labeling tool. It is worth noting that we did not calculate the average of F1 scores since each score only applies to one tool. We can see that ESCORT has a similar or better performance compared with the original tools used when labeling the vulnerability classes except for cl. 5.

Metrics	cl. 2	cl. 4	cl. 5	cl. 6	cl. 8
Underlying tool	Oyente	Vandal	Vandal	Mythril	Mythril
Positive samples	8	64	150	10	21
F1 tools	0.96	0.53	0.47	0.89	0.83
F1 ESCORT	0.96	0.73	0.31	0.90	0.88

TABLE III: Class-specific metrics for ground truth data.

We attribute the poor performance of ESCORT on cl.5 in Table III to two factors: *(i) Coding style changes over time.* For example, developers used exceptions to check return values in older contracts instead of "require()" or applied "require()" and "assert()" interchangeably. *(ii) The "REVERT" opcode was introduced and changed the function-to-opcode mapping.* Since our main dataset of 3.6mln smart contracts includes mostly older contracts, ESCORT learns older opcode mappings and coding styles. In contrast, the ground truth dataset mostly consists of recent smart contracts. We attribute the better performance in the other classes to the ability of ESCORT

to generalize attack patterns better than the underlying tools. In-depth manual analysis of mis-classified cases on the main dataset is non-trivial due to a large number of samples to explore (255 for cl.7 and 235 for cl.8), the absence of source codes for these contracts, and the fact that AI results are not trivially explainable.

VIII. Related Work

We discuss existing ML-based vulnerability detection methods in this section and provide a more detailed categorization of other related works in Appendix E.

ContractWard. ContractWard detects smart contracts vulnerability in the *opcode-level* by extracting *bigram* features from the simplified opcode and training individual binary ML classifiers for each vulnerability class [107]. It targets six vulnerabilities and experiments with Random Forests, K-Nearest Neighbors, SVM, AdaBoost, and XGBoost classifiers. Compare to our work, ContractWard has three main limitations: *(i) Requires opcode of smart contracts*. Obtaining opcode is non-trivial since smart contracts are typically stored in bytecodes on the blockchain. *(ii) Not extensible to new exploitation attacks*. It requires to build and train a new ML model for each new vulnerability type. *(iii) Not scalable to long contracts* since the *bigram* language model has a short window size.

LSTM-based. In [101] the authors proposes a sequence learning approach to detect weaknesses in the opcode of smart contracts. Particularly, it uses one-hot encoding and an embedding matrix to represent the contract's opcode. The obtained code vectors are used as input to train an LSTM model for determining whether the given smart contract is safe or vulnerable (i.e., binary classification). Compared to ESCORT, the paper [101] yields limited detection performance: (i) The reported F1 score of 86% is relatively low. We hypothesize the reason is that different vulnerability types have diverse behaviors, thus making it hard to distinguish the group of multiple vulnerabilities from the safe contracts. (ii) The LSTM model only provides a binary decision about contract security without distinguishing vulnerability types. Another work [64] formulates vulnerability detection as a multi-task learning problem [94], [112] and adjusts model architecture for each vulnerability type. Although [64] also works on the bytecode-level, this paper does not address the scalability challenge compared with ESCORT. Furthermore, [64] uses a single detection tool to label the contracts for training and assumes the labels are correct. This assumption is too strong and the ground-truth labels are not reliable. Our Demeter tool incorporates three different detection techniques and uses the most reliable one for each vulnerability type when constructing the labeled dataset (Section V-C), thus increasing the reliability of the obtained label.

AWD-LSTM based. The work in [82] adapts 'Average Stochastic Gradient Descent Weighted Dropped LSTM' (AWD-LSTM) for multi-class vulnerability detection. The proposed model consists of two parts: a pre-trained encoder for language tasks [63], and an LSTM-based classifier for vulnerability classification. This method works on the opcode-level and can detect three vulnerability types. Compared to ESCORT, the AWD-LSTM based detection method has the following constraints: *(i) Non-uniform effectiveness.* The multi-class detection performance of [58] is not uniformly effective across different vulnerabilities. In particular, [58] yields an F1 score of 95% on safe contracts and 30% on Prodigal contracts [89]. ESCORT features a much smaller performance divergence across different classes as can be seen from Table II. *(ii) Not extensible.* The AWD-LSTM based model [58] is a fixed design to detect pre-specified vulnerability types. The extension to incorporate new attacks is not considered in [58].

CNN-based. In [65], the proposed solution transforms the contract bytecode into fix-sized RGB color images and trains a convolution neural network for vulnerability detection. Similarly to ESCORT, the CNN-based classifier uses multi-label classification, which has a low confidence score when determining the exact vulnerability types. ABCNN [100] proposes a multinomial classification CNN model with self-attention mechanism to improve the detection performance. Compare to our work, these CNN-based detection schemes have the following limitations: (i) The multi-label classification performance is not satisfying due to its low confidence level. We hypothesize that this is because image representation of the bytecode and the CNN architecture ignore the sequential information existing in the contract. (ii) The extensibility/-generalization ability of the CNN-based detection method is neither discussed nor evaluated.

GNN-based. The solution in [114] proposes a graph neural network (GNN)-based approach. In particular, it builds a contract graph from the contract's source code where nodes and edges represent critical function calls/variables and temporal execution trace, respectively. This graph is normalized and passed to a temporal message propagation network for vulnerability detection. While supporting multi-class detection, the work [114] has the following drawbacks: first, they operates on contract source code, which is typically hard to obtain from the public blockchain; second, it has limited effectiveness, as it yields an average F1 score of 77% across all three vulnerabilities. We hypothesise that graph normalization in [114] does not preserve the malicious nodes responsible for vulnerability exploitation, leading to the low F1 score. The follow-up works [78], [79] extend GNN-based vulnerability detection by integrating expert-defined security patterns into the graph representation, thus providing explainability for the detection. However, these two detection methods do not solve the scalability challenge. Furthermore, they operate at the source-code level, thus only apply to smart contracts with public available source codes. Another source-level tool Peculiar [109] adapts the pre-trained GraphCodeBERT [61] as its model and extracts critical data flow graph from contract's source code for vulnerability detection. Peculiar demonstrates its effectiveness on the reentrancy bug only and does not support detection of other vulnerabilities.

IX. Conclusion

We present ESCORT, the first transfer learning-friendly vulnerability detection framework for smart contracts. Our tool is extensible to new vulnerability types using limited data with minimal model modification and re-training overhead. ESCORT can identify different vulnerability types in a single run and achieves effective (F1 score 97%) and efficient detection (0.15 second per contract). In contrast to previous ML-based vulnerability detection methods, ESCORT does not require source code access, supports concurrent detection of multiple vulnerability types, and is the first framework that allows

vulnerability type extension via transfer learning. Compared to non-ML techniques, ESCORT is a unified and automated framework that fully covers a smart contract with arbitrary complexity. In addition, ESCORT provides concurrent detection of various vulnerabilities through a single run, thus incurs much smaller deployment overhead compared to applying multiple non-ML based tools. ESCORT's multi-output RNN design is highly modular, scalable, efficient, and extensible. As a separate contribution, we devise Demeter, a toolchain for dataset construction. We will open-source Demeter and our dataset to promote research in this area.

ACKNOWLEDGMENT

We would like to thank Intel Private AI center and BMBF for their support of this research.

REFERENCES

[1] "CertiK Audits," Accessed 2022. [Online]. Available: https://www.certik.com/

[2] "ConsenSys Audits," Accessed 2022. [Online]. Available: https://consensys.net/diligence/audits/

[3] "Erigon," Accessed 2022. [Online]. Available: https://github.com/ledgerwatch/erigon

[4] "Ethereum Contract Library by Dedaub," Accessed 2022. [Online]. Available: https://library.dedaub.com/

[5] "Ethereum Whitepaper," Accessed 2022. [Online]. Available: https://ethereum.org/en/whitepaper/

[6] "Flask API," Accessed 2022. [Online]. Available: https://flask.palletsprojects.com/

[7] "Geth," Accessed 2022. [Online]. Available: https://geth.ethereum.org/

[8] "Google BigQuery," Accessed 2022. [Online]. Available: https://cloud.google.com/bigquery

[9] "Google Machine Learning Glossary," Accessed 2022. [Online]. Available: https://developers.google.com/machine-learning/glossary

[10] "Hyperledger project," Accessed 2022. [Online]. Available: https://www.hyperledger.org/

[11] "Introduction — Web3.py 5.12.1 documentation," Accessed 2022. [Online]. Available: https://web3py.readthedocs.io/en/stable/

[12] "Keras: Multiple outputs and multiple losses," Accessed 2022. [Online]. Available: https://www.pyimagesearch.com/2018/06/04/keras-multiple-outputs-and-multiple-losses/

[13] "Manticore," Accessed 2022. [Online]. Available: https://github.com/trailofbits/manticore

[14] "Metamask," Accessed 2022. [Online]. Available: https://github.com/MetaMask/metamask-mobile

[15] "Multi-Class Metrics Made Simple, Part II: the F1-score," Accessed 2022. [Online]. Available: https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1

[16] "MySQL," Accessed 2022. [Online]. Available: https://www.mysql.com/

[17] "Mythril," Accessed 2022. [Online]. Available: https://github.com/ConsenSys/mythril

[18] "MythX: Smart contract security service for Ethereum," Accessed 2022. [Online]. Available: https://mythx.io/

[19] "Mythx tech: Behind the scenes of smart contract security analysis," Accessed 2022. [Online]. Available: https://blog.mythx.io/features/mythx-tech-behind-the-scenes-of-smart-contract-analysis/

[20] "Networks — ethereum.org," Accessed 2022. [Online]. Available: https://ethereum.org/en/developers/docs/networks/

[21] "OpenZeppelin Security Audits," Accessed 2022. [Online]. Available: https://blog.openzeppelin.com/security-audits/

[22] "Overview · Smart Contract Weakness Classification and Test Cases," Accessed 2022. [Online]. Available: http://swcregistry.io/

[23] "Oyente: An analysis tool for smart contracts," Accessed 2022. [Online]. Available: https://github.com/melonproject/oyente

[24] "Quantstamp Security Audits," Accessed 2022. [Online]. Available: https://certificate.quantstamp.com/

[25] "Quorum," Accessed 2022. [Online]. Available: https://consensys.net/quorum/

[26] "Rootstock (RSK) platform for smart contracts," Accessed 2022. [Online]. Available: https://en.bitcoinwiki.org/wiki/Rootstock

[27] "scikit-learn: F-measure," Accessed 2022. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

[28] "Serpent assembly programming language," Accessed 2022. [Online]. Available: https://github.com/ethereum/serpent

[29] "The Solidity Contract-Oriented Programming Language," Accessed 2022. [Online]. Available: https://github.com/ethereum/solidity

[30] "TensorFlow," Accessed 2022. [Online]. Available: https://www.tensorflow.org/

[31] "Testnet Goerli," Accessed 2022. [Online]. Available: https://goerli.net/

[32] "Testnet Kovan," Accessed 2022. [Online]. Available: https://kovan-testnet.github.io/website/

[33] "Testnet Rinkeby," Accessed 2022. [Online]. Available: https://www.rinkeby.io/

[34] "Testnet Ropsten," Accessed 2022. [Online]. Available: https://ropsten.etherscan.io/

[35] "Trail of Bits Security Audits," Accessed 2022. [Online]. Available: https://github.com/trailofbits/publications/tree/master/reviews

[36] "Tron: Decentrailze the web." Accessed 2022. [Online]. Available: https://tron.network/

[37] "Vechain," Accessed 2022. [Online]. Available: https://www.vechain.com/

[38] "Vyper documentation," Accessed 2022. [Online]. Available: https://vyper.readthedocs.io/en/stable/

[39] "What are smart contracts on blockchain?" Accessed 2022. [Online]. Available: https://www.ibm.com/topics/smart-contracts

[40] "XML Path Language (XPath)," Accessed 2022. [Online]. Available: https://www.w3.org/TR/xpath20/

[41] E. Alpaydin, *Introduction to Machine Learning*. MIT press, 2020.

[42] L. Alt and C. Reitwießner, "SMT-Based Verification of Solidity Smart Contracts," in *International Symposium on Leveraging Applications of Formal Methods*, 2018.

[43] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *International Conference on Principles of Security and Trust*, 2017.

[44] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities," in *PLDI*, 2020.

[45] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[46] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-Based n-gram Models of Natural Language," *Computational linguistics*, 1992.

[47] J. Cai, S. Yang, J. Men, and J. He, "Automatic software vulnerability detection based on guided deep fuzzing," in *International Conference on Software Engineering and Service Science*, 2014.

[48] M. d. Castillo, "The DAO Attacked: Code Issue Leads to $60 Million Ether Theft," Accessed 2022. [Online]. Available: https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft

[49] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He *et al.*, "Soda: A generic online detection framework for smart contracts." in *NDSS*, 2020.

[50] X. Chen, S. Wang, B. Fu, M. Long, and J. Wang, "Catastrophic forgetting meets negative transfer: Batch spectral shrinkage for safe transfer learning," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[51] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, 2016.

[52] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," *arXiv preprint arXiv:1608.00771*, 2016.

[53] T. Cook, A. Latham, and J. H. Lee, "DappGuard : Active Monitoring and Defense for Solidity Smart Contracts," 2017.

[54] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International Conference on Financial Cryptography and Data Security*, 2016.

[55] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework For Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019.

[56] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "EVMFuzzer: detect EVM vulnerabilities via fuzz testing," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[57] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, 2012.

[58] A. K. Gogineni, S. Swayamjyoti, D. Sahoo, K. K. Sahu *et al.*, "Multi-class classification of vulnerabilities in Smart Contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing," *arXiv preprint arXiv:2004.00362*, 2020.

[59] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: surviving out-of-gas conditions in ethereum smart contracts," *ACM on Programming Languages*, 2018.

[60] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts," in *SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[61] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[62] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts," in *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[63] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[64] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart contract vulnerability detection model based on multi-task learning," *Sensors*, vol. 22, no. 5, p. 1829, 2022.

[65] T. H.-D. Huang, "Hunting the ethereum smart contract: Color-inspired inspection of potential attacks," *arXiv preprint arXiv:1807.01868*, 2018.

[66] N. Jaipuria, X. Zhang, R. Bhasin, M. Arafa, P. Chakravarty, S. Shrivastava, S. Manglani, and V. N. Murali, "Deflating dataset bias using synthetic data augmentation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2020, pp. 772–773.

[67] C. Jentzsch, "Decentralized autonomous organization to automate governance," *White paper, November*, 2016.

[68] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[69] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[70] J. Kim, M. El-Khamy, and J. Lee, "Residual LSTM: Design of a Deep Recurrent Architecture for Distant Speech Recognition," *arXiv preprint arXiv:1701.03360*, 2017.

[71] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 1976.

[72] S. Kiyono, J. Suzuki, and K. Inui, "Mixture of expert/imitator networks: Scalable semi-supervised learning framework," in *AAAI Conference on Artificial Intelligence*, 2019.

[73] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[74] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, 2015.

[75] S.-W. Lee, J.-H. Kim, J. Jun, J.-W. Ha, and B.-T. Zhang, "Overcoming catastrophic forgetting by incremental moment matching," *Advances in neural information processing systems*, vol. 30, 2017.

[76] K. Leino, M. Fredrikson, E. Black, S. Sen, and A. Datta, "Feature-wise bias amplification," *ArXiv*, vol. abs/1812.08999, 2019.

[77] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018.

[78] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in *IJCAI*, 2021, pp. 2751–2759.

[79] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[80] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, "NeuCheck: A more practical Ethereum smart contract security analysis tool," *Software: Practice and Experience*, 2019.

[81] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *ACM SIGSAC conference on computer and communications security*, 2016.

[82] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," *arXiv preprint arXiv:1708.02182*, 2017.

[83] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts," *arXiv preprint arXiv:1907.03890*, 2019.

[84] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," in *9th Annual HITB Security Conference (HITBSecConf)*, 2018, p. 54.

[85] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Accessed 2022. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[86] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts," *arXiv preprint arXiv:2004.08563*, 2020.

[87] I. Nikolić, "Maian," Accessed 2022. [Online]. Available: https://github.com/ivicanikolicsg/MAIAN

[88] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *Annual Computer Security Applications Conference*, 2018.

[89] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Annual Computer Security Applications Conference*, 2018.

[90] G. Pace and J. Ellul, "Runtime Verification of Ethereum Smart Contracts," in *2018 14th European Dependable Computing Conference (EDCC)*, 2018.

[91] D. M. Powers, "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[92] R. Price, "Digital currency Ethereum is cratering because of a $50 million hack," Accessed 2022. [Online]. Available: https://www.businessinsider.com/dao-hacked-ethereum-crashing-in-value-tens-of-millions-allegedly-stolen-2016-6

[93] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[94] S. Ruder, "An overview of multi-task learning in deep neural networks," *arXiv preprint arXiv:1706.05098*, 2017.

[95] A. Savelyev, "Contract law 2.0: 'Smart' contracts as the beginning of the end of classic contract law," *Information & Communications Technology Law*, 2017.

[96] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, *eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts*, 2020.

[97] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*, 2010.

[98] D. Siegel, "Understanding the DAO Attack," Accessed 2022. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[99] M. Suiche, "The $280M Ethereum's Parity bug," Accessed 2022. [Online]. Available: https://www.comae.com/posts/the-280m-ethereums-parity-bug./

[100] Y. Sun and L. Gu, "Attention-based machine learning model for smart contract vulnerability detection," in *Journal of Physics: Conference Series*, vol. 1820, no. 1. IOP Publishing, 2021, p. 012004.

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

[102] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of Ethereum smart contracts," in *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[103] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Annual Computer Security Applications Conference*, 2018.

[104] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *ACM Sigplan Notices*, 2009.

[105] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[106] H. M. Wallach, "Topic modeling: beyond bag-of-words," in *International Conference on Machine Learning*, 2006.

[107] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, 2020.

[108] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[109] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE*, 2021, pp. 378–389.

[110] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[111] H. Zamani and W. B. Croft, "Relevance-based word embedding," in *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017.

[112] Y. Zhang and Q. Yang, "A survey on multi-task learning," *IEEE Transactions on Knowledge and Data Engineering*, 2021.

[113] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *2017 IEEE international congress on big data (BigData congress)*, 2017.

[114] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural networks," 2020.

APPENDIX

In this section, we provide a detailed introduction about cryptocurrency systems, detailed background on deep learning as well as more related works.

A. Ethereum Platform

Blockchain is proposed as a distributed ledger that records transactions between two parties in a verifiable and permanent way [113]. Ethereum is an open-sourced cryptocurrency platform based on blockchain and provides a Turing-complete Ethereum Virtual Machine (EVM) that enables developers to deploy decentralized applications. A cryptocurrency platform has the following key characteristics:

Decentralized Nature. In contrast to conventional currencies, virtual money is not administered by a central authority but by a distributed peer-to-peer network. A network of nodes, the so-called *'miners'*, are responsible to perform money transactions, data storage, and updates [5]. Note that on a blockchain, all code, data, and transactions are shared and available for inspection on every single node. All actions performed inside of this network need to be confirmed by the majority of all participating nodes [95].

Mathematical Algorithm as a Basis of Cryptocurrency Value. Money, in the Ethereum context called Ether, can be initially earned by solving a complex mathematical problem that can be accepted by the other nodes. The process is called *mining*. Once a transaction is triggered, a state value for the new block is calculated and verified by the participating nodes in the network [5]. The amount of money available in the network is also ensured to be limited. As such, the Ether and the associated owners can be tracked at all times [54].

Resilience to Data Manipulations from Outside. The information of the mined money is stored in a public data structure, i.e., the blockchain. More specifically, modifications and transactions are stored in blocks that are appended to the chain in a chronological order after checking its validity [81]. Afterward, the network rejects any attempts to alter the blockchain entries. Therefore, the data is immutable and irreversible [54].

Pseudonymous Nature. In general, registration is not required to use cryptocurrency. Users that perform transactions inside the network are identified by a public key and a private key. All transactions are associated with the addresses instead of explicit users [5]. Therefore, it is hard to determine the identity of a user although all transactions are stored publicly on the blockchain.

B. Deep Learning

ESCORT operates on contract bytecode, which can be considered as a special case of text data. We briefly present the background of text representation and recurrent neural networks below.

Text Representation. The text modality is typically transformed into numerical vectors for usage in ML algorithms. This transformation can be realized in different ways, such as a bag of words [106], n-gram language model [46], and embedding layer [111]. The numerical vectors converted from the text data are then used as the direct input to the DL models.

Recurrent Neural Network. An RNN is a category of DNNs where the connections between neurons construct a direct computational graph along the temporal sequence [70]. A key property of the RNN is that the model can use its internal states as memory cells to store the knowledge about prior inputs, thus capturing the contextual information in sequential inputs (e.g., text document). As shown in Figure 9, the hidden states obtained from the previous input (s_t) affects the output in the current time step (o_{t+1}). The unfolded RNN diagram reveals the *'parameter sharing'* mechanism of RNNs where

the weight matrices (W, V, and U) are shared across different time steps. Parameter sharing makes RNNs generalizable to unseen sequences of different lengths.

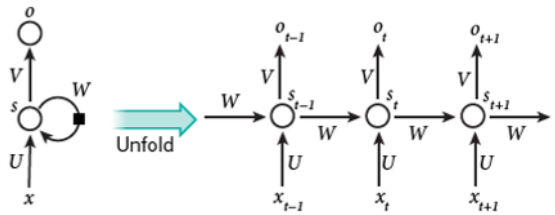


Fig. 9: A recurrent neural network and its unfolding in time [74]. The computation of the forward pass is shown.

C. Machine Learning-based Classification

Multi-label vs. Multi-class Classification. We introduce two types of ML architectures for classification tasks here and show how ESCORT leverages multi-label topology for concurrent vulnerability detection in Section IV. Smart contracts vulnerability detection can be realized with two different paradigms. The first one is known as *multi-class classification*, which refers to the case where the classification task has more than two classes that are mutually *exclusive*. In particular, each sample is assigned to one and only one class label. The second one, *multi-label classification*, also involves multiple classes while a data sample can have more than one associated labels. This is because the classes in multi-label tasks describe non-exclusive attributes of the input (e.g., color and length).

Multi-class vs. Multi-label Model Let us use an example to illustrate the difference between these two paradigms. Given a clothing dataset with three colors (black, blue, red) and four categories (jeans, dress, shirt, shoes), we want to train a model to predict these two clothing attributes simultaneously. Figure 10 shows the architecture of the multi-class and multi-label formulation of the clothing classification task. The multi-class design has only one set of dense layers (i.e., 'heads') at the bottom of the DNN where the last dense layer has $3 \times 4 = 12$ neurons. The network topology for multi-label classification has two sets of dense heads at the end of the DNN where the last dense layer in each branch has 4 and 3 neurons to learn the clothing category and color attribute, respectively. We call the network design of the *multi-label* classification with multiple sets of dense heads as *'multi-output'* architecture throughout this paper. The 'stem-branch' topology makes the multi-output architecture extensible to learn new attributes. ESCORT leverages this observation to devise an efficient and extensible smart contract inspection solution.

D. Evaluation Metrics

ESCORT provides concurrent detection of multiple vulnerability types. We evaluate the performance of ESCORT's multi-output DNN model with F1 score, precision, and recall.

Base Values. The results of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) are the base values to compute other metrics. The true values represent the number of correctly predicted results, which can be either true positive or true negative. The false values indicate that the DL model gives the wrong outputs [9].

Precision and Recall. The precision metric describes the ratio of truly positive values to all positive predictions. This indicates the reliability of the classifier's positive prediction [91].

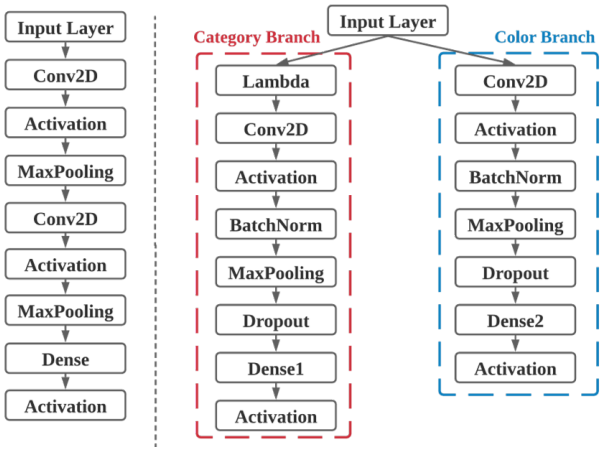


Fig. 10: DNN architectures of multi-class (left) and multi-label (right) formulation of clothing classification task [12].

The recall (or sensitivity) metric shows the proportion of actual positives that are correctly classified. The formulas to compute these two metrics are given below:

$$Precision = \frac{TP}{TP + FP}, \; Recall = \frac{TP}{TP + FN}. \quad (1)$$

F1 Score. The F1 score metric is commonly used in information retrieval and it quantifies the overall decision accuracy using precision and recall. The F1 score is defined as the *harmonic mean* of the precision and recall:

$$F1\_score = \frac{2 * Precision * Recall}{Precision + Recall}. \quad (2)$$

The best and the worst value of the F1 score is 1 and 0, respectively. The F1 score can be calculated for each class label or globally [27]. We use the weighted F1 score where the per-class F1 scores are weighted by the number of samples from that class [15].

E. Detailed Related Works

Table IV summarizes the qualitative comparison between ESCORT and representative counterparts. We discuss more detailed related works below.

1) Static Vulnerability Detection Methods: Static detection techniques analyze the smart contract in a static environment by examining its source code or bytecode. Main limitations of static detection method are expert formalisation of vulnerabilities, code coverage and detection time.

Information Flow Analysis-based. Slither [55] uses *taint analysis* [104] to detect vulnerabilities in Solidity source code. It can find nearly all vulnerabilities related to the user inputs or critical data flows while the inspection time might be prohibitively long. Dedaub's Contract-Library by Dedaub [4] provides multiple different features via an online API. It collects bytecode of the smart contracts and performs vulnerability classifications using tool MadMax [59] that performs flow and loop analysis to detect gas-focused vulnerabilities [44].

Symbolic Execution-based. Oyente [81] detects vulnerabilities in the source code or bytecode of Solidity contracts using *symbolic execution*. Symbolic execution represents the

Name	Method	Capability	Extensible	Required Input
This work	ML (GRU) + TL	Multi-label	Yes	Bytecode
Oyente [81]	Symbolic execution	Multi-class	No	Source and bytecode
Mythril [19]	Symbolic execution, TA, and SMT	Multi-class	No	Bytecode
Dedaub [4]	Flow and loop analysis	Gas-focused vulnerability	No	Source code
Securify [105]	Symbolic analysis	Binary decision	No	Bytecode
Vandal [45]	Static program analysis	Multi-class	No	Bytecode
ContractWard [107]	ML (bigram model)	Binary decision	No	Opcode
Towards Sequential [101]	ML (LSTM)	Binary decision	No	Opcode
NLP-inspried [58]	ML (AWD-LSTM)	Multi-class	No	Opcode
Color-inspried [65]	ML (CNN)	Multi-label	No	Bytecode
Graph NN-based [114]	ML (GNN)	Multi-class	No	Source code

TL: Transfer Learning, TA: Taint Analysis

TABLE IV: Qualitative comparison of ESCORT and existing smart contract vulnerability detection methods.

program's behavior as built formula and uses symbolic inputs to decide if a certain path can be reached [71]. As such, its performance depends on the number of explored paths and the program's complexity [47], [97]. Oyente constructs the control flow graph of the contract and uses it to create inputs for symbolic execution. Manticore [83] analyzes the contract by repeatedly executing symbolic transactions against the bytecode, tracking the discovered states, and verifying code invariants [13]. Securify [105] first obtains the contract's semantic information by performing symbolic analysis of the dependency graph, then checks the predefined compliance and violation patterns for vulnerability detection. teEther [73] searches for certain critical paths in the control flow graph of the smart contract and uses symbolic execution for vulnerability identification.

Logic Rules-based. Vandal [45] is a logic-driven static program analysis framework. It converts the low-level EVM bytecode to semantic logic relations and describes the security analysis problems with logic rules. The datalog engine executes the specifications for input relations and outputs the vulnerabilities. eThor [96] is a static analysis technique built on reachability analysis achieved by Horn clause resolution. NeuCheck [80] adopts a syntax tree in a syntactical analyzer to transform source code of smart contracts to an intermediate representation (IR). Vulnerabilities are identified by searching for detection patterns in the syntax tree. SmartCheck [102] converts the Solidity source code to XML-based IR and verifies it against detection patterns defined in XPath language [40].

Composite Methods-based. Mythril [19] combines multiple vulnerability detection approaches, including symbolic execution, taint analysis, and Satisfiability Modulo Theories (SMT). SMT solving converts the contract to SMT constraints to reveal program flaws. Zeus [69] uses symbolic model checking, abstract interpretation, and constrained horn clauses to verify contracts' security. Osiris [103] combines symbolic execution and taint analysis to precisely identify integer bugs in smart contracts.

2) Dynamic Vulnerability Detection Methods: Dynamic testing techniques execute the program and observe its behaviors to determine the vulnerability's existence. The main disadvantage of dynamic detection is that designing testing inputs to yield high software coverage is difficult.

Fuzzing-based. MythX [18] combines synthetic execution and *code fuzzing*. It provides a cloud-based API for developers to inspect smart contracts. Fuzzing [57] is a testing method that attempts to expose the vulnerabilities by executing the program with invalid, unexpected, or random inputs. The brute-force nature determines that fuzzing incurs large runtime overhead and might have poor code coverage due to its dependency on the inputs [47]. ReGuard [77] is another fuzzing tool specialized in the Reentrancy bug. It creates an IR for the smart contract. A fuzzing engine is used to generate random byte inputs and analyze the execution traces for reentrancy bugs detection. ContractFuzzer [68] generates fuzzing inputs based on the ABI specifications of smart contracts. Test oracles are defined to monitor and analyze the contract's runtime behaviors for vulnerability detection. Echidna [60] is a fuzzer that generates random tests to detect violations in assertions and custom properties. ILF [62] uses symbolic execution to generate contract inputs and employs imitation learning to design a neural network-based fuzzer from symbolic execution. sFuzz [86] is an adaptive fuzzer for smart contracts that combines the AFL fuzzer and multi-objective strategy to explore hard-to-cover branches. Harvey [110] is a greybox fuzzer that predicts new inputs to cover new paths and fuzzes the transaction sequence in a demand-driven manner.

Validation-based. ContractLarva [90] is a runtime verification tool for smart contracts where a violation of defined properties can lead to various handling strategies, such as a system stop. These properties can include undesired event traces of control or data flow. Maian [87], [88] combines symbolic analysis and concrete validation to inspect the smart contract's bytecode. In concrete validation, the contract is executed on a fork of Ethereum for tracing and validation. By passing symbolic inputs to the contract, the execution trace is analyzed to identify the vulnerabilities. Sereum [93] uses runtime monitoring and verification to protect existing smart contracts against reentrancy attacks without modifications or semantic knowledge of the contracts. It detects inconsistent states in the contract via dynamic taint tracking and data flow monitoring during contract execution. SODA [49] is a general online framework that detects various attacks exploiting different smart contract vulnerabilities. SODA can be integrated as a full node into EVM-compatible blockchains and allows users to develop attack detection apps using the real-time monitored information. While SOTA can perform online detection, as a non-ML based approach, it requires human expertise to identify insecure patterns. In contrast, our ML-based ESCORT framework supports runtime analysis and learns vulnerable features automatically.