Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

SmartChat: Exploring LLMs for Initial Seed Generation in Smart Contract Fuzzing with Vulnerability-Guided Prioritization

Fausto Carvalho Marques Silva

Dissertation presented in partial fulfillment of the
requirements for the Master's Degree in Informatics

Supervisor
Prof. Dr. Rodrigo Bonifácio

Brasília
2025

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

CC331s

CARVALHO MARQUES SILVA, FAUSTO
SmartChat: Exploring LLMs for Initial Seed Generation in Smart Contract Fuzzing with Vulnerability-Guided Prioritization / FAUSTO CARVALHO MARQUES SILVA; orientador Rodrigo Bonifácio. Brasília, 2025.
82 p.

Dissertação(Mestrado em Informática) Universidade de Brasília, 2025.

1. Fuzz Testing. 2. LLMs. 3. Smart Contracts. 4. Initial Seed Generation. I. Bonifácio, Rodrigo , orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

SmartChat: Exploring LLMs for Initial Seed Generation in Smart Contract Fuzzing with Vulnerability-Guided Prioritization

Fausto Carvalho Marques Silva

Dissertation presented in partial fulfillment of the
requirements for the Master's Degree in Informatics

Prof. Dr. Rodrigo Bonifácio (Supervisor)
CIC/UnB

Prof. Dr. Rohit Gheyi
UFCG

Dr.a Roberta de Souza Coelho
DIMAp/UFRN

Prof. Dr.a Cláudia Nalon
Coordinator of the Graduate Program in Informatics

Brasília, September 12, 2025

Dedicatória

Dedico esta dissertação de mestrado, e todo o suor e desespero aqui contidos, aos meus pais. É impossível descrevê-los de forma menos impressionante do que realmente são. Meu pai, aqui em memória, e minha mãe, aquela mineira forte e sensata. Ambos carregam consigo o sangue, suor e lágrimas que são tão presentes em nossas vidas, mas somente por eles eu vi o trajeto que poderia um dia tentar caminhar. Maria Magali Carvalho Silva e Carlos Alberto Marques da Silva, isso é por vocês e para vocês.

Tenho que fazer um grande adendo a esta dedicatória, para abarcar meus amores de dentro de casa. Minha esposa Rafiza Luziani, que, com sua paciência e perspicácia, acaba levando qualquer um, principalmente jovens e crianças, a acreditarem que são capazes, muitas vezes mais do que realmente o são. Te amo, Rafiza.

Só mais um me resta aqui, meu filho Theo de Carvalho. Desculpa qualquer coisa, meu filho, mas eu tenho que dedicar tudo meu para você, não por força de obrigação, mas porque você sempre será minha razão de viver. Obrigado, isso também é por você.

> "Now I'm a scientific expert; that means I know nothing about absolutely everything."
> — Arthur C. Clarke, *2001: A Space Odyssey*

Agradecimentos

Foi uma longa jornada até este momento, lutando contra inimigos invisíveis dentro da minha mente, e, por vezes, eles venceram. Mas hoje chego ao ponto de superar, definitivamente, o medo, o desespero e a falta de confiança. Para alcançar esse momento, preciso agradecer a muitas pessoas, porque é com outras pessoas que seguimos em frente. Somente um ser humano pode realmente ajudar outro ser humano.

Primeiramente, agradeço a alguém que sempre acreditou em mim, e isso não é comum. Agradeço ao meu orientador, Rodrigo Bonifácio de Almeida, uma pessoa que possui uma fé nas capacidades dos outros que é incomum na universidade. Foi somente por causa dele que consegui chegar até aqui. Gostaria de agradecer também a um amigo que fiz durante o mestrado, uma pessoa humilde e muito simpática: Walter Lucas, grande contribuidor e confidente ao longo da jornada acadêmica. Outras grandes pessoas que encontrei neste mestrado foram Leandro Oliveira, que sempre foi muito sincero e prestativo nesses últimos anos. Estendo meus agradecimentos ao colega Daniel Almendra, por toda a sua paciência e serenidade durante nossas interações.

Ademais, minha família merece todo o mérito e um agradecimento profundo por entenderem e suportarem minha ausência durante o tempo que dediquei ao mestrado e não a vocês. Sempre foram muito orgulhosos dessa conquista, e sinto um enorme desejo de retribuir tudo o que vocês me deram. Magali, Rafiza e Theo, vocês são a razão do meu caminhar adiante, neste momento e para sempre. Por fim, agradeço pelos anos de apoio e incentivo do meu pai, Carlos, cuja memória, mesmo ausente, permanece presente em cada conquista minha.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES), por meio do Acesso ao Portal de Periódicos.

SmartChat: Explorando LLMs para a Geração de Sementes Iniciais no Fuzzing de Smart Contracts com Priorização Guiada por Vulnerabilidades

Resumo

O alto valor financeiro do ecossistema da *blockchain* Ethereum tornou os contratos inteligentes alvos principais para atacantes, atraindo crescente atenção de pesquisadores da área e de auditores de segurança. Em resposta, ferramentas automatizadas—baseadas principalmente em análise estática ou dinâmica—tornaram-se centrais nos esforços para identificação de vulnerabilidades. No entanto, os atacantes frequentemente descobrem e exploram falhas mais rapidamente do que os mecanismos de defesa conseguem reagir, deixando claro que ainda são necessárias estratégias de detecção mais eficazes. Embora *fuzzing* tenha se mostrado útil na descoberta de vulnerabilidades em contratos inteligentes, muitos *fuzzers* ainda enfrentam dificuldades, sobretudo por iniciarem suas campanhas com sementes de entrada de baixa qualidade.

Com o objetivo de mitigar essa limitação, investigamos o uso de *Large Language Models* (LLMs) pré-treinados para gerar sementes iniciais de alta qualidade para o *fuzzing* de contratos inteligentes. Essa abordagem é combinada com um algoritmo de priorização guiado por vulnerabilidades, com o objetivo de direcionar o *fuzzing* para sequências de transações com maior probabilidade de revelar falhas de segurança. Enfrentamos a dependência crítica de *fuzzing* em relação à qualidade das sementes iniciais avaliando sete LLMs pré-treinados, abrangendo modelos de código aberto e fechado, quanto à capacidade de gerar sequências de transações estruturalmente válidas, sem a necessidade de customização.

Avaliamos empiricamente nossa ferramenta, SmartChat, por meio de uma série de experimentos, analisando a qualidade da geração das sementes quanto à validade estrutural, correção semântica e sintática, métricas de cobertura e a eficácia do *fuzzing* sob diferentes valores de amostragem de temperatura.

Os resultados experimentais demonstram que o SmartChat supera técnicas avançadas de geração de sementes baseadas em fluxo de dados. Em particular, nossa abordagem revela até 15,6% mais vulnerabilidades do que os *fuzzers* estado da arte, alcançando um *speedup* de 6,67× a 44×, ao mesmo tempo em que melhora a cobertura de código em diversas classes de vulnerabilidades.

Palavras-chave: Fuzz Testing, LLM, Initial Seed Generation, Smart Contracts.

Abstract

The high financial value of the Ethereum blockchain ecosystem has made smart contracts prime targets for attackers, attracting growing attention from both blockchain researchers and security auditors. In response, automated tools—relying largely on static or dynamic analysis—have become central to efforts in identifying vulnerabilities. Nevertheless, attackers often find and exploit vulnerabilities faster than defenders can react, making it clear that better detection strategies are still needed. While fuzz testing has been useful for uncovering bugs in smart contracts, many fuzzers still struggle, primarily because they start with low-quality input seeds.

To address this limitation, we investigate the use of pre-trained Large Language Models (LLMs) to generate semantically-aware initial seeds for smart contract fuzzing. This approach is combined with a vulnerability-guided prioritization algorithm to steer fuzzing toward transaction sequences likely to reveal security flaws. We address fuzzing's critical dependency on initial seed quality by evaluating seven pre-trained LLMs—spanning both open- and closed-weight models—based on their ability to generate structurally valid transaction sequences without requiring model customization.

We empirically evaluate our tool, SmartChat, through a series of experiments, assessing seed generation quality, structural validity, semantic and syntactic correctness, coverage metrics, and fuzzing effectiveness across varying sampling temperatures. The experimental results demonstrate that SmartChat outperforms advanced data-flow-based seed generation techniques. Specifically, our approach uncovers up to 15.6% more vulnerabilities than state-of-the-art fuzzers, achieving a speedup of 6.67× to 44× while also increasing code coverage across a wide range of vulnerability classes.

Keywords: Fuzz Testing, LLM, Initial Seed Generation, Smart Contracts.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

ABI Application Binary Interface.

dApps Decentralized Applications.

DeFi Decentralized Finance.

DGF Directed Grey-box fuzzing.

EOA Externally Owned Account.

EVM Ethereum Virtual Machine.

GenAI Generative Artificial Intelligence.

LLM Large Language Model.

NLP Natural Language Processing.

P2P Peer-to-Peer.

PoS Proof-of-Stake.

PoW Proof-of-Work.

PUT Program Under Test.

RLHF Reinforcement Learning from Human Feedback.

Chapter 1

Introduction

Blockchain technology has been proposed for numerous applications beyond its most common use in cryptocurrencies. These new applications range from supply chain systems and national economy tokenization to public health digitalization initiatives. On the Ethereum platform, the most prominent smart contract-enabled blockchain, smart contracts act as the supporting mechanisms for these new use cases. A smart contract is an executable program similar to scripts that are typically small in size when compared with traditional software but can manipulate the underlying structure of a blockchain, allowing the creation of protocols and agreements without involving a trusted intermediary. Several concerns appear during the lifecycle of smart contracts, particularly those related to security and correctness. Due to the immutable nature of transactions on a blockchain, the risk of deploying contracts that may jeopardize the platform's integrity is highly significant.

Ethereum smart contracts, as inherently stateful systems, present a complex and significant testing challenge. They continuously interact with other contracts within the underlying blockchain environment, maintaining state through a distributed ledger database. If a vulnerability can only be detected in a specific state, a test case[1] must include a specific sequence of transactions and inputs to guide the contract to that state before the potential risk can be exposed. This underscores the importance of vulnerability detection strategies, with generally effective native software methods—such as fuzzing—being an essential method for identifying vulnerabilities, exceptions, and bugs in smart contract-based systems.

Despite the widespread use of fuzz testing for smart contract security, it faces several key challenges [5]. First, smart contracts often require highly specific transaction sequences to reach vulnerable states, thus making random input generation inefficient.

[1] In the context of this dissertation, the terms *"transaction sequence," "test case,"* and *"seed"* are used interchangeably.

Second, traditional seed generation techniques, whether based on randomness, heuristics, or trained models, may produce low-quality initial seeds that fail to explore meaningful execution paths, where vulnerabilities are likely to reside. Finally, crafting effective initial seeds often requires domain expertise, large datasets, or complex program analysis pipelines, making the process complex and difficult to automate and generalize.

Current Generative Artificial Intelligence (GenAI) methods, through Large Language Model (LLM), are at the forefront of software engineering practices. They are being widely tested for tasks such as program repair [6, 7], code generation and test case generation [8, 9, 10, 11], and fuzzing [12, 13, 14]. While current research has focused on applying pre-trained LLMs to smart contract fuzzing for tasks such as seed scheduling and selection, a gap remains in understanding their broader potential. In particular, it is unclear whether LLMs can act similarly to domain experts—capable of understanding and reasoning about smart contract semantics—to assist in fuzz testing by generating structurally valid test cases as an initial seed corpus, without requiring task-specific domain adaptation (i.e., fine-tuning).

Motivated by this background, in this work, we propose SmartChat, an extension of the Smartian fuzzer [4] for Ethereum smart contracts. SmartChat leverages pre-trained LLMs and a vulnerability-guided fuzzing approach to address the challenges partially highlighted in [5]. The key insight is that LLMs, when guided by carefully designed prompting strategies, can synthesize transaction sequences that are not only syntactically correct and semantically meaningful but also structurally valid concerning the expected format. To further enhance fuzzing effectiveness, we introduce a vulnerability-guided prioritization mechanism that selects seeds based on their potential to uncover vulnerabilities.

1.1 Motivating Example

To illustrate the importance of constructing good initial seeds [15] that have the potential to uncover Ethereum vulnerabilities, consider the motivating example of a Solidity smart contract shown in Listing 1.1. We use a simplified version of an Ethereum ERC-20 fungible token implementation responsible for managing the distribution of tokens to users in a decentralized application. The contract defines a fixed total token supply upon deployment and implements the standard ERC-20 `transfer` function, enabling token transfers from the caller (`msg.sender`) to a specified recipient address (e.g., users or other contracts) (`_to`). Additionally, it features a `distribute` function, which allows batch distribution of a predefined token amount to multiple users.

```
1  pragma solidity ^0.4.13;
2
3  contract SampleToken is ERC20 {
4    uint256 _totalSupply = 21000000 * 10**8;
5    address public owner;
6    mapping(address => uint256) balances;
7
8    function SampleToken() {
9      owner = msg.sender;
10     balances[owner] = _totalSupply;
11   }
12
13   function distribute(address[] addresses) {
14     for (uint i = 0; i < addresses.length; i++) {
15       balances[owner] -= 2000 * 10**8;
16       balances[addresses[i]] += 2000 * 10**8;
17     }
18   }
19
20   function transfer(address to, uint256 amnt) returns (bool) {
21     if (balances[msg.sender] >= amnt && amnt > 0
22     && balances[to] + amnt > balances[to]) {
23       balances[msg.sender] -= amnt;
24       balances[to] += amnt;
25       return true;
26     } else {
27       return false;
28     }
29   }
30 }
```

Listing 1.1: Simplified token contract demonstrating an integer overflow vulnerability.

The contract contains a potential security risk: the `distribute` function is vulnerable to an integer underflow. In particular, the assignment on line 15 in the `distribute` function subtracts 2000×10^8 tokens from the owner's balance for each address in the input array, without verifying whether the owner's balance is sufficient to cover the total amount. If the loop executes more times than the owner's balance can support, an integer underflow may occur, causing the owner's balance to wrap around to a very large value. As a result, an attacker could exploit this vulnerability to transfer an effectively unlimited number of tokens.

One possible exploitation path involves draining the owner's balance through repeated calls to the `distribute` function. Given the total initial supply and the amount deducted per iteration, an attacker would need to invoke `distribute` approximately 10,500 times to trigger an integer underflow. A transaction sequence capable of triggering this vulnerability begins with a call to `transfer`, using a malicious contract address (`mal_addr`) as the first argument and an amount of $21,000,000 \times 10^8$ as the second. This is followed by a call to `distribute`, passing `mal_addr` as the sole element in the address array.

This sequence first drains the owner's token balance and then initiates the distribution logic, which attempts to subtract tokens from the already exhausted balance, ultimately resulting in an integer underflow.

Although the example may seem simple to domain experts, state-of-the-art fuzzers cannot reach this transaction sequence. When fuzzing this contract using a modern fuzzing tool, we were unable to discover the vulnerability even after a 60-minute campaign. Analyzing the test cases generated by this fuzzer, we observed that it fails to infer the total supply value ($21{,}000{,}000 \times 10^8$) and is unable to generate enough transactions to reach the vulnerable state. Moreover, it does not consider the possibility of using the `transfer` function to send a specific total amount, which is essential for generating the alternative sequence we presented. Consequently, it cannot explore the execution path that leads to the integer underflow in the `distribute` function.

1.2 Research Problem

Current state-of-the-art fuzzers for the Ethereum smart contract platform typically generate initial seed corpora by randomizing transaction sequences and using predefined upper or lower bounds for transaction argument values [5]. More advanced approaches, which integrate, for instance, static program analysis at the bytecode level, are useful but may neglect the semantic patterns of smart contract vulnerabilities. Additionally, some works [16, 17, 18] employ machine learning techniques, particularly pattern classification, to filter benign functions from suspicious ones or to simulate the behavior of a symbolic executor, which enhances the fuzzing process but may require retraining to operate on newer vulnerabilities. Therefore, building a high-quality initial seed corpus to achieve the fastest coverage increase in the initial stages and efficiently triggering specific vulnerabilities within a reduced timeframe is a topic that has not yet been thoroughly examined.

1.3 Research Goals

This section outlines the objectives explored in this study. The primary goal is to investigate how pre-trained LLMs can enhance Ethereum smart contract fuzzing by generating high-quality initial seeds that are both semantically aware and structurally valid, combined with vulnerability-guided prioritization mechanisms to accelerate vulnerability discovery.

To evaluate the effectiveness of our approach, we developed automated evaluation frameworks and introduced a new fuzzing tool, SmartChat. Building on this main objective, the study also pursues several related goals, which are detailed below:

- Develop a comprehensive LLM-driven seed generation framework to conduct a systematic empirical evaluation of pre-trained LLMs based on seed quality metrics, including structural validity, syntactic correctness, semantic meaningfulness, coverage, and uniqueness.
- Design and implement a universal seed format specification to enable the integration between LLM outputs and smart contract fuzzing tools.
- Implement and evaluate a vulnerability-guided seed prioritization algorithm.
- Conduct a comprehensive comparative study between LLM-generated seeds against traditional methods using established benchmarks.

1.4 Dissertation organization

The remaining chapters are organized as follows: Chapter 2 provides the necessary background and reviews related work to establish the foundation for this dissertation. Chapter 3 introduces the proposed tool, SmartChat, detailing its design, key decisions, and implementation. Next, Chapter 4 presents our empirical assessments, covering the experimental setup, research questions, and results. Finally, Chapter 5 outlines our main contributions, final considerations, and directions for future work.

Chapter 2

Background and Related Work

In this chapter, we will cover the necessary concepts to understand this work. First, we will discuss the concept and technical aspects of blockchain, as well as the main elements of the Ethereum platform. Next, we discuss the fundamentals of fuzz testing, including different types of fuzzers, common fuzzing strategies, and typical vulnerabilities found in Ethereum smart contracts. We then review state-of-the-art fuzzing tools developed specifically for this domain. Finally, we introduce relevant concepts in generative AI, with an emphasis on pre-trained LLMs and their recent applications to smart contract fuzzing.

2.1 Blockchain & Ethereum

The inception of blockchain is attributed to an enigmatic entity known as Satoshi Nakamoto, who, in 2008, posted a whitepaper [19] to a cryptography mailing list, introducing a fully decentralized and distributed electronic cash system, where the service remains available and functional even if a server on the network crashes or becomes unavailable. This marked the birth of Bitcoin and, consequently, blockchain as a system independent of trusted third parties such as governments or financial institutions.

As a decentralized digital currency, Bitcoin was able to address the double-spending issue (i.e., the number of coins an owner can transfer to others corresponds to the coins they previously possessed), enabling trustless transactions without a central authority or constant online presence. Emerging from the Bitcoin breakthrough, blockchain is regarded as one of the most disruptive technologies since the Internet. Companies worldwide have heavily invested in blockchain research to enhance their operations and security, as it has evolved into a principal cryptocurrency with a current market capitalization nearing 1.4 trillion USD.

Conceptually, a blockchain can be seen as a shared, publicly accessible database with an appendable, verifiable list of blocks, each linked to the previous one by a cryptographic hash. Therefore, modifying a block's data would invalidate its hash, and all subsequent blocks referencing that hash would become inconsistent. Computing new hashes would require immense computing power and would be noticed by the decentralized network that validates transactions on a blockchain. Maintained by a distributed Peer-to-Peer (P2P) network of untrusted nodes, blockchain operates on a consensus mechanism (a kind of agreement on data or state between distributed nodes) for adding new blocks. Before being added to the blockchain, blocks must be validated and signed by their creators; that way, transactions are broadcasted to the entire network for legitimacy verification. The predominant consensus mechanisms used in blockchain technology today are Proof-of-Work (PoW) and Proof-of-Stake (PoS). In PoW, miners race to solve complex math problems using cryptographic hashing functions. In contrast, PoS replaces miners with validators, who are chosen to create new blocks based on their token stake. Both systems use cryptography to secure transactions, enabling direct peer-to-peer exchanges without relying on a central authority.

2.1.1 Transactions, blocks, and addresses

At the core of blockchain technology are three concepts: transactions, blocks, and addresses. A transaction refers to an interaction between two entities. In the case of cryptocurrencies, it involves the transfer of coins or tokens from one user to another. In other usage scenarios, a transaction can refer to the transfer of ownership or the exchange of digital assets, such as non-fungible tokens (NFT), and associated activities. Transactions are the fundamental components of a blockchain system, with each block containing one or more transactions.

Regarding the blocks, Figure 2.1 illustrates the structure of blocks in a blockchain. Blocks in a blockchain are containers of transactions and are connected through cryptographic hashes, with each block containing a hash of its data and the previous block's hash forming a chain. The data in a block are divided into two main sections: the header, which contains all the metadata (hash, nonce, timestamp, etc), and the transaction data body. The first block, known as the Genesis Block, is unique because it has no previous hash value. Every block in the blockchain is visible to all nodes in the network, providing transparency and consistency [1].

Blockchain addresses, or accounts, are unique identifiers used to send and receive cryptocurrency. They're typically alphanumeric strings derived from public keys. While addresses serve as endpoints for blockchain transactions, the actual ownership and control

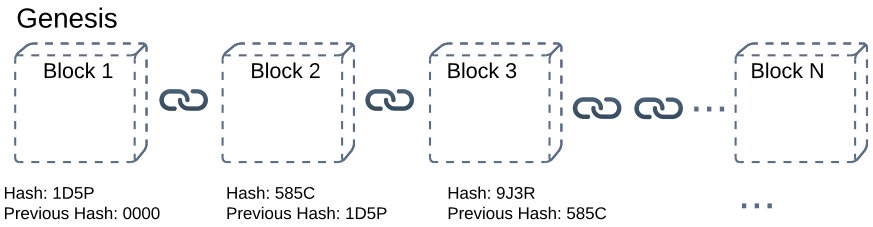

Figure 2.1: The structure of blocks in a Blockchain [1].

of assets are determined by the corresponding private key. Every transaction involving addresses is permanently recorded on the blockchain.

2.1.2 Ethereum

Ethereum is a second-generation public blockchain platform introduced in 2014 by Vitalik Buterin and Gavin Wood [20]. Building on the success of Bitcoin, Ethereum features its own cryptocurrency, ether (ETH), and it also allows developers to create new currencies in the form of tokens. The Ethereum Foundation has established the ERC-20 and ERC-721 standards for fungible and non-fungible tokens, respectively. These standards support a range of applications, including digital collectibles, stablecoins, and payment systems [21].

Ethereum provides a programmable interface called a smart contract, which is a Turing-complete, self-executing program that automatically runs when predefined conditions are met. For example, it can trigger a payment upon confirmation of goods received, thereby enforcing the terms of an agreement between parties [22]. Such characteristics provide the means for a broader monetary ecosystem, enabling decentralized finance (Decentralized Finance (DeFi)), decentralized applications (Decentralized Applications (dApps)), and various use cases beyond digital currency transactions.

Unlike Bitcoin, the Ethereum platform supports external accounts (i.e., user accounts) and contract accounts. Each Ethereum account is identified by its address, has an associated balance of ether (the native cryptocurrency of Ethereum), and is owned by a user. Contract accounts are linked to a program interface and a program state, known as storage. An Externally Owned Account (EOA) sends a transaction to the Ethereum network to interact with another contract or to send ether to other users [23].

As of August 2024, over 67 million smart contracts have been deployed on Ethereum, the largest blockchain-supporting smart contracts [24]. Additionally, there are Ethereum-compatible chains, allowing the programmer to write and deploy the same smart contracts without significant code changes. The total number of smart contracts deployed across

these chains has reached 1 billion [25]. Some of these chains include: Arbitrum, Avalanche, BNB Chain, Fantom, Gnosis Chain, Optimism, Polygon.

2.1.3 Smart Contracts & Solidity

The key elements of smart contracts were introduced by Nick Szabo in 1994 [26], describing them as agreements (legal contracts) between users and enforced by algorithms within a trustless system comprising self-executing computer programs. Nonetheless, the concept only became a reality with the launch of Ethereum.

Solidity is primarily considered to be the most widely used language for writing smart contracts [27], providing a high-level, statically typed language with a syntax similar to JavaScript and an object-oriented style, specifically designed to be compiled to Ethereum Virtual Machine (EVM) bytecode [28]. Solidity supports library reuse and creation and is an open-source project with numerous contributors and regular updates (as of August 2024, the current version is 0.8.26), serving as the primary language for Ethereum and other blockchain platforms. Developing smart contracts differs from programming in traditional languages like C++, Java, or Python. Due to the immutable and irreversible nature of the blockchain, once the bytecode of a smart contract is deployed on the Ethereum network, it cannot be modified.

In Solidity, the concept of null, as found in many other programming languages, does not exist. Instead, Solidity assigns default values to each data type when a variable is declared but not explicitly initialized. For example, integer types default to zero, and Booleans default to false [28]. Solidity supports integer types ranging from int8 to int256, with their unsigned counterparts being uint8 to uint256. The default integer type, if only int or uint is specified, is int256 or uint256, respectively. Internally, the EVM operates with fixed-size words of 256 bits. Therefore, even though Solidity allows for various integer sizes, values are often padded to fit into 256-bit words for efficient computation and storage.

```
pragma solidity ^0.8.26;

contract SimpleWallet {
  mapping(address => uint256) public balances;
  address public owner;

  constructor() { owner = msg.sender; }

  receive() external payable {}

  function deposit() public payable {
    require(msg.value > 0, "Deposit must be greater than zero");
    balances[msg.sender] += msg.value;
  }

  function withdraw(uint256 _amount) public {
    require(_amount <= balances[msg.sender], "Insufficient balance");
    balances[msg.sender] -= _amount;
    payable(msg.sender).transfer(_amount);
  }

  function sendTo(address payable _to, uint256 _amount) public {
    require(_amount <= address(this).balance, "Insufficient balance");
    require(_to.send(_amount), "Failed to send");
  }
}
```

Listing 2.1: A simple Solidity contract example.

Listing 2.1 shows a simple contract that illustrates the main Solidity language features. The contact uses a mapping to represent the variable balances; in Solidity, a mapping is a data structure that stores data in pairs, with each key associated with a corresponding value analogous to a dictionary or hash table in other languages. The address type is a 20-byte value that stores the address of an Ethereum account or a contract. It is fundamental to Ethereum and is commonly used as a key in mappings. It is used to uniquely identify and interact with accounts or smart contracts on the blockchain.

Functions in a Solidity contract are essential for defining how users and other contracts interact with the external world. In Listing 2.1, a simple wallet contract is implemented, allowing users to deposit Ether, withdraw funds, and transfer to other users. Similar to other object-oriented languages, Solidity supports function modifiers such as internal, external, pure, view, and payable. Functions marked as public or external can be called by external entities through the contract's ABI. Payable functions can also receive ethers through transaction invocations.

Solidity has evolved through 88 versions, from v0.1.2 to v0.8.26, since its inception in 2014. The rapid pace of updates may contribute to the challenges that smart contract developers face when trying to create secure programs [22].

2.1.4 Ethereum Virtual Machine

The Ethereum Virtual Machine is a stack-based, register-less virtual machine [2] with no I/O operations, featuring compact and highly optimized bytecode. Its instructions are restricted to manipulating the blockchain's state. Smart contracts are compiled into the EVM bytecode to be executed in Ethereum client nodes, targeting small size, simplicity, and deterministic execution. Ethereum smart contracts have only one entry point: the contract constructor.

The EVM operates over a world-state persistent environment like a transaction-based state machine. Each transaction is charged a gas fee, representing the computational effort required to execute it [20]. Ethereum uses the gas mechanism to ensure the termination of contracts and prevent denial-of-service attacks. The persistent storage associated with an account is organized as a key-value data structure to hold state variables persistently.

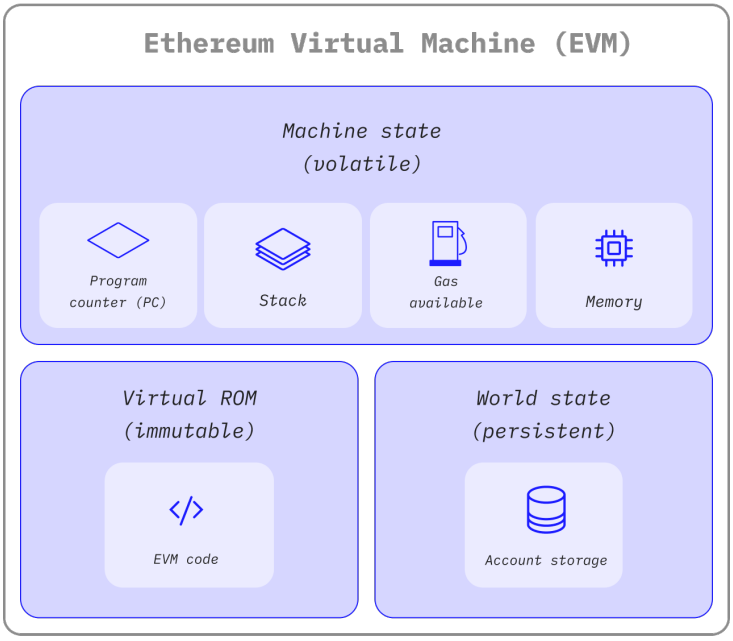

Figure 2.2: The Ethereum Virtual Machine (EVM) components [2].

Figure 2.2 describes the Ethereum Virtual Machine (EVM) internal key components, grouped into Machine State, Virtual ROM, and World State, where all elements are essential to the EVM's operation. All elements are clarified below:

Machine State (volatile)

- **Program Counter (PC)**: Is responsible for holding the next instruction that needs to be executed.

- **Stack**: The structure where values are pushed and popped to perform all arithmetic and control-transfer operations.
- **Gas Available**: This represents the amount of gas available for the transaction, considering that the sender specifies the gas limit.
- **Memory**: A temporary, byte-addressable storage space used during contract execution.

Virtual ROM (immutable)

- **EVM Code**: The immutable smart contract's bytecode.

World State (persistent)

- **Account Storage**: Is a key-value store with 256-bit keys and values, contents are kept on the blockchain, and hence, persists across multiple smart contract executions.

In an Ethereum Virtual Machine world state transition, the state changes from one configuration to another through the execution of a transaction. The world state is a comprehensive snapshot of all account balances, contract code, and storage on the blockchain at a specific point in time. The transition can be represented as follows:

$$\sigma \xrightarrow{T} \sigma'$$

Where:

- σ is the initial world state.
- T is the transaction.
- σ' is the new world state after the transaction has been applied.

The world state transition function γ can be defined as:

$$\sigma' = \gamma(\sigma, T)$$

2.2 Fuzzing and Fuzzing Smart Contracts

Detecting software vulnerabilities is a critical issue in today's technology-driven world, where software plays a key role in shaping economies. Various techniques are employed

to uncover vulnerable sections of code, ranging from static to dynamic analysis. Static vulnerability detection poses a challenge due to its high false positive rate compared to other methods like dynamic testing [5, 29], especially given the increasing complexity of software. In this context, fuzzing has emerged as a major method to improve vulnerability discovery effectiveness.

Fuzzing, or fuzz testing, involves exercising a Program Under Test (PUT) with invalid, unexpected, and random data to see if it crashes [30]. It is also used to discover software security vulnerabilities, logic flaws, and bugs. Fuzzing lies at the intersection of software testing and software security, with its primary purpose being the identification of security-related bugs.

The success of projects like OSS-Fuzz [31], a Google fuzzing system that combines various fuzzing techniques and tools for use in open-source software, has demonstrated the usefulness of fuzzing to the security community. As of August 2023, OSS-Fuzz has helped identify and fix over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects. In the following sections, we describe the types of fuzzers and fuzzing techniques, as well as their components.

2.2.1 Types of fuzzers

Fuzzers are commonly categorized into black-box, white-box, and grey-box [30]. Black-box fuzzing leverages no knowledge of the Program Under Test (PUT) while randomly trying to expose abnormal behavior. The inception of fuzz testing has primarily been related to black-box fuzzers; in 1988, Miller [32] used a random testing tool to investigate the reliability of userland UNIX tools. Black-box fuzzers are now considered less efficient because they struggle with complex conditions and cannot generate test cases that cover many paths in a target program [33, 34].

White-box fuzzers generate inputs by analyzing the internal structure of the PUT, specifically its source code. They rely on dynamic symbolic execution to explore the state space of the PUT, employing SMT (Satisfiability Modulo Theories) solvers [30]. This approach enables white-box fuzzers to systematically uncover vulnerabilities by considering the program's logical constraints. However, white-box fuzzers are only viable in situations where the source code is readily available, limiting their applicability in many scenarios, like binary fuzzing targets.

Grey-box fuzzers operate in between black-box and white-box fuzzers. It is considered more efficient than white-box fuzzing because it operates with minimal information about the target program and implements either instrumentation or lightweight program analysis. Modern fuzzers are often implemented as Directed Grey-box fuzzing (DGF), with the objective of efficiently reaching a given set of target program locations [33], having

various applications, such as testing recently modified code, reproducing crash reports as well as specific types of bugs, i.e., use after free, memory violations [35]

2.2.2 Fuzzing strategies

Fuzzing strategies are the methodologies or approaches used in fuzz testing, primarily in the domain of seed generation but also in areas like seed selection, mutation, scheduling, and feedback mechanisms. These strategies must consider the nature of the PUT while still accommodating the goals of the fuzzing campaign. The following sections describe three key strategies commonly employed in modern fuzz testing.

2.2.2.1 Grammar-based

Grammar-based fuzzing, also known as generation-based fuzzing, is a technique focused on synthesizing valid seeds by applying a set of rules in a grammar or model from the input domain of the target program, thereby achieving high syntactic correctness in test cases. Software that processes complex inputs, such as compilers and protocols, is often driven by models that describe the expected inputs or state transitions. In such cases, randomly generating input seeds is often ineffective, as it fails to explore code that resides deeper within the target program. The state-of-the-art in grammar-based fuzzing is progressing towards greater effectiveness through adopting hybrid fuzzing techniques [36], which combine the strengths of multiple fuzzing strategies to achieve a more complete coverage. Other advancements include the ability to target multiple programming languages with a single tool, as demonstrated by Polyglot [37], which employs a sophisticated method to bridge syntax and semantic differences across nine languages, leading to the discovery of over 173 new bugs in compilers.

2.2.2.2 Mutation-based

In mutation-based fuzzing, no model is used to describe the input format of the PUT, meaning seeds are generated without understanding the format or structure of the data. However, mutations are primarily performed by altering only a portion of valid input, referred to as the initial seed, which is why this approach is also known as seed-based input generation [30]. A fuzzer targeting a PNG library that uses randomly generated inputs might never surpass the initial image format validation, thereby failing to explore deeper paths where bugs may be located. By applying bit flips, byte operations, random deletions, and other mutation operators to a valid input seed, mutation-based fuzzers can generate unexpected or malformed inputs, increasing the chances of triggering bugs or vulnerabilities. Tools like zzuf [38] and Radamsa [39] are examples of pure mutation-based

fuzzers, applying various mutation operators to test cases using a predefined mutation ratio. Other mutation-based fuzzers, often referred to as hybrid fuzzers, use feedback mechanisms (mainly coverage) to guide mutations. Notable examples include AFL [40], AFLGo [33], AFL++ [41], Honggfuzz [42], and libAFL [43].

2.2.2.3 Coverage-guided

One of the challenges in fuzzing is generating meaningful inputs that explore all the different execution paths of a program, as the test case generation may get stuck at the local level of minimal code coverage, which means some paths will not be explored. Coverage-guided fuzzing aims to maximize the exploration of a program's code by using runtime feedback (code coverage, but also distance) to refine the fuzzing process.

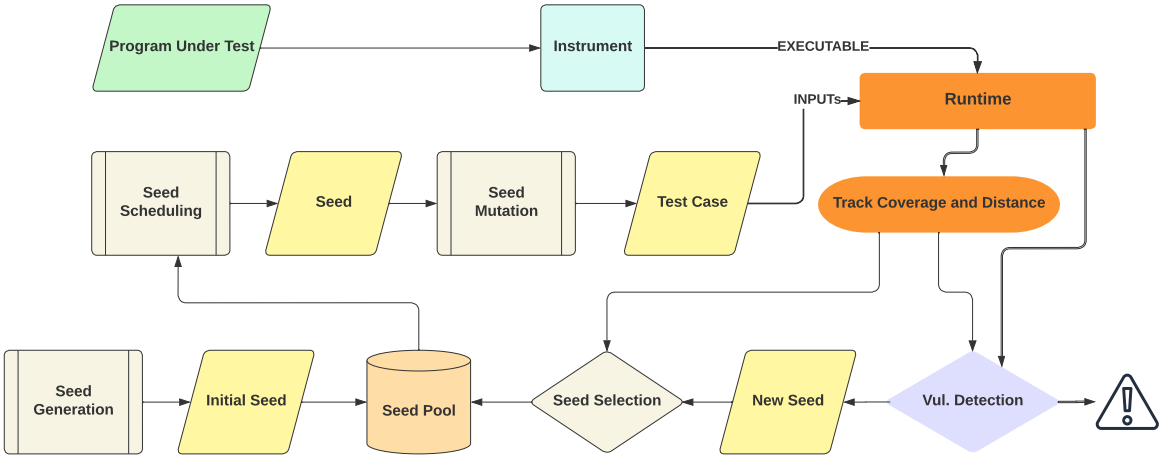

Figure 2.3: Overview of a coverage-guided grey-box fuzzer.

The diagram in Figure 2.3 provides an overview [1] of a coverage-guided grey-box fuzzer. The PUT, representing the software or application being tested, undergoes an instrumentation phase to collect specific runtime data, such as path coverage and distance metrics. Once instrumented, the program is executed in the runtime environment, where the PUT operates. Meanwhile, other components iteratively process input seeds and internal test cases, continuously refining them to maximize the likelihood of reaching a vulnerable code location. These components are responsible for:

- **Seed Scheduling:** Involves identifying which input in a seed pool is more likely to explore new code paths (or other metrics) when mutated. By implementing a

[1] Elements in Figure 2.3 may vary across fuzzers, with some including additional components and others having fewer due to different architectural choices and requirements.

scheduling algorithm, a fuzzer can prioritize seeds for execution, balancing exploration and exploitation.

- **Seed Selection:** Addresses the challenge of choosing seeds that, after execution, may be placed in the seed pool for later scheduling and mutation.
- **Seed Mutation:** Responsible for applying mutation operations to the input seeds to generate new test cases.
- **Seed Generation:** Involves creating the initial set of inputs that will be used to start the fuzzing campaign. The quality and diversity of these initial seeds influence the entire fuzzing process.

Fuzzers can operate as hybrid systems, combining different strategies and techniques. One prominent hybrid approach is Directed Grey-box fuzzing (DGF) as seen in Section 2.2.1. As a coverage-guided fuzzer, DGF specifically targets critical program areas, such as memory operations and API calls, which other fuzzers have difficulties reaching. DGF is a vital component of modern vulnerability discovery [44]; tools in this category include AFLGo [33], Driller [45], Beacon [46], and ParmeSan [47].

In the following sections, we will explain the typical set of vulnerabilities found in smart contracts and then dive deep into the current state-of-the-art of fuzzing tools for discovering smart contract vulnerabilities.

2.2.3 Smart Contracts Vulnerabilities

Smart contract vulnerabilities have been a subject of exploration since the inception of the Ethereum blockchain. The first majorly exploited and publicized incident was the TheDAO hack in 2016 [48]. Since this, over $8.58 billion has been hacked (Total Value Hacked, TVL) from various smart contract compatible chains [49]. Other notable attacks include the Parity Multisig Wallet attack [50], the Uniswap V2 liquidity pool hack [51], and the Rari Fuse Pool attack [52]

Atzei et al. [53] were the first to categorize smart contract vulnerabilities, providing a comprehensive catalog of common patterns associated with these security flaws. While some of these attacks are outdated and not widely explored in recent times, others continue to be prevalent in the current hacking landscape, such as reentrancy. Since Atzei et al. [53] taxonomy, several other vulnerability catalogs have emerged as important references for security threat mitigation tools within the community. Two notable examples are DASP10 [54] and SWC-Registry [55].

DASP10 (Decentralized Application Security Project) is similar to the OWASP taxonomy of the top 10 vulnerabilities in web applications. It was created by the NCC

Group and enumerates ten common bugs found in Ethereum Smart contracts. Although somewhat outdated, DASP10 still highlights some critical points relevant to current smart contracts, as some older contracts are still running on numerous mainnets across all chains, indicating that they are still actively deployed and operational. In contrast, SWC-Registry is a community-driven catalog similar to the Common Weakness Enumeration (CWE) project. It lists 37 types of vulnerabilities and is considered the standard taxonomy for EVM-like Smart Contracts.

Despite being a long list of vulnerabilities emerging from blockchain technology, we will only focus on those that are classified into the smart Contract layer of the Zhou et al. [56] classification. Other groups of threats in Zhou et al. [56] are Network, Consensus, DeFi Protocol, and Auxiliary Service layers. Modern dApps are intricate combinations of multiple contracts using auxiliary services and complex business logic, and vulnerabilities for this subset have yet to be tackled by vulnerability detection tools. Figure 2.4 presents threats classification for various blockchain layers.

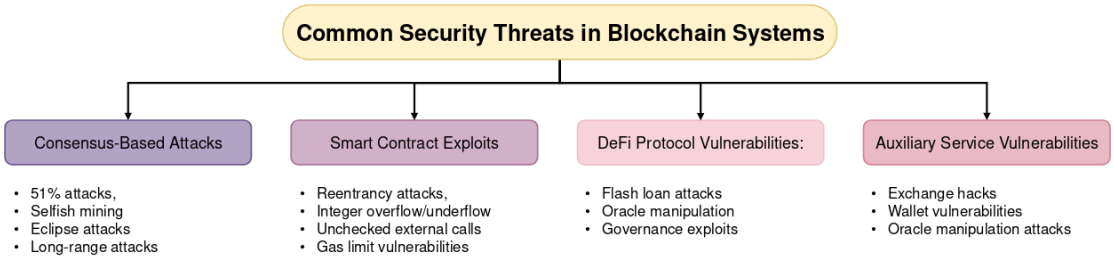

Figure 2.4: Security threats in blockchain systems [3].

Below, we provide a brief explanation of each relevant vulnerability detected by SmartChat, along with a mapping from our nomenclature to the corresponding SWC-IDs.[2]

Reentrancy - SWC-107: In programming language theory, a function is considered safely reentrant if it can be interrupted during its execution and then safely called again ('re-entered') before the original execution is complete, without causing unintended side effects. A smart contract is considered vulnerable to a reentrancy flaw if it lacks proper safety controls for when it is called again while still executing, potentially creating a race condition on state variables that could allow a malicious contract to steal funds [53].

Listing 2.2 demonstrates a simple contract that is vulnerable to reentrancy. In this contract, the `withdraw()` method sends ether to the caller before updating its internal state (the `balances` field). Since the `call` function can invoke another smart contract

[2]Since SmartChat is based on Smartian, we adopt its nomenclature and map these terms to the SWC-ID taxonomy.

and the EVM processes calls sequentially, a malicious contract can repeatedly invoke the `withdraw()` method within a single transaction. This allows the attacker to drain all the ether from the vulnerable contract before the assignment `balances[msg.sender] = 0` is executed. Other assets are also prone to be targeted by the reentrancy bug, such as ERC-20 or ERC-721 tokens, and even more complex ones like liquidity tokens, derivatives, or payment wallets.

To prevent a contract from having a reentrancy flaw, developers should consider using a reentrancy lock (e.g., a mutex or similar mechanism) or updating the contract's internal state before making any external calls. Even though static analysis tools can detect many reentrancy patterns, certain forms can only be uncovered through dynamic analysis, as they depend on runtime execution paths and inter-contract interactions.

```
contract Reentrancy {
  mapping (address => uint) private balances;
  function withdraw() public {
    uint amount = balances[msg.sender];
    (bool success, _) =
    msg.sender.call.value(amount)("");
    require(success);
    balances[msg.sender] = 0;
  }
}
```

Listing 2.2: A contract with the Reentrancy vulnerability.

Mishandled Exception - SWC-104: This vulnerability originates from the typical bad programming practice of failing to check the result of a function call or handle exceptions correctly. It is particularly risky when interacting with external contracts or functions, as it could lead to unexpected behavior or security issues. In Solidity, low-level calls (such as `call`, `delegatecall`, or `send`) do not halt execution or reverse the entire transaction in case of failure. This can allow the contract to continue running, potentially enabling an attacker to drain assets. Other names commonly used in literature for this vulnerability include exception disorder [57] or unchecked call return value [58]. In SmartChat, this vulnerability encompasses multiple types, including gasless send (out-of-gas Ether transfer).

Listing 2.3 describes a contract vulnerable to mishandled exceptions. The `execute()` function makes an external call without checking the result or handling any potential exceptions. To mitigate this issue, developers must either use Solidity's modern `require` statement to enforce conditions or manually control the result of the external call.

```
contract MishandledException {
  ExternalContract public externalContract;
  uint256 public result;

  constructor(address _externalContract) {
    externalContract = ExternalContract(_externalContract);
  }

  function execute() public {
    // External call without handling the result
    externalContract.performAction();
    // Additional logic that assumes the external call succeeded
    result = 1;
  }
}
```

Listing 2.3: A contract with the Mishandled Exception vulnerability.

Integer overflows/underflows - SWC-101: A contract contains an integer vulnerability if it fails to validate for underflow and overflow in arithmetic operations. This weakness arises when a value goes beyond the maximum limit of its type or falls below the minimum limit, causing it to wrap around to a different value and resulting in incorrect balance calculations or token amounts.

In Listing 2.4, there is an example of a contract that lacks arithmetic validation. The `deposit()` function takes an amount as input and adds it to the internal state variable `balance` without checking for overflow. Similarly, the `withdraw()` function does not verify whether subtracting the withdrawal amount will cause an underflow.

One way to mitigate integer bugs is to implement checks for overflow and underflow or use the well-known SafeMath library from OpenZeppelin [59], which provides built-in functions to handle arithmetic operations safely.

```
contract IntegerArithmetic {
  uint256 public balance;

  function deposit(uint256 _amount) public {
    balance += _amount;
  }

  function withdraw(uint256 _amount) public {
    balance -= _amount;
  }
}
```

Listing 2.4: A contract with the Integer overflow/underflow vulnerability.

Block State Dependency - SWC-120: Solidity does not provide built-in API functions for generating or obtaining random numbers. Nevertheless, some contracts

require randomness for their functionality. In such cases, contracts might depend on bad sources of randomness (i.e., values that are predictable, publicly observable, and in some cases even manipulable) provided by the underlying blockchain, such as `block.number`, `block.timestamp`, `block.difficulty`, or `blockhash`. Using these values can introduce security vulnerabilities, notably if they are used to determine critical operations like token or ether transfers, as they can be predictable and exploited by attackers.

Listing 2.5 presents a sample contract that uses `block.number` to set the winner address when the number is even. Being a predictable value (the block number is upper incremented and is public data), an attacker could use a timing technique to ensure that a transaction gets included in an even-numbered block. To reduce the risk of being targeted by this type of attack, developers should consider using external sources of randomness, such as oracles, or avoid relying solely on these attributes for critical decisions. Although static analysis tools are generally effective at detecting this type of vulnerability with relatively few false positives, the actual impact depends on runtime execution paths and the specific way the contract uses the block attributes.

```
contract BlockStateDependency {
  address public winner;

  function enter() public {
    if (block.number % 2 == 0) {
      winner = msg.sender;
    }
  }
}
```

Listing 2.5: A contract with the Block State Dependency vulnerability.

Ether Leak - SWC-105: In this vulnerability, Ether can be irreversibly lost when contracts lack proper access control, allowing arbitrary users to withdraw funds [60].

Listing 2.6 illustrates the conditions in the `batchRefund` function, where any user can invoke the function with a list of recipients of their choice and completely drain the contract's Ether. There is no balance check before sending Ether, and if any `send()` call fails, the loop continues while the contract's balance is still reduced. The balance is decreased regardless of the success of the transfers. Mitigating this vulnerability involves enforcing access control, validating balances, and using safe transfer patterns.

```
contract EtherLeak {
  mapping(address => uint256) public balances;

  function batchRefund(address[] users, uint256[] refundAmounts) external {
    for (uint i = 0; i < users.length; i++) {
      users[i].call{value: refundAmounts[i]}("");
      balances[users[i]] -= refundAmounts[i];
    }
  }
}
```

Listing 2.6: A contract with the Ether Leak vulnerability.

The remaining vulnerabilities will be briefly described as they are considered less relevant to our study:

Arbitrary Write - SWC-124: This vulnerability occurs when a contract allows an attacker to write data in an arbitrary storage location through a mismanaged array object, potentially overwriting critical contract variables or code. A detection method for this vulnerability involves intercepting data store instructions handlers to identify out-of-bounds array accesses that lead to arbitrary storage manipulation.

Assertion Failure - SWC-110: Ethereum prevents resource abuse by charging a fee, known as *gas*, for each computational operation. The `assert()` operation, normally used as a defensive programming construct, can have unintentional consequences in the context of Solidity. When an `assert()` fails, the transaction is reverted, and all the gas consumed by previous operations is lost. This behavior can potentially lead to denial-of-service (DoS) attacks, where an attacker deliberately triggers `assert()` failures to exhaust gas and render contracts inoperable through repeated transaction reverts.

Transaction Origin Use - SWC-115: A contract using `tx.origin` function for authorization may be vulnerable if it intends to identify the transaction sender. `tx.origin` refers to the original external account initiating the transaction, not necessarily the contract or account directly called the current function. A malicious attack contract can explore this to bypass authorization checks.

Suicidal Contract - SWC-106: When using Solidity's `selfdestruct` function in a contract, developers must implement strict access control mechanisms (such as `onlyOwner` modifiers or role-based access control) to prevent unauthorized users from triggering contract termination. The `selfdestruct` operation permanently deletes all contract code and storage, transferring any remaining ether balance to a specified address. An attacker could exploit a lack of access control to maliciously destroy the contract and seize its remaining funds.

Even though this work focuses on using fuzz testing to effectively find vulnerabilities in Ethereum smart contracts, static analysis methods can also be employed to detect most

of the weakness categories presented earlier. Nevertheless, static analysis approaches tend to be more efficient in terms of runtime but often suffer from a high false-positive rate [29]. Fuzzing, on the other hand, dynamically executes the contract code in a real EVM environment, ensuring that the detected vulnerabilities are guaranteed to be exploitable [61].

2.2.4 Smart Contracts Fuzzing Tools

We detail our literature review process, focusing on the collection of tools that implement smart contract fuzzing and related techniques. To gather relevant publications, we used the search query `("fuzzer" OR "fuzzing" OR "fuzz") AND ("smart contract" OR "ethereum")` on Google Scholar for the period from 2018 to 2025, which returned more than two thousand entries. The year 2018 was chosen as it marks the introduction of the first smart contract fuzzer [62], making it a starting point for our search. We then concentrated on articles that primarily discuss the development of tools, narrowing the selection down to 29 papers. In the next step, we excluded fuzzers designed for other blockchain platforms, retaining only those intended for Ethereum. Subsequently, we performed snowballing and reverse snowballing using Jiang et al. [62], the first fuzzer explicitly designed for smart contracts and a seminal work in smart contract security, to identify additional relevant literature on smart contract fuzzing. As a result, we compiled a final list of 17 Ethereum fuzzers that have been published in top peer-reviewed conferences or journals.

Table 2.1 presents an overview of the fuzzing tools studied for this dissertation. Some columns in the table need a detailed explanation for a better understanding of our intent, we will further provide its explanation.

Tool	Year	Source code	EVM	Base Tool	Pub. Venue	Oracles	Use ML	Avail. Bench.	Data Depen.	Seq. Aware
ContractFuzzer [62]	2018	✔	geth	-	ASE'18	7	✗	✔	✗	✗
ReGuard [63]	2018	✗	-	-	ICSE'18	1	✗	✗	✗	✗
ContraMaster [64]	2019	✔	geth	-	TDSC	5	✗	✔	✔	✔
ILF [16]	2019	✔	geth	-	CCS'19	6	✔	✗	✔	✗
Harvey [65]	2020	✗	-	-	FSE'20	7	✗	✗	✗	✔
Echidna [66]	2020	✔	hevm	-	ISSTA'20	-	✗	✔	✔	✗
sFuzz [57]	2020	✔	aleth	-	ICSE'20	9	✗	✗	✗	✗
ConFuzzius [67]	2021	✔	pyEVM	-	EuroS&P'21	10	✗	✔	✔	✔
Smartian [4]	2021	✔	nethermind	-	ASE'21	13	✗	✔	✔	✔
xFuzz [68]	2022	✔	aleth	sFuzz	TDSC	3	✔	✔	✗	✗
RLF [17]	2022	✔	geth	ILF	ASE'22	6	✔	✔	✗	✔
IR-Fuzz [18]	2023	✔	aleth	sFuzz	TIFS	8	✗	✔	✔	✔
ItyFuzz [69]	2023	✔	revm	-	ISSTA'23	-	✗	✗	✔	✔
EF/CF [23]	2023	✔	eEVM	-	EuroS&P'23	1	✗	✔	✗	✗
MuFuzz [70]	2024	✔	aleth	sFuzz	ICDE'24	9	✗	✔	✔	✔
CrossFuzz [71]	2024	✔	pyEVM	ConFuzzius	SCP	10	✗	✗	✔	✔
FunFuzz [72]	2024	✔	aleth	sFuzz	TOSEM	9	✗	✗	✗	✗
DogeFuzz [73]	2024	✔	geth	ContractFuzzer	SBSEG'24	6	✗	✔	✗	✗

Table 2.1: Summary of analyzed Ethereum smart contract fuzzers.

- **Source Code:** Indicates whether the fuzzer's source code is publicly available in a repository.
- **EVM:** Exhibit which EVM implementation each tool uses for the fuzzing process. As detailed in Section 2.1.4, which provides an overview of the EVM architecture, a fuzzer needs a runtime environment to execute the bytecode and interpret log traces for bugs and vulnerabilities. Some of these EVM clients are fully-fledged implementations, complex systems with block mining and consensus algorithms, while others are self-contained EVM implementations stripped down to focus solely on bytecode execution, thereby improving the throughput of a fuzzer.
- **Base Tool:** Specifies if the fuzzer is built upon a previous fuzzer release or designed from scratch.
- **Pub. Venue:** Lists the conference or journal where the fuzzer was published.
- **Oracles:** Shows the number of vulnerability detectors (also known as bug oracles, i.e., predefined rules that identify vulnerable runtime behaviors) implemented in the fuzzer.

For the remaining columns, Use ML and Avail. Bench, Data Depen., and Sequence Aware are checkmark-style columns indicating whether the fuzzer has a specific trait.

- **Use ML:** Indicates if the fuzzer utilizes Machine Learning in any of the core stages of a fuzzing campaign (as described in Section 2.2.2.3).
- **Avail. Bench:** Specifies whether the fuzzer has a publicly available benchmark or dataset for reproducibility.
- **Data Depen.:** Indicates if a fuzzer employs data flow or data dependency analysis during seed generation.
- **Sequence Aware:** States whether the fuzzer can use a sequence of transactions as a seed, rather than relying solely on random single transactions.

In the following section, we will provide a description of each tool listed in table 2.1 along with its primary contribution to the field.

2.2.4.1 Fuzzers

ContractFuzzer [62] is recognized as the first fuzzer for the Ethereum platform focused on smart contracts. It is a black-box fuzzer that requires the contract ABI to generate

transaction calls, implementing custom test oracles as predefined rules and patterns. However, Wu et al. [5] demonstrated that ContractFuzzer is inefficient in terms of the number of transactions over time, as it relies on a full EVM implementation. Additionally, it is also unable to generate inputs that trigger deeper or more complex code paths within the contract.

ContraMaster [64] was the first smart contract fuzzer to utilize data-flow dependency analysis to decide whether to mutate transaction sequences, and was also the first to be sequence-aware. **ILF** [16] employs an imitation learning approach as a symbolic execution expert, focusing on effectively generating high-quality initial inputs for transactions. **RLF** [17] is a fuzzer derived from ILF that uses a reinforcement learning network to generate meaningful transaction sequences while randomly producing transaction arguments, making it more efficient in test generation than ILF [5].

Wüstholz et al. [65] proposed **Harvey**, a closed-source grey-box fuzzer that utilizes heuristics from previous executions as an input prediction technique to explore deeper paths in contract code. However, because it is proprietary software, further evaluation is limited. **Echidna** [66], created by Trail of Bits, is based on both property-based and fuzz testing. To use Echidna, developers must write custom properties that the fuzzer will then verify, checking for violations. This approach allows for the discovery of business logic-related bugs, making Echidna the most widely used fuzzer among auditors and practitioners [5].

The **sFuzz** [57] fuzzer operates on a similar principle of AFL [40], using feedback from the runtime environment and implementing a branch distance metric to overcome difficult-to-cover branches in the contract. However, sFuzz cannot process transaction sequences as seeds. Nonetheless, it served as the foundation for several other tools in table 2.1, including xFuzz, IR-fuzz, MuFuzz, and FunFuzz.

xFuzz [68] is a sFuzz-derived fuzzer that utilizes a machine learning model trained for detecting benign functions and program paths directed towards detecting cross-contract vulnerabilities. The key intuition is that vulnerabilities in smart contracts are often located in specific error-prone code paths and that focusing the fuzzer energy on these areas can reduce the time spent on benign contracts.

Similar to xFuzz, **FunFuzz** [72], which is also derived from sFuzz, seeks to identify functions with security-critical operations by applying lightweight static analysis and determining critical opcodes in contract bytecode, then uses that information during fuzzing to guide seed mutation and selection to reach those critical functions. **IR-Fuzz** [18] builds on sFuzz by adding a layer of transaction sequence prolongation, using data dependencies analysis and prioritizing seeds based on minimum distances; Analogous to IR-Fuzz, **MuFuzz** [70] also employs data-flow-based feedback for sequence-aware mutation and

mask-guided seed mutation strategy.

Some fuzzers aim to improve vulnerability detection speed by focusing on increasing throughput. **ItyFuzz** [69] utilizes snapshot fuzzing, taking memory dumps from a highly efficient EVM and initiating a new fuzzing campaign from that point. The core idea is that smart contracts operate as stateful programs, and re-executing previous transactions to recreate a desired state comes with a high cost. **EF/CF** [23] also prioritizes high throughput, achieving this by translating smart contract bytecode into native C++ code through program transpilation.

ConFuzzius [67] is a state-of-the-art hybrid smart contract fuzzer that integrates symbolic execution and dynamic data analysis to identify Read-After-Write (RAW) dependencies in state variables. This approach seeks to generate meaningful transaction sequences while yet triggering and exploring various code branches. An extension of Confuzzius, **CrossFuzz** [71], addresses cross-contract vulnerabilities by leveraging inter-contract data flow information and optimizing mutation specifically for these types of cross-contract bugs.

DogeFuzz [73] is an extensible fuzzer that supports multiple fuzzing strategies, including black-box, coverage-guided grey-box, and directed grey-box fuzzing, targeting critical EVM instructions.

For our research, we selected Smartian [4] as the base fuzz testing tool, which we extended to develop our proposed approach, SmartChat. A brief overview of Smartian is provided in the next section.

2.2.4.2 Smartian

Smartian [4] is a hybrid smart contract fuzzer that uses static and dynamic data-flow analysis to generate interesting transaction sequences. It includes a concolic testing module to explore unexplored paths and edge cases during the fuzzing process. Smartian derives initial seeds by statically analyzing the smart contract bytecode—the input to the fuzzing process—to extract use-definition (use-def) facts. It assumes that vulnerabilities may arise when functions that define state variables must be called before those that write to the blockchain's persistent state.

Benchmark results demonstrate that Smartian significantly outperforms other fuzzers. Furthermore, the ablation experiment conducted during the evaluation study revealed that each component plays a crucial role in the overall results. It also showed that the static pre-processing step can be further enhanced by the dynamic data-flow instrumentation module. In a more recent study, Wu et al. [5] conducted a comprehensive evaluation of modern smart contract fuzzers and noted that Smartian achieves higher code coverage than other tools.

The architecture of Smartian is shown in Figure 2.5, which depicts three main components: INFOGATHER, SEEDPOOLINIT, and DATAFLOWFUZZ.

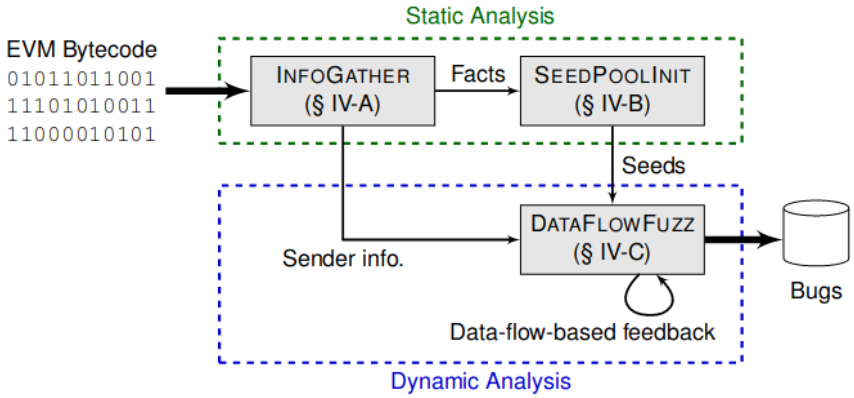

Figure 2.5: Smartian architecture [4].

The INFOGATHER module is responsible for statically analyzing contract bytecode to identify data-flow facts such as definition-use (Def-Use) chains and smart contract access control mechanisms. The collected data facts are then provided to the SEEDPOOLINIT, where high-quality initial seeds are generated based on the concept that the use and definition of state variables can guide the creation of meaningful transaction sequences. The DATAFLOWFUZZ module handles the dynamic data-flow analysis. Smartian uses a feedback loop to evaluate both code coverage gain and Def-Use gain, while monitoring storage accesses during execution.

In this work, we extend Smartian to evaluate Generative AI-generated initial seeds using LLM-based test case generation. To this end, we provide an overview of Generative AI and its applications in software testing and fuzzing, which we describe in the following section. We chose Smartian due to its superior performance reported in previous work [5, 74], even though we have extensive experience implementing our own fuzzing tool [73].

2.3 Generative AI

Generative-based AI systems are a prominent area of machine learning research, focusing on systematically creating new original content (e.g., text, images, audio, and videos) based on patterns learned from trained data. Two of the main techniques in the deep learning generative area are Generative Adversarial Networks (GANs) [75] and Transformer-based models [76]. The GAN framework consists of two neural networks (i.e., a generator and a discriminator) and is primarily used in image generation. Transformer-based mod-

els, which emerged from the self-attention mechanism and utilize parallel encode-decoder neural networks, are primarily employed in the area of Natural Language Processing (NLP). Generative Large Language Models are transformer-based models specialized for text and task instructions that have seen widespread adoption in software engineering activities; in the following sections, we briefly describe how these models work and their applications to fuzzing.

2.3.1 Pre-trained Large Language Models

Large Language Models (LLMs) are a type of machine learning model with a massive number of parameters (i.e., weights values and biases in transformer architectures) designed to predict the next token or generate a reasonable text continuation based on a given input [77]. Such models are trained on hundreds of gigabytes of text data from public data sources, like the Common Crawl dataset. LLMs excel in many tasks, most notably in text summarization, text reasoning, language translation, and conversational agents. Specifically, text-generating LLMs are autoregressive decoder-only models, meaning they are a class of transformer models that use only the decoder part of the transformer architecture. These models sequentially generate one token at a time, where each generated token is conditioned on the tokens generated before it, in an autoregressive way [78].

Furthermore, LLMs are versatile in the training process, as they can be used for both conversational and instruction tasks. Conversation-tuned models are optimized for human-like chats in a questions-and-answer format, while instruction-tuned models use a specialized instruction format for training. During inference, the model employs Reinforcement Learning from Human Feedback (RLHF) to enhance the responses, ensuring that the model follows user instructions accurately and generates texts that align closely with the given task.

2.3.1.1 LLMs metrics and concepts

Table 2.2 displays some of the main language models and their key metrics as of the time of writing this work. Key metrics elements are crucial for comparing LLMs; we will explain them below.

- **Context Size:** refers to the total length in tokens a model accepts as input and is able to process during the inference phase.
- **Training Cutoff Date:** is the final data collection date used to train the model.
- **Parameters:** is the number of weights and biases an LLM is trained with; generally, the bigger the value, the more accurate the final output of the model.

- **License:** refers to the terms or agreements governing the use of the model and the availability of the model weights files to the community.
- **Training Data Size:** is the size in number of tokens of the data collection used for training the neural network model; more data and more parameters mean more resources for processing and tuning the model.

Model	Context Window	Training Cutoff	Parameters (Billion)	License	Training Data (T)
Llama3-70B	8K	Mar 2023	70	Open	15
Llama3-8B	8K	Mar 2023	8	Open	15
Llama3-3-70B	128K	Dec 2023	70	Open	15
Mixtral-8x22B	32K	Dec 2023	22	Open	Unknown
Mixtral-8x7B	32K	Dec 2023	7	Open	Unknown
GPT-4.1	128K	Oct 2023	Unknown	Proprietary	Unknown
GPT-4.1 Mini	128K	Oct 2023	Unknown	Proprietary	Unknown
Claude 3.7 Sonnet	200K	Jun 2024	Unknown	Proprietary	Unknown
Gemini 1.5 Pro	1M	Mar 2024	Unknown	Proprietary	Unknown
Gemini 1.5 Flash	128K	Mar 2024	Unknown	Proprietary	Unknown

Table 2.2: Overview of major LLMs by context window, cutoff date, model size, license, and training data.

Other essential concepts in LLM usage relate to how they are employed and the extent to which the input interaction process is guided. While LLMs are pre-trained on vast amounts of data for numerous tasks, they can be further improved or tailored for specific tasks to provide better responses in particular contexts. This can be achieved through processes such as fine-tuning or prompt engineering [79].

Fine-tuning a model involves updating the model's weights and adapting it for a specific purpose, typically using a narrower dataset of labeled training examples. This process demands significant computational resources and incurs higher costs compared to using tailored prompts (i.e., prompt engineering). In this work, we focus on prompt engineering techniques, as they are more accessible and cost-effective.

Nevertheless, natural language prompts can also be used to achieve better generative results without additional training, through a technique known as prompt engineering. The process of selecting or designing specific inputs (i.e., prompts) does not require explicitly updating the model's weights and is used to elicit desired responses from an LLM without modifying the model itself.

In the context of prompt engineering, several techniques exist, including zero-shot, few-shot, and chain-of-thought (CoT) prompting [80]. In the zero-shot setting, the LLM is

provided only with task instructions and no examples. The few-shot variant supplements the prompt with a handful of demonstrations (e.g., output examples), allowing the model to infer the desired format and constraints. Lastly, CoT enhances the model's reasoning by incorporating intermediate steps (e.g., "Let us think step by step"). Also, CoT can be applied in a single-turn format, where reasoning is incorporated within a single prompt, or in a multi-turn setup, where the model is guided through sequential reasoning steps interactively [81].

2.3.2 LLM-Assisted Fuzzing

Since language models became widely available to the public, a significant amount of literature on LLM-assisted software engineering has been published. Key areas with LLM applicability include coding, design, requirements, repair, refactoring, testing, and documentation [79]. In software testing, fuzzing is a prominent area for LLM usage.

Deng et al. [14] used an LLM (i.e., Codex) to generate test cases code that follows API syntax and semantics of TensorFlow [82] and PyTorch [83] libraries. Their results show that in both seed mutation and generation, they achieved higher code coverage compared to existing fuzz driver generation tools. Fuzz4All [12] introduced a universal programming language fuzzer that leverages large language models in a fuzzing loop. This loop iteratively updates the initial input prompt with both code examples and generation strategies, aiming to produce diverse fuzzing seeds. The target languages under test include C, C++, SMT2, Go, Java, and Python, though the authors claim it can be easily adapted to other languages. In addition to improving coverage, the paper reports the discovery of 64 previously unknown compiler bugs.

Stateful systems are generally hard to fuzz due to the length of input sequences and combinations needed to reach a specific state (i.e., vulnerable state). Network protocol implementations are typically designed as state machines in a stateful architecture, making them hard to fuzz. Meng et al. [13] propose a protocol fuzzer using the intuition that pre-trained large language models embody a deep understanding of specification semantics and message format structure. ChatAFL is implemented on top of the AFLNet baseline, incorporating GPT-3.5 Turbo for protocol grammar generation and for reaching deeper new states. Overall, the results show that ChatAFL achieves an average of 5.8% to 6.7% more branch coverage than AFLNet.

Smart Contract fuzzing is also considered a stateful system involving all complex, hard-to-reach machine-state environments. LLM4Fuzz [74] is the first fuzzer for Ethereum smart contracts employing LLM for seed scheduling and prioritization. It uses Llama 2 70B to evaluate seed quality based on code fact producers. Producers evaluate invariants, complexity, sequential likelihood, and vulnerability probability through the use of hand-

tailored prompts aiming to obtain a numeric value. LLM4FUZZ can find exploits to vulnerable contracts in 1.9 hours, while the baseline fuzzer (Ityfuzz) takes almost 24 hours, meaning it guides the fuzzer to reach a complex state in less time.

Figure 2.6 illustrates a fuzzing workflow highlighting key points where LLMs can be applied for enhancements. Specifically:

- **Seed Generation:** LLMs can augment the generation process to provide diverse input seeds based on learned patterns from training data.
- **Seed Scheduling:** Using LLMs to assess the quality or likelihood of uncovering vulnerabilities.
- **Seed Mutation:** LLMs can play a significant role as a mutator, enhancing the generation of more effective and targeted test cases.
- **Vulnerability Detection:** LLMs can aid in identifying potential vulnerabilities or being used as test oracles.

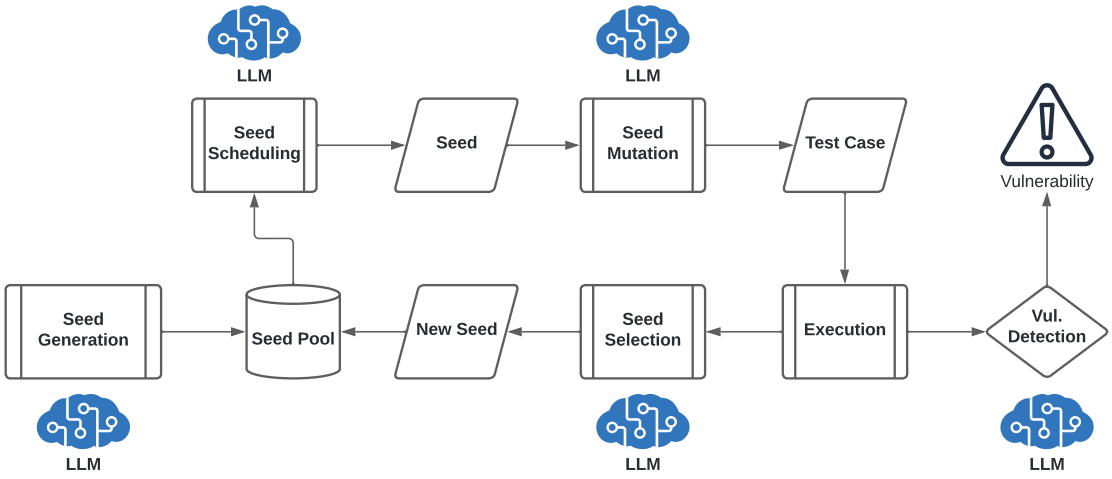

Figure 2.6: Fuzzing Workflow with LLM Integration.

While this integration presents several opportunities, it also introduces certain challenges that need to be addressed. Hallucinations in LLMs can be attributed to their inherently non-deterministic behavior, where the model may generate responses that are incorrect, irrelevant, or fabricated [84]. To reduce these occurrences in any key process entry where LLM-assisted fuzzing is used, some form of evaluation techniques or error-guided corrective measures must be applied. Furthermore, LLMs have a configurable

hyperparameter called temperature, which can reduce randomness, resulting in more deterministic outputs and lowering the likelihood of hallucinations [85].

Seed generation and seed mutation may also be prone to insufficient diversity of test cases [84], as well as inaccurate context understanding due to the small LLM context size (i.e., the number of tokens processed during inference). Fine-tuning on a specific corpus or using more precise prompt engineering methods could help mitigate these problems [79].

Chapter 3

SmartChat: LLM-Enhanced Smart Contract Fuzzing

This chapter presents the architecture and implementation of SmartChat, an enhanced Smartian-based fuzzer that integrates LLM-generated input seeds and vulnerability-guided scheduling. Section 3.1 describes the overall architecture, including the structured JSON seed format, automated seed generation, and the prioritization strategy. Section 3.2 covers the implementation, from prompt design and the multi-phase seed processing pipeline to the software modules developed to extend Smartian into SmartChat.

3.1 Architecture and Design

This work aims to develop a method for generating an initial seed set for smart contract fuzzing. To achieve this goal, we either needed to build a fuzzer or use an existing one from the research community. We chose to adopt Smartian, described in Section 2.2.4.2, due to its extensibility as an open-source project and its recognition as a high-performing tool in the field [5, 69]. Furthermore, our methodology involves developing a reliable extension tool that enables a comprehensive comparative study of experimental results.

During our evaluation of Smartian, we identified a key limitation to our research purpose: the tool does not expose its internal seed format nor support importing seeds from external sources. However, it does export the test case files that trigger vulnerabilities, which are the outcomes of its fuzzing process. Based on this, we designed an input seed format with its corresponding importing software module to enable the injection of externally generated seed inputs for testing and evaluation.

Moreover, integrating LLM-based structured output generation into the fuzzing process requires crafting natural language instructions (i.e., prompts), executing these queries, and processing the output. The LLM generates candidate seeds in JSON format first,

which are then passed to SmartChat, which only consumes these seeds once they have been converted into its internal format; therefore, the LLM precedes SmartChat in the overall process. With this integration in mind, and given that one of the goals of this research is to evaluate LLMs based on their ability to generate quality and well-structured seeds, we designed an automated generation, evaluation, and conversion pipeline.

Finally, while generating seeds that are well-structured and context-aware using LLMs represents a desirable outcome for seed generation, we also propose a vulnerability-guided seed selection method. This approach is primarily focused on steering the fuzzing process toward vulnerable execution paths.

3.1.1 Input Seed Format

In the realm of smart contract fuzzing, there is no universally defined format for input seeds. Some fuzzers use internal representations that are not exposed in a file format [67, 16], while others utilize internal binary formats [57]. Therefore, defining a generic format for capturing the low-level elements of blockchain transactions is crucial. Moreover, using a text-based format enables text-generative AI models to produce easily parseable test cases.

To address this requirement, we design a JSON format that is both general and expressive enough to support the testing of smart contracts under diverse and realistic conditions, ensuring complete preservation of all semantically relevant information needed to reproduce test executions and their outcomes. Smartian influences our design, and as the SmartChat tool extends it, we aim to ensure practical integration with its existing components. Still, the proposed format could also be used in other fuzzers, as it encompasses all key elements of the Ethereum smart contract execution environment.

An example of this format is shown in Listing 3.1. The JSON structure includes a deployment section, `DeployTx`, which represents the instantiation of a smart contract and emulates the constructor function along with its input parameters. Likewise, the `Txs` section captures the details of multiple transactions that interact with the deployed contract, other contracts, and the surrounding blockchain environment.

By specifying the sender and receiver, a value (in Ether), function, and function parameters, this seed format facilitates the simulation of diverse Ethereum contract interaction patterns, such as transferring tokens or Ether and interacting with specific contract states. Additionally, the `Timestamp` and `Blocknum` fields in the schema provide temporal and block-level context for the execution of each transaction.

```
1 "TestCase": {
2   "DeployTx": {
3     "From": "SmartianAgent1",
4     "Value": "0",
```

```
    "Function": "constructor",
    "Params": ["1000000", "TokenA", "18"],
    "Timestamp": "10000000",
    "Blocknum": "20000000"
  },
  "Txs": [{
    "From": "SmartianAgent3",
    "Value": "0",
    "Function": "transferFrom",
    "Params": ["SmartianAgent2", "5000"],
    "Timestamp": "10000161",
    "Blocknum": "20000007" }]
}
```

Listing 3.1: Example of a JSON seed generated by an LLM.

Specifying this structured format is particularly important because it enables the replication of a simulated environment, allowing for the testing of edge cases, boundary conditions, access controls, unexpected state changes, varying timestamps, and expiring conditions.

Furthermore, raw transaction calls in Ethereum are built with `calldata`, a low-level binary-encoded format that specifies the function signature and parameters. Because calldata is not human-readable and requires strict byte-level encoding, LLMs, designed to generate natural language text, tend to produce invalid or malformed outputs when asked to directly create it [86]. To address this challenge, we represent function parameters—such as strings, numbers, and addresses—in a human-readable format enclosed in JSON arrays. This abstraction allows LLMs to focus on producing structured and logical seeds without dealing with low-level encoding details, thereby simplifying test case creation and improving compatibility with smaller language models.

While the format presented in Listing 3.1 is designed for LLM generation to address challenges in binary encoding, the final format used as input for SmartChat fuzzing is shown in Listing 3.2. The main differences lie in the `data` field, which contains ABI-encoded function calls and parameters, formatted in binary for execution on the blockchain. Additionally, the `Entities` object defines all sender accounts available for the test case during the fuzzing campaign, including both attack contracts and EOA accounts. For each entity, it specifies the associated address and the initial Ether balance.

To enable this transition between human-readable and machine-executable representations,, the conversion is done by a validation and encoding pipeline that maps symbolic agent names and parameters to actual blockchain addresses, checks function calls against the ABI, and encodes inputs using Ethereum's standards. This produces binary-encoded transactions executable on the EVM, allowing the LLM to generate meaningful scenarios without handling low-level encoding details.

```
"Entities": [
```

```
  {
    "Balance": "10000000000000000000",
    "Account": "0xb7705ae4c6f81b66cdb323c65f4e8133690fc099",
    "Agent": "SmartianAgent",
    "Contract": "0x24cd2edba056b7c654a50e8201b619d4f624fdda"
  },
  [...]
],
"TargetDeployer": "0x24cd2edba056b7c654a50e8201b619d4f624fdda",
"TargetContract": "0x6b773032d99fb9aad6fc267651c446fa7f9301af",
"DeployTx": {
  "From": "0x24cd2edba056b7c654a50e8201b619d4f624fdda",
  "To": "0x6b773032d99fb9aad6fc267651c446fa7f9301af",
  "Value": "0",
  "Data": "0000000000000000000000000118a2c24808934116e6ab4c00ff48145d23b09e100000000",
  "Timestamp": "10000000",
  "Blocknum": "20000000",
  "Function": "constructor"
},
"Txs": [{
  "From": "0x24cd2edba056b7c654a50e8201b619d4f624fdda",
  "To": "0x6b773032d99fb9aad6fc267651c446fa7f9301af",
  "Value": "0",
  "Data": "095ea7b3000000000000000000000000033b808a5ae24c410e8739b5ca2d5ef3931d3e09f",
  "Timestamp": "10000141",
  "Blocknum": "20000198",
  "Function": "transferFrom(095ea7b3)"
},
[...]
]
```


Listing 3.2: Example of a final JSON seed used as input to SmartChat.

3.1.2 LLM-Based Automated Seed Generation

Since our focus is on generating initial fuzzing seeds to enhance early vulnerability discovery, the core components for automated seed generation must operate before the fuzzing campaign begins. To support this, we designed a multi-stage approach that processes, validates, and converts the structured JSON outputs produced by the LLM.

Figure 3.1 outlines the phases of SmartChat, which include (a) Static Analysis of Smart Contract, (b) Prompt Building and LLM-Guided Seed Generation, and (c) Seed Validation and Encoding.

In the first phase (Static Analysis of Smart Contracts), SmartChat relies on a dedicated software module to analyze and process the target contract's source code. Solidity smart contracts are compiled into EVM bytecode (see Section 2.1.4), resulting in a bytecode file and an ABI file. SmartChat uses a static analysis tool (i.e. Slither) to examine the contract and extract key information, including function signatures (with parameter types), access control modifiers (e.g., `onlyOwner`, `public`, `payable`), and the full source

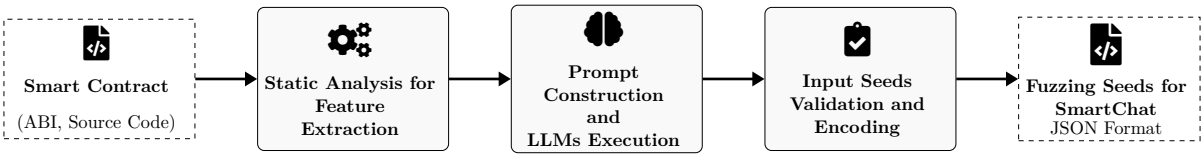

Figure 3.1: SmartChat system pipeline overview: (1) Smart contract files (ABI and source code) serve as input to the system, (2) Contract preprocessing and feature extraction component extracts function signatures, modifiers, and contract metadata, (3) Prompt construction and execution stage leverages LLM multi-model generation with diverse prompting strategies, (4) Seed validator and converter component performs JSON schema validation, ABI compliance checking, and argument encoding to produce initial seed sets for use in SmartChat fuzzing campaigns

code. We chose to use Slither as the static analysis tool primarily because it provides convenient helpers for parsing and querying smart contract files.

The second phase (Prompt Building and LLM-Guided Seed Generation) consists of modules designed to guide LLMs in generating transaction sequences. Using data from the previous phase (Static Analysis of Smart Contracts), the system formats prompt messages in the correct order, replacing placeholder markers with information extracted from the contract under test. A generic component enables the evaluation of different prompt strategies and also supports querying multiple LLM providers, local or remote, through a single LLM API.

In the third phase (Validation, Conversion, and Encoding), SmartChat validates the generated test cases to ensure conformance with the JSON schema, ABI compatibility, and correct argument types. It also performs conversion and encoding, resolves agent references, and detects duplicates using cryptographic hashing. This phase includes a validation and quality assessment module that extracts key metrics throughout the generation pipeline, supporting both immediate validation and longer-term quality analysis of the LLM-generated content. The outcomes of SmartChat, after discarding ill-formed seeds, are converted into the input format (see Listing 3.2), expected by the fuzzer and serve as the starting point for fuzzing campaigns.

3.1.3 Vulnerability-Guided Prioritization

While high-quality initial seeds provide a strong starting point, effective fuzzing also requires an evolutionary approach that iteratively mutates and selects seeds based on a fitness function, steering the search toward inputs more likely to expose vulnerabilities.

In this context, seed prioritization plays a critical role in smart contract fuzzing, as it influences which transaction sequences are selected to drive the exploration of the contract's code and state space [5]. Common strategies for prioritizing generated seeds include

coverage-based, distance-based, and data-flow approaches, each aiming to maximize the likelihood of discovering critical flaws (see Section 2.2.2 for more details).

Coverage-guided fuzzers aim to select inputs that can potentially drive deeper exploration of the code, and this has been the traditional approach adopted by many smart contract fuzzers [57, 66, 67]. However, traditional coverage-driven fuzzing is often considered ineffective for smart contracts due to their inherently stateful nature [74]. As a result, alternative strategies have been explored. For instance, Smartian leverages data-flow coverage based on the idea that tracking the definition and use of state variables can guide the generation of meaningful transaction sequences. Distance-based metrics are another alternative that prioritizes seeds based on their proximity to specific program points (e.g., dangerous instruction) [73].

SmartChat is designed upon RLF's approach to vulnerability-guided seed prioritization. RLF prioritizes seeds based on their potential to expose vulnerabilities by counting the number of detected bugs associated with specific function-call sequences while considering code coverage. Following this principle, SmartChat retains seeds that trigger vulnerabilities (regardless of whether they provide new coverage) alongside those that achieve traditional coverage gains, ensuring that promising attack vectors are preserved for further mutation and exploration.

Algorithm 1 illustrates the SmartChat seed evaluation method. Some fundamental concepts require clarification. Coverage gain (`covGain`) is computed as a boolean indicator of whether previously unseen control-flow edges were identified during execution by the EVM, where edges represent transitions between basic blocks in the smart contract bytecode. Similarly, def-use gain (`duGain`) indicates whether any new state variable definition and use chains were identified, where each chain is a data structure tracking data-flow from storage write operations to subsequent read operations. Bugs are detected dynamically during the execution of each seed on the instrumented EVM. The bug set (`bugSet`) is a data structure representing detected vulnerabilities, with bug detection occurring through runtime bug oracles that monitor contract execution patterns. The combination logic acts as a fitness function: it evaluates whether a seed is meaningful by checking vulnerability findings and coverage gains.

Algorithm 1 operates in a straightforward manner. It starts by initializing a seed queue with a set of initial seeds (Line 2). In our design, SmartChat leverage language models to build the initial seed set. Next, it enters a loop that runs until the time budget is exhausted (Line 3). In each iteration, a seed is dequeued for execution (Line 4), and its coverage gain, def-use gain, and total newly found vulnerabilities (Line 5) are collected. A flag is set if any bugs are discovered (Line 6). Depending on whether the vulnerability-guided option is enabled (Lines 7-8), the seed is considered "interesting" either if it reveals

Algorithm 1: Vulnerability-Guided Fuzzing Loop.

```
1:  Input: Initial seeds S_0
2:  Initialize seed queue: Q ← S_0
3:  while time budget not exhausted do
4:      seed ← dequeue(Q)
5:      covGain, duGain, bugSet ← execute(seed)
6:      bugGain ← |bugSet| > 0
7:      if vulnerabilityGuided option enabled then
8:          isInteresting ← bugGain OR covGain
9:      else
10:         isInteresting ← covGain OR duGain
11:     end if
12:     if isInteresting then
13:         S_mutated ← mutate(seed)
14:         Q ← Q ∪ S_mutated
15:     end if
16: end while
```

a bug or increases coverage. Otherwise, the default mode uses the coverage increase flag or the existence of any def-use gain. If the seed is signaled interesting (Line 12), it is mutated to generate new seeds (Line 13), which are then enqueued back into the queue for future exploration (Line 14). This process continues iteratively until the campaign's time budget has been reached.

3.2 Implementation

We implement SmartChat by extending Smartian with 341 lines of F# code and approximately 2500 lines of Python code. The F# component handles the processing and loading of the input seed, as well as the vulnerability-guided prioritization mechanism. The Python code includes the smart contract LLM-based automated seed generation module.

3.2.1 Prompting Strategies

Concerning the LLM prompting mechanisms used, we leverage in-context learning [87], augmented with examples that illustrate the desired output format. These examples help models infer patterns and generalize their behavior. To further enhance reasoning, we incorporate few-shot Chain-of-Thought (CoT) prompts in both single-turn and multi-turn formats. Single-turn CoT guides the model to reason step by step within a single prompt, while multi-turn CoT structures the analysis across multiple conversational steps.

Additionally, we adopt a few-shot learning approach, as a zero-shot prompting strategy was also evaluated but proved insufficient for our needs. The task requires generating a specific and complex JSON format, which benefits from illustrative examples, and also demands an understanding of Ethereum transaction structures.

SmartChat uses three distinct prompt templates that fall into the following categories: (i) ABI-focused few-shot prompt that leverages function signatures and modifiers extracted through source code analysis (Figure 3.2); (ii) Vulnerability-analysis single-turn prompt that combines source code examination with few-shot chain-of-thought reasoning (Figure 3.3); and (iii) a multi-turn vulnerability-analysis CoT strategy that separates the analysis and seed generation into distinct conversational rounds (Figure 3.4).

Prompt Template

System:
You are an expert assistant specializing in Solidity fuzzing with a deep understanding of SWC and DASP vulnerabilities. Your objective is to generate a diverse set of transactions and inputs targeting the main EVM/Solidity vulnerabilities. Respond strictly in JSON format following the provided instructions without any additional text.

Using only Contract ABI Functions

User:

Available Agents: You have four sender contracts: SmartianAgent1, SmartianAgent2, SmartianAgent3, and SmartianAgent4. Use their names in the parameters that need an address and in From fields as needed.
Contract Functions: `FUNCTIONS_OF_THE_CONTRACT`

JSON Grammar for EVM Test Case:

Root
- An array of ‘TestCase‘ objects.

TestCase
- **DeployTx**: An object representing the deployment transaction, using the constructor function.
- **Txs**: An array of transaction (Tx) objects.

DeployTx
- **From**: A string representing the deployer’s name or address.
- **Value**: A string representing the amount of Ether sent with the transaction (in Wei, use "0" if no Ether is sent).
- **Function**: A string representing the constructor function name being called.
- **Params** (optional): An array representing the parameters passed to the constructor, if it exists.
- **Timestamp**: A string representing the timestamp of the transaction.
- **Blocknum**: A string representing the block number when the transaction was included.

Tx (Transaction)
- **From**: A string representing the sender’s name.
- **Value**: A string representing the amount of Ether sent with the transaction, if function is payable (in Wei, use "0" if no Ether is sent).
- **Function**: A string representing the function name being called.
- **Params** (optional): An array representing the parameters passed to the function.
  - Parameters can be nested arrays.
- **Timestamp**: A string representing the timestamp of the transaction.
- **Blocknum**: A string representing the block number when the transaction was included.

Example: `SAMPLE_TEST_CASES`
Objective: Create `N_TESTCASES` new test case objects, each containing more than `N_TRANSACTIONS` transactions that might uncover bugs in the contract. Ensure transactions use raw values and respect data types in the function signatures and consider functions modifiers in your transactions. Provide the response as RFC8259 compliant JSON without explanations.
Notes:

- Each ‘TestCase‘ contains a ‘DeployTx‘ object and an array of ‘Tx‘ objects.
- Each transaction (‘Tx‘) includes details such as sender (‘From‘), value (‘Value‘), function name (‘Function‘), optional parameters (‘Params‘), timestamp (‘Timestamp‘), and block number (‘Blocknum‘).
- Parameters (‘Params‘) can be nested arrays to accommodate functions requiring multiple lists of parameters.

Figure 3.2: Prompt template using contract ABI functions

Prompt Template

System:
You are an expert assistant specializing in Solidity fuzzing with a deep understanding of SWC and DASP vulnerabilities. Your objective is to generate a diverse set of transactions and inputs targeting the main EVM/Solidity vulnerabilities. Respond strictly in JSON format following the provided instructions without any additional text.

Using the Contract Source Code

User:
Vulnerability Analysis: Is this contract vulnerable to any of `VULNERABILITY_TYPES`? Think step-by-step, write seeds that demonstrates the actual attack vector against this contract and only use contract ABI functions
Contract Code: `CONTRACT_CODE`
Available Agents: You have four sender contracts: SmartianAgent1, SmartianAgent2, SmartianAgent3, and SmartianAgent4. Use their names in the parameters that need an address and in From fields as needed.
Contract Functions: `FUNCTIONS_OF_THE_CONTRACT`

JSON Grammar for EVM Test Case:

Root
- An array of ‘TestCase‘ objects.

TestCase
- **DeployTx**: An object representing the deployment transaction, using the constructor function.
- **Txs**: An array of transaction (Tx) objects.

DeployTx
- **From**: A string representing the deployer’s name or address.
- **Value**: A string representing the amount of Ether sent with the transaction (in Wei, use "0" if no Ether is sent).
- **Function**: A string representing the constructor function name being called.
- **Params** (optional): An array representing the parameters passed to the constructor, if it exists.
- **Timestamp**: A string representing the timestamp of the transaction.
- **Blocknum**: A string representing the block number when the transaction was included.

Tx (Transaction)
- **From**: A string representing the sender’s name.
- **Value**: A string representing the amount of Ether sent with the transaction, if function is payable (in Wei, use "0" if no Ether is sent).
- **Function**: A string representing the function name being called.
- **Params** (optional): An array representing the parameters passed to the function.
  - Parameters can be nested arrays.
- **Timestamp**: A string representing the timestamp of the transaction.
- **Blocknum**: A string representing the block number when the transaction was included.

Example: `SAMPLE_TEST_CASES`
Objective: Create `N_TESTCASES` new test case objects, each containing more than `N_TRANSACTIONS` transactions that might uncover bugs in the contract. Ensure transactions use raw values and respect data types in the function signatures and consider functions modifiers in your transactions. Provide the response as RFC8259 compliant JSON without explanations.
Notes:

- Each ‘TestCase‘ contains a ‘DeployTx‘ object and an array of ‘Tx‘ objects.
- Each transaction (‘Tx‘) includes details such as sender (‘From‘), value (‘Value‘), function name (‘Function‘), optional parameters (‘Params‘), timestamp (‘Timestamp‘), and block number (‘Blocknum‘).
- Parameters (‘Params‘) can be nested arrays to accommodate functions requiring multiple lists of parameters.

Figure 3.3: Prompt template using contract ABI functions and Contract Source Code

Prompt Template

Turn 1 – Contract Analysis

System: You are an expert assistant specializing in Solidity fuzzing with a deep understanding of SWC and DASP vulnerabilities. **User:**
Vulnerability Analysis: Analyze this Solidity contract for the following specific vulnerabilities: `VULNERABILITY_TYPES`. Think step-by-step and identify possible attack vectors.
Contract Code: `CONTRACT_CODE`
Think step-by-step on how to exploit this.
For each vulnerability found, provide: all transaction calls that leads to this vulnerability (in plain text format) with exact input values, considering states changes. Print no other text, no explanations.
Constraints: Ensure the transactions use raw values and respect the data types in the function signatures and consider function modifiers in your transactions.

Turn 2 – Seed Generation

System: You are an expert assistant specializing in Solidity fuzzing with a deep understanding of SWC and DASP vulnerabilities. Your objective is to generate a diverse set of transactions and inputs targeting the main EVM/Solidity vulnerabilities. Respond strictly in JSON format following the provided instructions without any additional text.
User:
Contract Analysis Result: `ANALYSIS_RESULT_FROM_PREVIOUS_TURN`
Objective:
Create `N_TESTCASES` new test case objects, each containing more than `N_TRANSACTIONS` transactions, targeting the Contract Analysis.
Available Agents: You have four sender contracts: SmartianAgent1, SmartianAgent2, SmartianAgent3, and SmartianAgent4. Use their names in the parameters that need an address and in From fields as needed.
Contract Functions: `FUNCTIONS_OF_THE_CONTRACT`
JSON Grammar for EVM Test Case:

Root
- An array of ‘TestCase‘ objects.

TestCase
- **DeployTx**: An object representing the deployment transaction, using the constructor function.
- **Txs**: An array of transaction (Tx) objects.

DeployTx
- **From**: A string representing the deployer’s name or address.
- **Value**: A string representing the amount of Ether sent with the transaction (in Wei, use "0" if no Ether is sent).
- **Function**: A string representing the constructor function name being called.
- **Params** (optional): An array representing the parameters passed to the constructor, if it exists.
- **Timestamp**: A string representing the timestamp of the transaction.
- **Blocknum**: A string representing the block number when the transaction was included.

Tx (Transaction)
- **From**: A string representing the sender’s name.
- **Value**: A string representing the amount of Ether sent with the transaction, if function is payable (in Wei, use "0" if no Ether is sent).
- **Function**: A string representing the function name being called.
- **Params** (optional): An array representing the parameters passed to the function.
  - Parameters can be nested arrays.
- **Timestamp**: A string representing the timestamp of the transaction.
- **Blocknum**: A string representing the block number when the transaction was included.

Example: `SAMPLE_TEST_CASES`
Notes:

- Each ‘TestCase‘ contains a ‘DeployTx‘ object and an array of ‘Tx‘ objects.
- Each transaction (‘Tx‘) includes details such as sender (‘From‘), value (‘Value‘), function name (‘Function‘), optional parameters (‘Params‘), timestamp (‘Timestamp‘), and block number (‘Blocknum‘).
- Parameters (‘Params‘) can be nested arrays to accommodate functions requiring multiple lists of parameters.

Figure 3.4: Prompt template for multi-turn contract analysis followed by seed generation.

We define the system role message by specifying the objectives and the desired output format to guide the LLM in aligning the prompt's intent with the specific task. All other messages use the user role.

All prompt templates use placeholders [88] (i.e., variable interpolation) to dynamically incorporate content such as contract code, function signatures, generation parameters, and targeted vulnerability names. This design allows for flexible and programmatic prompt construction tailored to the specific fuzzing task. To ensure adherence to our seed format requirements, we employ format-restricting instructions, which have proven more effective for reasoning tasks than alternative approaches such as constrained generation [89].

Below, we detail each placeholder and its role:

- `CONTRACT_CODE`: The complete Solidity smart contract source code to be analyzed, with all comments removed. It provides the LLM with the full implementation needed for vulnerability analysis and is presented as raw Solidity code in string format.
- `FUNCTIONS_OF_THE_CONTRACT`: A list of all public and external functions available in the contract. This defines the set of function signatures that can be used for transaction generation and is presented in ABI-style format or as structured function descriptions, such as transfer(address,uint256), approve(address,uint256), or balanceOf(address).
- `VULNERABILITY_TYPES`: Specific vulnerability categories to be targeted during analysis, allowing the process to focus on particular security weaknesses such as those defined by SWC or DASP classifications. The categories are provided as a comma-separated list of vulnerability names, for example: Reentrancy, Integer Overflow.
- `N_TESTCASES`: Defines the number of test cases to generate in the current request, controlling the quantity of test cases produced.
- `N_TRANSACTIONS`: This input sets the minimum number of transactions per seed.
- `ANALYSIS_RESULT_FROM_PREVIOUS_TURN:` This contains the output from the first turn in the multi-turn prompting template, including the results of the vulnerability analysis. It provides context from the prior analysis to guide test case generation.
- `SAMPLE_TEST_CASES`: An example test case with the expected JSON format and structure, working as a template for the LLM to follow when generating new JSON-formatted fuzzing seeds.

In multi-turn Chain-of-Thought (CoT) prompting, the first turn incorporates `CONTRACT_CODE` and `VULNERABILITY_TYPES` to prompt the model for analysis. The second turn then builds on the previous output (`ANALYSIS_RESULT_FROM_PREVIOUS_TURN`), supplemented

by generation parameters. In single-turn prompting, the system either uses the full `CONTRACT_CODE` along with `FUNCTIONS_OF_THE_CONTRACT`, or, in ABI-based prompts, only the `FUNCTIONS_OF_THE_CONTRACT`, in both cases combined with the relevant generation parameters. All prompts must conform to a shared `JSON Grammar`, defined within each prompt template, which enforces a consistent test case format, specifies objects, and constrains data types.

3.2.2 Seed Generation, Validation and Conversion

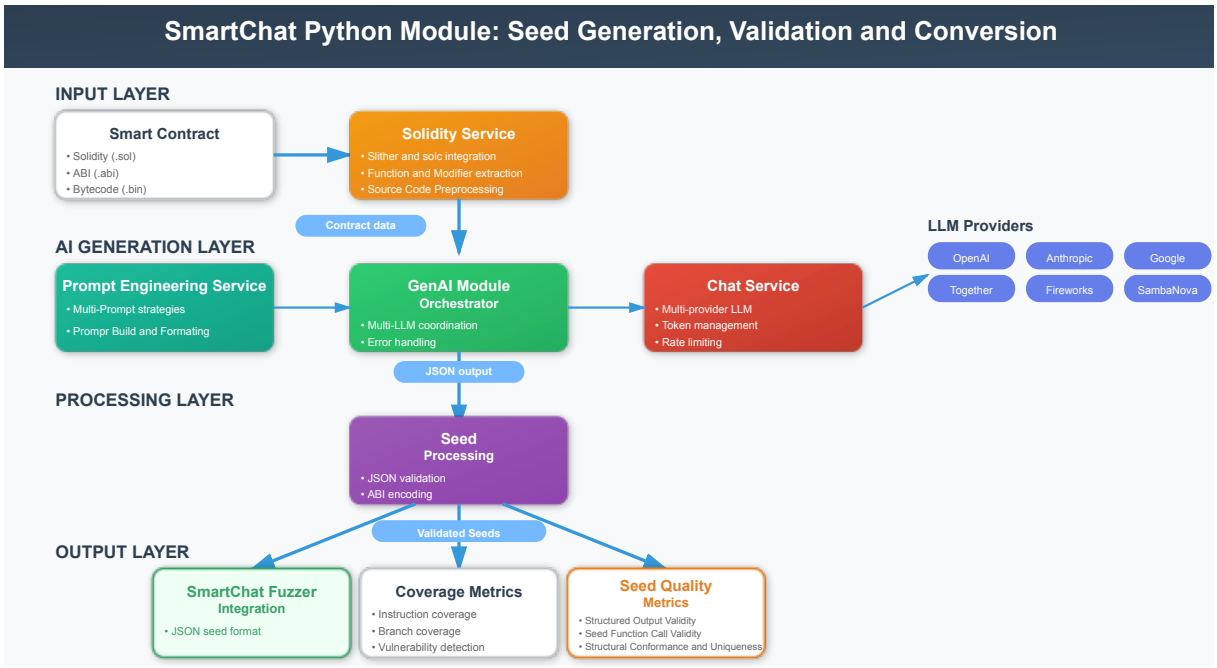

Figure 3.5: SmartChat Python subsystem for seed generation, validation and conversion.

Figure 3.5 illustrates the SmartChat Python subsystem, which follows a modular architecture centered around the `GenAI` module. This module serves as the main orchestrator for the entire seed generation, validation and conversion process. The system is organized into layers and core components, as outlined below:

Input Layer

Solidity Service: Performs contract analysis using the Slither framework to extract function signatures, modifiers, and interface specifications from smart contract ABIs and source code. It automatically handles Solidity compiler version management and provides cleaned contract representations for LLM consumption.

AI Generation Layer

Prompt Engineering Service: Provides abstraction to build and format multiple prompt templates designed to guide LLMs in generating test cases, building final executable prompts from contract data, and generating parameters.

Chat Service: Manages interactions with multiple LLM providers, including OpenAI GPT models, Anthropic Claude, Google Gemini, and various open-source models through providers like Together, Fireworks, and SambaNova. This layer is implemented using Python APIs for all providers (i.e., OpenAI HTTP API) and incorporates error handling for common exceptions, including HTTP request failures, token-related issues, and model-specific errors, while also applying provider-specific optimizations.

Processing Layer

Seed Processing: Implements transaction argument encoding and validation logic for EVM execution. This component handles JSON validation, type conversion, address resolution, and ABI-compliant parameter encoding.

Output Layer

The output layer consists of the outcomes from the entire process: seeds for input to the fuzzers, data files containing metrics for LLM seed generation evaluation, and metrics about the coverage of seeds executed in the EVM.

Chapter 4

Empirical Assessment

4.1 Introduction

This chapter presents the experimental evaluation of the proposed solution, including the SmartChat tool and associated modules, along with the observed results and discussion. The evaluation assesses the effectiveness of using Large Language Models (LLMs) to generate initial seeds for smart contract fuzzing, combined with vulnerability-guided prioritization mechanisms, against state-of-the-art fuzzing tools. The experiment is described through research questions, which are addressed based on the results obtained from evaluating the outputs generated by the LLM and the execution of the fuzzers against smart contract benchmarks.

4.1.1 Goals

We apply the Goal-Question-Metric (GQM) methodology [90] to structure our evaluation approach. Table 4.1 defines the primary goals of this study.

Table 4.1: GQM goal.

Goal: Effectiveness of SmartChat in Enhancing Smart Contract Fuzzing	
Purpose	Assess the effectiveness of SmartChat by evaluating the quality of LLM-generated seeds, the contribution of vulnerability-guided prioritization, and the comparative performance against state-of-the-art fuzzers
Issue	Structural validity, semantic correctness, coverage quality, vulnerability detection rate, and speedup
Object	Seeds produced by open- and closed-weight LLMs under varying temperatures; fuzzing campaigns with and without prioritization; SmartChat compared with Smartian and ConFuzzius under different prompt strategies
Viewpoint	Researchers in smart contract fuzzing and practitioners
Context	Evaluation of seed quality and vulnerability discovery using a dataset of smart contracts with multiple vulnerability classes

4.1.2 Research Questions

The main goal of this evaluation is to assess the effectiveness of the proposed SmartChat extensions in the context of smart contract fuzzing. Specifically, we seek to quantify improvements over state-of-the-art fuzzers, including Smartian and ConFuzzius. To this end, we conduct a set of experiments designed to answer the following research questions:

RQ1: How effective are open- and closed-weight LLMs at generating structurally valid smart contract seeds and maximizing code coverage under varying sampling temperatures?
Answering this question is important for identifying which models are better suited for integration with SmartChat.

RQ2: To what extent do SmartChat's core components—LLM-generated seeds and vulnerability-guided seed prioritization—contribute to improved bug detection compared to the original Smartian implementation?
This question assesses the impact of SmartChat extensions through an ablation-style comparison of LLM-generated seeds with and without vulnerability-guided prioritization, while also identifying an optimal temperature for later evaluations.

RQ3: How do the bug-finding capabilities of SmartChat compare to the state-of-the-art fuzzers Smartian and ConFuzzius under different prompt strategies?

This question enables the evaluation of different prompt strategies while benchmarking SmartChat performance against established fuzzing tools considering a larger set of vulnerability classes.

4.1.3 Metrics Definition

4.1.3.1 Metrics for RQ1

- **Structured Output Validity**: Percentage of valid JSON files generated by models (valid when JSON structure is properly formed with correct objects/arrays and proper closure)
- **Seed Structural Conformance**: Number of seeds conforming to JSON Schema specification, including type validation, nested structure consistency, and length constraints
- **Seed Uniqueness**: Total number of duplicate seeds detected using SHA-256 hash values across all seed collections
- **Function Call Error Rate**: Percentage of hallucinated calls to non-existent functions in the contract
- **Function Call Argument Error Rate**: Percentage of incorrect function signatures (wrong number or types of arguments)
- **Instruction Coverage**: Percentage of EVM instructions executed during processing of LLM-generated initial seeds using SmartChat's EVM replay feature

4.1.3.2 Metrics for RQ2

- **Vulnerability Detection Rate**: Total number of vulnerabilities discovered during fuzzing campaign executions

4.1.3.3 Metrics for RQ3

- **Vulnerability Detection Rate**: Total number of vulnerabilities found over 60-minute fuzzing campaigns at multiple time intervals (1-minute, 10-minute, 60-minute marks)
- **Instruction Coverage**: Percentage of EVM instructions covered during fuzzing campaigns

- **Time-to-Vulnerability Speedup**: Reduction factor in time required to reach specific vulnerability detection thresholds compared to baseline tools.

All collected measures are based on widely accepted metrics in fuzzing research [91], including the approach used for evaluating statistical significance [92].

4.2 Experimental Settings

4.2.1 Benchmarks

Our empirical assessment leverages two labeled benchmarks selected for their coverage of real-world contracts and vulnerability diversity. The first, Bench58, comprises 58 contracts from VeriSmart [93] with integer over/underflows, offering representative real-world usage patterns and enabling a focused evaluation of initial seed generation in the context of a critical vulnerability class.

To evaluate SmartChat's effectiveness across diverse vulnerability types and in comparison with different fuzzing tools, we constructed Bench78 by carefully selecting and combining contracts from two established benchmarks: Smartian[1] (69 contracts) and the ConFuzzius (9 contracts) evaluation dataset. Smartian primarily includes contracts with Block Dependency (BD), Mishandled Exception (ME), and Reentrancy (RE) vulnerabilities. However, it lacks important categories such as Integer Bugs (IB) and Ether Leaks (EL). To address this, we incorporated additional contracts from ConFuzzius to extend vulnerability diversity. Contracts with hardcoded Ethereum address checks or lacking public functions were excluded, as they pose no realistic detection condition. Bench78 captures a total of 14 BD, 48 ME (including Gasless Send, Dangerous Delegate Call, and Exception Disorder), 19 RE, 7 IB, and 7 EL cases.

The curated samples in the VeriSmart [93], Smartian [4], and ConFuzzius [67] benchmarks are widely adopted in the smart contract security literature [95, 96, 17, 18, 71], supporting direct comparison to prior work and facilitating reproducibility.

4.2.2 LLM Model Selection

For our experiments, we selected seven instruction-tuned models that are widely adopted in research and practice. Our selection includes four open-weight models (Llama3-8B, Llama3-70B, Llama3.3-70B, and Mixtral-8x7B) and three proprietary models (GPT-4o-Mini, GPT4.1-Mini, and Gemini-1.5-Flash). These models were chosen to ensure a representative and balanced comparison across different architectures, parameter scales, and

[1]Smartian extracted contracts from SmartBugs [94]

training strategies. Table 4.2 summarizes the metadata of the models used in our experiments, following best practices for study reproducibility [97]. Proprietary models were accessed exclusively via their official API providers. Open-source models were accessed through inference endpoints (remote APIs that host and serve the models) allowing queries without running them locally. All executions used full-precision weights provided by the API services, with no quantized variants involved.

Model	Parameters (Billion)	Model Version	Type	API Provider
Llama3.3-70B	70	Meta-Llama-3.3-70B-Instruct	Open-source	Together, Sambanova
Llama3-70B	70	Meta-Llama-3-70B-Instruct	Open-source	Together, Deepinfra, Hyperbolic
Llama3-8B	8	Meta-Llama-3-8B-Instruct	Open-source	Together, Grok, Deepinfra
Mixtral-8x7B	7	Mixtral-8x7B-Instruct	Open-source	Together, Grok, Deepinfra
Gemini-1.5-Flash	Unknown	gemini-1.5-flash-002	Proprietary	Google Cloud AI
GPT-4.1-Mini	Unknown	gpt-4.1-mini-2025-04-14	Proprietary	OpenAI API
GPT-4o-Mini	Unknown	gpt-4o-mini-2024-07-18	Proprietary	OpenAI API

Table 4.2: LLM model metadata details.

Based on the findings of [98], we evaluated the models across a temperature range from 0 to 1.2, in increments of 0.2, as higher temperatures (above 1.2) are known to produce ill-formed seeds in structured data generation. This evaluation allows us to identify the most suitable model configuration for generating smart contract seeds. For each model, we adopted consistent inference parameters: top-p was set to 1.0. No additional sampling parameters, such as frequency or presence penalties, were applied.

We performed eight runs of the same prompt at the same temperature using API providers. Each run is executed in a separate chat session. This approach accounts for the probabilistic nature of LLMs, which can produce varied responses to identical queries even at the same temperature [97]. To ensure a fair comparison, all models in this study are restricted to a maximum token limit of 8192, determined by the smallest token capacity among them, except in RQ3 where this limit is relaxed to investigate the impact of source code in prompts.

4.2.3 Comparison Tools

We used two smart contract fuzzers in our experiments: Smartian [4] and ConFuzzius [67]. Smartian was selected as our approach extends it with LLM-generated seeds and vulnerability-guided prioritization. ConFuzzius also represents the current state-of-the-art in vulnerability detection, coverage, and execution speed [5].

4.2.4 Test Environment

We conducted all our experiments using an Ubuntu 22.04.5 LTS server featuring a 13th-generation Intel® Core™ i5-13500 CPU @ 4.80GHz (14 cores, 20 threads) and 32GB RAM. Each fuzzing trial uses a dedicated core to ensure a fair comparison. We utilize Docker to provide a consistent and reproducible runtime environment, the corresponding image can be build from `https://github.com/PAMunb/SmartChat`

4.3 Results and Analysis

4.3.1 RQ1: LLMs for Initial Fuzzing Seed Generation

Producing structured output through the use of LLMs can present several significant challenges. LLMs can produce inconsistency and unpredictability, sometimes producing well-structured outputs and other times deviating from the expected format (ill-formed).

The temperature parameter in LLMs controls the trade-off between determinism and randomness: lower temperatures produce more deterministic outputs, whereas higher values introduce greater variability and creativity—though they also increase the risk of hallucinations, particularly as the temperature approaches upper bounds (e.g., values close to 2). With this, our aim here is to find the optimal model-temperature configuration for subsequent experiments.

In this experiment, we use only the ABI-focused few-shot prompt template (refer to Section 3.2.1), as our objective is to evaluate the seed generation quality of various LLMs.

We evaluate the models using key metrics: structural validity and coverage. Structural validity captures the syntactic and semantic correctness of the generated seeds across several dimensions, including (i) structured output validity, (ii) seed structural conformance and uniqueness, and the (iii) seed function call validity, the ability to produce valid function calls (i.e., valid transactions calls with appropriate arguments) without hallucinations or errors. Coverage measures how effectively the generated seeds explore the input space of the target smart contracts.All reported results correspond to the average of eight independent runs for each temperature setting.

4.3.1.1 Structured Output Validity

Figure 4.1 presents a comparative analysis of the total number of valid output files generated by different models across a range of temperature values. We consider an output file valid when a model generates valid JSON as requested for a container of initial fuzzing seeds, considering the benchmark Bench58 with prompt parameters of ten test cases with at least four transactions. The JSON structure is considered invalid when an object or

array is missing or the overall structure is not properly closed. In such cases, the entire output becomes unusable, losing all generated seeds and negatively impacting the model's performance.

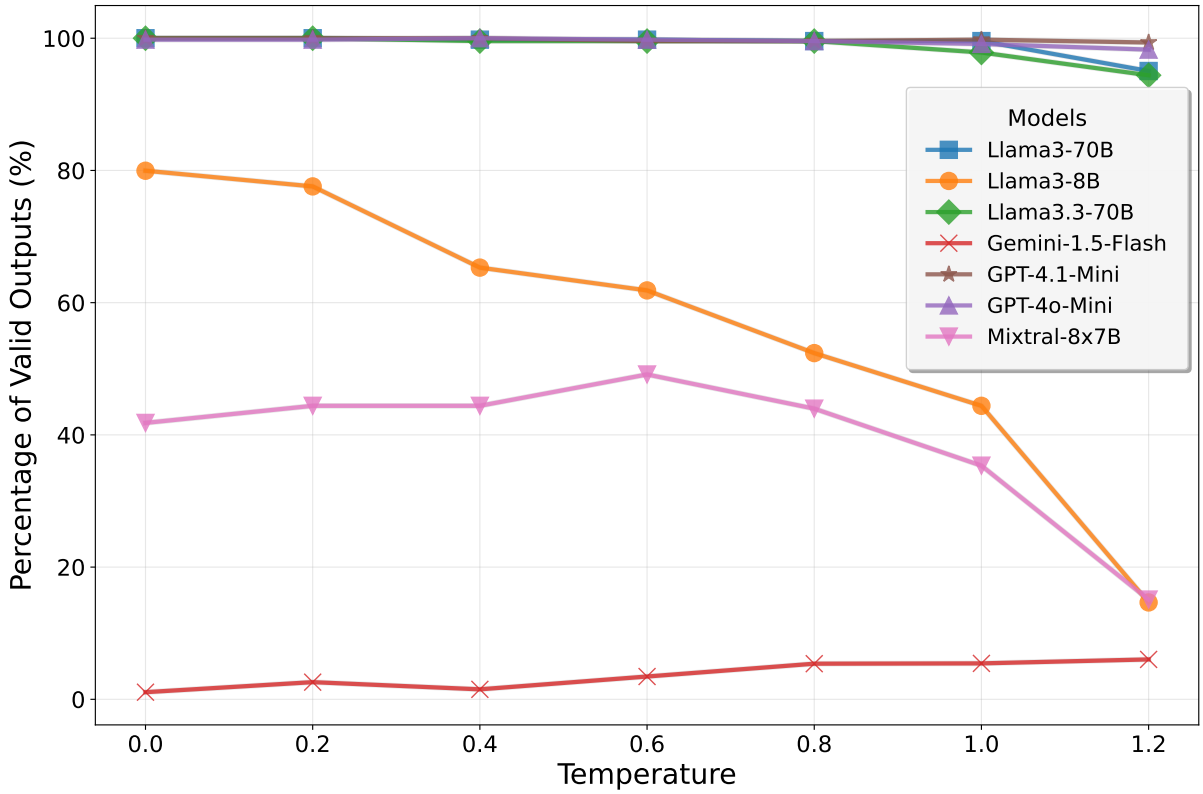

Figure 4.1: Valid structured outputs by temperature and model.

As seen in Figure 4.1, model performance in generating valid structured outputs declines as the temperature increases. While larger models such as Llama3-70B, Llama3.3-70B, GPT-4o-Mini, and GPT4.1-Mini maintain nearly 100% valid outputs up to temperature 0.8, only GPT4.1-Mini shows resilience even at temperatures above 1.0.

In contrast, Llama3-8B drops from 80% at a temperature of 0.0 to 15% at 1.2, while Mixtral-8x7B maintains a success rate of around 40–50% until experiencing a similar drop at higher temperatures. Gemini-1.5-Flash consistently underperforms, producing valid outputs for only 2–6% of contracts across all temperatures. This model often fails to complete JSON generation due to improper tracking of nested structures, suggesting architectural limitations or a preference for faster responses.

4.3.1.2 Seed Structural Conformance and Uniqueness

Another key aspect of structured output evaluation is the proportion of seeds that successfully pass both syntactic and semantic validation. This validation process enforces

conformance to a predefined specification expressed in JSON Schema, ensuring that generated outputs adhere to the expected structural and semantic requirements. For this metric, we compute the number of seeds that do not conform to the seed structure model regarding type validation, nested structure consistency, and length constraints. As a result, a seed is deemed unusable if it has an invalid format and cannot be parsed as an initial fuzzing seed for SmartChat. Additionally, we compute the total number of duplicate seeds across all seed collections for each contract by calculating the SHA-256 hash values for each test case structure element.

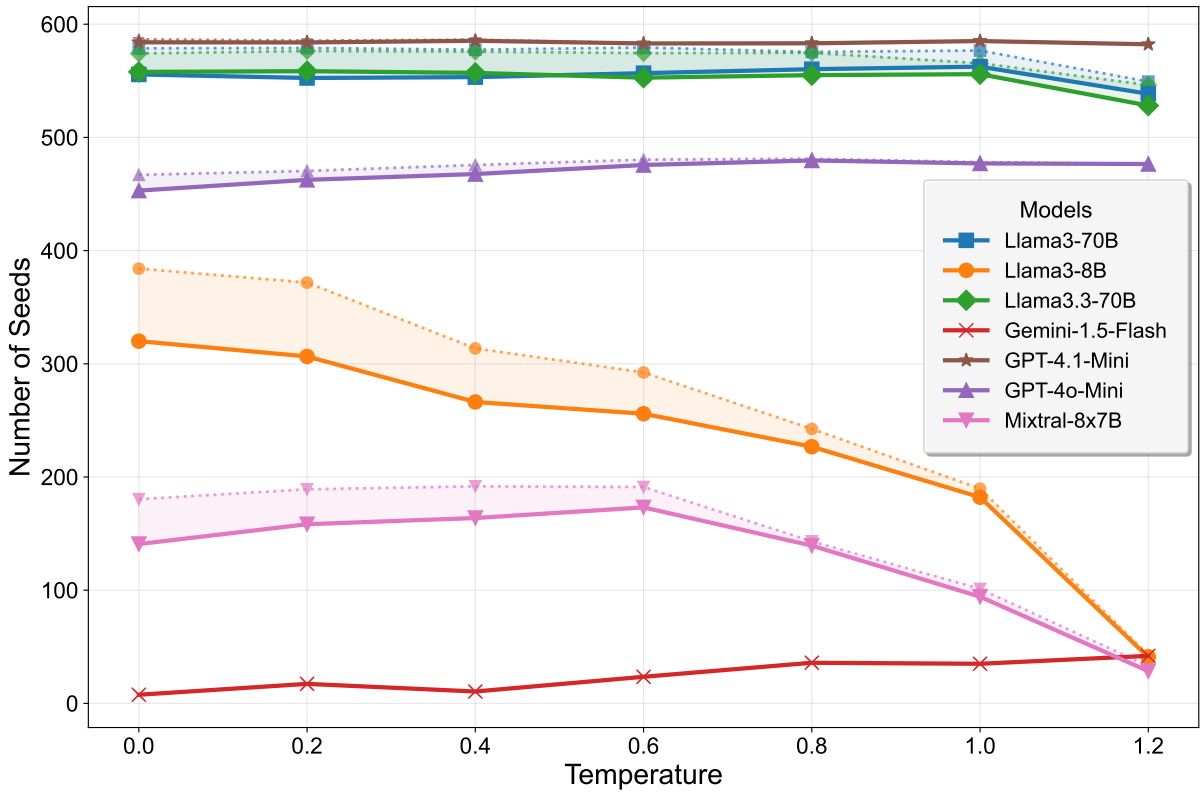

Figure 4.2: Seed generation quality by temperature and model. *Note: Solid lines show valid seeds, dotted lines represent total seeds, and shaded areas correspond to invalid and duplicate seeds.*

Figure 4.2 shows that the top-performing models (Llama3.3-70B, Llama3-70B, GPT4.1-Mini) consistently generate around 550-580 valid seeds across the different temperatures, with GPT4.1-Mini balancing the total seed generated with the minor average of duplicate and invalid seeds across all temperatures. GPT-4o-Mini also has almost all seeds being valid and with fewer duplicates, but generates fewer seeds than those requested by the instruction prompt. The figure confirms that higher temperatures (> 1.0) generally lead to quality degradation across most models. Llama3-8B and Mixtral-8x7B show signif-

icant degradation in valid seed generation as temperature increases. More specifically, Gemini-1.5-Flash shows the poorest performance with very few valid seeds (under 50).

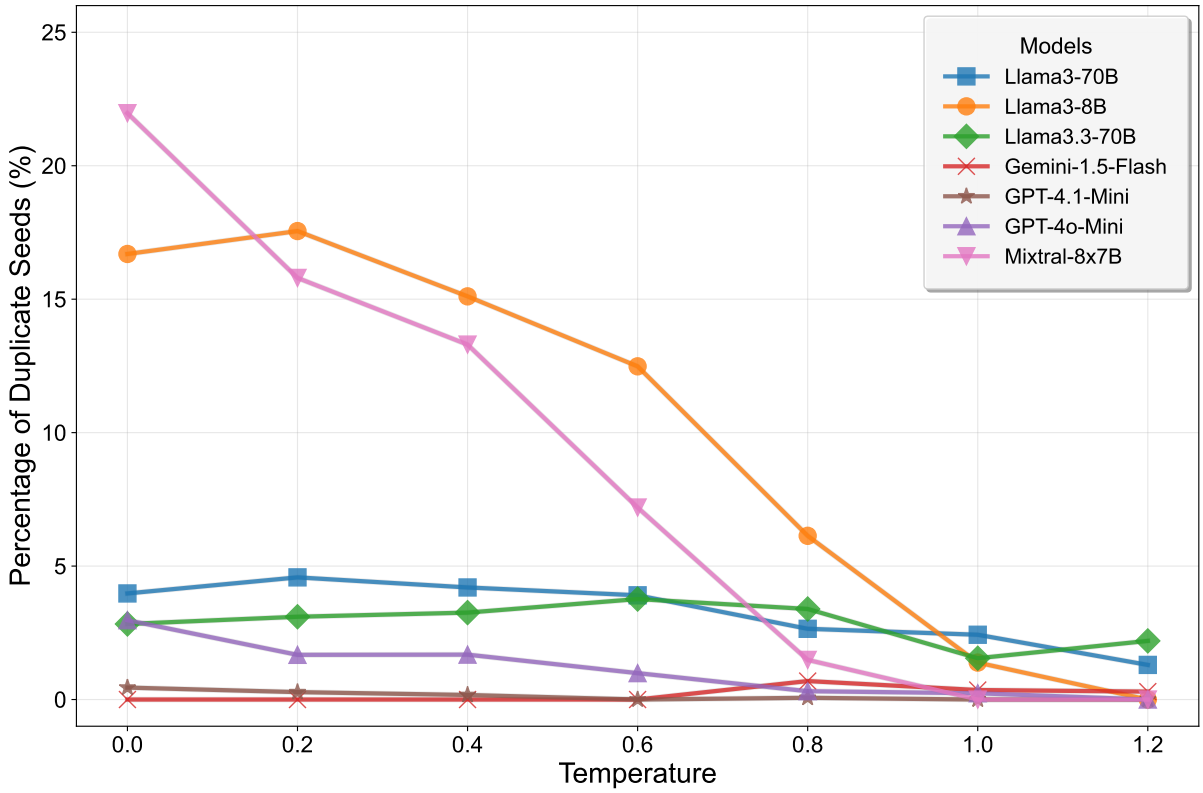

Figure 4.3: Duplicate seeds vs. temperature for different models.

A more nuanced representation can be seen in Figure 4.3, showcasing that higher temperatures generally reduce duplicate seeds, with most models reaching near-zero duplicates at $T = 1.2$. However, this comes at the cost of more invalid structured seeds, as illustrated in Figure 4.4, highlighting a trade-off between diversity and structural validity when tuning temperature.

Among the evaluated models, GPT4.1-Mini and GPT-4o-Mini exhibit almost no trade-off, with both duplicate and invalid seed rates remaining close to 0% across the entire temperature range, while Llama3.3-70B and Llama3-70B also maintain low duplicate rates (2–4%) and invalid rates (under 2%) across most temperatures, with only minor increases above $T = 1.0$. In contrast, Mixtral-8x7B and Llama3-8B show a clearer trade-off: higher temperatures reduce duplicates but raise invalids beyond $T = 0.8$, with the best balance at $T = 0.8$–1.0, though both generate few seeds overall. Gemini-1.5-Flash also yields near-zero duplicates and invalids but consistently produces under 5% valid structured output files (see Figure 4.1), making it impractical for seed generation.

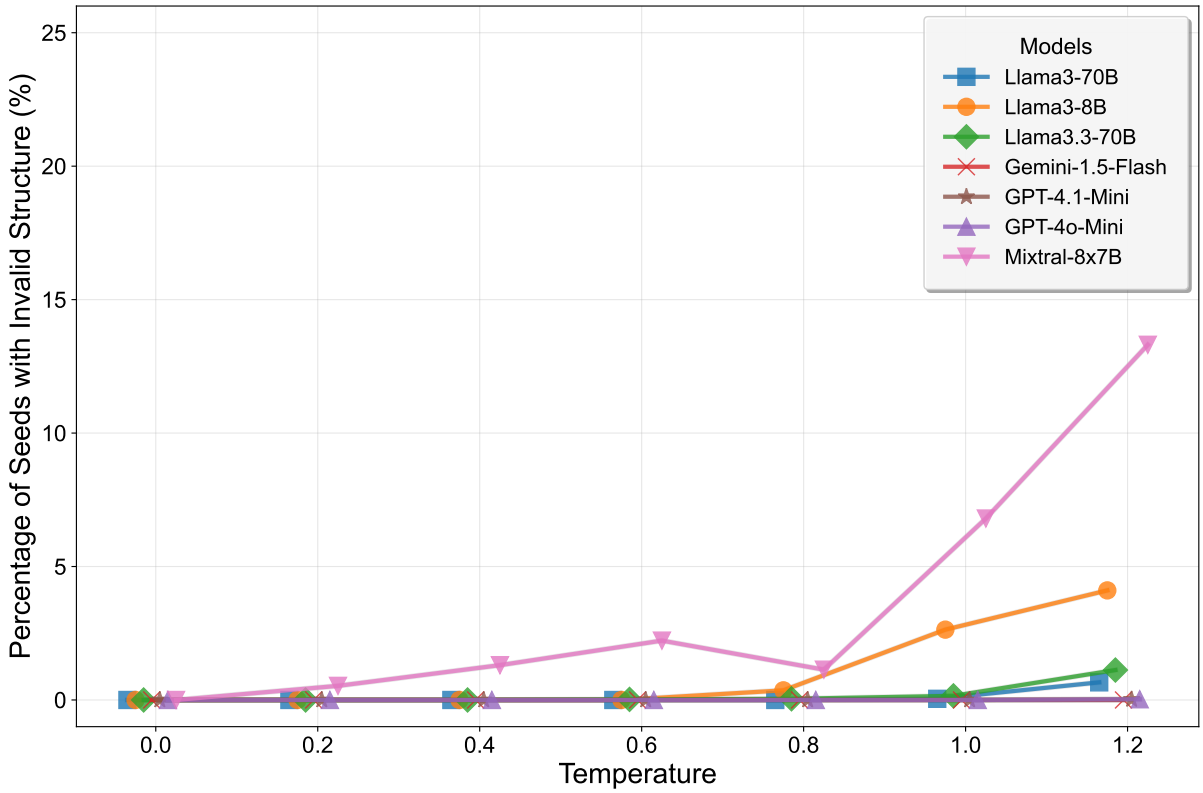

Figure 4.4: Structurally invalid seeds vs. temperature for different models.

Listing 4.1 shows an example of a seed with an invalid structure. Note between lines 17-23 that the `Value` field is missing for the second `Tx` object, as required by the prompt, which specifies that this field is not optional. Another example is found in line 12, where the `Value` field contains a negative value ("-1000000000000000000"), while the desired type is a `UInt256` value which must be non-negative. Additionally, line 22 shows an invalid `Blocknum` field containing spaces ("15 000 000"), which violates the numeric-only validation requirement. As a result, this seed has an invalid format and cannot be parsed as an initial fuzzing seed for SmartChat.

```
1  "TestCase": {
2    "DeployTx": {
3      "From": "SmartianAgent1",
4      "Value": "1000000000000000000",
5      "Function": "constructor",
6      "Params": [],
7      "Timestamp": "1640995200",
8      "Blocknum": "15000000"
9    },
10   "Txs": [{
11     "From": "SmartianAgent4",
12     "Value": "-1000000000000000000",
13     "Function": "deposit",
14     "Params": [],
```

```
    "Timestamp": "11000181",
    "Blocknum": "21000162"
  },{
    "From": "SmartianAgent3",
    "Function": "decimals",
    "Params": [],
    "Timestamp": "10000251",
    "Blocknum": "15 000 000",
  }]
}
```

Listing 4.1: A seed with invalid structure.

4.3.1.3 Seed Function Call Validity

When generating initial seeds, LLMs must adhere to the smart contract's ABI, which defines its external and public functions. However, models often misinterpret this interface, even when explicitly provided with a list of functions in the prompt. This leads to the loss of transaction elements in generated fuzzing seeds, since including transactions with invalid arguments or calls to non-existent function introduces noise and may hinder the effectiveness of the fuzzing process.

To assess this behavior, we evaluate how frequently the tested models produce invalid function calls as the sampling temperature varies. Specifically, we examine two aspects: (i) incorrect function signatures (e.g., wrong number or types of arguments), and (ii) hallucinated calls to functions that do not exist in the contract under test.

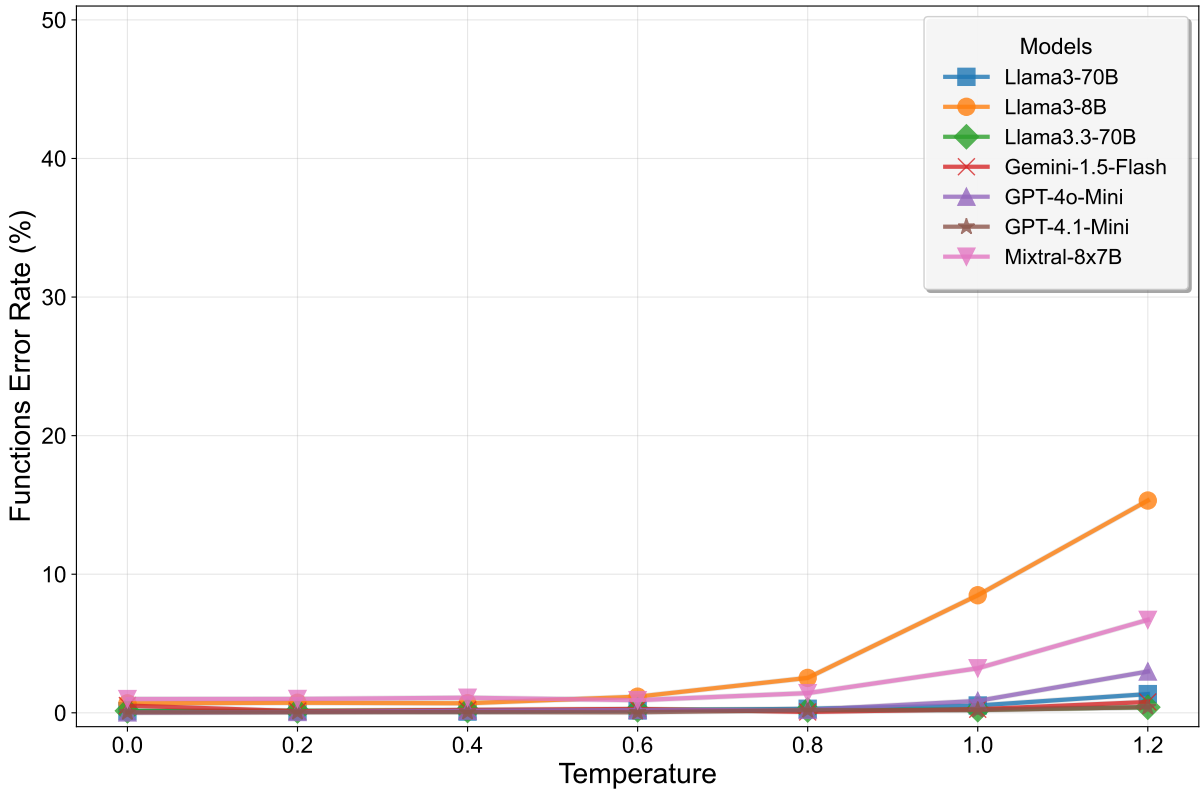

Figure 4.5: Invalid function calls vs. temperature.

Figures 4.5 and 4.6 present our findings on these aspects. Larger models, such as Llama3.3-70B and GPT4.1-Mini, consistently achieve the highest accuracy across all temperature settings, exhibiting minimal function call argument error rates (below 1.25%) and extremely low function call error rates (generally below 0.25% except at higher temperatures). GPT-4o-Mini also performs well, though with slightly higher error rates. In contrast, smaller models like Llama3-8B show substantially higher error rates, with function call argument errors ranging from 26.70% to 37.96%, which worsen as the temperature increases. Llama3-8B function call error rate rises dramatically from 0.68% at temperature 0.0 to 15.32% at temperature 1.2.

Llama3-70B maintains strong performance, with function call argument error rates consistently below 2.64%, though it does not match the accuracy of Llama3.3-70B. Mixtral-8x7B exhibits stable function call argument error rates between 11% and 14% across all temperatures. In contrast, Gemini's performance is more sensitive to temperature variation. Gemini-1.5-Flash demonstrates intermediate performance; however, its reduced number of valid output seed files—and consequently fewer function calls—limits its ability to follow instructions effectively and likely obscures its true error rate.

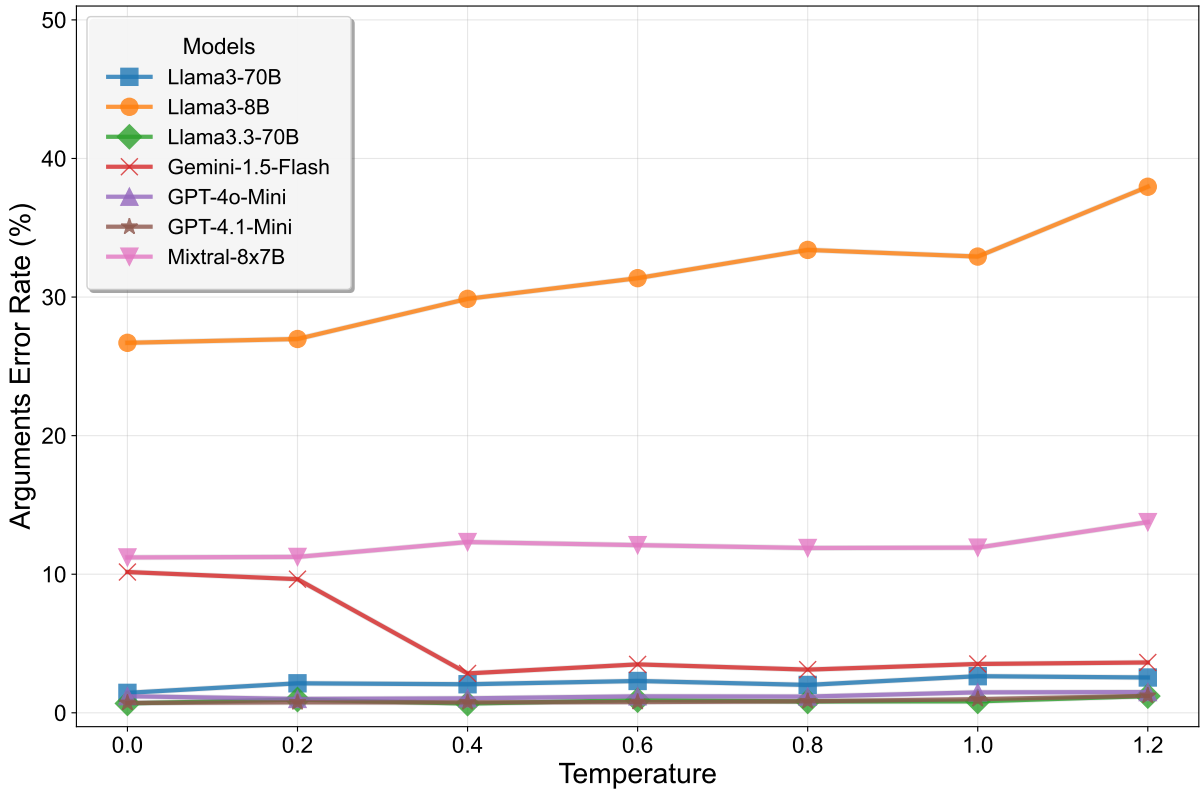

Figure 4.6: Invalid function call arguments vs. temperature.

Function call argument errors are significantly more prevalent than function call hallucinations across all models, indicating that LLMs sometimes struggle more with Solidity function call argument structure than function call recognition itself. These issues indicate, to some extent, fundamental shortcomings in type system comprehension, more prominent in smaller models but still present in larger models at higher sampling temperatures; however, while scaling model size improves Solidity function call structure, it does not fully eliminate these errors.

4.3.1.4 Seed Coverage

To evaluate the effectiveness of LLM-generated seeds, we measured the number of EVM instructions executed during the processing of these initial seeds, utilizing SmartChat's EVM replay feature. Figure 4.7 presents a heatmap illustrating instruction coverage of the models across various temperature settings. GPT4.1-Mini demonstrates the highest coverage overall (31.1% at temperature 0.0), followed by Llama3.3-70B and Llama3-70B.

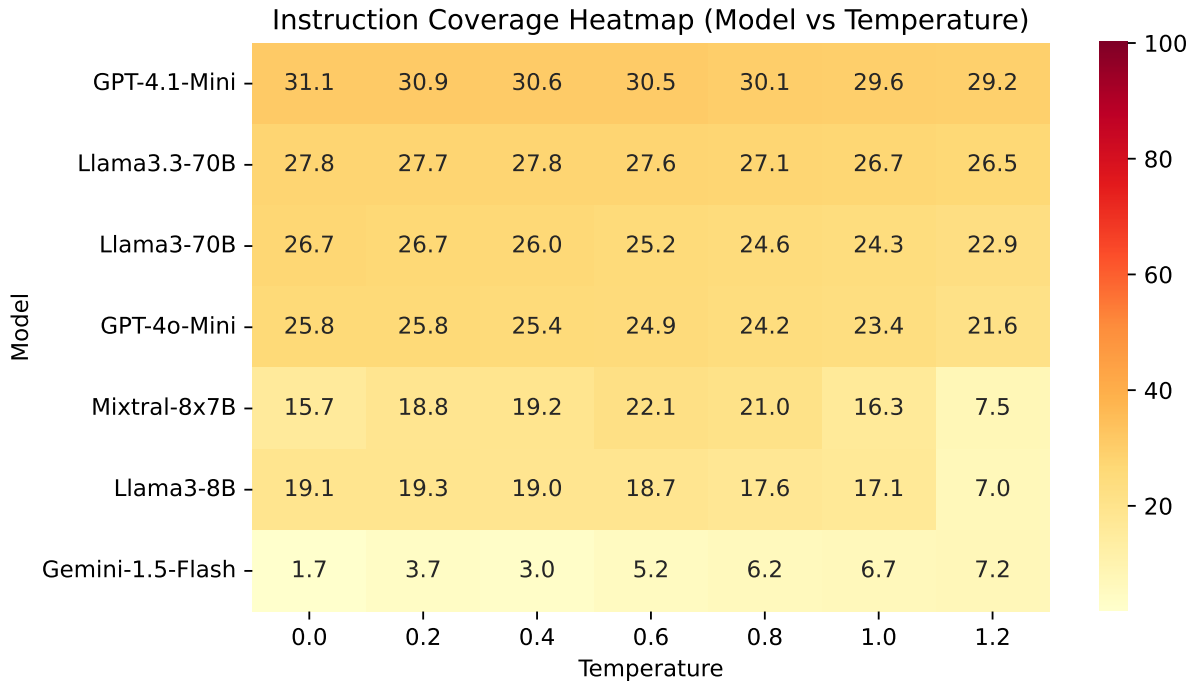

Figure 4.7: Average instruction coverage Heatmap (Model vs Temperature).

Most models show declining performance as temperature increases, particularly at the highest setting (1.2). In particular, Gemini-1.5-Flash shows the lowest coverage overall. The data suggests that larger models generally outperform smaller ones, with optimal instruction coverage typically occurring at lower temperature settings (0.0-0.4), though Mixtral-8x7B shows an exception with peak performance at temperature 0.6. This shows that the negative impact on coverage is more evident when models aim to be more random (creative) and, therefore, more likely to violate the structural rules.

Answer for Research Question (RQ1)

GPT4.1-Mini, followed by Llama3.3-70B, at low temperatures ($T < 1.0$), provides enough determinism for syntactic correctness while still allowing some variation in the generated seeds. Both models maintain high validity rates (>98%), generate a large number of seeds, and achieve the highest instruction coverage across all temperatures. In contrast, higher temperatures significantly degrade performance, particularly in smaller models, which exhibit substantial instability.

4.3.2 RQ2: Ablation and Temperature comparison

In this experiment, we evaluate the fuzzing effectiveness (i.e., vulnerability detection rate) of SmartChat under varying model temperature settings, with and without the vulnerability-guided prioritization algorithm. We compare these results against the original Smartian implementation, which supports separate fuzzing modes using either randomly generated or static data-flow seeds.

Here we leverage the two best-performing models identified when addressing RQ1: Llama3.3-70B and GPT4.1-Mini. SmartChat fuzzer is first executed with only code coverage prioritization enabled, using initial seeds generated by these models. Subsequently, the vulnerability-guided prioritization mechanism is enabled to assess its additional impact. The initial seeds used in this experiment are those generated in RQ1 using the ABI-focused few-shot prompt template.

We evaluate each configuration by measuring the number of vulnerabilities discovered over five one-hour campaigns executions, following the setup used by Choi et al. [4]. This process is repeated for each sampling temperature. Results are then compared against values obtained from Smartian's two fuzzing modes: one using randomly generated seeds and the other using full data-flow analysis. The Bench58 benchmark, previously introduced, is used for this evaluation. When executed with only randomly generated seeds, Smartian detects 44 integer vulnerabilities out of a possible 58. With the full data-flow configuration (combining static and dynamic analysis), Smartian achieves an average of 52.8 detected vulnerabilities across five runs. These values are used as baselines in the subsequent figures.

Figure 4.8 presents the SmartChat vulnerabilities detection results when using LLM-generated initial seeds. Both models outperform Smartian's random seed configuration, with GPT4.1-Mini detecting between 46 and 48 vulnerabilities, representing an improvement of 4.5% to 9.1%, and Llama3.3-70B detecting between 45.6 and 47.6, representing an improvement of 3.6% to 8.2%. An exception occurs at the higher temperature setting (1.2) for Llama3.3-70B, where performance drops below the random baseline. Nevertheless, both models still underperform compared to Smartian's full data-flow configuration, which detects an average of 52.8 vulnerabilities.

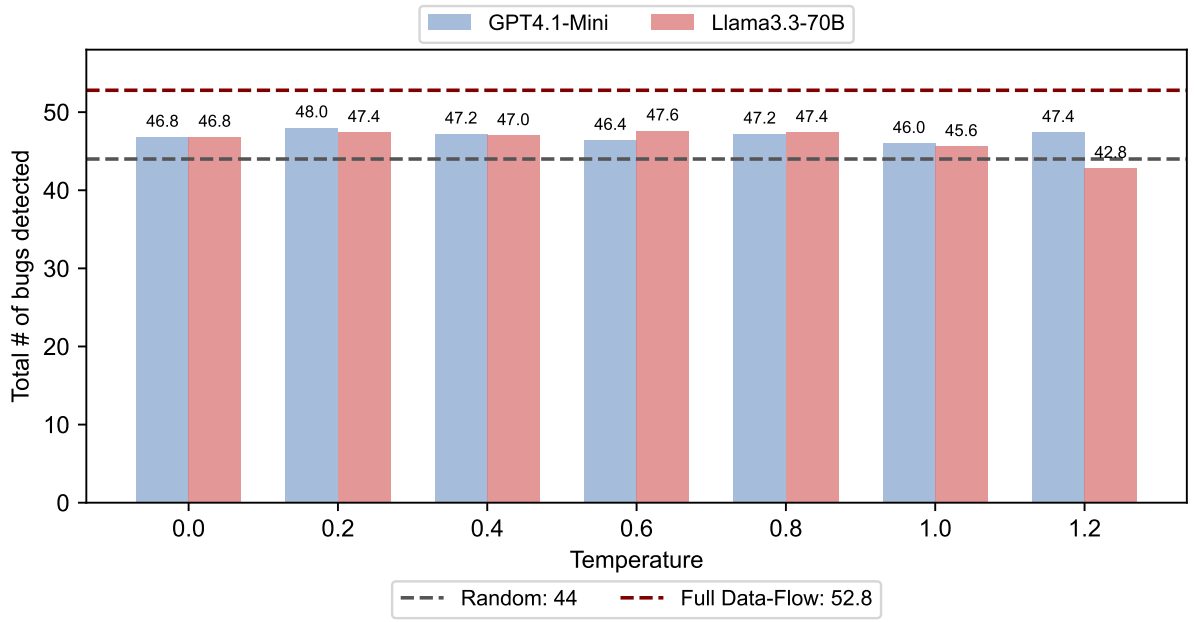

Figure 4.8: Comparison of models across sampling temperatures using LLM-generated initial seeds on Bench58.

After enabling vulnerability-guided prioritization, the performance improves significantly (see Figure 4.9). GPT4.1-Mini achieves detection of 52.0–53.6 bugs across different temperatures, with peak performance at temperatures 0.8 and 1.2 (53.6 bugs), representing a 1.52% improvement over the data-flow baseline. Similarly, Llama3.3-70B shows enhanced results, detecting 49.4–52.6 bugs. Its peak occurs at temperatures 0.0 and 0.6 (52.6 bugs), but it exhibits greater sensitivity to temperature, with performance dropping to 49.4 bugs at temperature 1.2. These results demonstrate that vulnerability-guided prioritization consistently boosts vulnerability detection across both models, though the magnitude of improvement and sensitivity to temperature vary by model.

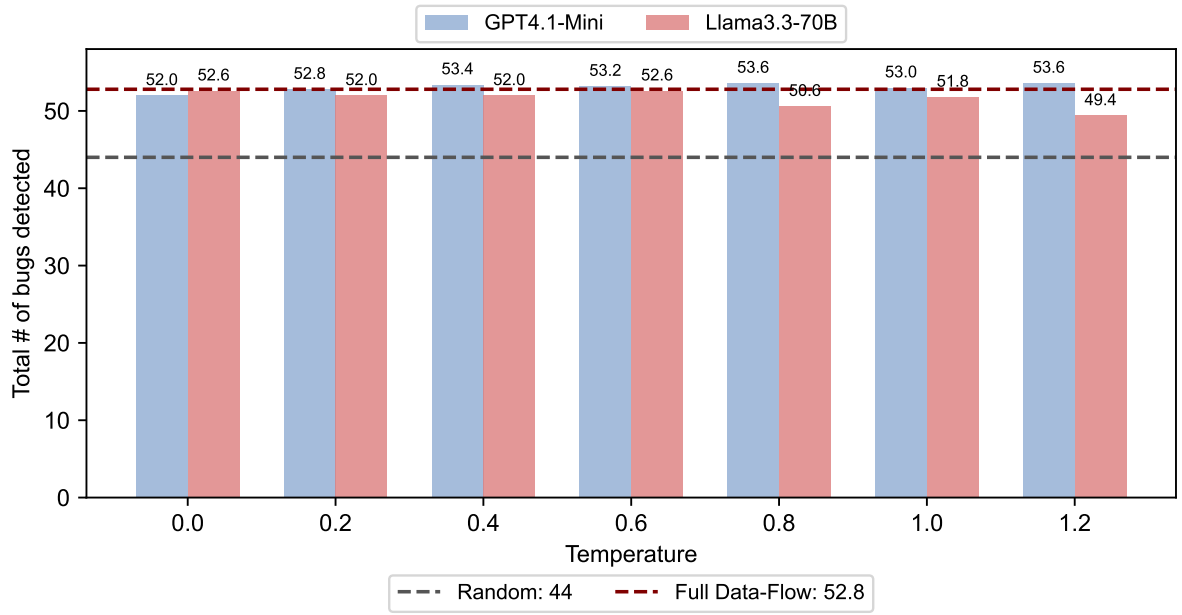

Figure 4.9: Comparison of models across sampling temperatures using LLM-generated initial seeds and vulnerability-guided prioritization on Bench58.

Following established practices in fuzzing evaluation [92, 91], we conducted a statistical analysis of the results using the Mann–Whitney U-test to compare the performance of each model–temperature configuration against Smartian results.

The null hypothesis states that there is no difference in detection performance between each model-temperature configuration and Smartian data-flow version. We fail to reject the null hypothesis for all GPT4.1-Mini configurations across all temperature settings, indicating no statistically detectable difference from Smartian performance. For Llama3.3-70B, we fail to reject the null hypothesis at temperatures 0.0–1.0, but reject it at temperature 1.2 ($p < 0.05$), providing evidence of significantly worse results at this highest temperature.

The rejection of the null hypothesis only at Llama3.3-70B's temperature setting of 1.2 suggests that higher temperature values may compromise the model's ability to generate effective fuzzing seeds. In contrast, our failure to reject the null hypothesis for GPT4.1-Mini across all temperatures indicates that its performance remains statistically stable regardless of temperature configuration, demonstrating greater robustness to this hyperparameter variation.

Answer for Research Question (RQ2)

SmartChat LLM-generated fuzzing seeds with vulnerability-guided prioritization substantially outperform random seeds and achieve performance statistically comparable to traditional data-flow analysis methods. While GPT4.1-Mini consistently matches the effectiveness of Smartian's full data-flow configuration across all temperature settings, Llama3.3-70B exhibits degradation at higher temperatures.

4.3.3 RQ3: Comparison with Smartian and ConFuzzius

In this research question, we present the results of our empirical investigation comparing SmartChat with Smartian and ConFuzzius, and evaluating the impact of different prompting strategies for generating initial seeds in SmartChat. Specifically, we assess the three prompting strategies described in Chapter 3:

- **(SmartChat ABI)** A few-shot prompting strategy that uses only the smart contract's Application Binary Interface (ABI) as input.
- **(SmartChat Code)** A single-turn, few-shot, chain-of-thought prompting strategy that incorporates the full smart contract source code.
- **(SmartChat CoT Code)** A multi-turn, few-shot, chain-of-thought prompting strategy that leverages iterative prompting rounds with the full smart contract source code.

The hypothesis guiding this experiment is that incorporating full source code into prompts leads to higher bug detection rates than using ABI-only prompts, and that multi-turn CoT prompting yields better performance than single-turn CoT prompting. As in the previous experiment, we executed each tool on every contract for one hour and repeated the process five times to obtain average results, this time using a benchmark with a substantially larger set of vulnerabilities (Bench78). For statistical significance, we employ the Mann–Whitney U-test with a threshold set at $\alpha = 0.05$.

The temperature used in this experiment is set to 0.4, as it provides the best trade-off between fuzzing effectiveness and the minimization of seed generation errors, whereas higher values—particularly above 1.0—result in notable degradation for Llama3.3-70B by reducing stability and increasing errors. Unlike previous experiments, which limited the input-output length to 8,192 tokens, this experiment increases the LLM token limit to 32,768 tokens to allow for the inclusion of full smart contract source code.

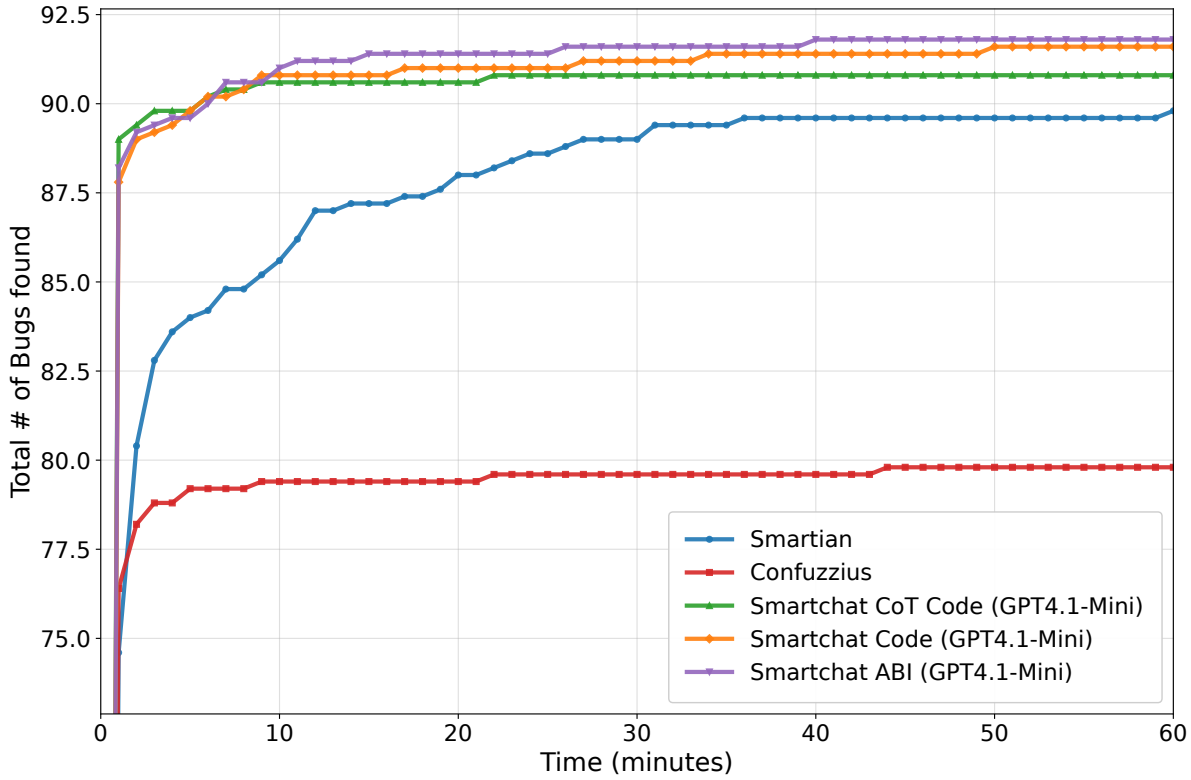

Figure 4.10: Vulnerability discovery over time for SmartChat GPT4.1-Mini variants, compared to Smartian and ConFuzzius across a 60-minute fuzzing campaign.

Figure 4.10 presents the results on vulnerability discovery performance across Smartian, ConFuzzius, and three SmartChat configurations using GPT4.1-Mini. All SmartChat GPT4.1-Mini variants consistently achieved superior initial performance, with 17.6–19.3% higher detection rates at the 1-minute stage compared to Smartian (87.8–89.0 vs. 74.6 vulnerabilities). They also outperformed ConFuzzius, detecting 14.9–16.49% more vulnerabilities within the first minute (compared to 76.4 bugs for ConFuzzius).

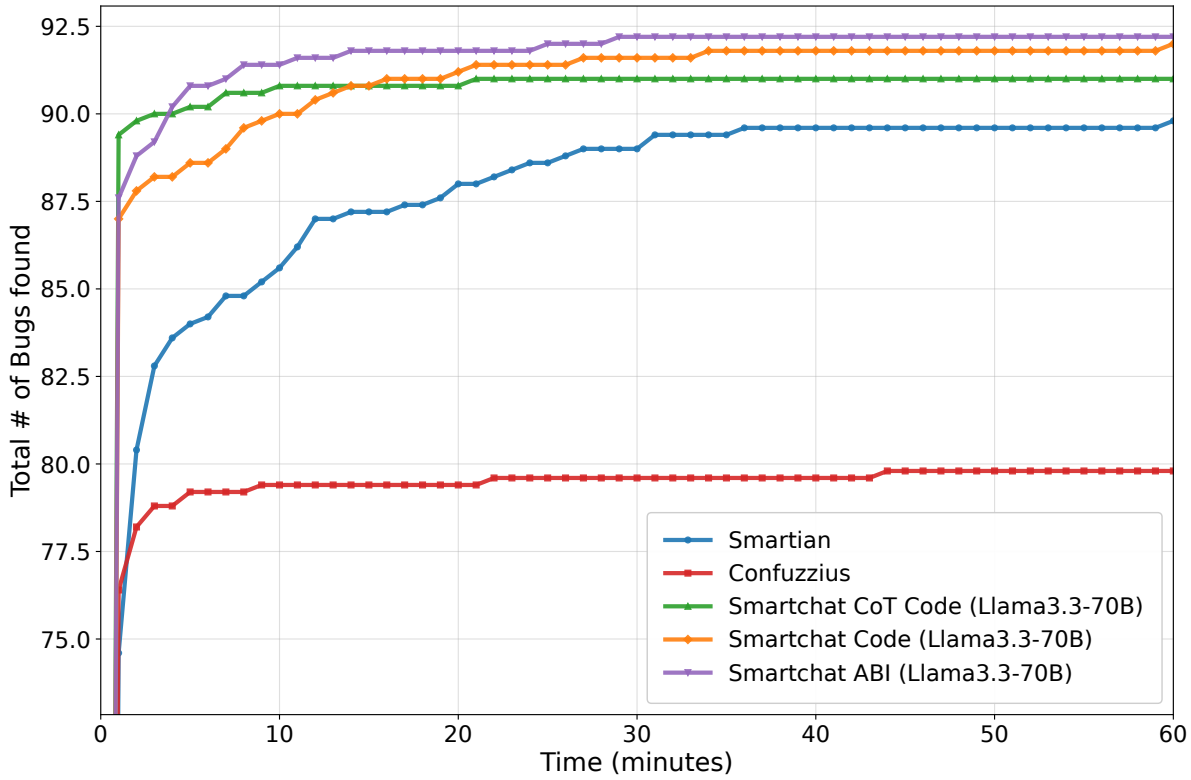

Figure 4.11: Vulnerability discovery over time for SmartChat Llama3.3-70B variants, compared to Smartian and ConFuzzius across a 60-minute fuzzing campaign.

By the end of the fuzzing period, all SmartChat GPT4.1-Mini based approaches surpassed Smartian by 0.8%–2.2% and ConFuzzius by 13.5%–15% in final bugs found. Statistical analysis confirmed that both the SmartChat ABI ($p < 0.05$) and SmartChat Code ($p < 0.05$) configurations achieved significant improvements over Smartian, while the SmartChat CoT Code variant shows no significant difference.

Regarding SmartChat Llama3.3-70B, illustrated in Figure 4.11, all variants demonstrated superior initial performance, achieving 16.6%–19.8% higher vulnerability detection rates at the 1-minute mark compared to Smartian, and outperformed ConFuzzius by 9.9%–17.0% in the same interval. By the end of the fuzzing period, all SmartChat-based approaches maintained a consistent lead over Smartian, with final counts exceeding Smartian by 1.33%–2.67%. Notably, the SmartChat ABI configuration achieved the highest final vulnerabilities count (92.2), representing a 15.6% improvement over ConFuzzius. Statistical analysis confirmed similar results as the SmartChat GPT4.1-Mini variants: both the ABI ($p < 0.05$) and Code ($p < 0.05$) configurations achieved significant improvements over Smartian, while the CoT Code variant showed no significant difference.

Examining instruction coverage in Figures 4.12 and 4.13, we observe a consistent trend across both Llama3.3-70B and GPT4.1-Mini SmartChat variants, which achieve higher coverage than Smartian within the first minute of fuzzing. Specifically, all SmartChat configurations reach between 84.6% and 85.4% coverage at the 1-minute mark, surpassing Smartian's 82.5% and significantly outperforming ConFuzzius at 65.6%. After 60 minutes, all SmartChat variants maintain a slight lead, with final coverage ranging from 85.66% to 85.82%, compared to 85.3% for Smartian and 69.5% for ConFuzzius. It is important to note, however, that as Inozemtseva et al. [99] highlight, the correlation between code coverage and bug-finding capability can be weak. This insight is particularly relevant in the context of smart contract fuzzing, where coverage feedback may be less effective due to the inherently stateful nature of the systems [69].

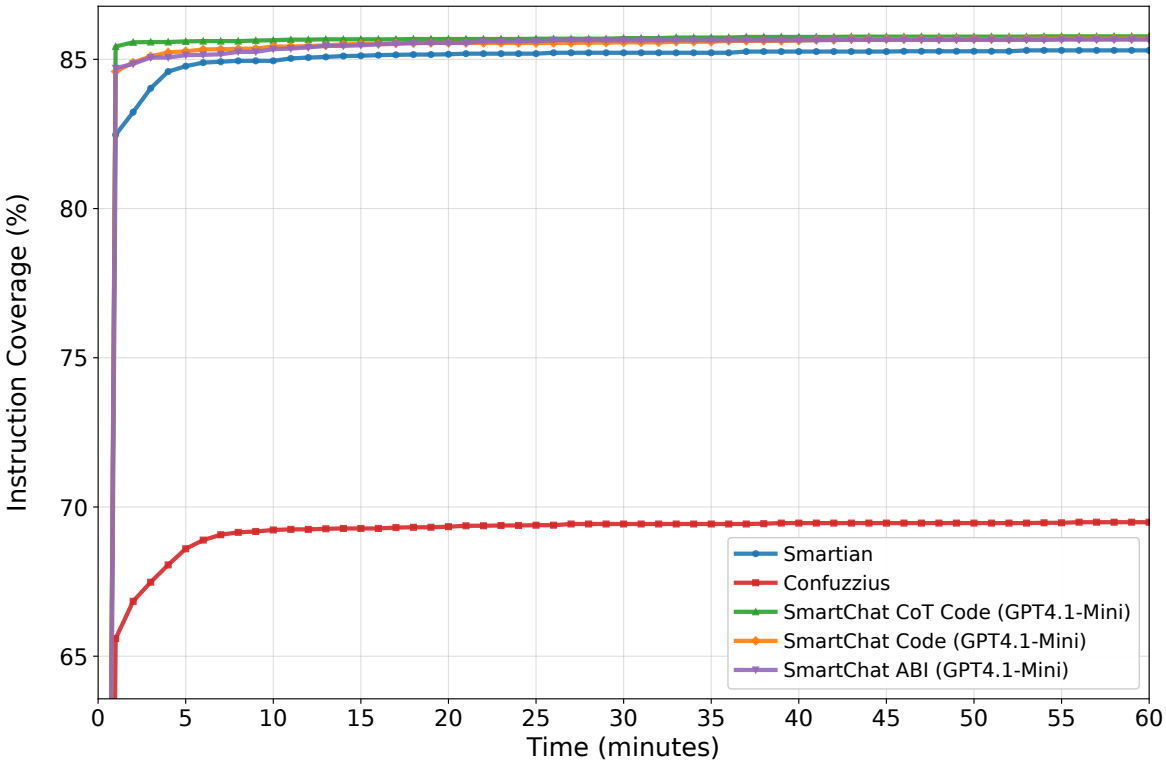

Figure 4.12: Instruction coverage over time for SmartChat GPT4.1-Mini variants, compared to Smartian and ConFuzzius across a 60-minute fuzzing campaign.

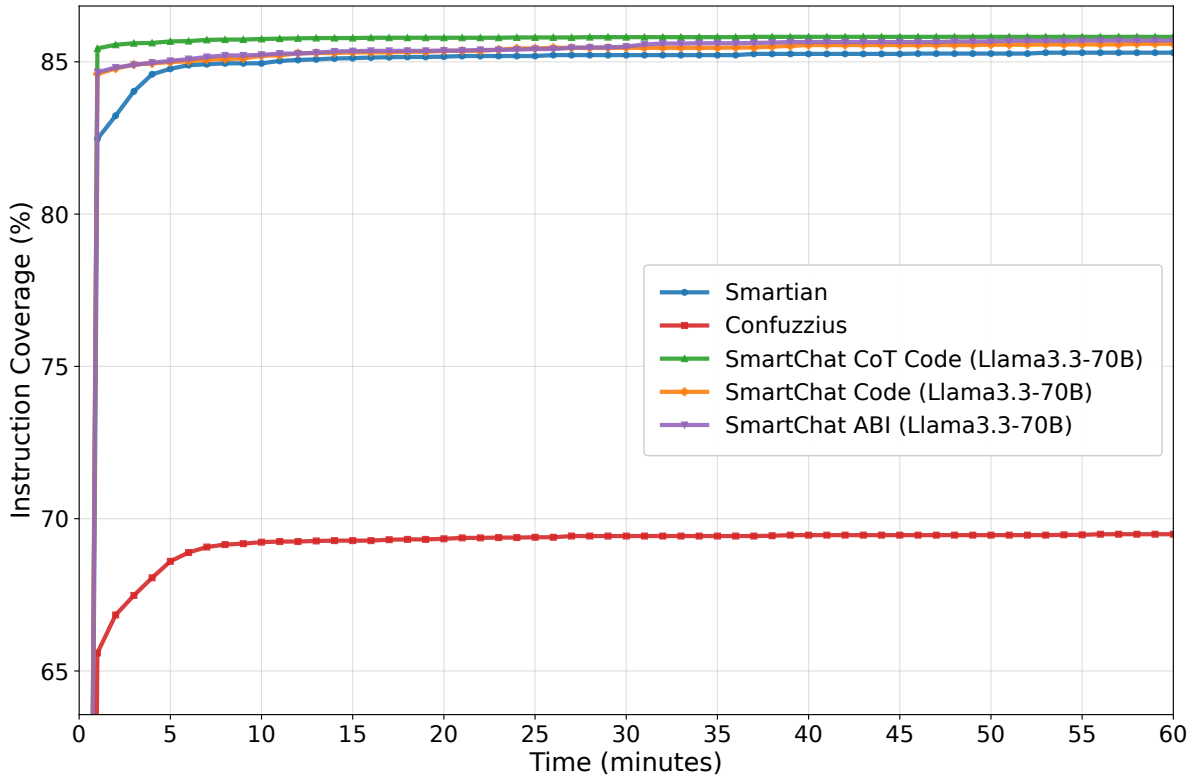

Figure 4.13: Instruction coverage over time for SmartChat Llama3.3-70B variants, compared to Smartian and ConFuzzius across a 60-minute fuzzing campaign.

Table 4.3 presents speedup results for vulnerability detection, showing the number of bugs found at multiple time intervals for each tool and SmartChat prompting strategy. SmartChat configurations consistently achieve substantial speedups over both Smartian and ConFuzzius, reducing the time required to reach high vulnerability detection coverage by up to 30× (Llama3.3-70B, CoT) and 20× (GPT4.1-Mini, CoT) compared to Smartian, and by 44× over ConFuzzius.

Tool	Time	Speedup vs Smartian / Confuzzius	@1m	@10m	@60m
Smartian	60m	–	74.6	85.6	89.8
Confuzzius	60m	–	76.4	79.4	79.8
SmartChat (Llama3.3-70B-based)					
+ Code	9m	6.67× / 44×	87.0	90.0	92.0
+ ABI	4m	15× / 44×	87.6	91.4	92.2
+ CoT	2m	30× / 44×	89.4	90.8	91.0
SmartChat (GPT-4.1-Mini-based)					
+ Code	5m	12× / 44×	87.8	90.8	91.6
+ ABI	6m	10× / 44×	88.2	91.0	91.8
+ CoT	3m	20× / 44×	89.0	90.6	90.6

Table 4.3: Speedup in vulnerability detection of SmartChat variants compared to other fuzzing tools.

Our results show that both the ABI and Code-based SmartChat prompting strategies significantly enhance bug discovery rates relative to traditional data-flow analysis, while the CoT Code variant, despite exhibiting a positive trend, did not achieve statistical significance under the experimental conditions. Nevertheless, contrary to our expectations, the inclusion of full contract source code in SmartChat prompts did not lead to improved fuzzing effectiveness compared to the ABI-only strategy. Even with a multi-turn CoT code variant, which incorporates separate code reasoning about potential vulnerabilities, performance improvements were minimal or non-existent compared to one-turn approaches.

Answer to Research Question (RQ3-a)

Incorporating full contract source code in SmartChat prompts resulted in only marginal improvements over ABI-only prompting, with no statistically significant advantage in bug discovery rates. These findings suggest that simply adding source code to prompts does not yield superior fuzzing outcomes.

Table 4.4 presents a breakdown of the detection rates for each vulnerability group by the SmartChat variants, compared against the baseline fuzzers Smartian and ConFuzzius.These detection rates are determined by bug oracles, which are predefined vulnerability detectors embedded in the execution engine that identify specific runtime behaviors indicating potential flaws. Thus, the detection rate reflects how often the fuzzer triggers

executions that satisfy these oracle conditions, rather than generic program crashes or exceptions.

Vulnerability	SmartChat	Smartian	ConFuzzius
Block Dependency (BD)	92.9% – 85.7%	78.6%	71.4%
Ether Leak (EL)	85.7% – 71.4%	57.1%	14.3%
Integer Bug (IB)	85.7%	85.7%	28.6%
Mishandled Exception (ME)	100.0%	100.0%	97.9%
Reentrancy (RE)	100.0%	100.0%	89.5%

Table 4.4: Detection Rates (%) of Vulnerability Groups by SmartChat Variants and Baseline Fuzzers.

The **Block Dependency** vulnerability group shows that SmartChat variants, employing prompts with Llama3.3-70B and GPT4.1-Mini models, achieved the highest detection rates, ranging from 92.9% to 85.7%, outperforming traditional fuzzers such as Smartian and ConFuzzius. This suggests that SmartChat 's approach of generating transaction sequences with contextual knowledge more effectively uncovers subtle contract state dependencies. A closer inspection of some missed cases revealed that Smartian's bug oracles (the same ones used by SmartChat) exhibit detection issues when checking block-related information within conditional branches, as noted by [5], making it infeasible to detect certain vulnerable contracts.

For the **EtherLeak** vulnerability class, SmartChat leads again, with variants reaching detection rates of up to 85.7%. Smartian performs moderately at 57.1%, while ConFuzzius struggles at 14.3%. Ether leaks often arise from specific logical conditions, such as failing to properly transfer funds or from rare contract states. These conditions can be subtle and require specific sequences of transactions or state setups to trigger. Nonetheless, the improvement achieved by SmartChat 's LLM-enhanced method is significant.

For the **IntegerBug** category, SmartChat's configurations display consistent detection rates of 85.7%, matching Smartian's highest performance, suggesting that such bugs may be more easily triggered by systematic input variation. ConFuzzius, with only 28.6%, exhibits less effective coverage of integer edge cases.

For the **Mishandled Exception** and **Reentrancy** vulnerability groups, SmartChat variants and Smartian achieved 100% detection rates, with ConFuzzius only marginally behind at 97.9% and 89.5%, respectively, indicating that mishandled exceptions and reentrancy issues may be relatively easier to trigger and detect, reflecting well-defined and common contract behaviors. Nonetheless, SmartChat 's method provided a major speedup in overall vulnerability detection, indicating that even classical and well-studied problems can be enhanced through LLM-based and vulnerability-guided approaches.

Answer to Research Question (RQ3-b)

All prompt variants of SmartChat outperform Smartian and ConFuzzius in terms of bug-finding capabilities and code coverage. In particular, our results provide evidence that SmartChat detects more bugs and covers more instructions in less time than the other tools considered in our evaluation.

4.4 Discussion

4.4.1 Key Findings and Implications

Our findings have direct implications for security auditors, as LLM-generated seeds can reduce fuzzing time by up to 44× and increase bug detection by as much as 15.6% compared to existing tools. This improvement can shorten overall audit durations and enable a more comprehensive security inspection within existing time constraints.

Moreover, by defining a clear seed format and identifying the ideal temperature settings, we demonstrate to tool builders that LLMs can generate high-quality, structured fuzzing seeds without requiring fine-tuning—facilitating easier integration of generative AI into existing smart contract fuzzing tools.

Another key observation from our results is that the effectiveness of LLM-generated initial seeds depends largely on both temperature and model size. Our experiments demonstrate that lower temperature values (below 1.0) generally produce more deterministic and structurally valid outputs, reducing hallucinations and syntax errors without compromising coverage or diversity. Moreover, model size significantly contributes to overall performance, with larger models (e.g., GPT4.1-Mini, Llama3.3-70B) consistently outperforming their smaller counterparts across all key metrics.

Finally, our findings reveal that seed generation with LLMs, when combined with vulnerability-guided prioritization, reaches a fuzzing effectiveness that is on par with—or even superior to—advanced static and dynamic data-flow-based approaches. Such results have direct implications for developers and researchers, requiring less engineering effort in terms of analysis and implementation, while demanding no domain expertise in the area. This highlights the practical viability of LLMs as intelligent seed generators in security testing pipelines, especially when fine-tuning is not feasible.

4.5 Threats to Validity

Our evaluation has some limitations. While we compare our method against Smartian [4] and ConFuzzius [67], direct comparison with other LLM-based initial seed generation tools

for smart contract fuzzing is limited due to the lack of publicly available implementations.

Our evaluation's reliance on two carefully selected benchmarks—Bench58 (58 real contracts from VeriSmart [93]) and Bench78 (78 contracts with 95 known bugs from SmartBugs [94] and ConFuzzius [67])—presents a potential threat to external validity due to the limited scope. However, we argue that Bench58 provides sufficient diversity for general Solidity test case generation, as it is based on real-world contracts, and Bench78 is composed of commonly used datasets [5, 72, 70, 100], which are widely recognized in the fuzzing community as a solid experimental foundation. Nevertheless, other benchmarks may yield different results.

One might question the stability of LLMs (e.g., measured by Total Agreement Rate), but since our work targets seed generation for fuzzing, output variability is not a threat—diversity is in fact desirable.

Another threat arises from the fact that these benchmarks predominantly contain contracts written in Solidity versions prior to 0.8. This reflects both the constraints of existing public datasets and the compatibility of our baseline fuzzer (Smartian). While integer overflow and underflow are largely mitigated in Solidity version 0.8.0 and subsequent releases through built-in arithmetic checks, other vulnerabilities present in our benchmarks, such as block dependency, mishandled exceptions, reentrancy, and Ether leaks, remain possible in modern versions. Moreover, such vulnerabilities continue to be observed in newly deployed contracts, making our evaluation applicable beyond legacy code. Given the cutoff dates of the LLMs used, their training data should also include contracts written in newer Solidity versions. Future work should incorporate updated datasets, including synthetic smart contracts with injected vulnerabilities, to assess applicability to newer Solidity versions and reduce data contamination.

Regarding computational costs, our approach, which utilizes API-based LLMs, presents a monetary impact. However, for open-weight models (Llama3-70B, Llama3.3-70B, Llama3-8B, and Mixtral-8x7B), this limitation can be mitigated by deploying the models locally on organizational GPU clusters, reducing the cost to computational resources only.

Our findings may not generalize beyond the specific prompt engineering approaches used in this study, as the effectiveness of the approach is closely tied to the quality of the prompts. Different prompts might yield varying results, and the optimal prompting strategies may evolve as LLM capabilities advance.

To mitigate these threats, we have made our full implementation and dataset publicly available, enabling the research community to evaluate SmartChat on additional benchmarks and with emerging LLMs.

All data, scripts, and source code can be accessed for further research at `https://github.com/PAMunb/SmartChat-Artifact` and `https://github.com/PAMunb/SmartChat`.

Chapter 5

Conclusion and Future Work

In this work, we present an extensive evaluation of Large Language Models (LLMs) for initial fuzzing seed generation and introduce SmartChat, a novel framework that leverages LLM-generated seed corpora and prioritization techniques for smart contract security testing. By evaluating seven pre-trained LLMs across different sampling temperatures and prompting strategies, we demonstrate that LLMs can synthesize structurally and semantically valid transaction sequences without the need for fine-tuning.

Our key findings reveal that LLMs, particularly GPT4.1-Mini and Llama3.3-70B, can effectively generate structurally valid transaction sequences at low temperatures without requiring task-specific fine-tuning. Furthermore, the vulnerability-guided prioritization mechanism enhances fuzzing effectiveness, allowing SmartChat to outperform traditional approaches by detecting up to 15.6% more vulnerabilities than state-of-the-art fuzzers. Most notably, SmartChat achieves speedups in vulnerability detection, reducing time-to-discovery by 6.67x to 44x compared to Smartian and ConFuzzius.

5.1 Contributions

The main contributions of our work are as follows:

- A generic format for input seeds that simulates realistic smart contract execution contexts and integrates seamlessly with existing fuzzing tools.
- A novel LLM-driven seed generation mechanism that synthesizes structured transaction sequences, supports multiple prompting strategies, and remains easily extensible.
- A new fuzzing strategy that prioritizes vulnerability-oriented seeds.

- A systematic evaluation of seven pre-trained LLMs across a range of sampling temperatures and prompt styles, helping reduce the overhead of model selection and temperature tuning.
- A demonstration of high-quality seed generation without fine-tuning, eliminating the need for resource-intensive training or large domain-specific datasets.
- An open-source implementation and reproducible benchmark.

5.2 Limitations and Future Work

Future work includes extending our methodology to broader benchmarks, which will require constructing experiments involving a large number of smart contracts with a more diverse set of vulnerability classes, including contracts written in more recent versions of the Solidity language.

It will also be important to consider the potential impact of data contamination on the results. Future work will include carefully designed experiments to assess how much pre-training exposure to Solidity code or vulnerability datasets influences LLM effectiveness in seed generation. To mitigate this concern, controlled benchmarks will be constructed to explicitly measure robustness against contamination.

We also aim to investigate the use of information extracted from static analysis tools, such as control flow graphs and function call dependencies, to generate prompts that target specific execution paths. This may increase the likelihood of generating seeds that trigger particular vulnerability patterns.

Given that including full contract source code in prompts provided no gains over ABI-only strategies, we plan to investigate LLM-based evaluations of contract subsets—specifically targeting portions of the public interface or individual functions—to reduce context noise and enable the decomposition of complex contracts into smaller, more targeted fuzzable elements.

Another future direction is to refine vulnerability reasoning through advanced prompting techniques and explore reasoning-focused LLMs to develop deeper source code understanding strategies.

In addition, we plan to broaden the comparison beyond fuzzing by incorporating alternative vulnerability detection techniques, such as symbolic execution, statis analysis, and hybrid testing approaches. Contrasting these methodologies with LLM-guided fuzzing will provide better insights into their respective strengths and limitations.

References

[1] Panda, Sandeep Kumar, Sanjay Kumar Das, and Jyotir Moy Chatterjee (editors): *Blockchain Technology: Applications and Challenges*. Springer, Singapore, 2021, ISBN 9789813345147. x, 7, 8

[2] Ethereum: *Ethereum virtual machine (evm) | ethereum documentation*. `https://ethereum.org/en/developers/docs/evm/`. x, 11

[3] He, Zheyuan, Zihao Li, and Sen Yang: *Large language models for blockchain security: A systematic literature review*. arXiv preprint arXiv:2403.14280, 2024. x, 17

[4] Choi, Jaeseung, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha: *Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses*. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 227–239. IEEE, 2021. x, 2, 22, 25, 26, 49, 50, 60, 70

[5] Wu, Shuohan, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou: *Are we there yet? unraveling the state-of-the-art smart contract fuzzers*. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. 1, 2, 4, 13, 24, 25, 26, 32, 36, 50, 69, 71

[6] Fan, Zhiyu, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan: *Automated repair of programs from large language models*. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023. 2

[7] Sobania, Dominik, Martin Briesch, Carol Hanna, and Justyna Petke: *An analysis of the automatic bug fixing performance of chatgpt*. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30. IEEE, 2023. 2

[8] Yang, Chen, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang: *Enhancing llm-based test generation for hard-to-cover branches via program analysis*, 2024. `https://arxiv.org/abs/2404.04966`. 2

[9] Lemieux, Caroline, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen: *Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models*. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023. 2

[10] Souza, Débora, Rohit Gheyi, Lucas Albuquerque, Gustavo Soares, and Márcio Ribeiro: *Code generation with small language models: A deep evaluation on codeforces*, 2025. https://arxiv.org/abs/2504.07343. 2

[11] Ni, Ansong, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, *et al.*: *L2ceval: Evaluating language-to-code generation capabilities of large language models*. Transactions of the Association for Computational Linguistics, 12:1311–1329, 2024. 2

[12] Xia, Chunqiu Steven, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang: *Fuzz4all: Universal fuzzing with large language models*. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. 2, 29

[13] Meng, Ruijie, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury: *Large language model guided protocol fuzzing*. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024. 2, 29

[14] Deng, Yinlin, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang: *Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models*. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023. 2, 29

[15] Herrera, Adrian, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking: *Seed selection for successful fuzzing*. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pages 230–243, 2021. 2

[16] He, Jingxuan, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev: *Learning to fuzz from symbolic execution with application to smart contracts*. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 531–548, New York, NY, USA, 2019. ACM, ISBN 978-1-4503-6747-9. http://doi.acm.org/10.1145/3319535.3363230. 4, 22, 24, 33

[17] Su, Jianzhong, Hong Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo: *Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing*. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery, ISBN 9781450394758. https://doi.org/10.1145/3551349.3560429. 4, 22, 24, 49

[18] Liu, Zhenguang, Peng Qian, Jiaxu Yang, Lingfeng Liu, Xiaojun Xu, Qinming He, and Xiaosong Zhang: *Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting*. IEEE Transactions on Information Forensics and Security, 18:1237–1251, 2023. 4, 22, 24, 49

[19] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system*, 2008. http://www.bitcoin.org/bitcoin.pdf. 6

[20] Wood, Gavin *et al.*: *Ethereum: A secure decentralised generalised transaction ledger*, 2014. 8, 11

[21] Foundation, Ethereum: *Ethereum request for comments 20: Token standard*. `https://ethereum.org/en/developers/docs/standards/tokens/erc-20/`, 2015. Accessed: 2025-06-05. 8

[22] Zhou, Teng, Kui Liu, Li Li, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé: *Smartgift: Learning to generate practical inputs for testing smart contracts*. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 23–34. IEEE, 2021. 8, 10

[23] Rodler, Michael, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi: *Ef/cf: High performance smart contract fuzzing for exploit generation*. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023. 8, 22, 25

[24] *Ethereum data in bigquery*. `https://console.cloud.google.com/bigquery?ws=!1m4!1m3!3m2!1sbigquery-public-data!2scrypto_ethereum`. Accessed: 2025-06-04. 8

[25] Caversaccio, Pascal: *Smart contract deployment statistics*. `https://dune.com/pcaversaccio/smart-contract-deployment-statistics`, 2024. Accessed: 2025-06-04. 9

[26] Szabo, Nick: *Formalizing and securing relationships on public networks*. First monday, 1997. 9

[27] DeFi Llama Language: *Languages*, 2024. `https://defillama.com/languages`, Accessed: 2025-06-26. 9

[28] Solidity Team: *Solidity programming language*, 2024. `https://soliditylang.org/`, Accessed: 2025-06-04. 9

[29] Chen, Ting, Zihao Li, Yufei Zhang, Xiapu Luo, Ting Wang, Teng Hu, Xiuzhuo Xiao, Dong Wang, Jin Huang, and Xiaosong Zhang: *A large-scale empirical study on control flow identification of smart contracts*. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019. 13, 22

[30] Manes, Valentin JM, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo: *Fuzzing: Art, science, and engineering*. arXiv preprint arXiv:1812.00140, 2018. 13, 14

[31] Google: *Oss-fuzz: Fuzzing the planet*, 2023. `https://github.com/google/oss-fuzz`, Accessed: 2025-06-06. 13

[32] Miller, Barton P., Louis Fredriksen, and Bryan So: *An empirical study of the reliability of unix utilities*. Communications of the ACM, 33(12):32–44, December 1990. 13

[33] Böhme, Marcel, Van Thuan Pham, Manh Dung Nguyen, and Abhik Roychoudhury: *Directed greybox fuzzing.* In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017. 13, 15, 16

[34] Wang, Pengfei, Xu Zhou, Tai Yue, Peihong Lin, Yingying Liu, and Kai Lu: *The progress, challenges, and perspectives of directed greybox fuzzing.* Software Testing, Verification and Reliability, 34(2), 2023, ISSN 1099-1689. `http://dx.doi.org/10.1002/stvr.1869`. 13

[35] Manes, Valentin J. M., HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo: *The art, science, and engineering of fuzzing: A survey*, 2019. `https://arxiv.org/abs/1812.00140`. 14

[36] Aschermann, Cornelius, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad Reza Sadeghi, and Daniel Teuchert: *Nautilus: Fishing for deep bugs with grammars.* In *NDSS*, 2019. 14

[37] Chen, Yongheng, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee: *One engine to fuzz'em all: Generic language processor testing with semantic validation.* In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 642–658. IEEE, 2021. 14

[38] Sam Hocevar: *Zzuf: A Transparent Application Input Fuzzer.* `http://caca.zoy.org/wiki/zzuf`, 2006. 14

[39] Helin, Aki: *Radamsa.* `https://gitlab.com/akihe/radamsa`, 2006. A mutation-based fuzzer. 14

[40] Zalewski, Michal: *American fuzzy lop (afl).* `http://lcamtuf.coredump.cx/afl/`, 2014. A coverage-guided fuzzer. 15, 24

[41] Contributors, AFL++: *Afl++.* `https://aflplus.plus/`, 2020. An improved version of AFL with additional features and enhancements. 15

[42] Honggfuzz: *Honggfuzz.* `https://github.com/google/honggfuzz`, 2018. A security fuzzer with feedback-driven fuzzing capabilities. 15

[43] Contributors libAFL: *libafl.* `https://github.com/AFLplusplus/libafl`, 2020. A library for fuzzing that provides core fuzzing functionalities. 15

[44] Kim, Tae Eun, Jaeseung Choi, Seongjae Im, Kihong Heo, and Sang Kil Cha: *Evaluating directed fuzzers: Are we heading in the right direction?* Proceedings of the ACM on Software Engineering, 1(FSE):316–337, 2024. 16

[45] Stephens, Nick, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna: *Driller: Augmenting fuzzing through selective symbolic execution.* In *NDSS*, volume 16, pages 1–16, 2016. 16

[46] Huang, Heqing, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang: *Beacon: Directed grey-box fuzzing with provable path pruning.* In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022. 16

[47] Österlund, Sebastian, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida: {*ParmeSan*}*: Sanitizer-guided greybox fuzzing.* In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306, 2020. 16

[48] Distributed, Hacking: *Analysis of the dao exploit*, 2016. `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`, Accessed: 2025-06-26. 16

[49] DeFi Llama Hacks: *Hacks*, 2024. `https://defillama.com/hacks`, Accessed: 2025-06-26. 16

[50] Comae: *The 280m ethereum bug*, 2020. `https://medium.com/comae/the-280m-ethereums-bug-f28e5de43513`, Accessed: 2025-06-26. 16

[51] Kaiko: *The nomad bridge attack and the mystery of uniswap v2*, 2022. `https://blog.kaiko.com/the-nomad-bridge-attack-and-the-mystery-of-uniswap-v2-5f888355218d`, Accessed: 2025-06-26. 16

[52] Immunebytes: *Rari capital re-entrancy attack april 2022: Detailed analysis*, 2022. `https://www.immunebytes.com/blog/rari-capital-re-entrancy-attack-april-2022-detailed-analysis/`, Accessed: 2025-06-26. 16

[53] Atzei, Nicola, Massimo Bartoletti, and Tiziana Cimoli: *A survey of attacks on ethereum smart contracts (sok).* In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*, pages 164–186. Springer, 2017. 16, 17

[54] (DASP), Decentralized Application Security Project: *Dasp10: The top 10 smart contract vulnerabilities*, 2020. `https://dasp.org/dasp-top-10`, Accessed: 2025-06-27. 16

[55] Registry, SWC: *Swc registry: Smart contract weakness classification registry*, 2023. `https://swcregistry.io`, Accessed: 2025-06-27. 16

[56] Zhou, Liyi, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais: *Sok: Decentralized finance (defi) attacks.* In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461. IEEE, 2023. 17

[57] Nguyen, Tai D, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh: *sfuzz: An efficient adaptive fuzzer for solidity smart contracts.* In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020. 18, 22, 24, 33, 37

[58] Ashraf, Imran, Xiaoxue Ma, Bo Jiang, and Wing Kwong Chan: *Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities*. IEEE Access, 8:99552–99564, 2020. 18

[59] OpenZeppelin: *Safemath*. `https://docs.openzeppelin.com/contracts/4.x/api/utils#SafeMath`, 2024. Accessed: 2025-06-27. 19

[60] Nikolic, Ivica, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor: *Finding the greedy, prodigal, and suicidal contracts at scale*, 2018. `https://arxiv.org/abs/1802.06038`. 20

[61] Wang, Haijun, Ye Liu, Yi Li, Shang Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu: *Oracle-supported dynamic exploit generation for smart contracts*. IEEE Transactions on Dependable and Secure Computing, 19(3):1795–1809, 2020. 22

[62] Jiang, Bo, Ye Liu, and Wing Kwong Chan: *Contractfuzzer: Fuzzing smart contracts for vulnerability detection*. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 259–269, 2018. 22, 23

[63] Liu, Chao, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe: *Reguard: finding reentrancy bugs in smart contracts*. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 65–68, 2018. 22

[64] Wang, Haijun, Yi Li, Shang Wei Lin, Lei Ma, and Yang Liu: *Vultron: catching vulnerable smart contracts once and for all*. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 1–4. IEEE, 2019. 22, 24

[65] Wüstholz, Valentin and Maria Christakis: *Harvey: A greybox fuzzer for smart contracts*. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1398–1409, 2020. 22, 24

[66] Grieco, Gustavo, Will Song, Artur Cygan, Josselin Feist, and Alex Groce: *Echidna: effective, usable, and fast fuzzing for smart contracts*. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 557–560, 2020. 22, 24, 37

[67] Torres, Christof Ferreira, Antonio Ken Iannillo, Arthur Gervais, and Radu State: *Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts*. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 103–119. IEEE, 2021. 22, 25, 33, 37, 49, 50, 70, 71

[68] Xue, Yinxing, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao: *xfuzz: Machine learning guided cross-contract fuzzing*. IEEE Transactions on Dependable and Secure Computing, 21(2):515–529, 2022. 22, 24

[69] Shou, Chaofan, Shangyin Tan, and Koushik Sen: *Ityfuzz: Snapshot-based fuzzer for smart contract*. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 322–333, 2023. 22, 25, 32, 66

[70] Qian, Peng, Hanjie Wu, Zeren Du, Turan Vural, Dazhong Rong, Zheng Cao, Lun Zhang, Yanbin Wang, Jianhai Chen, and Qinming He: *Mufuzz: Sequence-aware mutation and seed mask guidance for blockchain smart contract fuzzing.* In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1972–1985. IEEE, 2024. 22, 24, 71

[71] Yang, Huiwen, Xiguo Gu, Xiang Chen, Liwei Zheng, and Zhanqi Cui: *Crossfuzz: Cross-contract fuzzing for smart contract vulnerability detection.* Science of Computer Programming, 234:103076, 2024. 22, 25, 49

[72] Ye, Mingxi, Yuhong Nan, Hong Ning Dai, Shuo Yang, Xiapu Luo, and Zibin Zheng: *Funfuzz: A function-oriented fuzzer for smart contract vulnerability detection with high effectiveness and efficiency.* ACM Transactions on Software Engineering and Methodology, 2025. 22, 24, 71

[73] Medeiros, Ismael, Fausto Carvalho, Alexandre Ferreira, Rodrigo Bonifácio, and Fabiano Cavalcanti Fernandes: *Dogefuzz: A simple yet efficient grey-box fuzzer for ethereum smart contracts.* CoRR, abs/2409.01788, 2024. `https://doi.org/10.48550/arXiv.2409.01788`. 22, 25, 26, 37

[74] Shou, Chaofan, Jing Liu, Doudou Lu, and Koushik Sen: *Llm4fuzz: Guided fuzzing of smart contracts with large language models.* arXiv preprint arXiv:2401.11108, 2024. 26, 29, 37

[75] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio: *Generative adversarial networks.* Communications of the ACM, 63(11):139–144, 2020. 26

[76] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin: *Attention is all you need.* CoRR, abs/1706.03762, 2017. `http://arxiv.org/abs/1706.03762`. 26

[77] Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei: *Language models are few-shot learners*, 2020. `https://arxiv.org/abs/2005.14165`. 27

[78] Raschka, Sebastian: *Build a Large Language Model (From Scratch).* Pragmatic Bookshelf, 2024, ISBN 978-1-63343-716-6. Printed in black & white. 27

[79] Fan, Angela, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang: *Large language models for software engineering: Survey and open problems.* In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023. 28, 29, 31

[80] Liu, Pengfei, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig: *Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing.* ACM Computing Surveys, 55(9):1–35, 2023. 28

[81] Yao, Shinn, Jeffrey Zhao, Dian Yu, Izhak Shafran Zhao, Zhou Yu, Karthik Narasimhan, and Yuan Cao: *Tree of thoughts: Deliberate problem solving with large language models.* arXiv preprint arXiv:2305.10601, 2023. 29

[82] *Tensorflow.* `https://www.tensorflow.org`, 2024. Accessed: 2025-06-12. 29

[83] *Pytorch.* `http://pytorch.org`, 2024. Accessed: 2025-06-12. 29

[84] Jiang, Yu, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, *et al.*: *When fuzzing meets llms: Challenges and opportunities.* In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 492–496, 2024. 30, 31

[85] Renze, Matthew and Erhan Guven: *The effect of sampling temperature on problem solving in large language models.* arXiv preprint arXiv:2402.05201, 2024. 31

[86] Shi, Wenxuan, Yunhang Zhang, Xinyu Xing, and Jun Xu: *Harnessing large language models for seed generation in greybox fuzzing.* arXiv preprint arXiv:2411.18143, 2024. 34

[87] Schulhoff, Sander, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, H Han, Sevien Schulhoff, *et al.*: *The prompt report: A systematic survey of prompting techniques.* arXiv preprint arXiv:2406.06608, 5, 2024. 38

[88] PromptLayer: *Template variables.* `https://docs.promptlayer.com/features/prompt-registry/template-variables`, 2025. Accessed: 2025-09-18. 43

[89] Tam, Zhi Rui, Cheng Kuang Wu, Yi Lin Tsai, Chieh Yen Lin, Hung yi Lee, and Yun Nung Chen: *Let me speak freely? a study on the impact of format restrictions on performance of large language models.* arXiv preprint arXiv:2408.02442, 2024. 43

[90] Caldiera, Victor R Basili1 Gianluigi and H Dieter Rombach: *The goal question metric approach.* Encyclopedia of software engineering, pages 528–532, 1994. 46

[91] Schloegel, Moritz, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz: *Sok: Prudent evaluation practices for fuzzing.* In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993. IEEE, 2024. 49, 62

[92] Arcuri, Andrea and Lionel Briand: *A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering.* Software Testing, Verification and Reliability, 24(3):219–250, 2014. 49, 62

[93] So, S., M. Lee, J. Park, H. Lee, and H. Oh: *VERISMART: a highly precise safety verifier for Ethereum smart contracts*. In *Proc. 2020 IEEE Symp. Security and Privacy (S&P '20)*, pages 1678–1694, San Francisco, CA, USA, 2020. IEEE. 49, 71

[94] Ferreira, J. F., P. Cruz, T. Durieux, and R. Abreu: *SmartBugs: a framework to analyse Solidity smart contracts*. In *Proc. 35th IEEE/ACM Int. Conf. Automated Software Engineering (ASE '20)*, pages 1349–1352, Virtual Event, 2020. ACM. 49, 71

[95] Chen, Weimin, Xiapu Luo, Haipeng Cai, and Haoyu Wang: *Towards smart contract fuzzing on gpus*. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2255–2272. IEEE, 2024. 49

[96] Torres, Christof Ferreira, Julian Schütte, and Radu State: *Osiris: Hunting for integer bugs in ethereum smart contracts*. In *Proceedings of the 34th annual computer security applications conference*, pages 664–676, 2018. 49

[97] Sallou, June, Thomas Durieux, and Annibale Panichella: *Breaking the silence: the threats of using llms in software engineering*. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, page 102–106. ACM, April 2024. `http://dx.doi.org/10.1145/3639476.3639764`. 50

[98] Hu, J., Q. Zhang, and H. Yin: *Augmenting grey-box fuzzing with generative ai*. arXiv preprint arXiv:2306.06782, 2023. 50

[99] Inozemtseva, Laura and Reid Holmes: *Coverage is not strongly correlated with test suite effectiveness*. In *Proceedings of the 36th international conference on software engineering*, pages 435–445, 2014. 66

[100] Feng, Peixuan, Wenrui Cao, Siqi Lu, Yongjuan Wang, Haoyuan Xue, and Runnan Yang: *Acofuzz: an ant colony algorithm-based fuzzer for smart contracts*. Blockchain: Research and Applications, page 100279, 2025. 71