# Smart Contract Vulnerability Detection: A Literature Review

Mark Kane

## Intro

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

Blockchain has fostered the exchange of value between two peers in a decentralized manner across the internet which was never possible before its existence. Any technology which allows for the transfer of value will attract financially motivated bad actors. Real world attacks have already demonstrated the need for smart contract security. For example, the DAO (Decentralized Autonomous Organization) attack in 2016 resulted in the loss of over $50 million worth of Ethereum due to a vulnerability in the smart contract code [5]. In this case, an attacker was able to exploit a security bug in the smart contract's code (called reentrancy) to drain funds from the DAO's account. Similarly, the Parity wallet hack in 2017 resulted in the loss of over $30 million worth of Ethereum due to a bug in the smart contract code. In this case, a vulnerability in the code allowed an attacker to take control of the Parity wallet and steal the funds held within it [5].

These real world examples highlight the importance of smart contract security. The security problems caused by smart contracts are more complicated and some could argue higher consequence than traditional software programs, and analysing smart contracts to detect these vulnerabilities can also be more difficult [2]. Failure to adequately secure smart contracts can lead to significant financial loss and damage to the reputation of the blockchain ecosystem as a whole. This paper presents a review of the literature around smart contract vulnerability detection and security verification. Section 1 discusses smart contract vulnerabilities and how they are classified. Section 2 surveys the different approaches to smart contract vulnerability detection and the state of the art vulnerability detection tools. Section 3 compares these different approaches. Section 4 then takes a closer look at the different approaches to deep learning based smart contract vulnerability detection. Section 5 deduces the current “gaps” in the research related to deep learning based smart contract vulnerability detection and illustrates the potential research directions which this practicum should consider. Finally, this literature review is concluded with a well-defined research question which this practicum will aim to answer.

## 1. Smart Contract Vulnerabilities

The sharp rise in smart contract development over recent years has caused researchers and developers alike to focus their attention towards vulnerabilities in smart contracts which can be exploited by attackers. In 2021, 680 million US dollars' worth of digital assets managed by smart contracts were stolen/hacked through the exploitation of security vulnerabilities [6]. Security has now become a prioritised concern when it comes to smart contract development, but up until the study carried out by [6] it was unclear how security was approached by developers when writing smart contracts and deploying them to the blockchain. [6] conducts a qualitative study with 29 smart contract developers of varied levels of experience: 10 junior smart contract developers (< 1 year of experience) and 19 more experienced smart contract developers (2-5 years of experience). The study carried out by [6] consists of a code review task along with a semi-structured interview for each of the participants. [6] finds a wide variety of perceptions and practices used by the participants with respect to the vulnerability detection tools and resources they used. The vast majority (83%) of participants did not even mention security as a priority to their smart contract development process. The study found that the more junior developers had a far lower success rate at identifying security vulnerabilities in the code review task (15%) as compared to their more experienced counterparts (55%). It was found that many developers rely on external auditing of code to ensure the security of their smart contracts and when they do assess the security themselves, they find a lack of tools and resources to do so and do it manually. Although [6] conducts their study on a small sample size, it is clear from that the tools and resources in this domain are not yet adequate to support developers in deploying secure smart contracts to the blockchain.

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

In traditional software there exists systematic security guidelines such as the NIST (National Institute of Standards and Technologies) [11] or OWASP (Open Web Application Security

Project) [12] which aim to provide a framework for software developers in writing secure code. These aid software developers in mitigating against exploitation of their code and help them to understand the root cause of their code vulnerabilities to help future proof their code [6]. There exists some resources related to writing secure smart contracts (e.g. Solidity documentation, ConsenSys) which provide code design patterns for developers to follow [6]. However, after the sharp increase in smart contract development over the last few years, there lacked a framework to support formal assessment of smart contract security over the entire development life cycle [6]. The proliferation of smart contract vulnerabilities deployed to the blockchain is largely a consequence of their meagre documentation and classification within a formal framework [5]. This is a problem which [5] attempts to help solve by formally classifying Ethereum smart contract vulnerabilities using the NIST Bugs Framework. [5] collects, analyses and categorizes different smart contract bugs that exist on the Ethereum blockchain and documents a comprehensive master list of smart contract vulnerabilities written in Solidity and deployed to the Ethereum blockchain. This master list comprises of 49 vulnerabilities and as well as classifying each into a class of the NIST Bug Framework, [5] also categorizes each under the following descriptions: Operational, Functional, Security, Developmental. [5] also proposes two new bug classes: Distributed System Protocol (DSP) and Distributed System Resource Management (DRM), for smart contract vulnerabilities found which fall outside the scope of the NIST Bug Framework at the time the research was carried out.

As can be seen, the tally of possible smart contract vulnerabilities in Ethereum varies from study to study which makes it difficult to discern which vulnerabilities pose the biggest risk to real world attacks. Luckily, many recent studies have carried out systematic reviews of the literature in relation to security vulnerabilities on Ethereum blockchain smart contracts. One such study is done by [1], who, similar to [7], categorize vulnerabilities with respect to their root cause: Solidity programming language, features of Ethereum Virtual Machine or design features of the Ethereum blockchain. After surveying 119 peer reviewed articles on the subject, [1] discusses 23 vulnerabilities and the tools/prevention methods available to detect them. Similarly, [3] focuses on 16 different types of attacks on smart contracts in Ethereum through exploiting vulnerabilities in the code and does a comprehensive study on the different vulnerability detection tools available today. Similar studies are done by [2] and [4]. From reviewing the literature thus far, a set of 16-23 of the same vulnerabilities appear over and over. This bank of vulnerabilities are considered to cover the root causes to all or most of the known attacks that can be made on Ethereum smart contracts. This bank of vulnerabilities will be the focus for this practicum.

## 2. Smart Contract Verification Tools

[14] is a recent study (up to 2021) which presents a systematic survey of existing smart contract analysis tools for the Ethereum blockchain discussing a total of 86 tools – the most

discussed in a paper by far. [14] gives two categories of analysis tools: static and dynamic. Each category has sub categories based on the actual input to the tool. Static and dynamic can both receive Solidity code as input and EVM Byte code. There is also a static analysis tool, FSolidM which takes input from some formal specifications and generates Solidity code to analyse. [14] finds that the top five vulnerabilities checked/detected by most of the tools are re-entrance, arithmetic overflow/underflow, gas-related, timestamp dependency, and transaction ordering dependency. The top five most popular approaches employed by most tools are: symbolic execution, fuzz testing, constraint solving, code instrumentation and code transformation. It also mentions the following analysis approaches:

- Symbolic Execution
- Dis-assembler
- Graphic Visualizer
- Fuzz testing
- Machine Learning
- Code Instrumentation
- Mutation Testing
- Formal Verification
- Constraint Solving  (Static)
- Code Transformation (Static)
- Abstract Interpretation (Static)
- Model based testing (Dynamic)
- Taint Analysis (Dynamic)

[14] classifies tools using a wide range of vulnerability detection methods but fails to provide a broad description of any of these categories. [14] compares tools with respect to aspects like analysis type, input to tool and checked vulnerabilities but finds that most of the tools' source code is not openly accessible. [14] selects some tools from each category (16 in total) and attempts to practically evaluate them using 30 contracts from the SolidiFI Benchmark dataset [17] tagged with the top 5 vulnerabilities checked by static or dynamic analysis tools. It finds that the tools Slither [16], Mythril [15] and Oyente [8] perform the best with respect to vulnerability detection. It seems that the tools to evaluate were chosen arbitrarily and the evaluation was done using a small number of contracts. Therefore, a more thorough and comprehensive evaluation would be required to draw any conclusions regarding the effectiveness of each tool. Finally, [14] remarks that most of the tools' source code is not made public or accessible for the tools to be evaluated, and so open sourcing new smart contract vulnerability detection tools will help foster trust among the research community in this domain. [14] also notes that the majority of smart contract analysis tools employ static analysis only. For comprehensive smart contract analysis, static *and* dynamic analysis is necessary to detect all vulnerabilities. These are important points to consider when formulating a research question that this practicum should attempt to answer at the end of this literature review.

[2] is a recent study which carries out its own survey on the current state of smart contract vulnerability detection. Like most of the studies in this area, [2] summarizes the most common types of smart contract vulnerabilities using the original taxonomy provided by [7] (discussed in section 1) of the three layers: Solidity code layer, EVM execution layer and the block dependency layer. [2] presents 15 of the most common Ethereum smart contract vulnerabilities with 8 of these classified as being Solidity code layer vulnerabilities, 4 being at the EVM execution layer and 3 being at the block dependency layer. [2] defines a much simpler classification system for classifying smart contract verification tools than [14], classifying the tool based on its vulnerability detection method coming from one of the 5 following categories: formal verification, symbolic execution, fuzzing detection, intermediate representation and deep learning.

***Formal Verification***

Formal verification attempts to eliminate the non-universality and ambiguity in a contract program through using a formal language to transform the ratiocinations into a smart contract model [2]. This method then uses rigorous logic/proof to verify the correctness and safety of smart contract functions [2]. One such formal verification tool is the F* framework proposed by [19], which aims to analyse and verify the functional correctness and runtime safety of Ethereum contracts by translation to a functional programming language called F* that is aimed at program verification. The smart contract Solidity code and bytecode are translated into F* through two modules, *Solidity** and *EVM** and uses an equivalence proof to verify the functional equivalence between source code and bytecode, ensuring functional correctness and runtime safety [2]. Other tools which also implement a formal analysis framework are: *KEVM Framework, Isabelle/HOL, ZEUS,* and *VaaS* [2]. One problem with formal verification tools is that many of them are semi-automatic [2]. They do, however, provide support for detecting a good variety of vulnerabilities with the *F* Framework* and *ZEUS* capable of detecting 6 vulnerabilities and *VaaS* capable of detecting 8 vulnerabilities.

***Symbolic Execution***

The approach taken under the category of symbolic execution is to reason about a program symbolically instead of actually executing the code. Inputs are treated as symbolic variables rather than concrete values and the symbolic execution engine then explores all possible smart contract execution paths, considering all possible combinations of input values [2]. A constraint system is created by the engine which represents the conditions under which each path can be taken, and using this constraint system test cases can be generated to exercise different parts of the contract [2]. The advantage of symbolic execution is that it can discover potential vulnerabilities in a smart contract without having to actually execute the code. This can be particularly useful for finding corner cases and edge conditions that might be missed by traditional testing techniques [2]. It can also be used to generate test cases that cover all possible execution paths of a contract, providing more thorough testing and a higher level of confidence in the contract's behaviour [2]. *Oyente* [8] is one of the pioneer smart contract vulnerability detection tools which was made open source. It achieves symbolic execution to detect vulnerabilities by using a control flow graph (CFG), which takes as input the state and bytecode of a smart contract to traverse different execution paths [2]. If Oyente finds a potential vulnerability, it will generate a report describing the issue including information around the location in code that the issue was found and a description of the conditions under which the issue can occur [8]. Many of the popular tools fall under the category of symbolic execution such as *Mythril, Osiris, Gasper, Maian, Securify, TeEther* and *Sereum*. Many of the tools in this category also have support for a good spread of vulnerabilities like Mythril which can detect 9 vulnerabilities and Securify which can detect 13 vulnerabilities [2].

***Fuzzing Test***

Fuzzing test is a popular vulnerability detection technique where the tool generates a large number of test inputs for the program and feeds them into the program and observes how it behaves [2]. If abnormal behaviour occurs such as the program crashing, freezing or producing an unexpected result, this could indicate a security vulnerability which could be exploited by malicious actors [2]. Fuzzing test also has good scalability and applicability when compared to the other testing methods. *ContractFuzzer* is the first dynamic analysis method which uses fuzzing for detecting vulnerabilities in Ethereum smart contracts [2]. *ContractFuzzer* uses the ABI specification for smart contracts to generate fuzzing inputs [2]. It first configures the EVM and records the runtime behaviour of the contract [2]. It then analyses these recorded logs to detect vulnerabilities and is capable of detecting 6 vulnerabilities in Ethereum smart contracts. *Regurad* and *ILF* are examples of other fuzzing analysers [2].

***Intermediate Representation***

This approach to smart contract vulnerability detection is self-explanatory: the smart contract source code or bytecode is converted into an intermediate representation with highly semantic information [2]. This intermediate representation of the smart contract code is then analysed to discover security vulnerabilities [2]. One well known tool which employs this approach is *Slither* – a tool which converts Ethereum smart contract code into an intermediate representation named *SlithIR*, which is designed to simplify the analysis process whilst retaining the lost semantic information that occurs when the source code is compiled to EVM bytecode [2]. *SlithIR* uses a static single allocation (SSA) form and reduced instruction set, which refers to the representation of the code in a form that assigns each variable a unique value only once, and reduces the number of instructions used to represent the code [2]. This form is commonly used in compilers to optimize code and make it easier to perform analysis and optimization [2]. By converting the smart contract source code into the SlithIR representation, the Slither framework is able to simplify the analysis process and make it easier to identify potential security vulnerabilities and other issues in the code [2]. Slither provides support for the detection of 7 Ethereum smart contract vulnerabilities [2]. Other tools which take the approach of intermediate representation are: *Vandal, Madmax, Ethir, Smartcheck* and *ContractGuard* [2].

***Deep Learning***

The progress in deep learning technology over recent years has helped promote its use in various disciplines including security detection in programs [2]. Deep learning involves the training of artificial neural networks on large datasets, allowing the network to learn and make decisions based on the input data. A deep learning model consists of multiple layers of artificial neurons, each of which is responsible for processing a different aspect of the input data. The input data is transformed and processed through each layer, and the output of the final layer is used to make predictions or decisions. Deep learning has already been very successful in areas such as image classification and speech recognition. *SaferSC* [20] is the first smart contract vulnerability detection system using a deep learning model. [20] employs an LSTM network to build a sequence network where the input is smart contract opcodes. [20] focuses on 3 vulnerabilities – the same three defined by Maian [21] – and uses the accuracy results of the Maian tool to evaluate the effectiveness of the model. SaferSC achieves a higher detection than Maian. Other deep learning based smart contract verification tools are: *RecChecker, DR-GCN, TMP, ContractWard, CGE* and *AME* [2]. Many of the deep learning based tools only support detection of one or two vulnerabilities, with the exception of ContractWard which supports five [2].

## 3. Comparative Analysis of Smart Contract Vulnerability Detection Methods

In the previous section we review the current state of the art Ethereum smart contract verification tools under 5 categories: Formal Verification, Symbolic Execution, Fuzz testing, intermediate representation and deep learning as presented by the survey carried out by [2]. [2] then compares representative methods from these 5 categories using 300 real world smart contracts deployed on the Ethereum blockchain as test samples with respect to detection accuracy, F1-Score and average detection time. Using these three measurements, the following representative tools for each of the 5 categories were compared: VaaS, Oyente, Smartcheck, ContractFuzzer, TMP. The effectiveness of these tools was tested on the following three vulnerabilities: reentrancy, integer overflow and timestamp dependency.

[2] finds that for reentrancy, TMP achieves the highest accuracy and F1-score closely followed by VaaS. For the integer overflow vulnerability, VaaS has the highest accuracy and performance compared to Oyente and Smartcheck. ContractFuzzer and TMP could not be tested for this vulnerability as they do not support it. Finally, for the timestamp dependency VaaS achieves the highest accuracy and F1-Score closely followed by TMP. Across all three vulnerabilities, VaaS and TMP achieve a significantly higher accuracy and F1-Score than the other tools. ContractFuzzer takes the longest with 352.2 seconds. VaaS also takes a significant

amount of time with 159.4 seconds. All tools are grossly outperformed by TMP which achieved an average detection time of 2.1-2.5 seconds. The evaluation carried out by [2] is not hugely comprehensive as it only tests a small number of representative tools on a small number of contracts, but it does illustrate the strengths and limitations of each of the five tools and hence each of the approaches to smart contract vulnerability detection.

The formal verification method performed well in the evaluation done by [2] but tools in this category are often semi-automated at best because the mathematical process they rely on requires interactive verification and judgement [2]. Its reliance on heavy mathematical derivation and verification means it cannot perform dynamic analysis [2]. Losing detection and judgement of the executable path results in more false-positives and leakage [2]. This behaviour is observed on evaluation of tools in this category.

The symbolic execution, fuzzing test and intermediate representation tools all provide an insufficient performance at detecting vulnerabilities in the evaluation done by [2]. These results are consistent with the results of the study carried out by [4], which did an empirical evaluation of the effectiveness of the following three smart contract tools: Securify2, Slither and Mythril. We know from the previous section that Securify2 and Mythril relies on symbolic execution, while Slither uses intermediate representation. [4] defines their own smart contract defect classification scheme based on the structure of the Orthogonal Defect Classification [22] and then extracts contracts from multiple datasets to construct a dataset of 222 smart contracts. [4] finds that the effectiveness of these tools is quite low due to the fact that they generate a high number of false positives and low number of true positives.

Finally, deep learning usually involves the pre-processing of a dataset of smart contracts to make them conducive to training a model [2]. The training of artificial deep neural networks allows the model to extract high level features of the smart contract code. These models have a black-box nature in that they cannot output the exact line of code where a potential vulnerability may exist in the same way other tools can [2]. The opaqueness of the inner workings of these models makes them less interpretable and with a lack of explanations for the vulnerability detection results, the results in turn seem less convincing [2]. However, with deep learning used in this area still in its infancy, the accuracy of these models so far is very promising and even more impressive is the near constant detection time which makes these models far superior when the number of contracts to be analysed is scaled. In the previous section we saw that deep learning based solutions only support a small number of vulnerabilities. This is due to the lack of a standard dataset of smart contracts which covers all of the main vulnerabilities on which to train models on. Most deep learning models so far have focused on just a small number of vulnerabilities, with reentrancy being the most popular [2]. The lack of variety of supported vulnerabilities by deep learning based tools is another area which needs improvement. The following section will discuss further the current deep learning based tools available.

## 4. Deep Learning for Smart Contract Verification

In the previous section we have already discussed one deep learning solution, *SaferSC* [20], which employs an LSTM network to analyse smart contract operation code (opcode) to build a sequence model. SaferSC [20] used the Maian tool [21] as a benchmark for performance comparison and so used the same dataset consisting of just three vulnerabilities. Although [20] was therefore able to conclude that SaferSC achieves a higher detection than Maian, it is limited to just the same three vulnerabilities covered by Maian. Nevertheless, the experimentation carried out by [20] on 620,000 smart contracts proves that the LSTM model implemented maintains a near constant analysis time as the complexity of the smart contract grows. [20] claims that their sequential learning model achieved a detection accuracy of 99.57% and f1-score of 86.04%, but more impressively correctly detects up to 92.86% of the particularly challenging smart contracts which, when applied to the Maian symbolic analysis tool, were deemed to be false-positives. SaferSC [20] does have another limitation, however. The model makes the assumption that sequence of opcode features in the smart contract can be used to identify and analyse its vulnerabilities [3]. However, this assumption may not always hold true and the model does not support checking certain properties like data and control flow properties [3]. In contracts where this assumption does not hold true, the model will not perform well.

Where much of the deep learning solutions to smart contract vulnerability detection attempt to extract features from the smart contract opcode/assembly code, [9] uses a Solidity parser to generate an AST (Abstract Syntax Tree) of the smart contract source code from which to extract features. This approach allows the deep learning model implemented by [9] to extract features directly from the source code. [9] experiments with three different classification techniques: binary, multi class and multi label classification. The binary classifier implemented by [9] uses an ANN and an Auto Encoder to detect if a smart contract has the reentrancy vulnerability and claims that this classifier achieves 100% accuracy. [9] does not, however, provide much detail on how it generates the dataset. It also seems to evaluate the performance of the classifier using the same dataset that was used to train the model as opposed to training it on test data which the trained model has not seen before. This would not help to evaluate how the model might perform on real world smart contracts. The multi class classifier is claimed by [9] to have achieved 100% accuracy at detecting whether a smart contract has the reentrancy vulnerability, or if it has the transaction origin vulnerability, or if it has neither. [9] claims that their multi label classifier achieves an accuracy of 97.4% at predicting if a smart contract has either of the following three vulnerabilities: reentrancy, transaction origin and denial of service. Despite these strong claims of accuracy by [9] it is difficult to determine how these models would fair with real world smart contracts as it seems that [9] evaluated the models using the same data with which the model was trained on. [9] also has not open-sourced the classifier so it has not been formally tested.

*ESCORT*, proposed by [10], is the first Deep Neural Network (DNN) based vulnerability detection framework for Ethereum smart contracts which also supports transfer learning for new unseen vulnerabilities. ESCORT is a multi-output (multi-label) model with the following two main components: feature extractor and vulnerability branches [10]. The feature extractor is a common stack of layers shared by all the vulnerability branches that learns general fundamental features of the input data. The feature extractor component of ESCORT takes smart contract bytecode as input and is trained to learn the semantic/syntactic information from this. The second component of ESCORT's DNN architecture is multiple vulnerability branches, where each branch consists of a stack of layers which are trained to learn features/patterns related to the branches corresponding vulnerability class. The input to each branch will be the same, i.e., the output of the feature extractor. The actual deep learning model incorporates *text representation*, the transformation of the smart contract bytecode from a text modality into numerical vectors, and a *Recurrent Neural Network (RNN)*, a category of DNNs where the connections between neurons construct a direct computational graph along the temporal sequence. Additionally, one of the key differentiators of ESCORT with respect to other deep learning based approaches to smart contract verification is that it is extensible and generalizable because it supports lightweight transfer learning on new vulnerability types. [10] states that ESCORT performs strongly on 6 vulnerabilities which the model was pre-trained for, achieving an average f1-score of 95% at detecting these vulnerabilities with a detection time of 0.02 seconds per contract. Its ability to generalize through transfer learning is also promising, yielding an average f1-score of 93% for the new extended vulnerability types [10]. [10] also adds that the slight reduction in performance for vulnerability types added through transfer learning is well compensated by the reduction in time for training when compared to training a model from scratch. ESCORT does, however, have the limitation that [10] does not provide the source code for their architecture and so ESCORT cannot be formally evaluated. Notwithstanding this, [10] does make publicly available the dataset used as well as ContractScraper, the toolchain built by [10] to construct and label their contract dataset.

*ContractWard* [18] is a tool that uses machine learning techniques to detect vulnerabilities in Ethereum smart contracts. The tool collects a large number of verified smart contracts and simplifies the opcodes. It then adds bigram features and labels using the symbolic checking tool called Oyente [8]. The dataset is classified using One Vs Rest (OvR) algorithms for multi-label classification, and sampling algorithms are used to balance the dataset before training [3]. Five classification algorithms are used to train the models, and XGBoost classifier on a balanced training set with SMOTETomek is used to achieve the best results. ContractWard is capable of detecting six vulnerabilities with high accuracy, but has limitations such as dependence on Oyente for labelling and an inability to generalize to other vulnerabilities. It should also be noted that although many surveys have categorized ContractWard as a deep learning based tool, it uses more traditional machine learning techniques over neural networks.

## 5. Research Directions

Through the review of the literature detailed in the past few sections it has been clearly established that using deep learning for smart contract vulnerability detection has many advantages over the more commonly used methods such as faster vulnerability detection time and superior performance with respect to detection accuracy and f1-score. We have also seen that many of the deep learning based tools implemented to date are either binary classifiers for one vulnerability type or they are multi-class/multi-label classifiers for a small number of vulnerability types (usually trained for a maximum of three different vulnerabilities). Tools which go beyond this or are extensible to incorporating further vulnerabilities through transfer learning, ESCORT [10] being the only known solution which supports this, have not been made open source. For these reasons, the author believes that exploring deep learning for smart contract vulnerability detection is a good proposal for this research practicum. There are a number of research questions in this domain that this practicum could aim to answer:

- [3] comments that deep learning approaches lack a unified evaluation method, where most of the frameworks are evaluated using their own personally constructed database of smart contracts collected from etherscan. Therefore, it is currently very difficult to conduct an independent comparative analysis of the tools which have been made open source by the author. Lack of a standard dataset is also a contributing factor as to why many of the deep learning based solutions to smart contract vulnerability detection are trained to detect only a small number of vulnerabilities. Therefore, a useful contribution would be to establish a large, standardized dataset with a diverse range of vulnerability types covering all of the well-established vulnerabilities suitable for the training and evaluation of deep learning based smart contract verification tools. This dataset could then be made open source so it can be used by the research community and also added to as new vulnerabilities arise.
- [2] notes how, with deep learning based smart contract vulnerability detection still in its infancy, most methods only analyse the contract at the level of static smart contract bytecode or source code. We know that static analysis often will miss existing execution paths. With a lack of dynamic interaction with external contracts, a high false-negative/false-positive rate can occur [2]. Therefore, [2] suggests that combining dynamic and static analysis to build a more comprehensive model should be considered.
- [2] also acknowledges how existing vulnerability detection methods based on deep learning focus on training models for a specific set of known vulnerabilities. [2] believes that whether these models can be quickly adapted to the new kinds of vulnerabilities should be further studied. This literature review has already detailed how ESCORT [10] is the only known exception to this, achieving generalizability through transfer learning. However, ESCORT [10] has not been formally tested.

- Another area of improvement for deep learning based solutions to smart contract vulnerability detection is related to the lack of an explanation provided to the user for vulnerability detection results, e.g. giving the user a specific code line which may be responsible for the detected vulnerability [2]. This opaque nature of deep learning models can sometimes make detection results seem unconvincing. [2] mentions expert rules, which refer to pre-defined rules/heuristics designed to identify potential vulnerabilities in smart contracts which can be used to complement other smart contract vulnerability detection techniques like static analysis. [2] suggests that future deep learning models should consider integrating expert rules to improve the tools vulnerability detection.
- Another research gap in this domain is around experimenting with the mode of input to deep learning models. The review of the existing tools showed how most models accept smart contract bytecode, or in the case of [9], an AST of the smart contract source code. To the author's knowledge, there has been no experimentation done by previous research with multi modal inputs to the deep learning model to see if better features of one modality can be learned at the stage of feature learning if multiple modalities of the smart contract source code are present.

## Conclusion

Through the review of the literature we have seen the necessity for being able to detect smart contract vulnerabilities and the consequences of insecure contracts. We have summarized the main Ethereum based smart contract vulnerabilities and how different studies within this research area have chosen to define and classify them. We then discussed the different approaches to smart contract vulnerability detection and the state of the art verification tools. It is clear from the literature that the deep learning approach to smart contract vulnerability detection is superior in many aspects such as detection time, detection accuracy and detection f1-score as compared to other approaches (like formal analysis and symbolic execution) taken by the more popular tools. Also discussed are the many research directions this practicum could take due to the fact that research in applying deep learning for smart contract vulnerability detection is still in its infancy.

This practicum will look to follow the path of research described in the final point of the previous section related to experimenting with multi-modal or cross-modal deep learning for smart contract vulnerability detection. This has been chosen as the research question this practicum will attempt to answer as it does not seem to have been answered by previous research.

## Bibliography

[1]: Kushwaha SS, Joshi S, Singh D, Kaur M, Lee HN (2022) Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract. IEEE Access 10:6605–6621

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

[3]: Vani, S. & Doshi, M. & Nanavati, A. & Kundu, A.. (2022). Vulnerability Analysis of Smart Contracts. 10.48550/arXiv.2212.07387.

[4]: Dias, Bruno & Ivaki, Naghmeh & Laranjeiro, Nuno. (2021). An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools. 10.1109/PRDC53464.2021.00013.

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

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

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

[8]: Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[9]: Narayana, K. & Sathiyamurthy, K.. (2021). Automation and smart materials in detecting smart contracts vulnerabilities in Blockchain using deep learning. Materials Today: Proceedings. 10.1016/j.matpr.2021.04.125.

[10]: Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad Reza Sadeghi, and Farinaz Koushanfar.Escort: Ethereum Smart Contracts Vulnerability Detection Using Deep Neural Network andTtransfer Learning. arXiv preprint arXiv:2103.12607,2021

[11]: NIST, NIST BF Welcome. (NIST Website: https://csrc.nist.gov/Projects/ssdf)

[12]: OWASP. (OWASP Website: https://owasp.org/www-project-security-knowledge-framework/)

[13]: N. Fatima Samreen and M. H Alafi, "A Survey of Security Vulnerabilities in Ethereum Smart Contracts," in 30th Annual International Conference on Computer Science and Software Engineering, Toronto, 2020.

[14]: S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur and H. -N. Lee, "Ethereum Smart Contract Analysis Tools: A Systematic Review," in IEEE Access, vol. 10, pp. 57037-57062, 2022, doi: 10.1109/ACCESS.2022.3169902.

[15]: Mythril, Aug. 2021, [online] Available:https://github.com/ConsenSys/mithril

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

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

[18]: Wang, Wei & Song, Jingjing & Xu, Guangquan & Li, Yidong & Wang, Hao & Su, Chunhua. (2020). ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. IEEE Transactions on Network Science and Engineering. PP. 1-1. 10.1109/TNSE.2020.2968505.

[19]: I. Grishchenko, M. Maffei, and C. Schneidewind, “A semantic framework for the security analysis of ethereum smart contracts,” in International Conference on Principles of Security and Trust. Springer, 2018, pp. 243–269.

[20]: 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.

[21]: I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding ´ the greedy, prodigal, and suicidal contracts at scale,” in Proceedings of the 34th annual computer security applications conference, 2018, pp. 653–663.

[22]: IBM. (2013, Sep.) Orthogonal Defect Classification v 5.2 for Software Design and Code. [Online]. Available: https://researcher.watson.ibm. com/researcher/files/us-pasanth/ODC-5-2.pdf