Smart Contract Vulnerability Detection: The Role of Large Language Model (LLM)

Biagio Boi
University of Salerno
Fisciano, Italy
bboi@unisa.it

Christian Esposito
University of Salerno
Fisciano, Italy
esposito@unisa.it

Sokjoon Lee
Gachon University
Seongnam-si, Republic of Korea
junny@gachon.ac.kr

ABSTRACT

Smart contracts are susceptible to various vulnerabilities that can lead to significant financial losses. The usage of tools for vulnerabilities is reducing the threats but presents some limitations related to the approach used by the tool itself. This paper presents a novel approach to smart contract vulnerability detection utilizing Large Language Models (LLMs), as a tool to detect all the vulnerabilities at once. Our proposed tool leverages the advanced natural language processing capabilities of LLMs to analyze smart contract code and identify potential security flaws. By training the LLM on a diverse dataset of known smart contract vulnerabilities and secure coding practices, we enhance its ability to recognize subtle and complex vulnerabilities that traditional static analysis tools might miss. The evaluation of our tool demonstrates its effectiveness in detecting a wide range of vulnerabilities with satisfaction and accuracy, providing developers with a robust mechanism to improve the security of their smart contracts before deployment. This approach signifies a significant advancement in the application of artificial intelligence for blockchain security, highlighting the potential of LLMs to enhance the reliability and safety of decentralized applications.

CCS CONCEPTS

• **Security and privacy** → **Vulnerability scanners**; **Distributed systems security**;

KEYWORDS

Large Language Models, LLama, Vulnerabilities Detection, Smart Contract

1 INTRODUCTION

Smart contracts have gained significant traction across various industries due to their potential to streamline processes and reduce reliance on intermediaries [1]. These are self-executing contracts with the terms of the agreement directly written into code. They run on blockchain platforms, such as Ethereum, and automatically enforce the terms of the contract when predefined conditions are met [33]. Despite their potential benefits, smart contracts are susceptible to vulnerabilities that can lead to financial losses and reputational damage [38]. These vulnerabilities often stem from various sources, including programming errors, where developers inadvertently introduce bugs or vulnerabilities during the coding process. Additionally, design flaws in the architecture of the smart contract or unexpected interactions with the underlying blockchain platform can create exploitable weaknesses. These vulnerabilities may go unnoticed during development or deployment but can be exploited by malicious actors to manipulate contract logic, steal funds, or disrupt the intended functionality of the smart contract [16]. Exploiting these vulnerabilities can have severe consequences, including unauthorized fund transfers, denial of service attacks, or manipulation of contract logic. Such exploits can result in financial losses for contract participants, undermine trust in the smart contract ecosystem, and tarnish the reputation of blockchain platforms. Therefore, it is crucial for smart contract developers to be aware of these vulnerabilities and implement robust security measures to mitigate their risks [29].

Security analysis within this context has evolved over time, initially relying on manual code review and later incorporating static analysis tools. While these methods have been effective to some extent, they often fall short in detecting subtle vulnerabilities and require significant human effort to conduct thorough assessments. Manual code review, although meticulous, can be time-consuming and prone to oversight, particularly for complex smart contracts with intricate logic. Similarly, static analysis tools, while providing automated checks for known vulnerabilities, may struggle to identify novel or nuanced security threats. As smart contracts become increasingly integral to various industries, there is a growing need for more sophisticated and efficient security analysis techniques [20].

Language model-based vulnerability analysis presents a promising alternative to traditional methods, leveraging advancements in Natural Language Processing (NLP) and Machine Learning (ML) [25]. This enables them to interpret code snippets, identify patterns indicative of vulnerabilities, and generate responses or predictions based on learned associations [22]. In the last few years, the evolution of ML created a new generation of models, the so-called Large Language Model (LLM), whose adoption in the context of vulnerability assessment is currently under investigation in various domains and can only be done after a fine-tuning phase using the domain-specific task of vulnerability analysis [12]. The adoption of LLM-based approaches poses challenges, such as the need for large and representative datasets, potential biases in model predictions, and difficulties in interpreting model decisions. Addressing these challenges will be crucial in realizing the full potential of language model-based vulnerability analysis in practice [35].

This research aims to bridge this gap by focusing specifically on the use of language model-based approaches for smart contract vulnerability analysis. By fine-tuning language models such as *Llama-2-7b-chat-hf* on a dataset of smart contract vulnerabilities, this study seeks to adapt these models to the unique characteristics

of blockchain-based applications. The considered dataset has been carefully curated to include examples of common smart contract vulnerabilities, annotated with relevant metadata such as vulnerability type and related lines of code. Through a rigorous evaluation process, this study intends to assess the effectiveness of LLM-based approaches in enhancing smart contract security. By comparing the performance of the fine-tuned models against traditional methods and benchmark datasets, the research aims to provide valuable insights into the potential benefits and limitations of LLM-based vulnerability analysis in the context of blockchain-based applications. Ultimately, the findings of this study will contribute to advancing the state-of-the-art in smart contract security and inform the development of more robust and resilient blockchain systems.

The primary objectives of this research are multifaceted and aim to contribute significantly to the field of smart contract security:

(1) **Mapping between Vulnerabilities**: To summarize the vulnerabilities and offer a unique perspective we unified the possible vulnerabilities listed according to OWASP and SWC, creating a mapping between them.
(2) **Fine-Tuning of Language Models:** We propose a fine-tuned model based on existing *Llama-2-7b-chat-hf* trained on a dataset specifically curated for smart contract vulnerabilities. This process entails adapting the pre-trained language model to recognize and understand the nuances of smart contract code, ensuring that it can effectively identify and categorize vulnerabilities.
(3) **Performance Comparison:** The third objective involves comparing the performance of the fine-tuned *Llama-2-7b-chat-hf* with respect to current popular analysis tools in detecting smart contract vulnerabilities. This comparative analysis will provide insights into the strengths and weaknesses of each model, helping to identify which approach is more effective for smart contract security analysis.
(4) **Evaluation of Practical Implications:** Lastly, the third objective aims to evaluate the practical implications of language model-based vulnerability analysis for smart contract development and auditing.

By achieving these objectives, the research seeks to advance our understanding of language model-based vulnerability analysis in the context of smart contracts and contribute to the development of more effective and robust security measures for blockchain-based applications.

The thesis is organized into six chapters, described as follows:

Section 2 - *Background*: A complete overview of existing vulnerabilities is given in this section, by splitting the discussion between OWASP and SWC.

Section 3 - *State of The Art*: In this section, a discussion on current works implementing ML-based vulnerability analysis is provided in order to highlight key enhancements of current research.

Section 4 - *Methodology* : It outlines the methodology employed in fine-tuning the *Llama-2-7b-chat-hf* model for smart contract vulnerability analysis. It provides a detailed description of the experimental setup, including the selection of datasets, preprocessing steps, and hyperparameter tuning techniques. The chapter also discusses the process of fine-tuning the language models and describes the evaluation metrics used to assess their performance.

Section 5 - *Results*: This chapter presents the experimental results obtained from evaluating the performance of the fine-tuned *Llama-2-7b-chat-hf* model. It includes quantitative analyses, such as accuracy, precision, recall, and F1 score, as well as qualitative assessments of the models' capabilities in detecting and mitigating smart contract vulnerabilities. The section also discusses any observed trends or patterns in the results and compares them with existing literature and benchmark datasets.

Section 6 - *Conclusion*: The final section concludes the manuscript by summarizing the key findings, highlighting contributions, and suggesting avenues for future research.

2 BACKGROUND

The increased integration of blockchain technology into software solutions has ushered in a new era of security measures, but it has also brought about its own set of challenges. While blockchain offers a decentralized and secure method for data storage, it introduces constraints, notably the immutability of deployed smart contracts. Once a smart contract is deployed, it cannot be altered, posing significant implications for security and risk management [27].

Possible vulnerabilities range from coding errors and logical flaws to design weaknesses, with examples including reentrancy attacks, integer overflow/underflow, and unchecked external calls [24]. OWASP Top 10 and SWC Registry provide a complete overview of the possible vulnerabilities included in smart contracts.

2.1 OWASP Top 10

The vulnerabilities are more general and may include multiple pattern. In this section we include an overview of these.

- **Reentrancy:** A reentrancy attack happens when a smart contract function is callable directly or indirectly by an external malicious contract before the first execution is completed. This can lead to unexpected behaviors, such as withdrawing funds multiple times. The most famous example is the DAO attack. To mitigate this, developers use the Checks-Effects-Interactions pattern to ensure all internal state changes are completed before calling external contracts[18]. *Attack Vector*: A smart contract traces the balance of a series of external address and allows them to withdraw money with a public function *withdraw()*. A malicious smart contract
- **Access Control:** Incorrectly setting a function's visibility (public, external, internal, private) can expose contracts to unauthorized access. Similarly, failing to use modifiers or other access control mechanisms properly can lead to unauthorized actions being performed on the contract [30]. It's crucial to review and restrict access to sensitive functions carefully[9].
  Attack Vector: A very popular schema for giving root privileges is to define the initialization address as the owner of the contract. Anyway, the initialization function can be called by anyone else, also after it has been previously called, enabling anyone to be the contract owner.
- **Arithmetic Problems:** Solidity uses fixed-size integer variables. An overflow occurs when a calculation exceeds the maximum value for a type, wrapping around to zero. Underflows happen in the opposite scenario, where a calculation

goes below zero and wraps around to the maximum value. These issues can lead to serious logical errors in contract execution. Using SafeMath library or newer Solidity versions with built-in overflow/underflow checks can prevent this[34].
Attack Vector: A function *withdraw()* is responsible for giving to the users the amount of Ether that users previously deposited in the contract. An attacker may try to withdraw an amount higher than his current balance modifying the check defined by a hypothetical function *withdraw()*, making the check always positive.

- **Denial of Service:** Denial of Service (DoS) attacks on smart contracts can unfold through various tactics, such as rendering functions inoperative by exploiting contract logic—for instance, by overcrowding contract slots to block new entries. To counteract these threats, it's essential to architect contracts that lessen dependency on external addresses for vital functions, incorporate safeguarding measures or contingencies to maintain contract integrity under attack, and circumvent infinite loops while managing gas expenditures to avert execution interruptions due to gas depletion[21].
Attack Vector: each block has an upper bound for the gas fee, namely the Block Gas limit. If such a limit is overcome, the transaction will be rejected. It is particularly dangerous if the attacker is able to modify the gas fee necessary.
- **Unchecked Low Level Calls:** In Solidity, low-level calls such as 'call', 'callcode', 'delegatecall', and 'send' present a unique challenge as they do not raise exceptions upon failure but instead return a boolean value indicating success or failure. Neglecting to verify this return value can erroneously lead a contract to assume a call was successful, introducing potential logic errors and security vulnerabilities. To mitigate these risks, it is critical to consistently check the outcome of low-level calls and, where possible, opt for higher-level constructs like transfer, which automatically reverts on failure
Attack Vector: When programmers do not check the return value of *call()*, which is used to send Ether to a smart contract that is not able to accept them, the EVM substitute return value with false. Since this value is not checked, the modifications done by smart contract are applied and cannot be reverted.
- **Bad Randomness:** Smart contracts, particularly those facilitating lotteries or games, often depend on pseudo-random number generators (PRNGs) to achieve necessary randomness. However, the inherently deterministic blockchain environment and the public visibility of data inputs for randomness generation, such as block hashes and timestamps, can compromise randomness, rendering it predictable and vulnerable to exploitation by attackers. To combat this, adopting external, reputable randomness sources, like Chainlink's Verifiable Random Function (VRF), is advised[5].
Attack Vector: The smart contract leverages on a number of blocks as a seed for randomness in a game; the attacker leverages on this to win the game.
- **Front Running:** Front-running occurs when someone with access to transaction information places their transaction in a way that benefits them, based on pending transactions they can see in the mempool. This is a broader blockchain issue but can affect contracts, for example, in decentralized exchanges. Mitigation strategies include using commit-reveal schemes or adjusting the contract design to reduce the predictability of outcomes[29].
Attack Vector: A public smart contract publish an RSA number; with a call to *submitSolution()*, the caller is rewarded. Consider the case in which someone submits the solution but someone else sees the transaction and sends it with a higher fee causing that the second transaction will be accepted before.
- **Short Addresses:** The Short Address Attack vulnerability arises when a smart contract fails to adequately validate the length of an address input, allowing attackers to manipulate the payload by appending fewer bytes. This manipulation can cause the smart contract to misinterpret the address, resulting in funds being misdirected. To mitigate this risk, it is crucial to rigorously validate address inputs for both length and format, leveraging Solidity's native address types that inherently manage such validations.
Attack vector: Considering the case in which an API uses a function that accepts a 20-byte address and an amount and transfers the amount to the address passed to the function. The attacker adds 12 zero bytes to the address in order to create a 32-byte address; and an amount of 20 tokens. Then the smart contract wrongly computes the receiving address and sends money to the attacker.
- **Timestamp Dependence:** Some contracts use block timestamps as a source of randomness or for critical functionality timing. However, miners have some control over the timestamp, which can lead to manipulation. Avoiding reliance on block timestamps for critical contract logic or using it in conjunction with other sources of randomness is advisable[29].
Attack vector: A game pays for the first player of the day. A malicious miner can set the timestamp to midnight and since the current time is really near to midnight (also if it is before), the other nodes decide to accept the block.

2.2 Smart Contract Weakness Classification (SWC) Registry

The Smart Contract Weakness Classification (SWC) Registry is a comprehensive repository of known vulnerabilities and weaknesses that can affect smart contracts. This registry serves as a crucial resource for developers, auditors, and researchers, providing detailed information on various types of vulnerabilities, their potential impacts, and mitigation strategies.

Traditional methods of vulnerability detection, such as manual code review and automated static analysis tools, have limitations in detecting subtle or complex vulnerabilities. However, recent advancements in vulnerability analysis, including the integration of machine learning and natural language processing techniques, such as sentiment analysis [15], offer promising avenues for enhancing the efficiency and effectiveness of vulnerability detection in smart contracts[4].

Table 1 reports all the most relevant SWC vulnerabilities. As it is possible to notice, these are much more detailed with respect to the

Table 1: Description of vulnerabilities in SWC.

SWC ID	Name	Description
SWC-101	Integer Arithmetic Bugs	Vulnerabilities related to arithmetic operations that can cause overflows or underflows, leading to incorrect calculations and potential exploits.
SWC-104	Unchecked Low Level Calls	This vulnerability arises when the return value of a low-level call, such as `call()`, `delegatecall()`, or `send()`, is not checked. Failure to handle errors can lead to unexpected behavior and security risks.
SWC-105	Unprotected Ether Withdrawal	Occurs when a contract lacks proper access controls for functions that withdraw Ether, enabling unauthorized users to withdraw funds.
SWC-106	Unprotected Selfdestruct	A vulnerability where the `selfdestruct()` function is accessible without adequate access control, allowing anyone to destroy the contract and potentially claim the remaining Ether.
SWC-107	Reentrancy	A vulnerability that occurs when a function makes an external call to another contract before resolving the current state, potentially allowing attackers to exploit this sequence and perform unintended actions such as draining funds.
SWC-113	Multiple Calls in a Single Transaction	This vulnerability allows an attacker to make multiple calls to a function within a single transaction, potentially bypassing intended logic or limits.
SWC-114	Transaction Order Dependence	Vulnerability where the outcome of a transaction depends on its order within the block, leading to potential exploitation through techniques like front-running.
SWC-115	Dependence on tx.origin	This vulnerability involves relying on the `tx.origin` variable for authorization. Attackers can exploit this by tricking a contract into believing a transaction originated from a trusted address.
SWC-116	Dependence on Predictable Environment Variable	This vulnerability occurs when a contract relies on environment variables that can be predicted or manipulated, such as block timestamps, compromising the contract's integrity.
SWC-120	Dependence on Predictable Environment Variable	Similar to SWC-116, this involves reliance on predictable environment variables for randomness, which can be manipulated to an attacker's advantage.

OWASP Top 10 and related to singular aspects of the smart contracts. This is a crucial aspect in the identification of vulnerabilities since more generic descriptions are hard to encode in ML-based tools. In the following sections, we will show a clear mapping between these two categories that will be leveraged for the creation of our model.

By understanding and addressing these vulnerabilities, researchers, and practitioners can strengthen the security posture of blockchain-based applications and mitigate the risks associated with immutable smart contracts.

3 STATE OF THE ART

Security analysis tools for smart contracts play a crucial role in identifying vulnerabilities and ensuring the reliability and safety of smart contract code before deployment. These tools vary in their approach, from static analysis, which examines code without executing it, to dynamic analysis, which involves running the code, and formal verification, which mathematically proves the properties of the code. In our previous work, we investigated the role of classical tools and their characteristics [3], while in this manuscript, we focus on ML-based frameworks. A primary approach has been proposed by Xingxin et al. [39] with the tool DeeSCVHunter. The research demonstrates how the deep learning framework automates the detection process. It is based on the concept of Vulnerability Candidate Slice (VCS) which is able to detect critical instructions through semantic dependence. Through extensive testing on real-world datasets, VCS has proven to enhance detection capabilities significantly, achieving up to a 25.76% improvement in the F1-score compared to classical methods. This demonstrates superior performance in accurately identifying smart contract vulnerabilities and validates semantical similarity in this field. The limitation of this work is the number of vulnerabilities detected, which only include reentrancy and time dependence. Peng et al. [26] review highlights an innovative approach utilizing deep learning, specifically a bidirectional Long Short Term Memory (LSTM) network with an attention mechanism (BLSTM-ATT), to accurately detect reentrancy bugs. It introduces the concept of contract snippet representations, designed to capture essential semantic information and control flow dependencies in smart contracts. An extensive analysis of over 42,000 real-world smart contracts demonstrates that this method significantly surpasses existing state-of-the-art techniques. This review underscores the practicality and potential of applying deep learning technologies in the realm of smart contract vulnerability detection, paving the way for further research in this critical area. Huang et al. [13] introduce a multi-task learning-based model for smart contract vulnerability detection, which enhances detection capabilities by incorporating auxiliary tasks focused on specific vulnerability features. Utilizing a hard-sharing architecture, the model employs a bottom-sharing layer for semantic learning and a task-specific layer for targeted function execution, leveraging word and positional embedding, attention mechanisms, and convolutional neural networks to improve the identification of different vulnerability types. Experimental results demonstrate the model's effectiveness in recognizing three distinct vulnerability types, showing superior performance in efficiency and resource consumption compared to single-task models.

One of the primary uses of LLM within the context of security assessment has been investigated by Gabriel et al. [23]. The research explores the potential of ChatGPT in providing recommendations for secure solution implementations through effective prompt engineering. This study underscores the dual role of prompt engineering in both unlocking ChatGPT's capability to aid in security solutions and mitigating the risk of generating advice that could compromise system integrity. By focusing on the precision of the interaction with AI models, the authors showcase the nuanced balance between leveraging AI for security enhancements and ensuring the

AI's guidance does not become a source of vulnerability itself. Sihao et al. [10] presents a detailed examination of using LLMs, like GPT-4, to uncover vulnerabilities in smart contracts, highlighting the balance between detecting true vulnerabilities and minimizing false positives. Through empirical research, it reveals that increased randomness in answers boosts the chance of correct detection but raises false positives. To address this, the paper introduces GPTLens, an adversarial framework dividing the detection process into two phases: generation and discrimination, where the LLM serves as both auditor and critic. This dual-role approach, aiming to expand vulnerability detection while reducing inaccuracies, significantly outperforms traditional methods, showcasing GPTLens as a versatile, LLM-based solution without requiring smart contract expertise. A more detailed study has been conducted by LLM4Vuln [32] with the aim of decoupling LLMs' vulnerability reasoning capability from their other capabilities. Results shown an interesting results of the adoption these models in the vulnerabilities assessment.

There are advantages of each tool, such as precision in detecting specific vulnerabilities or innovative analysis techniques. Conversely, there are also tool's limitations, pointing out areas where improvements are needed or where their applicability might be restricted. This comparative view is essential for understanding the landscape of smart contract security tools, and guiding future research and development efforts towards creating more comprehensive and effective solutions. Fine-tuned large language models (LLMs) introduce a novel paradigm in vulnerability analysis by adopting an approach that leverages training on specific datasets of smart contracts. This enables the models to focus explicitly on identifying and addressing vulnerabilities within these contracts. Our study embarks on a unique trajectory within the realm of ML techniques for smart contract vulnerability analysis. We are pioneering the use of fine-tuned LLMs, specifically *Llama-2-7b-chat-hf* and *GPT-2-XL*, by training them on a curated, preprocessed dataset used in VulnHunt-GPT [3] enriched with targeted vulnerability remediation strategies for Ethereum smart contracts. This process allows the models to not only learn from but also to precisely identify vulnerable segments within smart contracts, offering tailored remediation solutions.

Different from other work in this field, our aim is to demonstrate the efficacy of a relatively low-weight model. Other work in this field demonstrates the feasibility of LLMs in this field without a clear fine-tuning process needed for correct knowledge enrichment. In this manuscript, we want to address this limitation by extending our previous finding [3] to another model.

4 METHODOLOGY

This chapter explores the deployment of Large Language Models (LLMs) for identifying vulnerabilities within Ethereum smart contracts. LLMs are at the forefront of artificial intelligence (AI) research, representing a significant leap in the ability of machines to process, understand, and generate human language [40]. These models are built upon the foundation of neural networks, more specifically, a type known as transformers, which have revolutionized natural language processing (NLP) due to their efficiency and scalability. The core idea behind LLMs is to create a model that can learn the complexities of language from vast amounts of text data, enabling it to perform a wide array of language-related tasks without needing task-specific training. Fine-tuning allows language models to adapt their representations to the nuances of the target domain, improving their performance on specific tasks such as vulnerability detection and classification. In this section, we describe the process of utilizing advanced LLMs, such as *Llama-2-7b-chat-hf* and *GPT-2-XL*, to analyze and detect potential security flaws in smart contracts. This methodology not only highlights the technical steps involved in preparing, training, and fine-tuning these models on Ethereum contract data but also emphasizes the broader aim of leveraging AI's deep learning capabilities to enhance blockchain security. Through a detailed examination of model selection, data preprocessing, and evaluation strategies, this section aims to provide a comprehensive framework for applying LLMs in the ongoing effort to safeguard smart contract deployments against vulnerabilities.

4.1 Vulnerabilities Mapping

In Section 2.1, we explored two distinct approaches to vulnerability classification. Nevertheless, it is crucial to establish a unified method before proceeding with fine-tuning. OWASP vulnerabilities are too broad and challenging to map to a single aspect or specific line of a smart contract. As shown in Table 2, *Access Control* can correspond to *SWC 105, 106, 115* due to the complexity of this threat [28]. This could involve an unprotected withdrawal (SWC 105), an unprotected self-destruct (SWC 106), or a more conventional dependence on *tx.origin* (SWC 115). Similarly, two SWC vulnerabilities can overlap within the OWASP classification. For instance, *SWC 107*, while clearly identified as reentrancy, can also result from unchecked low-level calls. Some other vulnerabilities instead admit an injective mapping between OWASP and SWC.

Table 2: Mapping between OWASP Top 10 and SWC Vulnerability.

OWASP Vulnerability	SWC Vulnerability
Reentrancy	SWC 107
Unchecked Low Level Calls	SWC 107
	SWC 104
Access Control	SWC 105
	SWC 106
	SWC 114
	SWC 115
Time Manipulation	SWC 116
Bad Randomness	SWC 120
Other	SWC 114
Short Addresses	SWC 105
Front Running	SWC 114
Arithmetic	SWC 101
Denial of Service	SWC 113

To improve the quality of our fine-tuned LLM, we moved to SWC classification, and then, as we will show in the results section, we used this mapping to compare with the results of other tools.

4.2 Model Selection

For the fine-tuning process, two models, *Llama-2-7b-chat-hf* and *GPT-2-XL*, were selected for their novelty in research applications and their lightweight architecture, making them suitable for fine-tuning in environments like Google Colab using the GPU T4 runtime. More advanced models are not currently available for fine-tuning tasks.

- ***Llama-2-7b-chat-hf***: LLaMA-2 is a series of large language models created by Meta AI, known for their competitive performance despite smaller sizes. Released in multiple parameter sizes (7B to 65B), LLaMA-2 is notable for its accessibility and Meta's focus on addressing issues of bias and safety. As an open-source model, LLaMA-2 facilitates a wide range of potential applications in content generation, research, and conversational AI, while highlighting the importance of responsible use within a rapidly evolving AI landscape. For our experiment we selected the version of 7B parameters due to the constraints of our setup.
- ***GPT-2-XL***: GPT-2 is a large language model developed by OpenAI, renowned for its exceptional text generation capabilities. It stands out for its ability to produce different creative text formats, demonstrating a grasp of language nuances and patterns. OpenAI's staged release of GPT-2 brought attention to concerns around the potential misuse of LLMs, emphasizing the need for responsible development and deployment of such powerful AI models. For the experiment, we selected the *GPT-2-XL*, as it is the powerful version of GPT-2 with 1.5B parameters.

Considering the differences in the number of parameters of the selected models, as anticipated, they perform differently both in terms of accuracy and the time required to generate a response, even when using the same set of domain-specific knowledge as a dataset for the fine-tuning process. Fig. 1 illustrates the main differences in response quality between these two models. *GPT-2-XL* provides an incoherent response since it correctly identifies the vulnerability but associates it with the wrong line of code. In contrast, *Llama-2-7b-chat-hf* not only offers a good classification for a vulnerability but also provides insights into possible remediation. We computed an average response generation time of 20 seconds for the first model and 30 seconds for the second. Despite this result, which seems to favor *GPT-2-XL* in terms of speed, the accuracy of the first model is lower compared to the second. The *Llama-2-7b-chat-hf* model was right **82%** of the time, which means it's pretty effective at picking out the contracts that might cause problems. The *GPT-2-XL* model was accurate **62%** of the time on the same set of 20 smart contracts. So, while it is still uncertain to find vulnerabilities, it's not as accurate as the other. This comparison shows that the *Llama-2-7b-chat-hf* model is a stronger tool for identifying vulnerabilities in smart contracts, suggesting it might be a better choice for this specific task.

One of the hallmarks of LLMs is their scale. These models consist of billions of parameters (the parts of the model that are learned from training data). This scale allows LLMs to capture a wide range of language nuances, but it also requires significant computational resources for training and inference, raising questions about energy consumption and accessibility.

As these models scale, their ability to understand and generate text improves, often reaching a point where their output is indistinguishable from that of a human. This capability has led to groundbreaking applications in AI but has also prompted discussions about ethical considerations, including the potential for generating misleading information, the environmental impact of training large models, and concerns about copyright and content authenticity. Some of the famous LLM models are:

- **GPT Series by OpenAI:** *GPT-2-XL*, stands out as a highly recognized Large Language Model (LLM) for its ability to produce text that closely mimics human writing. It has been applied in diverse areas, such as aiding in content creation and programming. The model has also inspired various specialized versions tailored to specific languages or professional domains. Despite its extensive language comprehension capabilities, the vast scale of GPT-3 demands considerable computational resources.
- **LLaMA-2 by Meta:** *Llama-2-7b-chat-hf* marks Meta's entry into the realm of efficient and scalable Large Language Models (LLMs), aiming to deliver top-tier language understanding and generation while minimizing computational demands. Its architecture supports various sizes and specializations, facilitating the creation of models optimized for different computational settings or targeted applications. With a strong emphasis on efficiency and accessibility, LLaMA-2 seeks to extend the reach of LLM technologies across a broader array of devices and platforms, reducing the hefty resource requirements associated with earlier models.

4.3 Architecture

Fig. 2 depicts the process and comparative results of fine-tuning a machine learning model for the purpose of vulnerability detection in smart contracts. Initially, the model, which has 7 billion parameters, outputs results that are somewhat imprecise. It correctly identifies the type of vulnerability, such as arithmetic overflow or underflow, but it falls short in pinpointing the exact vulnerable line of code, showing the entire function instead. Furthermore, it suggests numerous remediation strategies, many of which are irrelevant to the specific vulnerability at hand. The central part of the diagram represents the fine-tuning process, where the predefined model undergoes training using a dataset composed of smart contracts containing known vulnerabilities. This specialized training is key to enhancing the model's accuracy in vulnerability detection.

After the fine-tuning process, the model's output results are markedly improved. The type of vulnerability is still accurately identified. However, now the model specifies the vulnerable line of code, such as *count -= input* which directly aids in locating the precise point of concern within the code. In terms of remediation, the model provides targeted advice, recommending the use of the SafeMath library, which is known for mitigating risks of arithmetic overflow and underflow.

4.4 Dataset Preparation

After establishing a model to use for performing fine-tuning, it is necessary to prepare the dataset in adequate form, according to the tokenizer. The goal is to ensure the data accurately represents the

Figure 1: *GPT-2-XL* vs. *Llama-2-7b-chat-hf* short address vulnerability detection.

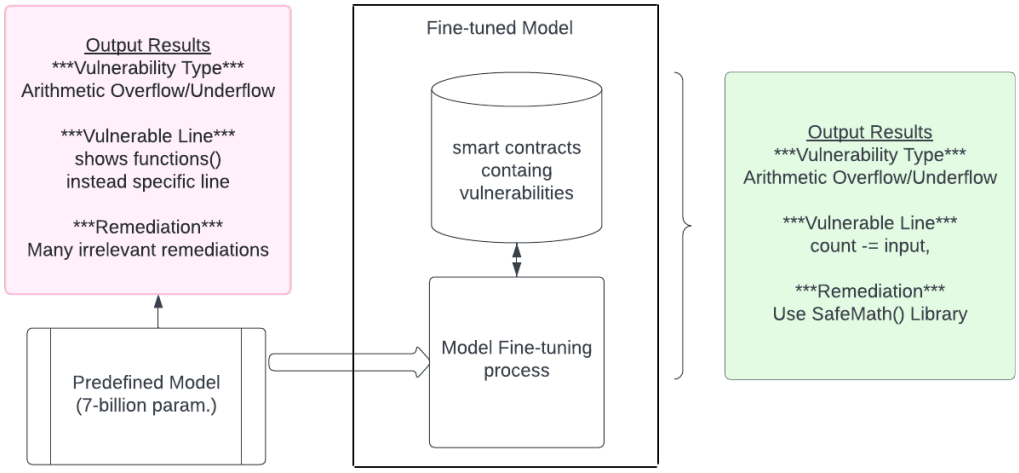

Figure 2: Fine-Tuning Architecture.

task, enabling the pre-trained model to adapt its knowledge to the specific domain or problem. Proper data preparation is important for the success of fine-tuning, as it directly affects the model's learning and performance.

- **Data Collection**: To enhance language model performance to identify smart contract vulnerabilities we leveraged two sources of data. The first dataset [14] contains 6003 GPT-generated human instruction and Solidity source code data pairs needed to tune the model in a smart contract generator. This is needed to increase the capabilities of the model in generating code and associate it with real-world scenarios. The second dataset comes from benchmark dataset of Solidity smart contract [11], in its re-labeled form used introduced in [31]. This dataset includes 609 vulnerable contracts, containing 1117 vulnerabilities meticulously organized according to OWASP's classification.
- **Data Processing**: Considering the structure of these datasets, which is in a tabular form, a more language-oriented approach is needed for the fine-tuning task. In particular, dataset processing involves organizing the collected data into a specific structure according to the tokenizer used by *Llama-2-7b-chat-hf* that allows the model to interpret and generate results from the provided data. The process of dataset re-creation consists of a phase where we encoded conversation in tags according to the given tokenizer and a phase where we included the vulnerabilities in a discursive form. The resulting dataset has been stored in a Hugging Face repository [2].

4.5 Fine-Tuning Process

LLMs utilize a transformer architecture, characterized by its attention mechanism. This mechanism allows the model to weigh the

importance of different words in a sentence, enabling it to understand context and meaning more deeply than previous models [37]. The architecture is designed to handle sequences of data, such as sentences, and can process all parts of the sequence simultaneously, making it highly efficient for language tasks. Fine-tuning is a critical step in adapting Large Language Models (LLMs) to perform specific tasks with high accuracy. While these models are initially trained on vast amounts of general text data, which helps them understand and generate language, they may not be optimized for specialized tasks straight out of the box. That's where fine-tuning comes into play [7]. LLMs are pretrained on an extensive corpus of text. Model is fine-tuned using base model *Llama-2-7b-chat-hf* on a L4 GPU with high RAM using Google Colab (4.82 credits/hour). Note that a L4 only has 22.5 GB of GPU, which is enough to store *Llama-2-7b-chat-hf* weights (7b × 2 bytes = 14 GB in FP16). In addition, we need to consider the overhead due to optimizer states, gradients, and forward activations. In order to reduce memory consumption, parameter-efficient fine-tuning (PEFT) techniques **QLoRA** [6] is used. QLoRA It is an efficient fine-tuning method that significantly reduces the memory requirements for fine-tuning LLMs. It utilizes a novel 4-bit quantization technique and introduces several innovations such as 4-bit NormalFloat (NF4) for optimal weight representation, Double Quantization to further reduce memory footprint, and Paged Optimizers to manage memory spikes effectively [17]. This approach allows for state-of-the-art performance with reduced memory usage, making large model fine-tuning more accessible

By leveraging the Hugging Face ecosystem with the transformers, accelerate, peft, trl, and bitsandbytes libraries, the necessary components are installed and loaded. Bitsandbytes is configured for 4-bit quantization, and the *Llama-2-7b-chat-hf* model is loaded in 4-bit precision on a GPU along with the corresponding tokenizer. Configurations for QLoRA, regular training parameters, and the setup for passing everything to the SFTTrainer are established. The number of epochs is fixed at 5 to avoid overfitting and overspecialization of the model while ensuring a satisfactory level of accuracy. As reported in Fig. 3, the model achieves a final loss of 0.366200.

5 RESULTS

This chapter presents the quantitative results from fine-tuned *Llama-2-7b-chat-hf* for smart contract vulnerability analysis, with a focus on model accuracy. Accuracy, as a key performance measure, indicates the models' proficiency in correctly identifying vulnerabilities within smart contracts. The evaluation involved comparing model predictions against a dataset with known vulnerabilities, thereby determining the accuracy rate. This metric is crucial for assessing the practical utility of the models in identifying security issues. The results highlight the models' effectiveness and their potential application in enhancing smart contract security.

5.1 Dataset

To get the accuracy and generalizability of our model, we decided to take as validation dataset the Smartbugs Curated Dataset [8], widely used in the context of smart contract vulnerability assessment. The last version of this dataset contains 143 vulnerable smart contracts, divided into ten categories according to OWASP Top 10 Vulnerabilities described in Section 2.1.

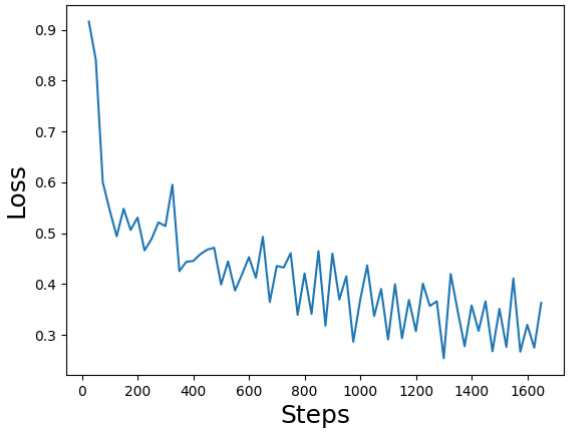

Figure 3: Steps vs. Loss of fine-tuned *Llama-2-7b-chat-hf*.

5.2 Analysis

After fine-tuning the *Llama-2-7b-chat-hf* model, we used it to predict which smart contracts might have vulnerabilities and then checked these predictions against a list of vulnerabilities we already knew about. The fine-tuned *Llama-2-7b-chat-hf* model was right **59,5%** of the time, which means it's pretty effective at picking out the contracts that might cause problems.

Despite not being perfect in its analysis, it demonstrates efficacy in almost all categories. In contrast, tools such as Osiris [36] and Oyente do not cover all vulnerabilities. Mythril performs exceptionally well but has a significantly longer execution time, as analyzed in our previous research [3]. Although this comparison shows that the LLaMA-2-7B model is not the best tool for identifying vulnerabilities in smart contracts, it opens new avenues for investigation.

In particular, as highlighted in Section 3, an LLM-based tool can be used by a more general audience and not only by security experts. Additionally, our framework leverages SWC, which offers a more extensive classification compared to the OWASP Top 10 and provides a mapping between SWC and the OWASP Top 10.

The precision of our model is particularly high in identifying arithmetic and reentrancy vulnerabilities, highlighting the model's capability to effectively identify and analyze the most represented vulnerability categories in the research dataset. However, Unchecked Low Level Calls include multiple different patterns, which might explain why the model is not very effective in their identification.

5.3 Challenges and Limitations

During the fine-tuning process for vulnerability analysis of smart contract several limitations were encountered, which also impact the results.

The decision to utilize models such as *Llama-2-7b-chat-hf*, which possess 8B parameters, presented initial obstacles. These models

Table 3: Comparison of the proposed approach with respect to other existing tools.

OWASP Vulnerability	Mythril	Osiris	Oyente	Securify	Conkas	LLM Proposed
Reentrancy (31)	30 (96.8%)	28 (90.3%)	28 (90.3%)	19 (61.3%)	31 (100%)	22 (71%)
Unc. Low Level Calls (52)	46 (88.5%)	14 (26.9%)	41 (78.8%)	21 (40.4%)	0 (0%)	28 (53.8%)
Arithmetic (15)	11 (73.3%)	15 (100%)	15 (100%)	0 (0%)	15 (100%)	14 (93.3%)
Front Running (4)	1 (25%)	0 (0%)	4 (100%)	2 (50%)	0 (0%)	2 (50%)
Access Control (17)	9 (52.9%)	0 (0%)	3 (17.6%)	7 (41.2%)	0 (0%)	8 (47.1%)
Denial of Service (6)	1 (16.7%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	3 (50%)
Bad Randomness (8)	3 (37.5%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	4 (50%)
Other (3)	0 (0%)	0 (0%)	2 (66.7%)	0 (0%)	0 (0%)	0 (0%)
Time Manipulation (5)	3 (60%)	2 (40%)	0 (0%)	0 (0%)	5 (100%)	4 (80%)
Short Addresses (1)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
	104 (73.2%)	**59 (41.5%)**	**93 (65.5%)**	**49 (34.5%)**	**51 (35.9%)**	**85 (59.9%)**

were pre-trained on relatively limited datasets, resulting in pre-trained outputs that lacked relevance to the specific task of vulnerability analysis in smart contracts. The inadequacy of pre-trained results underscores the importance of selecting a model not just based on its size or popularity, but also considering the nature and scope of its initial training data in relation to the target application.

To address the specificity of the task at hand, extensive effort was invested in preparing a comprehensive dataset. This dataset aimed to provide the models with a broad and deep understanding of smart contract vulnerabilities, facilitating the generation of pertinent outcomes. The preparation of such a dataset is critical in fine-tuning processes, especially when dealing with specialized domains like smart contract security, which require nuanced understanding and accurate identification of vulnerabilities.

The proposed research, in any case, wants to highlight the role of LLM in the security context, particularly in smart contract vulnerability assessment. Results are encouraging, and bigger models might be used to increase the accuracy, as well as to provide more accurate suggestions. Usage of Retrieval-Augmented Generation (RAG) can surely increase the performance of these models as demonstrated in [19].

6 CONCLUSION

The rise of smart contracts has revolutionized the way digital transactions are conducted, offering unprecedented efficiency and security. However, their immutable nature and the complexity of their code make them susceptible to vulnerabilities that can lead to significant financial losses and breaches of trust.

Through our research, we have demonstrated that LLMs, are adept at identifying and categorizing various types of smart contract vulnerabilities. The ability of LLMs to understand context, semantics, and patterns within the code allows for a more comprehensive and accurate detection process compared to traditional methods. Limitations of the current research are related to the performance of the proposed framework, which is in part caused by the weights of the base model used for fine-tuning. RAG and other contextualization strategies could be leveraged to enhance the performance of the approach while maintaining a reduced size.

ACKNOWLEDGEMENTS

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

REFERENCES

[1] Abdulaziz Aljaloud and Abdul Razzaq. 2023. Modernizing the Legacy Healthcare System to Decentralize Platform Using Blockchain Technology. *Technologies* 11, 4 (2023), 84.

[2] Biagio Boi. 2024. Solidity Smart Contract Vulnerability LLM Dataset. https://huggingface.co/datasets/biagioboi/ETH_SmartContractVulnerability_LLM

[3] Biagio Boi, Christian Esposito, and Sokjoon Lee. 2024. VulnHunt-GPT: a Smart Contract vulnerabilities detector based on OpenAI chatGPT. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*. 1517–1524.

[4] Saikat Chakraborty, R. Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48 (2020), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[5] K. Chatterjee, A. K. Goharshady, and Arash Pourdamghani. 2019. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (2019), 403–412. https://doi.org/10.1109/BLOC.2019.8751326

[6] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems* 36 (2024).

[7] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah A. Smith. 2020. Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping. *ArXiv* abs/2002.06305 (2020).

[8] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*. 1349–1352.

[9] Christof Ferreira Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. 2019. ÆGIS: Smart shielding of smart contracts. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2589–2591.

[10] Sihao Hu, Tiansheng Huang, Fatih Ilhan, Selim Fukan Tekin, and Ling Liu. 2023. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. *ArXiv* abs/2310.01152 (2023). https://doi.org/10.48550/arXiv.2310.01152

[11] Tianyuan Hu. 2023. A benchmark dataset of Solidity smart contracts. https://doi.org/10.5281/zenodo.7744053

[12] JIN Huan and LI Qinying. 2023. Fine-Tuning Pre-Trained CodeBERT for Code Search in Smart Contract. *Wuhan University Journal of Natural Sciences* 28, 3 (2023), 237–245.

[13] Jing Huang, Kuo Zhou, Ao Xiong, and Dongmeng Li. 2022. Smart Contract Vulnerability Detection Model Based on Multi-Task Learning. *Sensors (Basel, Switzerland)* 22 (2022). https://doi.org/10.3390/s22051829

[14] A. Kuhlman. 2023. Smart Contracts Instructions Dataset. https://huggingface.co/datasets/AlfredPros/smart-contracts-instructions.

[15] Junhee Lee, Flavius Frasincar, and Maria Mihaela Truşcă. 2023. DIWS-LCR-Rothop++: A Domain-Independent Word Selector for Cross-Domain Aspect-Based Sentiment Classification. *SIGAPP Appl. Comput. Rev.* 23, 3 (sep 2023), 19–31. https://doi.org/10.1145/3626307.3626309

[16] Hui Li, Ranran Dang, Yao Yao, and Han Wang. 2023. A Review of Approaches for Detecting Vulnerabilities in Smart Contracts within Web 3.0 Applications. *Blockchains* 1, 1 (2023), 3–18. https://doi.org/10.3390/blockchains1010002

[17] Yixiao Li, Yifan Yu, Chen Liang, Pengcheng He, Nikos Karampatziakis, Weizhu Chen, and Tuo Zhao. 2023. LoftQ: LoRA-Fine-Tuning-Aware Quantization for Large Language Models. *ArXiv* abs/2310.08659 (2023). https://doi.org/10.48550/arXiv.2310.08659

[18] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and B. Roscoe. 2018. ReGuard: Finding Reentrancy Bugs in Smart Contracts. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (2018), 65–68. https://doi.org/10.1145/3183440.3183495

[19] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. *arXiv preprint arXiv:2405.02580* (2024).

[20] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2021. Combining Graph Neural Networks With Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Transactions on Knowledge and Data Engineering* 35 (2021), 1296–1310. https://doi.org/10.1109/TKDE.2021.3095196

[21] G. Loukas and Gülay Öke Günel. 2010. Protection Against Denial of Service Attacks: A Survey. *Comput. J.* 53 (2010), 1020–1037. https://doi.org/10.1093/comjnl/bxp078

[22] Pouyan Momeni, Yu Wang, and Reza Samavi. 2019. Machine Learning Model for Smart Contracts Security Analysis. *2019 17th International Conference on Privacy, Security and Trust (PST)* (2019), 1–6. https://doi.org/10.1109/PST47121.2019.8949045

[23] Gabriel Poesia, Oleksandr Polozov, Vu Le, A. Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *ArXiv* abs/2201.11227 (2022).

[24] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph K. Liu, and R. Doss. 2019. Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey. *ArXiv* abs/1908.08605 (2019).

[25] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software Vulnerability Detection using Large Language Models. *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2023), 112–119. https://doi.org/10.1109/ISSREW60843.2023.00058

[26] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. *IEEE Access* 8 (2020), 19685–19695. https://doi.org/10.1109/ACCESS.2020.2969429

[27] Sara Rouhani and R. Deters. 2019. Security, Performance, and Applications of Smart Contracts: A Systematic Survey. *IEEE Access* 7 (2019), 50759–50779. https://doi.org/10.1109/ACCESS.2019.2911031

[28] Mersedeh Sadeghi, Luca Sartor, and Matteo Rossi. 2022. A semantic-based access control approach for systems of systems. *SIGAPP Appl. Comput. Rev.* 21, 4 (jan 2022), 5–19. https://doi.org/10.1145/3512753.3512754

[29] Dr Sarwar Sayeed, Héctor Marco-Gisbert, and T. Caira. 2020. Smart Contract: Attacks and Protections. *IEEE Access* 8 (2020), 24416–24427. https://doi.org/10.1109/ACCESS.2020.2970495

[30] Christoph Stach, Clémentine Gritti, Dennis Przytarski, and Bernhard Mitschang. 2022. Assessment and treatment of privacy issues in blockchain systems. *SIGAPP Appl. Comput. Rev.* 22, 3 (nov 2022), 5–24. https://doi.org/10.1145/3570733.3570734

[31] André Storhaug, Jingyue Li, and Tianyuan Hu. 2023. Efficient avoidance of vulnerabilities in auto-completed smart contract code using vulnerability-constrained decoding. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 683–693.

[32] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[33] Hamed Taherdoost. 2023. Smart Contracts in Blockchain Technology: A Critical Review. *Information* 14, 2 (2023). https://doi.org/10.3390/info14020117

[34] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Işil Dillig, and Yu Feng. 2021. SolType: refinement types for arithmetic overflow in solidity. *Proceedings of the ACM on Programming Languages* 6 (2021), 1 – 29. https://doi.org/10.1145/3498665

[35] Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, S. Çamtepe, J. Pieprzyk, and S. Nepal. 2022. Transformer-Based Language Models for Software Vulnerability Detection. *Proceedings of the 38th Annual Computer Security Applications Conference* (2022). https://doi.org/10.1145/3564625.3567985

[36] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th annual computer security applications conference*. 664–676.

[37] Jesse Vig and Yonatan Belinkov. 2019. Analyzing the Structure of Attention in a Transformer Language Model. (2019), 63–76. https://doi.org/10.18653/v1/W19-4808

[38] Guangfu Wu, HaiPing Wang, Xin Lai, Mengmeng Wang, Daojing He, and Sammy Chan. 2024. A comprehensive survey of smart contract security: State of the art and research directions. *Journal of Network and Computer Applications* (2024), 103882.

[39] Xingxin Yu, Haoyue Zhao, Botao Hou, Zonghao Ying, and Bin Wu. 2021. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[40] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, J. Nie, and Ji rong Wen. 2023. A Survey of Large Language Models. *ArXiv* abs/2303.18223 (2023). https://doi.org/10.48550/arXiv.2303.18223

ABOUT THE AUTHORS:

Biagio BOI obtained master's degree in 2022 at the University of Salerno, in Italy. He is currently pursuing his PhD at the University of Salerno, investigating the issues of digital transformation in the upcoming society. His interests include the internet of things, blockchain, authentication and authorization

Christian ESPOSITO is currently an Associate Professor at the University of Salerno, since December 2022. He was a Tenured Assistant Professor at the University of Salerno, an Assistant Professor at the University of Napoli "Federico II". He is an Associate Editor of the IEEE Access and has served as a guest editor for various special issues at international journals. His interests include positioning systems, reliable and secure communications, game theory, and multi-objective optimization.

Sokjoon LEE is currently an Associate Professor at Dept. of Computer Engineering (Smart Security) at Gachon University, since March 2022. He pursed his PhD at Korea Advanced Institute of Science and Technology. His research interests include IoT, Quantum, and Automotive Security.