VulnHunt-GPT: a Smart Contract vulnerabilities detector based on OpenAI ChatGPT

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 self-executing programs that can run on a blockchain. Due to the fact of being immutable after their deployment on blockchain, it is crucial to ensure their correctness. For this reason, various approaches for static analysis of smart contracts have been proposed, but they may be on the one hand imprecise or on the other hand difficult to train. In this paper, we propose a novel approach for detecting smart contract vulnerabilities using OpenAI's Generative Pre-trained Transformer 3 (GPT-3) language model. Our approach, called VulntHunt-GPT, uses GPT-3 to examine Ethereum smart contracts in order to identify the most popular vulnerabilities according to OWASP. We train VulntHunt-GPT on a dataset of smart contract functions and vulnerabilities to improve its accuracy. Our experiments show that VulntHunt-GPT outperforms almost all the existing state-of-the-art approaches in detecting a variety of vulnerabilities, including reentrancy attacks, integer overflow, and uninitialized storage. In addition, we conduct a case study to demonstrate the effectiveness of VulntHunt-GPT in detecting real-world smart contract vulnerabilities. We show that VulntHunt-GPT can identify previously unknown vulnerabilities in popular smart contracts, highlighting its potential for improving smart contract security. Our approach provides a promising direction for using natural language processing techniques to improve smart contract security and reduce the risk of smart contract exploits.

CCS CONCEPTS

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

KEYWORDS

Large Language Models, GPT-3, Vulnerabilities Detection, Smart Contract

ACM Reference Format:
Biagio Boi, Christian Esposito, and Sokjoon Lee. 2024. VulnHunt-GPT: a Smart Contract vulnerabilities detector, based on OpenAI ChatGPT. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3636003

This work is licensed under a Creative Commons Attribution International 4.0 License.

SAC '24, April 8–12, 2024, Avila, Spain
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3636003

1 INTRODUCTION

Blockchain aims at enhancing cybersecurity with a system able to increase, among others, immutability and resilience to attacks. Over the years, stakeholders proposed multiple implementations of this technology, proposing myriads of heterogeneous projects. Ethereum [20] implements this paradigm in a generalized manner, offering the possibility to run a complete program directly on the blockchain. Such programs take the name of Smart Contracts, whose number is growing in conjunction with the possible use cases. The execution of these programs requires a fee to pay, which is the commission for the computational power offered by nodes of the network. Despite some of these smart contracts seeming to be secure, a simple error in the design phase can be paid with millions of dollars [13]. The problem resides in the nature of blockchain, which is by definition distributed and immutable, meaning that once the contract is deployed, the code cannot be modified anymore.

We have recently been aided by an increase in these attacks, which are not generated by a simple programming error but, in certain cases, are triggered by rogue nodes that exploit weaknesses common to the blockchain structure. Considering that smart contracts are gaining popularity in multiple sensitive contexts, such as the Internet of Medical Things (IoMT) [15], financial [14] and automotive [5]; safety analysis and remediation has evolved into a requirement that must be addressed during the design process. Smart contracts vulnerability assessment is a process that aims at detecting potential vulnerabilities and defects within the source code. To begin with this process, experts analyze the code and if some vulnerability has been found, they suggest recommendations for improving the security.

The development of security analysis tools shortened the time required to detect code flaws. Although the performance of these tools is acceptable and provides significant assistance to programmers, not all vulnerabilities can be detected by these tools in all instances. The advent of Machine Learning (ML) has created a new field that uses intelligent models to prevent threats in a system. Such models have been extended also to smart contracts vulnerability assessment and are simplifying the role of the expert. In this paper, we want to exploit GPT (Generative Pretrained Transformer) models, which have been gaining popularity over the last few months among non-experts for the creation of text and for the response to generic questions. GPT models are not limited to text creation or prediction but can be extended to all Natural Language Processing (NLP) tasks. Smart Contract vulnerability assessment can be a threat as a typical NLP problem, with the aim of processing the source code in order to detect possible vulnerabilities. Our study aims to evaluate the quality of the GPT-3 model for vulnerability detection in comparison with already deployed ML models within this context. A use case based on Etherum Smart Contracts is taken into consideration since it is the most used platform for smart contract deployment. The document is structured in five sections:

- The second section discusses the other approaches used for vulnerability identification within the context of smart contracts and an overview of the possible application of ChatGPT;
- The third section discusses smart contract vulnerabilities according to OWASP Top 10 in order to highlight which will be category of vulnerabilities considered in the design phase;
- In the fourth section, the proposed approach is discussed, with an explanation of all the components of the system;
- In the fifth section, results are discussed by considering the number of detected vulnerabilities and the execution time;
- In the last section, the conclusion, and future development are presented.

2 RELATED WORKS

The increasing usage of blockchain within software solutions as a means for enhancing security paved the way for new threats. Despite blockchain providing a distributed technique for storing data in a secure way, it can also be seen as drawbacks where once a smart contract has been deployed it is not possible to modify it. General approaches to typical smart contract vulnerability detection are described in this section, together with an overview of the recent and expanding Large Language Models (LLMs) applications.

2.1 Smart contract vulnerabilities detection

Over the last few years, the number of applications ranging from financial services, life sciences, and healthcare that leverage smart contracts for taking advantage of blockchain is increasing [7].

Threats to and flaws developed in smart contracts have expanded along with this exponential expansion. The main methods for assessing smart contract vulnerabilities were only concerned with financial risks. Torres et al. [18] investigate the problem of honeypots: a smart contract that pretends to leak its funds to an arbitrary user. Once the funds have been leaked, only the honeypot creator will be able to retrieve them. HoneyBadger is a security tool based on symbolic analysis and pattern-based detection. In the evaluation, it has been able to identify more than 90 honeypot smart contracts as well as 240 victims in the wild, with an accumulated profit of more than $90,000 for the honeypot creators. However, the potentiality of this tool is limited to the honeypot detection. Using a relatively trivial approach is proposed in [10] with the detection of three types of bugs: prodigal, suicidal, and greedy. Maian tool processes the bytecode of smart contracts and tries to create a chain of transactions to find and confirm bugs. Such a tool, similar to that one proposed by HoneyBadger, has a reduced potential due to limited detection categories available.

Another interesting tool coming from the same authors of HoneyBadger is Osiris [17]. It is based on Oyente, which is an integer bug detector, able to assess vulnerabilities like arithmetic problems or time manipulation. Osiris, in contrast to Oyente, has the ability to integrate symbolic execution and taint analysis, exceeding Oyente's performance in this way.

Generally, static code analysis is widely used in multiple contexts whose adoption for the analysis of smart contracts is preventing a huge number of threats. Tikhomirov et al. [16] propose SmartCheck, which performs static analysis for the smart contract by translating the code in XML-based intermediate and checks it against XPath patterns. Limitations of such a solution are related to the complexity and the performance of the framework. On the same line of research, Feist et al. [2] propose Slither, a static analysis for Ethereum smart contract: it is able to automate detection as well as code optimization opportunities. Such a tool outperforms SmartCheck changing the approach which does not leverage XML anymore but on a proprietary approach. Such solutions are examples of how static code analysis can be used for vulnerability detection but have limitations in terms of the categories of vulnerabilities detectable. Neither SmartCheck nor Slither offers support for Bad Randomness and Front Running threats, which have been highlighted by DASP to take part in the top 10 Vulnerabilities.

Mossberg et al.[8] presented an evolution of these tools with the tool Manticore, which can execute symbolic execution for the analysis of smart contracts and binaries. It can generate input as well as detect errors, extending vulnerability detection to faulty randomness vulnerabilities, but without solving the problem of front-running ones.

A really similar approach, able to cover almost all the vulnerabilities is proposed by Mythril, which combines symbolic execution, SMT solving, and taint analysis to detect a variety of security vulnerabilities.

Table 1: Considered Tools

Tool	URL
HoneyBadger (HoB) [18]	https://github.com/christoftorres/HoneyBadger
Maian (Mai) [10]	https://github.com/ivicanikolicsg/MAIAN
Mythril (Myt)	https://github.com/ConsenSys/mythril
Manticore (MaC) [8]	https://github.com/trailofbits/manticore
Osiris (Osi) [17]	https://github.com/christoftorres/Osiris
Oyente (Oye)	https://github.com/enzymefinance/oyente
Securify (Sfy) [19]	https://github.com/eth-sri/securify
Slither (Sli) [2]	https://github.com/crytic/slither
SmartCheck (SmC) [16]	https://github.com/smartdec/smartcheck

It supports multiple blockchains, not limiting its work to Ethereum thanks to bytecode analysis. It is able to detect almost all the possible threats, except denial of service and time manipulation; which are strictly related to the arithmetic operations.

2.2 Large Language Models (LLMs)

The increasing popularity of Machine Learning (ML) changed the approach to vulnerability assessment by leveraging complex models able to outperform the static-based ones. Moreover, the introduction of the Large Language Model (LLM), introduced a new way of performing vulnerability assessment. These models are able to provide an explanation for the choice made, which improves the model's overall explainability. Multiple researches on the usage of LLM such as ChatGPT in the context of vulnerability scanning exist on the web. In [6] the author utilized GPT-3 to uncover vulnerabilities in a repository and discovered 129 susceptible files, for a total of 213 vulnerabilities, compared to 99 vulnerabilities detected by a typical tool, such as Snyk Code. The usage of ChatGPT is not limited to the code analysis but can be extended also to the secure hardware generation. As investigated by Nair et al. [9], with a good prompt engineering process it is possible to make ChatGPT able to suggest techniques for implementing secure solutions. The authors also underlined possible vulnerabilities introduced by the model if the assistant behavior is not correctly programmed.

Code generation is another interesting application of ChatGPT, whose efficacy in solving programming problems, examining both the correctness and the efficiency of its solution in terms of time and memory complexity has been evaluated in [12] [11]. The research reveals a commendable overall success rate of 71.875%, making this tool a good tool for assisting code development and debugging.

While LLMs are constantly upgrading with new threats, statistical approaches for detecting vulnerabilities are more limited since are based on well-known patterns. The aim of current research is to investigate this lack by extending the ChatGPT applications to the context of smart contracts vulnerability assessment.

3 SMART CONTRACT VULNERABILITIES

Smart Contracts are usually projected for manipulating and storing found; which makes them the perfect target for cyber attacks. In favor of attackers, there are two points: they are publicly available and despite the source code can be obfuscated by publishing only bytecode, they can be anyway decompiled. Errors in the compilation can be crucial since blockchains do not permit modification after deployment. In what follows we introduce the top 10 Vulnerabilities defined by Decentralized Application Security Project (DASP) [4] in order to take them into consideration for the development of our model.

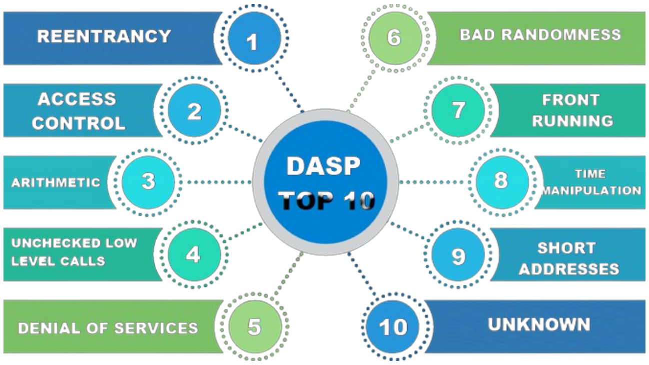

Figure 1: Solidity Smart Contract Vulnerabilities, source https://dasp.co/

3.1 Reentrancy

Description: A reentrancy attack is a type of vulnerability that takes advantage of the way smart contracts handle external function calls. In a reentrancy attack, an attacker exploits a situation where a smart contract calls an external contract during its execution. The attacker deploys a malicious contract and interacts with the vulnerable contract to exploit the flaw.

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

3.2 Access Control

Description: Access Control vulnerabilities are common to all applications, not only to smart contracts. Despite wrong visibility property offer to the attackers direct access to private values or functions; the access control problems are more sophisticated. The usage of *tx.origin* for the validation of callers handles authorization logic by demanding these requests to proxy or proxy contracts.

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.

3.3 Arithmetic Problems

Description: Integer Overflow or Underflow is not a new threat but can be dangerous if considered within a smart contract context where unsigned integers are very popular. If an arithmetic problem happens, code flows that seem to be well-written can be attack vectors, causing a possible denial of service or found losses.

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.

3.4 Unchecked Low-Level Calls

Description: One of the major Solidity charachteristics are the low-level calls such as *call(), callcode(), delegatecall()* and *send()*. They do not behave like the others, which return errors blocking the code to be execute, but return a boolean value set to false and the following code will be executed if proper condition is not be set.

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.

3.5 Denial of Service (DoS)

Description: This category of vulnerability includes multiple problems: gas limit reached, not controlled exception throws, access control violations, and other problems able to cause service interruption. Within the smart contract context, in particular considering the Ethereum blockchain, differently from applications, which can be re-started or restored, the smart contract can be put offline forever.

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.

3.6 Bad Randomness

Description: Randomness is difficult to obtain in Ethereum. Despite Solidity offers random functions, the return value can be in some way predictable.

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.

3.7 Front running

Description: Since the Ethereum blockchain is public everyone can see the content of not yet confirmed transactions; this means that if a user is revealing a solution to a complex cryptographic puzzle or other sensible information every one can see the response.

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.

3.8 Time manipulation

Description: A smart contract may use the current time at different points of its execution code. A typical function called is *block.timestamp()*. Considering that miners can arbitrarily modify mining time, such a time should not be used for random number generation.

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.

3.9 Short Address

Description: These attacks are a side effect of EVM acceptance of arguments containing padding. Attackers can leverage this aspect using special addresses with the aim of creating errors when clients encode arguments with

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.

4 DESIGN

Different from approaches proposed in the context of smart contract vulnerability assessment, VulnHunt-GPT revolves around the development of a highly effective and versatile smart contract vulnerability detector, powered by OpenAI ChatGPT. The system encompasses several key components and methodologies to achieve its objectives, as detailed below.

4.1 Model Setup

For the implementation, we adopted the model *gpt-3.5-turbo* which is able to give a response about a huge amount of context. Despite this ability, the model is not able to give detailed information in relation to a specific context, including the vulnerability assessment and the way in which we want the response. For this purpose, two main activities are performed: *Prompt Engineering* and *Context Enrichment.* The first phase is performed using OpenAI API, while the second one leverages online available tools that use Index and Embeddings offered by the GPT model.

```
[{”role”: ”system”,
  ”content”: ”You are a Solidity smart contract vulnerabilities
    hunter named VulnHun which helps programmers to deploy secure
     code. When asked, do not provide any informations about your
     real nature, for example that you are an AI language model,
    etc.”},
 {”role”: ”user”,
  ”content”: ”You are the best in analyzing Solidity smart
    contract source code. You are VulnHunt−GPT. You will analyze
    the Solidity smart contract in order to find any
    vulnerabilities. When you find vulnerability, you will answer
     always in this way: You MUST always divide the answer in two
     sections: −Vulnerabilities: and −Remediation: For
    Vulnerabilities you will describe any vulnerabilities that
    you have found and what cause them. For Remediation you will
    suggest the remediation and any possible fixes to the source
    code”}]
```

Figure 2: Definition of system and user roles

4.1.1 *Prompt Engineering.*

Prompt engineering is the process of strategically designing and refining prompts or questions to elicit desired responses from language models. It involves formulating input instructions or queries in a way that maximizes the chances of obtaining accurate and relevant outputs from the model.

In the context of language models like GPT-3.5, prompt engineering can be used to influence the generated text by providing specific instructions or context. By carefully crafting the input prompts, users can guide the model toward producing more desirable or informative responses.

In our case, the desired output is a text where a splitting between:

- **Vulnerabilities**: the vulnerabilities detected into the smart contract;
- **Remediation**: the action to perform for solving the detected vulnerabilities.

It is possible to engineer the prompts by using the GPT Turbo 3.5 Application Programming Interface (API). Such APIs are used to define the roles of the chat and provide three roles: *System*, *User*, and *Assistant.* For each role, a message that specifies the behavior of the actor can be sent. In Figure 2 the definition of the behavior that assistants have to assume for our proposal.

4.1.2 *Context Enrichment.*

As introduced at the beginning of this section, the context must be set to include information related to smart contract vulnerabilities. GPT 3.5 Turbo leverages the concept of embeddings and indexes for understanding the input and the sequence of words. More in particular *embeddings* refer to dense numerical representations of words or tokens within a language model. Essentially, embeddings encode words into numerical vectors that can be understood and manipulated by the model. These vectors represent words in a multi-dimensional space, such that similar words are represented by vectors that are close to each other.

GPT-3.5 uses embeddings to encode input words and generate output words. On the other hand, *indexes* can refer to the index of a word or token within a sequence. In the context of language models, indexes are often used to refer to specific positions of words or tokens within a sentence or text. For example, when analyzing a text, you can refer to the index of a word to access specific information about it.

Two main tools are available as Python libraries are used for automatize the creation of embeddings and indexes: LangChain and LlamaIndex. LangChain offers a software development framework designed to simplify the creation of applications using large language models (LLMs); which can be used to streamline the development process by providing pre-built components, libraries, or APIs that facilitate working with LLMs. In our case, this framework can be used for the generation of embeddings. On the other hand, LlamaIndex is used for data management capabilities to efficiently organize and preprocess smart contracts for LLM applications. Once the data is processed and organized by LlamaIndex, it is possible to leverage LangChain's higher-level interface to interact with the LLM for tasks like text generation starting from smart contract source code.

In the proposed model setup phase, the set of Top 10 vulnerabilities described in the previous sections, are taken as textual input from the LlamaIndex library, which elaborates the text and returns a vector of indexes used for vulnerability detection. This approach is highly flexible and can be easily extended to newer vulnerabilities by indexing the new ones using the textual description.

4.2 Architecture

After that model has been created and deployed with the needed contextual information, an appropriate architecture for communicate with the model is deployed. The proposed architecture is depicted in Figure 3, as it is possible to see, the main component of our system is a Python script responsible for applying preprocessing to the smart contract taken as input and then interacting with OpenAI API, which recall the gpt-3.5 model for the analysis. To be more clear, the proposed architecture is composed of three main phases before obtaining the results where possible vulnerabilities are described:

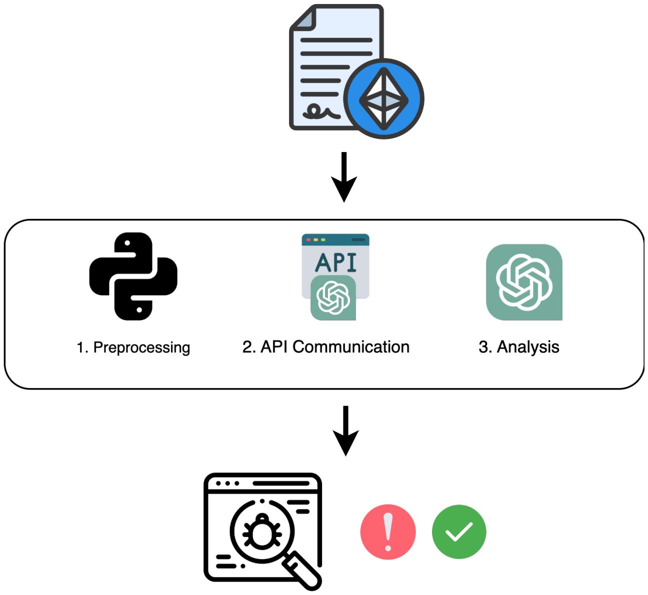

Figure 3: Architecture of proposed tool.

```
Vulnerabilities:
1. The randomness source used in the makeBet() function is not
   secure and can be easily predicted. This can lead to a
   situation where the attacker can always win the lottery by
   predicting the block number.
2. The destroy() function allows anyone with the organizer's
   address to destroy the contract, which can lead to loss of
   funds.

Remediation:
1. Use a more secure source of randomness like Chainlink VRF or
   Oraclize to generate random numbers.
2. Use access control mechanisms like OpenZeppelin's Ownable
   contract to ensure that only the contract owner can call the
   destroy() function.
```

Figure 4: Bad Randomness and Access Control detection.

1. **Preprocessing**. In this phase, the textual representation of smart contract code is taken as input from the GUI. Preprocessing is needed because it is possible that the user inserts additional information not relevant for the purpose of the vulnerabilities detector (e.g. *I have this smart contract deployed*) and must be removed. Moreover, in this phase, the text is formatted according to OpenAI API in order to avoid errors in communication.
2. **API Communication**. Once the text has been correctly preprocessed, the request is created and sent to the OpenAI servers by using the available API. In this phase, a token key is used for two reasons. In the first instance, the previous model setup phase is restored, without the need for recreating the environment each time; and secondly, for guaranteeing the privacy of users and preventing access from malicious nodes.
3. **Analysis**. When an OpenAI server receives the request from the Python script, it automatically forwards it to the gpt-3.5 model for analysis. At this time the vulnerability analysis happens on the smart contract, and once completed, the response is given back to the user throw the GUI.

The response generated from the model follows the definition declared in the model setup phase and will be composed of both detected vulnerabilities and suggested remediations.

4.3 Running Example

To better understand the use case of the proposed system, a running example has been executed. Two smart contracts have been considered for the example: in the first one access control and bad randomness vulnerabilities were introduced, while the second one does not contain vulnerabilities.

```
Vulnerabilities:
This contract seems to be well written and there are no apparent
    vulnerabilities.
Remediation:
However, there is one suggestion for improvement. In the `
    deleteOwner` function, it would be better to change `owners[
    _owner] = 0;` to `delete owners[_owner];` to reduce the
    contract's storage usage..
```

Figure 5: Smart Contract without vulnerabilities.

The response related to the first smart contract is depicted in Figure 4 and as reported it detects two vulnerabilities. The first vulnerability arises from the function *makeBet()* which refers to a problem related to a typical bad randomness vulnerability. More in particular, the function leverages the number of blocks to determine if the current caller is a winner or not. As remediation, the tool suggests using a verifiable source of randomnesses such as Chainlink VRF, or Oraclize. The second vulnerability, instead, arises from the function *destroy()*, which is a typical function inserted in smart contracts for self-destroy the smart contract from the blockchain and withdraw all the funds to the owner. An **access control** vulnerability can happen since the check is performed on the sender address but is done in a static way by only the address without ensuring which is the current owner of the smart contract. The remediation actions are in line with the typical countermeasure, and they also suggest some practical mechanisms for preventing attacks such as OpenZepplin's Ownable contract.

Figure 5 depicts the response to the second smart contract, which was supposed to not detect any vulnerability. As expected, it does not detect any vulnerability but gives the user a suggestion for the improvement related to the storage usage required from the structure owners used in the **deleteOwner()** function. This last example puts in light another relevant aspect of LLM, which is able to give suggestions derived from the existing knowledge of the model.

5 RESULTS

To begin with the results analysis it's necessary to define which are the parameters used for the selection of the tool. SmartBugs [3] is a framework for the analysis of smart contracts; it contains 19 tools for automation and two datasets. This represents a good starting point for the comparison but we will limit the selection to only 9 tools, which have been chosen on the basis of four criteria, which leverage the same parameters of our proposal:

- **Availability**: tools must be publicly available and must support a Command Line Interface (CLI);
- **Input**: tools must accept Solidity smart contract, tools which require EVM bytecode are excluded;
- **Vulnerabilities Identification**: tools described as analysis tools that are limited only to the artifact build such as graphs of flow control are excluded.

These tools are evaluated against the SmartBugs Curated dataset, which considers 69 smart contracts split into ten categories referring to the top 10 OWASP vulnerabilities. Table 2 shows the number of contracts in association with each category. Among them, access control and arithmetic vulnerabilities are the most relevant ones since can lead to unauthorized execution and error in smart contracts. Similarly to what is done in [1], the performance of the proposed approach will be measured by considering:

Table 2: Dataset composition

Category	Level	# Smart Contracts
Access Control	Solidity	17
Arithmetic	Solidity	14
Bad Randomness	Blockchain	8
Denial of Service	Solidity	6
Front Running	Blockchain	4
Reentrancy	Solidity	7
Short Address	EVM	1
Time Manipulation	Blockchain	4
Unchecked Low-Level Cell	Solidity	5
Unknown	N/A	3

- **Vulnerabilities detected**: the number of vulnerabilities identified using the tool.
- **Execution time**: the time needed for the analysis of a smart contract, till the response.

In order to perform the evaluation, an iMac with a 3.33 GHz Intel Core i5 6 core processor and 16 GB 2667 MHz DDR4 RAM has been used.

5.1 Vulnerabilities Detected

Analysis of the number of vulnerabilities found is achievable after the execution of all the instruments under consideration. Since it is possible for a smart contract to contain several vulnerabilities, the number of identified vulnerabilities is greater than the number of smart contracts that have been taken into consideration.

Table 2 reports the number of vulnerabilities detected in the smart contract. In the initial investigation, it can be seen that extremely specialized tools like HoneyBadger and Maian perform dreadfully in categories unrelated to their intended use, as was expected. Statistical analysis tools like Oyente, Osiris, Slither, and Smartchecker outperform specialist ones, indicating that a generalized approach can be used. In any case, none of the eight vulnerabilities related to bad randomness were discovered, proving the limitations of statistical analysis.

Symbolic analysis introduced by Manticore performs similarly to the other tools, with an increasing number of vulnerabilities detected for the group of access control. Finally, Mythril can be considered as the most complete tool, able to detect almost all the vulnerabilities, since it combines symbolic analysis with SMT and tain analysis.

The proposed approach provides a thorough analysis across all categories, making it the most comprehensive tool. Despite the fact that it cannot detect a large number of vulnerabilities, such as Mythril, it can detect at least one from each category, unlike all of the tools analyzed.

Table 3: Vulnerability detection taxonomy in considered tools

Category / Tool	HoB	Mai	MaC	Myt	Osi	Oye	Sfy	Sli	SmC	**VH-GPT**
Access Control	0	10	28	24	0	0	6	20	3	17
Arithmetic	0	0	11	92	62	69	0	0	23	40
Denial of Service	0	0	0	0	27	11	0	2	19	11
Bad Randomness	0	0	0	4	1	0	0	0	0	8
Front Running	0	0	4	21	0	0	55	0	0	6
Reentrancy	0	0	4	16	5	5	32	15	7	16
Time Manipulation	0	0	4	0	4	5	0	5	2	7
Unchecked Low Calls	0	0	4	30	0	0	21	13	14	7
Other	5	2	25	32	0	0	0	28	8	16
	5	**12**	**76**	**219**	**99**	**90**	**114**	**83**	**76**	**128**

Table 4: Execution Time

Tool	**Avg. Ex. Time**	**Total Ex. Time**
HoB	0:00:46	0:53:11
Mai	0:02:57	3:23:50
MaC	0:08:11	5:03:04
Myt	0:01:13	1:23:42
Osi	0:00:44	0:50:03
Oye	0:00:36	0:41:29
Sfy	0:01:00	1:09:08
Sli	0:00:03	0:03:35
SmC	0:00:06	0:06:34
VH-GPT	**0:00:23**	**0:16:28**

5.2 Execution Time

The execution time has been considered as the total and averaged execution time over all the smart contracts contained in the SmartBugs Curated dataset. Table 4 reports these two metrics for all the considered tools. HoneyBadger and more in particular Main have a relatively high execution time which can be caused by the approach used by these tools: the first one is interested only in finding honeypots using pattern-based detection; while the second one spends a lot of time for the creation of the entire trace of transactions.

Manticore is the worst tool in terms of execution time due to the symbolic execution Statistical-based tools like Slither and Smartcheck have reduced execution time tanks to the strategies put in place, which is related to the translation of code and analysis using static tools. Oyente, Osiris, and Honeybadger have good detection times but their detection rate is really reduced if compared with the others due to the limitations described in the previous subsection.

Differently from the other tools within this context, VulnHunt-GPT execution time depends on OpenAI servers, which are able to detect vulnerabilities in a really short time with respect to others.

6 CONCLUSION

The usage of LLMs is not limited to the content generation and to the Machine Learning and, in particular, LLMs are increasing in popularity within the context of cybersecurity. The possibility of analyzing and interpreting code snippets, together with knowledge representation capability, makes this model the most suitable for such tasks. Moreover, they are always more user-friendly, making them available to a huge audience. The efficiency of such models highly depends on the training phase, so the, more recent the models will be the more performant the predictions will be.

Conventional methods of evaluating the vulnerability of smart contracts suffer from a lack of completeness in detection, as every instrument may identify every threat listed in the OWASP Top 10. From this perspective, the suggested method performs better than the others since it finds at least one vulnerability across all ten categories. Despite this benefit, it's crucial to underline that, in order to get better outcomes, GPT-based models should be viewed as a component of a more comprehensive testing set. They can assist in identifying vulnerabilities, but they cannot offer a complete fix for the issue. Future developments in this research will focus on assessing the existing methodology against more recent LLM models, such as Bard AI, and the evaluation of false positives detected by the models.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*. 530–541.

[2] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[3] 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.

[4] NCC Group et al. 2018. Decentralized application security project (DASP) top 10. *Retrieved August* 5 (2018), 2021.

[5] Olivér Hornyák and George Farid Alkhoury. 2020. Smart contracts in the automotive industry. In *Vehicle and automotive engineering*. Springer, 148–157.

[6] C. Koch. 2023. Experimenting with GPT-3 for Detecting Security Vulnerabilities in Code. https://github.com/chris-koch-penn/gpt3_security_vulnerability_scanner

[7] Daniel Macrinici, Cristian Cartofeanu, and Shang Gao. 2018. Smart contract applications within blockchain technology: A systematic mapping study. *Telematics and Informatics* 35, 8 (2018), 2337–2354.

[8] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[9] Madhav Nair, Rajat Sadhukhan, and Debdeep Mukhopadhyay. 2023. Generating secure hardware using chatgpt resistant to cwes. *Cryptology ePrint Archive* (2023).

[10] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*. 653–663.

[11] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227* (2022).

[12] Fardin Ahsan Sakib, Saadat Hasan Khan, and AHM Karim. 2023. Extending the Frontier of ChatGPT: Code Generation and Debugging. *arXiv preprint arXiv:2307.08260* (2023).

[13] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. 2020. Smart contract: Attacks and protections. *IEEE Access* 8 (2020), 24416–24427.

[14] Fabian Schär. 2021. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review* (2021).

[15] Ashutosh Sharma, Sarishma, Ravi Tomar, Naveen Chilamkurti, and Byung-Gyu Kim. 2020. Blockchain based smart contracts for internet of medical things in e-healthcare. *Electronics* 9, 10 (2020), 1609.

[16] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. ACM, Gothenburg Sweden, 9–16. https://doi.org/10.1145/3194113.3194115

[17] 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.

[18] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

[19] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[20] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.