On the Use of Deep Neural Networks for Security Vulnerabilities Detection in Smart Contracts

Martina Rossini
University of Bologna
Bologna, Italy
martina.rossini3@studio.unibo.it

Mirco Zichichi
Universidad Politécnica de Madrid
Madrid, Spain
mirko.zichichi@upm.es

Stefano Ferretti
University of Urbino Carlo Bo
Urbino, Italy
stefano.ferretti@uniurb.it

***Abstract*—In this paper, we investigate the use of deep learning techniques to identify and classify smart contract code vulnerabilities. We collected a large-scale dataset of smart contracts that we used to train different Convolutional Neural Networks (CNNs) models. In particular, we used two variants of 2-dimensional CNNs working on RGB images corresponding to contract bytecode, a 1-dimensional CNN working on the bytecode directly, and a Long Short-Term Memory (LSTM) neural network. Given a set of vulnerability detectors, we employed five classes of vulnerabilities. Our results show that CNNs provide a good level of accuracy and demonstrate the viability of using deep learning techniques to identify smart contract vulnerabilities.**

***Index Terms*—smart contract, code vulnerability, blockchain, deep learning, convolutional neural networks**

I. Introduction

Smart contracts gained momentum in recent years. The general interest concerns their unique features, such as immutability, being executed in blockchains, and automatic enforceability. This has led to lively research activity regarding the development of tools for building contracts, as well as careful analysis comparing smart contracts to legal contracts. A recent European Union regulation proposal confirms this general interest [1]. In particular, in the Data Act proposal, smart contracts are defined as programs on digital ledgers that execute and settle transactions based on pre-determined conditions. They are considered a system to promote data sharing, providing data holders and recipients with guarantees that conditions for sharing data are respected [2]. However, this need for "guarantees" raises smart contract security concerns that need to be solved [3].

At the time of writing, the most popular blockchain platform to run smart contracts is Ethereum. Ethereum's smart contracts are written in Solidity, a Turing complete programming language. This allows blockchain developers to implement complex business logic solutions and to develop decentralized applications (dApps) [4]–[6]. However, it also allows for a bigger chance of bugs and code vulnerabilities. This is a severe issue since these problems cannot be patched after deployment due to the immutable nature of the ledger [7]. Thus, a developer should check potentially vulnerable pieces of code before deploying its contracts: this is often done by comparing against security patterns, which help ensure that the code is reliable but is expert-made and expensive to produce. Automated vulnerability detection methods only focus on known bugs and are too time-consuming or rely on expert-made detection rules. For this reason, some machine learning and deep learning-based techniques have been proposed, making it possible to not rely on a heavy feature engineering phase [8].

This paper explores deep learning techniques, particularly Convolutional Neural Networks (CNNs), for detecting and classifying vulnerabilities in smart contracts deployed on the Ethereum main net. Deep Learning techniques based on CNNs have long since shown promising results in malware detection and classification [8], [9]. Our study compares four different types of neural networks, i.e., a baseline LSTM, a ResNet 1D CNN, and two 2D CNNs (ResNet-18 and Inception v3) that work on the code bytecode transformed into an RGB image. We provide an in-depth analysis of these techniques to classify a dataset of smart contracts we have collected. To classify our smart contracts, we first employed a set of vulnerability detectors that we then mapped to 5 different classes, i.e., access-control, arithmetic, reentrancy, unchecked calls, and others.

Results show the viability of the proposal as a promising technique to automatically assess smart contracts' correctness and classify their potential vulnerabilities. In particular, according to the specific configuration and the available computational capabilities used during the tests, the ResNet 1D CNN working directly on the smart contract bytecode seems to offer the best results in terms of classification capabilities. Moreover, due to the unbalanced sizes of the different classes, the classification resulted in more effectiveness for the *unchecked calls* and *reentrancy* classes while still providing good results for others.

To sum up, the contributions of this paper are the following. First, we collected a large set of smart contracts and labeled them based on the different vulnerability detectors and classes identified above. The dataset is available for public use. Second, we compare four different types of deep neural networks trained over our dataset, showing that they can provide good results in classifying smart contracts. This allows for code vulnerability detection. With respect to previous

This work has received funding from the EU H2020 research and innovation programme under the MSCA ITN European Joint Doctorate grant agreement No 814177 LAST-JD - RIoE and from the University of Urbino Carlo Bo through the "Bit4Food" research project.

proposals focusing on the use of CNNs for smart contract vulnerability classification, we thus exploit a different set of neural networks and a novel, wider dataset which also considers the possibility of having contracts that are affected by more than one vulnerability at a time.

The remainder of this paper is organized as follows. Section II presents the background. In Section III, we describe the methodology related to constructing the dataset and the deep neural networks. Section IV discusses our results. Finally, Section V provides some concluding remarks.

II. Background and Motivation

A. Smart Contracts

An immutable set of instructions whose execution is calculated deterministically by peers in the DLT network is embraced by the definition of the smart contract. Each node executing the instructions receives the same inputs and produces the same outputs, thanks to a shared protocol. This allows the issuer of a smart contract not to require the presence of a trusted third-party validator to check the terms of an agreement. However, since it consists of executable code, the issuer must also be sure that the behavior implemented is correct (e.g., through code verification). In Ethereum, the smart contract is a set of instructions and a state. The latter is modified utilizing transactions that enclose data and references to the former. The state evolution is completely traced in the ledger. This protocol allows computing (*quasi*-)Turing-complete programs, i.e., smart contracts capable of processing any type of calculation where steps are bounded. A price, measured in a unit called "gas", is associated with each smart contract execution, and a gas limit is set to avoid infinite computation.

Smart contracts offer a variety of applications, ranging from decentralized finance and traceability of processes up to novel systems for smart services in different domains [10]–[12].

B. Auditing and Data Act

The European Union Data Act is a regulation that lays down harmonized rules on making data generated by the use of a product or related service available to the user of that product or service or other data recipients [1]. In this act, smart contract is considered as an appropriate technical protection measure to prevent unauthorized data access. Indeed, many proposals already implement Ethereum smart contracts for the protection of personal data and their sharing [2]. Of course, all this provided that the smart contract offers some degree of robustness to avoid functional errors and to withstand manipulation by third parties.

Auditing contracts for vulnerabilities is a common practice. This process should preferably be performed while contracts are still in the testing phase. There have been automated or semi-automated tools to perform the contract audit:

- **manual**: manual audits are performed by security engineers through an analysis of the source code and the dynamics of its execution. The advantage of manual auditing is the feasibility of understanding the code logic and identifying logical loopholes, while the downsides are the auditing speed and expense.
- **software tools**: automated tools offer scalable solutions in terms of vulnerability analysis. These are based on methods such as symbolic execution, fuzzing test, and taint analysis. Most of these tools analyze either the contract source code or its compiled bytecode and look for known security issues.

Defining security patterns requires having a deep knowledge of the internal workings of the blockchain and Solidity code. Thus, manual audits can be performed only by field experts. Moreover, the manual method requires time and effort, especially considering the rates at which new potentially exploitable vulnerabilities are discovered. With this in view, in this paper, we focus on the automation of the auditing process. Some automated vulnerability detection tools have been proposed already: the majority rely on symbolic execution (Oyente [13], Mythril [14]) or are rule-based (Slither [15], Smartcheck [16]). These tools can reach a high detection accuracy for known bugs but are either too time-consuming (symbolic execution) or rely on expert-made detection rules, thus not resolving our problem. We follow the approach that some scholars recently undertook, i.e., developing machine learning and deep learning-based solutions that are usually less time-consuming and do not rely on a heavy feature engineering phase [8], [17], [18].

III. Methodology

This section describes the dataset and the models we used to classify smart contract vulnerabilities.

A. Dataset

Since smart contract vulnerability classification is a relatively new research area, there are few open-source datasets of labeled smart contracts, and most are quite small. Two of them are the SmartBugs [19] wild and the ScrawlID [20] datasets: they were labeled using different tools and thus reduce the probability of false-positives. However, they only contain 6.7k and 47k elements, respectively, making them too small for training a deep model from scratch. With this in view, we decided to put together and release our large-scale dataset, which is available on the HuggingFace hub[1], a platform that hosts more than 6K open source datasets as well as thousands of models. To build this dataset, we first obtained a list of verified smart contracts on the Ethereum main net from Smart Contract Sanctuary [21]. The source code of every contract was then either downloaded from the aforementioned repository or obtained via the Etherscan API[2]. In contrast, the bytecode was downloaded using the `web3` Python library, which allows us to interact with a local or remote Ethereum node. Moreover, when the source code was organized in multiple files, we flattened it using the designated Slither tool. Note that while the source code is not employed

[1]https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts

[2]https://docs.etherscan.io/

anywhere in our analysis, we still release it along with the contracts' bytecode. This makes our dataset easy to reuse for other types of tasks, including Solidity code generation and vulnerability classification based on Solidity code.

The final dataset accounts for more than 100k smart contracts labeled using the Slither static analyzer. This tool passes the code through several rule-based detectors and returns a JSON file containing details about where those detectors found a vulnerability. The 38 detectors that found a match in our dataset were then mapped to the following five classes, according to the guidelines provided by the Smart Contract Weakness Classification registry [22], and by the Decentralized Application Security Project (DASP) [23], two community-based projects that aim to offer a taxonomy of smart contract vulnerabilities. In case of ambiguities, we followed the same classification used in SmartBugs [19].

- **Access-control**: this vulnerability is common in all languages, hence not related to blockchain technologies. Usually, a contract's functionality is accessed through its public or external functions. However, if the visibility of some fields/functions is not correctly set to private, malicious users could have access to them.
- **Arithmetic**: this class is related to integer underflow and overflow errors, which are particularly dangerous in smart contracts where unsigned integers are prevalent. In case of overflow, many benevolent pieces of code may be turned into tools for DoS attacks and theft.
- **Reentrancy** [24]: this is probably the most famous Ethereum vulnerability. It occurs when a call to an external contract is allowed to make new calls to the calling contract before the initial execution is complete. This means that the contract state may change in the middle of the execution of a function.
- **Unchecked-calls**: Solidity offers some low-level functions like *call()*, *callcode()*, *delegatedcall()* and *send()*, which do not propagate errors. These functions return false, but the code will continue to run; thus, developers should always check the return value of such low-level calls. Note that, in alignment with SmartBugs Wild, we include the results of the Slither detector unused-return in this category, which checks if the return value of an external call is not stored in a local or state variable.
- **Others**: this class groups together the results of all the other relevant Solidity detectors that were not included in the previous classes. Examples are: uninitialized-state, which checks whether the contract has some uninitialized state variables, incorrect-equality, which checks whether strict equalities were used to determine if an account has enough Ether or tokens (something that an attacker can easily manipulate) and backdoor, which detects the presence of a function called "backdoor".

During the dataset construction phase, we collected and labeled more than 100k verified smart contracts on the Ethereum chain. We then separated them into training, validation, and test sets taking care to maintain the proportion between classes. In the end, our dataset comprises 79.6k, 10.8k, and 15.9k elements in training, validation, and test set, respectively. It is also important to note that our approach to vulnerability detection in smart contracts accounts for the fact that, in real life, a single contract may have more than one vulnerability. Indeed, as stated in Section I, we focus on a multi-label classification task, and approximately 40% of all the elements in our training set are part of two or more classes. Table I shows the number of contracts our training set has per vulnerability class and the percentage of the contracts in that class that also have one or more other vulnerabilities. It is possible to notice how the elements in class `safe` do not belong to any other class, while we have widespread overlapping for all the other labels. It is also important to point out how the classes are strongly unbalanced, with `unchecked-calls`, `safe` and `reentrancy` the most populated classes, while the `access-control` vulnerability is only present in about 11k smart contract, making it the minority class.

TABLE I
NUMBER OF ELEMENTS PER CLASS IN THE TRAINING SET, ALONG WITH THE PERCENTAGE OF CONTRACTS OF THAT CLASS THAT ALSO HAVE OTHER VULNERABILITIES

Vulnerability	Contracts	Multi-label Contracts (%)
`unchecked-calls`	36353	72.32 %
`safe`	27036	00.00 %
`reentrancy`	24161	91.51 %
`other`	20993	84.17 %
`arithmetic`	13530	77.05 %
`access-control`	11704	87.27 %

Once we have our dataset of labeled contracts, we can use the bytecode to produce an RGB image with a procedure similar to what is done in [9]: suppose we have the piece of bytecode 606080, then in the RGB image, the three channels will be (R:60, G:60, B:80). Some of the images produced with this technique are shown in Figure 1. Note that when passing them as input to our CNNs, we center crop and resize them to achieve a single image size.

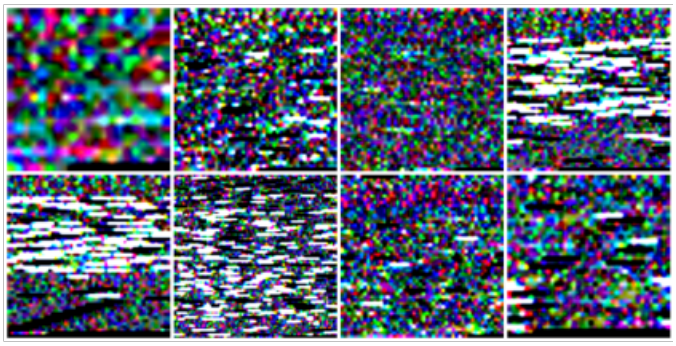

Fig. 1. Examples of results obtained from transforming smart contract bytecodes into RGB images.

B. Models

We experimented with four deep neural networks architectures:

- two traditional 2D CNNs, namely ResNet-18 and Inception v3, both working on RGB images corresponding to contract bytecode (generated as described above);
- a 1D CNN applied directly to the contract bytecode, which was treated as a signal and normalized to be between -1 and 1;
- a baseline Long Short-Term Memory (LSTM) model, which was trained only on the sequences of opcodes in the contract bytecode.

More details about each of these architectures are given below.

1) LSTM baseline: This is a simple network composed of an Embedding layer, which was trained from scratch to produce an embedding for each opcode, three stacked bidirectional LSTM layers [25], and two linear layers that served as the classification head. These linear layers took as input the concatenation of the final hidden states of the last forward and backward LSTM and computed the prediction. Figure 2 shows the complete network architecture.

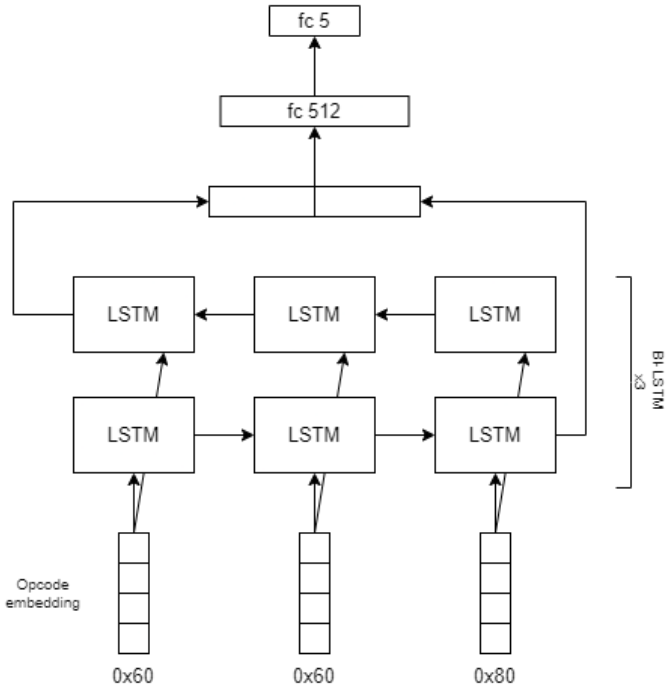

Fig. 2. LSTM baseline architecture

2) Conv2D models: We used two common 2D CNN models from the computer vision literature, specifically, ResNet-18 [26], and Inception v3 [27]. According to the literature, the ResNet model was chosen as a baseline convolutional model, while the Inception network was selected since it provides good results in malware detection and classification. Both models were not trained from scratch but initialized from ImageNet weights. Indeed, literature on malware classification [28] showed that this pre-training is beneficial and improves performance even on domains that are quite different from the natural images in ImageNet.

3) Conv1D model: Finally, some literature [8], [29] suggested that 1D convolutions may be a good fit for this task. Indeed, traditional 2D CNNs are structured so that the shallow layers capture low-level features, which then get aggregated into high-level ones in subsequent layers. However, the useful patterns to detect code vulnerability are most likely low-level pixel-by-pixel ones. In practice, as the network grows deeper, we tend to lose some of the pixel-level information. As a result, the semantics and context of the smart contract can be destroyed. At the same time, applying 1D convolutions over the contract bytecode used as a signal (i.e., not reshaped as an RGB image) may be a better strategy to maintain this information.

Thus, we also implemented a ResNet-inspired 1D CNN. Figure 3 shows how we defined the 1D ResBlock, while Figure 4 shows the architecture as a whole.

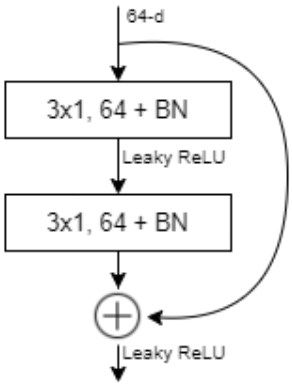

Fig. 3. 1-dimensional residual block

IV. Results

We tried different training configurations for every model type varying: the learning rate, the eventual l2 penalty, the optimizer (Adam or SGD), the statistics (mean and standard deviation) used when normalizing RGB images (either computed ad-hoc or taken from ImageNet), which layers were fine-tuned and which, if any, were kept fixed, the loss (either binary-crossentropy or focal loss), and the use of class weights.

The ResNet-inspired 1D CNN achieved the best performances. Since this architecture was only inspired by the literature, no pre-trained (ImageNet) weights of the model were available. Thus, the model was trained from scratch, and the signal was normalized between -1 and 1. We employed an SGD optimizer with a learning rate set to 0.001 and an L2 penalty of 0.0001, applied only on convolutional and dense layer weights, as commonly done in computer vision applications [30]. Finally, the optimized loss for this model was binary-cross-entropy, and – even though class weights are a common way to try and improve performances in case of strongly unbalanced classes – in our case, we found the performance to be better without them. Table II reports the best results we could achieve on the validation set for every

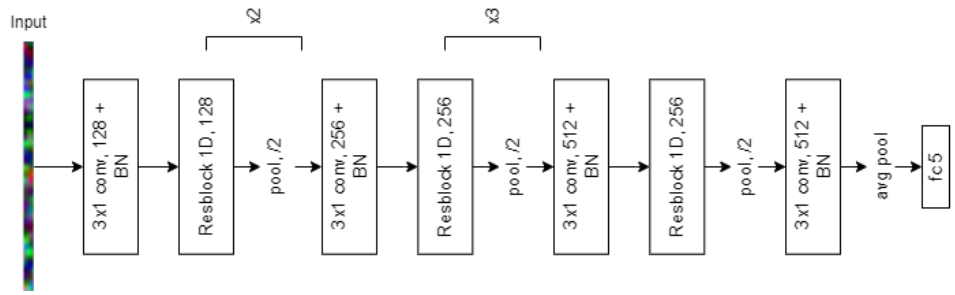

Fig. 4. Resnet-inspired 1D convolutional network

architecture. In this context, where every element can be in more than one class, and the class labels are not balanced, accuracy is not an ideal metric. Indeed, we also consider a micro-averaged version of the F1 score, which aggregates the contributions of all classes to compute the average metric and thus treats the examples of each class with equal weight.

TABLE II
MODELS' RESULTS ON THE VALIDATION SET

Model name	Accuracy	Micro F1
ResNet1D	0.7353	0.8381
ResNet	0.6841	0.7928
Inception	0.6988	0.8015
LSTM Baseline	0.6934	0.7953

From Table II, we can see that the LSTM baseline obtains poor results. This is probably because we cut the bytecode to just 512 opcodes due to limited memory and computational resources. However, our analysis shows that most bytecodes have a length of about 5000 opcodes, meaning that the portion we use probably corresponds to only a tiny first portion of the contract code.

2D CNNs have the advantage of not requiring the input to be truncated in any way: we first create the images using all the bytecode and then resize them as needed. However, their performance is comparable to our baseline, indicating that they may not be ideal for this task. This might seem surprising, as we saw them successfully employed in malware detection; however, we should note that malicious code patterns in that domain are usually quite big and easy to detect, even to the human eye. By contrast, patterns for code vulnerability detection in Solidity may only be at the level of a small sequence of opcodes.

The 1D CNN again requires the input to be cut off, but the network's nature lets us use a larger maximum length of 16384 (corresponding to a flattened 128x128 image). As shown in Table II, this architecture is the one that achieves the best results.

We show in Figure 5 the confusion matrices relative to the performance of our best model (i.e., ResNet1D) on the test set. The class unchecked-calls has significantly few misclassified examples, while the percentage of errors increases when we consider the other classes. This result was predictable since unchecked-calls are a vulnerability in more than 35k of the original train contracts (majority class). Among the other classes, a notable percentage of false negatives was experienced for classes with fewer training examples, i.e., access-control and arithmetic. Finally, classes others and reentrancy have approximately the same number of samples in the train set. However, the first one is mis-classified a lot more: this is due to the inherent nature of this class, which groups all the interesting vulnerabilities that are not part of the other four classes. This variety may indeed generate some confusion for the detector.

V. CONCLUSIONS

This paper showed that deep learning techniques could be a viable tool for identifying and classifying smart contract code vulnerabilities. Based on the dataset we built (and made available), we have trained and tested four different neural network architectures. Using the aforementioned dataset, we then approach the problem of vulnerability classification as a problem of multi-label classification: this is quite different from previous works, which focused either on vulnerability detection (i.e., the binary problem of detecting whether the contract is `safe` or `unsafe`) or on single-label vulnerability classification, meaning that each contract can only belong to one vulnerability class. Results show that, according to our configuration based on the available computational capabilities, the 1D ResNet CNN working on the smart contract bytecodes can provide the best results in terms of accuracy and Micro F1.

Our study can be considered a starting point for several future works. Indeed, to the best of our knowledge, the dataset we made available can be employed for several other types of tasks, i.e., Solidity code generation, vulnerability classification from source code, etc. Moreover, the dataset is constructed so that every unsafe smart contract can be in one or more vulnerability classes. This closely reflects the real world, where a single smart contract can open to several potential threats. We plan to continue this study by comparing the considered architectures, which according to state of the art are among the best ones for this purpose, with other classification methods available in the literature. Moreover, due to the extensive computation requirements needed to deploy effective machine learning models, we plan to perform more comprehensive tests

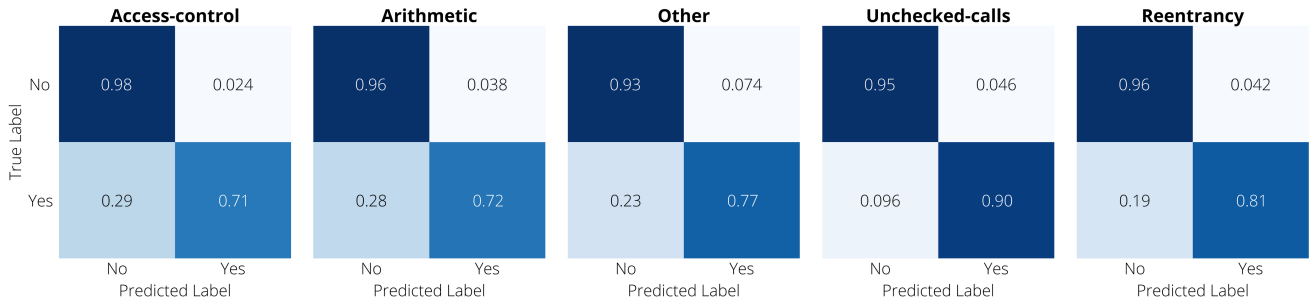

Fig. 5. Per-class confusion matrices obtained by ResNet1D on the test set

over more powerful servers that would allow us to tune the employed networks better.

REFERENCES

[1] European Commission, "Data act proposal," pp. 1–63, 2022.

[2] M. Zichichi, S. Ferretti, G. D'Angelo, and V. Rodríguez-Doncel, "Data governance through a multi-dlt architecture in view of the gdpr," *Cluster Computing*, Aug 2022.

[3] "Security analysis of distributed ledgers and blockchains through agent-based simulation," *Simulation Modelling Practice and Theory*, vol. 114, p. 102413, 2022.

[4] M. Zichichi, S. Ferretti, and G. D'Angelo, "A distributed ledger based infrastructure for smart transportation system and social good," in *2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC)*, 2020, pp. 1–6.

[5] M. Zichichi, M. Contu, S. Ferretti, and G. D'Angelo, "Likestarter: a smart-contract based social DAO for crowdfunding," in *Proc. of the 2nd Workshop on Cryptocurrencies and Blockchains for Distributed Systems (CryBlock)*, 2019.

[6] A. Spătaru, L. Ricci, D. Petcu, and B. Guidi, "Decentralized cloud scheduling via smart contracts. operational constraints and costs," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2019, pp. 482–489.

[7] R. Agarwal, T. Thapliyal, and S. K. Shukla, "Vulnerability and transaction behavior based detection of malicious smart contracts," 2021. [Online]. Available: https://arxiv.org/abs/2106.13422

[8] S.-J. Hwang, S.-H. Choi, J. Shin, and Y.-H. Choi, "Codenet: Code-targeted convolutional neural network architecture for smart contract vulnerability detection," *IEEE Access*, vol. 10, pp. 32 595–32 607, 2022.

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

[10] G. D'Angelo, S. Ferretti, and M. Marzolla, "A blockchain-based flight data recorder for cloud accountability," in *Proc. of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, ser. CryBlock'18. Association for Computing Machinery, 2018, p. 93–98.

[11] M. Zichichi, S. Ferretti, and G. D'angelo, "A framework based on distributed ledger technologies for data management and services in intelligent transportation systems," *IEEE Access*, vol. 8, pp. 100 384–100 402, 2020.

[12] G. Bigini, M. Zichichi, E. Lattanzi, S. Ferretti, and G. D'Angelo, "Decentralized health data distribution: A dlt-based architecture for data protection," in *2022 IEEE International Conference on Blockchain (Blockchain)*, 2022, pp. 97–104.

[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. of 2016 ACM SIGSAC*, ser. CCS '16, 2016, p. 254–269.

[14] B. Mueller, "Smashing Ethereum Smart Contracts for Fun and Real Profit," in *9th HITB Security Conference*, 2018, p. 54.

[15] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd Int. Work. on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019.

[16] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 9–16. [Online]. Available: https://doi.org/10.1145/3194113.3194115

[17] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning," *arXiv:2103.12607 [cs]*, Mar. 2021, arXiv: 2103.12607. [Online]. Available: http://arxiv.org/abs/2103.12607

[18] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python," *Information and Software Technology*, vol. 144, p. 106809, Apr. 2022, arXiv: 2201.08441. [Online]. Available: http://arxiv.org/abs/2201.08441

[19] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[20] C. S. Yashavant, S. Kumar, and A. Karkare, "Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities," *arXiv preprint arXiv:2202.11409*, 2022.

[21] M. Ortner and S. Eskandari, "Smart contract sanctuary," Jan. 2022. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary

[22] S. Registry, "Smart Contract Weakness Classification and Test Cases," Aug. 2018. [Online]. Available: http://swcregistry.io/

[23] N. Group, "Decentralized application security project (DASP) - top 10," Mar. 2018. [Online]. Available: https://www.dasp.co/

[24] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Berlin, Heidelberg: Springer-Verlag, 2017, p. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6_8

[25] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM networks," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 4. Montreal, Que., Canada: IEEE, 2005, pp. 2047–2052. [Online]. Available: http://ieeexplore.ieee.org/document/1556215/

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[27] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[28] L. Chen, "Deep Transfer Learning for Static Malware Classification," Dec. 2018, arXiv:1812.07606 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1812.07606

[29] W.-C. Lin and Y.-R. Yeh, "Efficient malware classification by binary sequences with one-dimensional convolutional neural networks," *Mathematics*, vol. 10, no. 4, p. 608, 2022.

[30] A. Brock, S. De, S. L. Smith, and K. Simonyan, "High-performance large-scale image recognition without normalization," 2021. [Online]. Available: https://arxiv.org/abs/2102.06171