Soleker: Uncovering Vulnerabilities in Solana Smart Contracts

Kunsong Zhao[†], Yunpeng Tian[†], Zuchao Ma[†], Xiapu Luo[†*]
[†]*The Hong Kong Polytechnic University, China*
{*kunsong.zhao, yun-peng.tian, zuchao.ma*}*@connect.polyu.hk, csxluo@comp.polyu.edu.hk*

***Abstract*—Solana has rapidly evolved into a leading next generation platform for supporting decentralized applications due to its high performance and low transaction costs. Its new contract execution model, which decouples code logic from states, gives rise to new vulnerability threats that can result in significant financial losses for users within the ecosystem. However, existing studies towards detecting vulnerabilities are predominantly tailored for Ethereum smart contracts, which are unsuitable for Solana platform because of the variations in implementation languages and runtime semantics. In this paper, we propose *Soleker*, a novel approach that leverages learning-based techniques to automatically identifying potential vulnerabilities in Solana smart contract bytecode. More specifically, *Soleker* captures runtime semantic information from instructions that are associated with blockchain interactions and extracts vulnerability-specific localized features. Then, a prefix-guided graph learning model is introduced to learn and integrate extracted features, enabling effective vulnerability detection. We conduct experiments on a newly constructed contract dataset and the results demonstrate that *Soleker* significantly outperforms the baseline methods, achieving an average effectiveness improvement of 126.4% and a 335× boost in efficiency.**

***Index Terms*—Blockchain, Solana Smart Contract, Vulnerability Detection**

I. Introduction

Solana has emerged as a next-generation blockchain ecosystem, quickly establishing itself as a leading platform for decentralized applications and non-fungible token marketplaces since its launch in 2020 [1], [2]. It is specifically designed to overcome the speed limitations and high transaction costs of traditional platforms like Ethereum [3]. In particular, Solana can process about 1,000 times more transactions per second while keeping transaction fees that are three orders of magnitude lower [4], [5]. This improvement arises from the design of the execution runtime, which decouples program logic (e.g., smart contracts written in Rust) from associated states and enables transactions to process distinct data in parallel [6], [7], [8].

Unfortunately, the introduction of a new smart contract execution model has also given rise to new attack surfaces, leading to significant financial losses for users within the Solana ecosystem [9], [10]. For instance, the infamous Wormhole incident results in a staggering losses of 326 million USD because of the absence of proper key checking [10]. Given the immutability of the blockchain, identifying vulnerabilities in Solana smart contracts and preventing their exploitation are crucial for protecting user digital assets [11].

*Corresponding author.

Existing studies on identifying smart contract vulnerabilities primarily focus on the Ethereum platform by utilizing various techniques such as program analysis [12], [13], [14], deep learning [11], [15], and formal verification [16], [17], [18], [19]. However, these tools cannot be applied to the Solana ecosystem due to the differences in implementation languages, execution runtimes, and vulnerability manifestations [20]. Although recent studies [7], [8] leverage static analysis and fuzzing techniques to detect vulnerabilities in Solana smart contracts, they still have the following limitations. **L1**: The existing method relies heavily on the availability of the source code. However, since over 98% of smart contracts on Solana are not open source [5], its applicability is significantly limited. **L2**: These approaches depend entirely on predefined patterns designed by experts. Although such rules can benefit the detection, the rapid increase of smart contracts makes it prohibitively expensive to continuously craft new rules to accommodate diverse vulnerability patterns. **L3**: They either focus solely on capturing static features while overlooking the program dynamics [7], or suffer from limited code coverage due to the randomness of input generation [8]. Consequently, their ability to capture code semantic behavior is constrained, leading to suboptimal detection performance.

In this paper, we present *Soleker*, a learning-based approach designed specifically for Solana smart contracts, which directly leverages code semantics and features extracted from contract bytecode to enable vulnerability detection. However, it is non-trivial to design *Soleker* due to the following challenges. **C1**: To achieve high performance and flexibility, Solana compiles smart contracts into corresponding eBPF bytecode. Such low-level representation consists of a vast number of instructions, making it difficult to comprehend the underlying logic and detect potential vulnerable behaviors. Therefore, we create a code snapshot to ensure the model focuses exclusively on instructions associated with blockchain interactions since they will affect blockchain states and incur possible vulnerabilities (§ III-B). **C2**: Solely learning from instruction snippets fails to capture dynamic semantics, which are crucial for the model to grasp the vulnerability-related instruction context. Consequently, we develop a lightweight symbolic analyzer that simulates the instruction execution to obtain runtime semantics and treat them as instruction features (§ III-C). **C3**: While previous steps capture overall code semantics, localized features related to specific vulnerabilities will be forgot during the learning process [21], because the vulnerability could be triggered by merely several instructions. For this reason, we construct a prefix vector as a complementary component to enrich the model with vulnerability-specific knowledge (§ III-D). **C4**: Accurately detecting vulnerabilities presents another challenge, as it requires the effective integration of all relevant features to enable the model to understand code semantics while distinguishing vulnerable details from benign ones. Accordingly, we design a prefix-guided graph neural network to aggregate extracted features and generate detection results (§ III-E).

We conduct experiments on a newly constructed dataset comprising over 1,700 labeled Solana smart contracts, covering representative vulnerabilities such as missing signer checking, missing owner checking, missing key checking, and arbitrary cross program invocations. Experimental results demonstrate that *Soleker* achieves average performance scores of 94.97%, 91.11%, 89.04%, and 92.38% on each of the vulnerabilities, with significant improvements of 162.8%, 128.1%, 72.0%, and 158.9% over the baseline methods. In addition, extended experiments and analyses highlight the effectiveness and efficiency of each component in *Soleker*, along with its capability to identify vulnerabilities in the wild (§ IV).

In summary, this paper makes the following contributions.

- We tackle the practical challenge of detecting vulnerabilities in Solana smart contracts. To the best of our knowledge, this is the first work to leverage learning-based techniques for automatically identifying potential vulnerabilities directly from contract bytecode.
- We propose *Soleker*, a novel approach that captures runtime semantics of contract bytecode and introduces a prefix-guided learning model to enhance the vulnerability detection.
- We conduct extensive experiments to evaluate the performance of *Soleker* and the results indicate that *Soleker* outperforms the baselines in both effectiveness and efficiency, with each design playing a key role in its success.
- We also contribute to the community by providing a new dataset focused on vulnerabilities in Solana smart contracts to support future research. To the best of our knowledge, this is the largest labeled dataset of Solana contracts to date.

II. Background

A. Basic Concepts of the Solana Blockchain

Solana is an ideal platform for decentralized applications that require high speed and cost efficiency [1]. To interact with the Solana ecosystem, users submit a transaction containing operations and the associated data to be processed. Solana's execution environment then validates the transaction, invokes the specified program deployed on the platform, and updates the relevant states [7], [22]. Readers can find more details in the official Solana documentation [1].

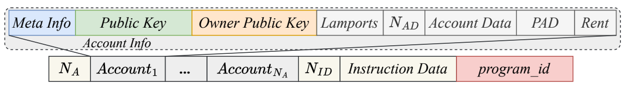


Fig. 1. The layout of contract input.

Account Model. Account is the core of Solana blockchain, which is responsible for storing data in the way of key-value pairs. Different from Ethereum [23], Solana splits the executable code logic from the raw data storage, i.e., states. Each account contains the following elements: *key* is a 32-byte unique identifier for the account; *data* stores either the state or executable code; *executable* is a symbol to specify whether the account is executable; *lamports* specifies the account balance, where lamports is the smallest unit (1 SOL = 1B lamports); *owner* represents the public key of the program that owns the account; *rent epoch* is a time window for managing account storage data and rent payment [6].

Smart Contract & Execution Runtime. Smart contracts are referred to as programs in the Solana ecosystem [22]. There are two types of programs: native programs, which provide the ability to interact with the underlying blockchain protocol, and on-chain programs, which are developed using high-level languages like Rust and then compiled into extended Berkeley Packet Filter (eBPF) bytecode deploying on the blockchain. Here we refer to the bytecode deployed on the Solana platform as Solana eBPF (SBPF) to distinguish it from the original eBPF bytecode. This distinction is necessary because Solana's bytecode can be viewed as a subset or an extension of the standard eBPF instruction set, specifically tailored for Solana's contract execution environment [24]. Although programs are stateless, they include many functions that can modify the storage data of accounts whose owner matches the public key of the program [22]. Programs are executed by the SBPF virtual machine (VM) in either interpreter or just-in-time mode [24]. Before executing a program on the VM, Solana's runtime serializes the input accounts to be processed, allowing the VM to directly access and manipulate the relevant data. Fig. 1 illustrates the input layout after serialization. The *Meta Info* section contains flags such as *is_signer*, *writable*, and *executable*, etc. N_{AD}, N_A, and N_{ID} represent the number of account data, accounts, and instruction data, respectively. *PAD* refers to the padding operation used to align offsets to 8-byte boundaries. In addition, the SBPF VM imposes restrictions on resource consumption and invocation depth to mitigate potential threats, such as reentrancy [25].

Program Derived Address & Cross Program Invocation. A program derived address (PDA) is a 32-byte string that lies outside the Ed25519 curve, meaning it does not have an associated private key [26]. This design allows Solana developers to deterministically generate an account address by combining a *program id* with predefined inputs and a bump seed. The *program id* is the public key of the account that holds the program. The bump seed is a value ranging from 255 to 0, which is used to ensure the PDA falls off the Ed25519 curve [26]. Moreover, programs can sign for PDAs derived from their *program id*, enabling the cross program invocation (CPI) [27]. During execution, one program can invoke functions in other programs via CPI, with a maximum call depth of 4. In this process, the caller's privileges are extended to the callee, since the PDA is derived from the public key of the caller [27].

B. Vulnerabilities in Solana Smart Contracts

Although smart contracts on Solana are written in the memory-safe Rust language, they still inevitably harbor vulnerabilities that pose a significant threat to the security of the ecosystem. In this paper, we focus on four representative vulnerabilities within the Solana ecosystem. We choose these four vulnerability types based on three considerations: (1) Relevance to Solana's stateless design: Input verification is critical in Solana contracts due to the stateless nature. The selected vulnerabilities cover key aspects of validation, with different manifestation and exploitation methods that directly impact contract security. (2) Prevalence: These vulnerabilities are among the most prevalent on Solana, as reported by security auditing companies [28], [29], [30], [31]. More importantly, they are also highlighted in the program security section of the official Solana developer documentation [32], [33], serving as a critical warning to developers about potential risks. (3) Real-world impact: They have resulted in significant financial losses within the Solana ecosystem. For instance, the Cashio hack led to a $52M loss due to the lack of signer checking [34], while CremaFinance lost $8.8M because of a missing owner checking [35]. In addition, Wormhole suffered a staggering $323M loss due to a missing key checking [10], and Slope Wallet incurred an $8M loss as a result of an arbitrary cross program invocation vulnerability [36]. These incidents highlight the critical importance of detecting these vulnerabilities.

1) Missing Owner Checking (MOC): Due to the design that separates state from executable code, all accounts involved in a function call must be provided as inputs by the caller. Each account has an associated *owner*, which specifies the *program id* authorized to modify the account. If the contract fails to verify that the account being operated on is owned by the correct *program id*, an attacker can craft malicious data under their control and feed it into the contract, potentially enabling money transfer from an account that does not belong to him.

```
 1 fn withdraw(program_id: &Pubkey, accounts: &[
       AccountInfo], amount: u64) -> ProgramResult
       {
 2   let accs_iter = &mut accounts.iter();
 3   let wallet = next_account_info(accs_iter)?;
 4   let vault = Wallet::deserialize(&mut &(*wallet
       .data).borrow_mut()[..])?;
 5   let authority = next_account_info(accs_iter)?;
 6   let dest = next_account_info(accs_iter)?;
 7   assert_eq!(vault.authority, *authority.key);
 8   if amount > **wallet.lamports.borrow_mut() {
 9     return Err(InsufficientFundsError);}
10   // + Fix: Missing Owner Checking
11   // + assert_eq!(wallet.owner, program_id);
12   // + Fix: Missing Signer Checking
13   // + assert!(authority.is_signer);
14   ... // Token transfer operations are omitted.
15   Ok(())}
```

Listing 1. A vulnerable Solana smart contract for withdrawing funds.

Example: Listing 1 demonstrates a smart contract for withdrawing funds from a vault. The contract takes three parameters as input: *program_id*, which is the public key of the contract; *accounts*, which contains all accounts involved in the execution; and *amount*, which specifies the amount to be transferred. After iterating through the accounts from *accs_iter*, the contract then retrieves three accounts: the *wallet* account holding the funds, the *authority* account saving the authorization operation, and the *dest* account specifying the fund receiver. The contract then transfers lamports from the *wallet* account to the *dest* account. However, as highlighted in Lines 10-11, the contract code fails to verify whether the *program_id* is the legitimate owner of the *wallet*, which will allow attackers to redirect the funds to their own account.

2) Missing Signer Checking (MSC): When the caller operates account data, the smart contract must verify that the caller has the appropriate access rights, i.e., the initiated transaction must be signed by the authorized user. If the contract fails to validate the signer, malicious users can forge an authority account and execute illegal transactions.

Example: As shown in Lines 12-13 of Listing 1, the contract ignores the signature validation for *authority* account, i.e., the check on the flag of the *is_signer* field. This vulnerability will enable the attacker to launch an unsigned transaction and take control of the authority account, thereby unlawfully withdrawing funds from the vault.

```
1 fn verify_signatures(ctx: &ExectionCtx, accs: &
     mut VerifySigns, data: VerifySignsData) ->
     Result {...
2   let instr_sysvar_acc = &accs.instruction_acc;
3   // + Fix: Missing Key Checking
4   // + if instr_sysvar_acc.key != solana_program
      ::sysvar::instructions::id() {
5   // +   return Err(UnsupportedSysvarError); }
6   let instr_sysvar_data = instr_sysvar_acc.
      try_borrow_mut_data()?;
7   let cur_instr = solana_program::sysvar::
      instructions::load_current_index(
      instr_sysvar_data)?; ...}
```

Listing 2. A smart contract with missing key checking.

3) Missing Key Checking (MKC): When a smart contract is designed to modify storage data associated with a specific account, it is essential to verify the account key to ensure that the operation is being performed on the intended account. If the contract overlooks verifying the intended account, malicious attackers can substitute another account they control for profit.

Example: Listing 2 presents an example contract from Wormhole bridge, which has caused $320 million loss [10]. This signature verification function retrieves the instructions to be executed using the *load_current_index* function, with the parameter derived from the user input *accs*. The code assumes that the input account key corresponds to the *sysvar* in the Solana protocol, but neglects to validate this assumption (Lines 3-5). As a result, an attacker can forge the account and execute their own crafted instructions, resulting in the loss of funds.

4) Arbitrary Cross Program Invocation (ACPI): Cross program invocation facilitates the interaction between various Solana smart contracts, which is essential for decentralized applications. However, if a smart contract neglects to verify that the invocation target is the intended contract, malicious users can modify the associated storage data arbitrarily.

Example: Listing 3 illustrates an example contract about arbitrary cross program invocation. After retrieving the target contract from the input *accounts* (Line 3), the code calls the *invoked_signed* function to perform cross program invocation with the parameter *cnt.key*. However, it fails to verify that the target contract is trusted (Lines 4-5), potentially allowing arbitrary data modification or unauthorized actions.

```
1 fn invoke_transfer(program_id: &Pubkey, accounts
      : &[AccountInfo], amount: u64) ->
      ProgramResult{
2   // ... The same code as shown in Listing 1.
3   cnt = next_account_info(accs_iter)?
4   // + Fix: Arbitrary Cross Program Invocation
5   // + assert_eq!(cnt.key,&spl_token::id());
6   invoked_signed(&spl_token::instruction::
        transfer_checked(cnt.key, ...),...)?;
7   Ok(())}
```

Listing 3. A contract with arbitrary cross program invocation.

III. Methodology

A. Overview

The overall framework of *Soleker* is elaborated in Fig. 2. We focus on realistic on-chain vulnerability detection scenarios where only the compiled SBPF bytecode is accessible. *Soleker* comprises four main components: *Snapshot Builder*, *Runtime Analyzer*, *Prefix Extractor*, and *Vulnerability Detector*. More specifically, *Snapshot Builder* captures the instructions associated with Solana blockchain interactions from the SBPF bytecode to preserve potential operations related to vulnerabilities (§ III-B). *Runtime Analyzer* introduces a lightweight symbolic analyzer to simulate the execution of SBPF instructions for obtaining runtime semantics of each instruction (§ III-C). *Prefix Extractor* constructs a prefix vector based on vulnerability-specific rules, enriching the model with localized features (§ III-D). Finally, *Vulnerability Detector* integrates the extracted features and generates detection results (§ III-E). In the following subsections, we provide a detailed explanation of each designed component individually.

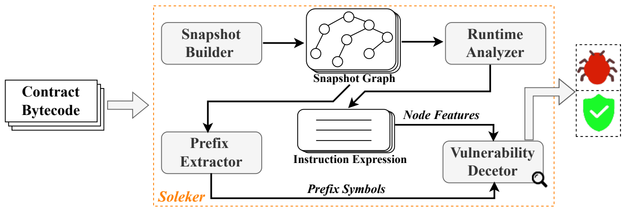


Fig. 2. The overall framework of *Soleker*.

B. Snapshot Builder

When a Solana smart contract is compiled, all the source code information is compressed into the SBPF bytecode. The instruction set provides a special instruction `syscall` to facilitate the interaction of the virtual machine with the external Solana blockchain environment. Only those instruction operations related to `syscall` will actually affect the blockchain states, i.e., they can potentially trigger vulnerabilities. Therefore, we define the snapshot as the instructions that are associated with blockchain system calls. *Soleker* employs the Datalog engine [37] to capture these snapshots for analysis. As a declarative logic language, Datalog is widely used in program analysis tasks due to its simplicity of expression and efficient reasoning ability [37], [38], [39]. It enables users to define facts and inference rules, facilitating the reasoning and derivation of new facts during analysis [40].

TABLE I
Facts and Inference Rules

```
Instruction(pc,opcode,des_reg,src_reg,offset,imm).
Flow(X, Y).
-----------------------------------------------------
FindSource(pc) :- Instruction(pc, "syscall", _, _,
          CALL_TARGET,_).
FindImpacts(pc, pc) :- FindSource(pc).
FindImpacts(sopc, dpc) :- FindImpacts(sopc, spc),
             Flow(spc, dpc).
FindImpactsBy(pc, pc) :- FindSource(pc).
FindImpactsBy(spc, tpc) :- FindImpactsBy(dpc, tpc),
              Flow(spc, dpc).
```

Table I demonstrates the facts and inference rules defined for our analysis. We directly utilize the Datalog syntax to formalize them for clarity. We define each instruction fact according to its logical expression where `pc` means the program counter which holds the instruction location. `opcode` refers to the name of operation code. `des_reg` is the destination register that stores the instruction execution results. `src_reg` is the source register that provides the data to be operated on by the instruction. `offset` refers to the offset value used for memory access or jump instructions. `imm` is a constant value that is included directly in the instruction. The fact `Flow(X, Y)` means a data flow between instruction variable `X` to `Y`. With the facts, we can further define the inference rules $\mathcal{R}$ for extracting the snapshots as shown at the bottom of the table.

We start with the rule `FindSource(pc)`, which looks for locations that interact with the blockchain environment using the `syscall` instruction. `CALL_TARGET` refers to the built-in system calls in the Solana platform. Once the instruction location is found, we define the rules `FindImpacts` to find those instructions that will be affected by specific system calls. The rules describe two aspects: (1) retaining the instructions that interact with the blockchain environment; (2) if an instruction `spc` is affected by a `syscall` (i.e., `FindImpacts(sopc,spc)` is satisfied) and there exists a data flow from `spc` to another instruction `dpc` (i.e., `Flow(spc,dpc)` holds), then the instruction `dpc` is also affected by this system call. Similarly, we utilize the rules `FindImpactsBy` to infer the instructions that will impact the system calls.

Since the manipulation of blockchain state relies on reads and writes to storage, only system calls that involve storage access can present potential vulnerabilities. Therefore, we mainly focus on the following five system calls. *sol_memcpy_* copies *n* bytes from a memory region to another. *sol_memset_* sets a specified value to the first *n* bytes of a memory location. *sol_log_* emits log messages to the blockchains for tracking program behaviors. *sol_try_find_program_address* discovers a valid PDA with the given bump seed. *sol_invoke_signed_rust* facilitates interactions with other smart contracts [41]. Among these, the first two are associated with storage access, while

TABLE II
Abbreviated Description of SBPF Syntax and Symbolic Expressions

Syntax of SBPF Bytecode		Syntax of Symbolic Expressions	
SC	$::= \mathit{stmt}\star$	e	$::= e_{form} \mid e_{set}$
stmt	$::= \mathtt{ASSIGN}\ \mathit{reg}\ \mathit{expr} \mid \mathtt{RETURN} \mid \mathtt{BRANCH}\ \mathit{expr}$ $\mid \mathtt{LOAD}\ \mathit{reg}\ \mathit{expr} \mid \mathtt{STORE}\ \mathit{expr}_1,\ \mathit{expr}_2 \mid \mathtt{CALL}\ \mathit{expr}$ $\mid \mathtt{SYSCALL}\ \mathit{expr} \mid \mathtt{CDBRANCH}\ \mathit{expr}_1,\ \mathit{expr}_2,\ \mathit{expr}_3$	e_{form}	$::= e_{src} \mid \mathit{MEM}\ [e_{form}] \mid \mathit{const}_n \mid \ominus_u\ e_{form}$ $\mid e_{form_1} \oplus_b e_{form_2}$
expr	$::= \mathit{reg} \mid \mathtt{const} \mid \mathtt{func} \mid \mathit{mem[reg + offset]}$ $\mid \ominus_{\mathbb{U}}\ \mathit{expr} \mid \oplus_{\mathbb{B}}\ \mathit{expr}_1,\ \mathit{expr}_2$ $\mid \lozenge_{\mathbb{C}}\ \mathit{expr}_1,\ \mathit{expr}_2,\ \mathit{expr}_3$	e_{set}	$::= \varnothing \mid e_{src} \mid \{e_{src_1}, e_{src_2}, \ldots\}$
$\ominus_{\mathbb{U}}$	$::= \mathtt{neg32} \mid \mathtt{neg64}$	e_{src}	$::= \mathit{String} \mid \mathit{Symbol}_r$
$\oplus_{\mathbb{B}}$	$::= \mathtt{add32} \mid \mathtt{sub32} \mid \mathtt{mul32} \mid \mathtt{div32} \mid \ldots$	$\ominus_u$	$:= \neg$
$\lozenge_{\mathbb{C}}$	$::= \mathtt{jeq} \mid \mathtt{jgt} \mid \mathtt{jge} \mid \mathtt{jset} \mid \mathtt{jne} \mid \mathtt{jsgt} \mid \mathtt{jsne} \mid \ldots$	$\oplus_b$	$:= + \mid - \mid * \mid \div \mid \& \mid \wedge \mid \ldots$

the last two pertain to program address and cross program invocations. Readers seeking more details about the system calls in Solana can refer to [41].

After extracting the relevant instructions for each system call, we integrate them and construct a graph representation according to their dependencies, as illustrated in Algorithm 1. Specifically, we retain only the instructions related to system calls and treat them as nodes, while representing the relationships between nodes as edges. As a result, we construct a graph $G(V, E)$ where V represents the nodes corresponding to each instruction, and E denotes the relationships between instructions. We refer to this built graph as a snapshot graph since it preserves partial information from the original bytecode. According to our statistics, the instructions in the snapshot, on average, represent only 63.78% of the original SBPF bytecode.

Algorithm 1: Snapshot Graph Construction

```
  Input: The inference results by our rules $\mathcal{R}$ and the facts in $Flow$
  Output: The generated Snapshot Graph $G$
1 Function SnapshotGraph ($\mathcal{R}$):
2     $V \leftarrow \emptyset$ ; $E \leftarrow \emptyset$ ;
3     foreach node $v$ in $\mathcal{R}$ do
4         $V \leftarrow V \cup \{v\}$ ;
5     foreach edge $e(v, u)$ in $Flow$ do
6         if $v \in V \wedge u \in V$ then
7             $E \leftarrow E \cup \{e(v, u)\}$ ;
8     return $G(V, E)$
```

C. Runtime Analyzer

After creating the snapshot graph representation, we further extract the initial features of each node. A common solution is to use opcodes as the initial node embeddings; however, this method captures only the static semantic information of instructions and fails to reflect their dynamic behavior at runtime [15], [42], [43], [44]. As a low-level representation, the opcode in SBPF bytecode is insufficient to reveal the deeper contract logic. Instead, runtime information can help understand code semantics by providing contextual knowledge to facilitate our vulnerability detection task [43]. Therefore, we develop a runtime analyzer that performs lightweight symbolic analysis on the snapshot graph and generates symbolic expressions for each instruction. These expressions are then used to build the node features, serving as the initial node embeddings for learning graph-level features.

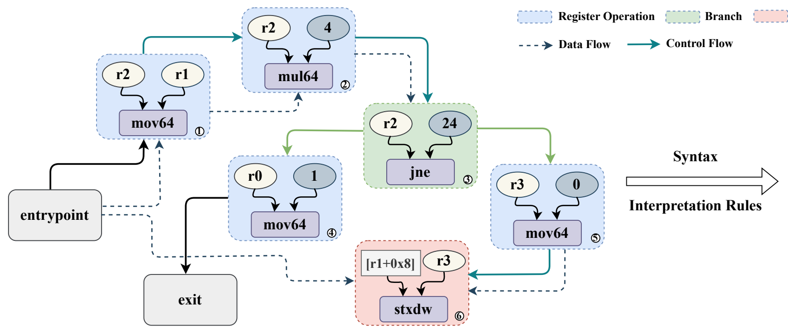
1) Syntax: We begin by formally defining the syntax used to represent SBPF bytecode (SC), as shown in Table II (left). Specifically, `ASSIGN` moves the value of expr to the target register reg. `STORE` and `LOAD` writes the value of expr_2 into the address pointed by expr_1 and reads the value of the address pointed to by expr into a target register reg, respectively. `BRANCH` jumps to the location specified by expr. `CDBRANCH` compares the values of expr_1 and expr_2, and then jumps to the location specified by expr_3 if the condition is satisfied. `CALL` performs internal function calls, while `SYSCALL` handles built-in blockchain interactions.

We then formalize the syntax of symbolic expressions (e), which is derived from the syntax definition of SBPF and further approximate the underlying semantics of variables (see Fig. 3). As shown in Table II (right), a symbolic expression can be either a simple or nested formula e_{form}, or a set of symbols e_{set}. In particular, $\mathit{MEM}\ [e_{form}]$ refers to the memory location addressed by e_{form}. const_n specifies an arbitrary constant n. e_{src} is defined as the basic expression such as String or the symbols associated with register input Symbol_r. During the symbolic analysis, a register variable may contain multiple symbolic expressions due to the presence of various execution flows [45]. Following previous work [43], we merge the original expression (e_{src}) as the semantic-aware representation, allowing us to preserve all runtime information within the execution flows.

TABLE III
Interpretation for Instruction Expression

$$
\begin{array}{l}
\phi(\mathit{reg} := \mathit{expr}) \rightsquigarrow \mathit{reg} := \phi(\mathit{expr}) \\
\phi(\ominus_{\mathbb{U}}\ \mathit{expr}) \rightsquigarrow \ominus_u\ \phi(\mathit{expr}) \\
\phi(\oplus_{\mathbb{B}}\ \mathit{expr}_1,\ \mathit{expr}_2) \rightsquigarrow \phi(\mathit{expr}_1) \oplus_b \phi(\mathit{expr}_2) \\
\phi(\lozenge_{\mathbb{C}}\ \mathit{expr}_1,\ \mathit{expr}_2,\ \mathit{expr}_3) \rightsquigarrow \lozenge_c\ \phi(\mathit{expr}_1)\ \phi(\mathit{expr}_2)\ \phi(\mathit{expr}_3) \\
\phi(\mathtt{LOAD}\ \mathit{reg},\ \mathit{expr}) \rightsquigarrow \mathit{MEM}\ [\phi(\mathit{expr})]\ \mathtt{LOAD}\ \phi(\mathit{reg}) \\
\phi(\mathtt{STORE}\ \mathit{expr}_1,\ \mathit{expr}_2) \rightsquigarrow \mathtt{STORE}\ \mathit{MEM}\ [\phi(\mathit{expr}_1)]\ \phi(\mathit{expr}_2) \\
\phi(\mathtt{MERGE}\ (e_{set_1}, e_{set_2}, \ldots)) \rightsquigarrow \phi(e_{set_1}) \cup \phi(e_{set_2}) \cup \phi(\ldots)
\end{array}
$$

2) Instruction Symbolic Expression: Building on the above syntax, we next define an interpretation function ϕ to translate register variables and statement expressions into instruction-level symbolic expressions. As demonstrated in Table III, the first rule focuses on interpreting assignments to register variables. The subsequent three expressions are interpreted directly based on their manipulation semantics. For the `LOAD`



PC	Runtime Symbolism	Instruction Symbolic Expressions
①	ϕ (r1) = REG1 ϕ (r2) = REG1	REG1 MOV64 REG1
②	ϕ (r2) = REG1 $\times$ 4	REG1 $\times$ 4 MUL64 REG1 4
③	ϕ (r2) = REG1 $\times$ 4	JNE REG1 $\times$ 4 24
④	ϕ (r0) = 1	RET MOV64 1
⑤	ϕ (r3) = 0	REG3 MOV64 0
⑥	ϕ ([r1+0x8]) = MEM [REG1_8] ϕ (r3) = 0	STXDW MEM [REG1_8] 0

Fig. 3. An example of constructing node features using the lightweight symbolic analysis.

expression, the rule first resolves the actual memory location and then loads the corresponding data into the target register. In the case of the `STORE` expression, the rule interprets the value to be stored by $\phi(expr_2)$ and assigns it to the memory address specified by $\phi(expr_1)$. Note that the opcodes are retained as an integral part of the instruction-level symbolic expression. The final rule describes the merge operation, designed to capture all runtime information when a specific instruction point is impacted by multiple execution flows.

3) Node Feature Generation: To reveal the runtime semantics of each instruction at the bytecode level, we leverage the symbolic analysis output to generate node embeddings. This operation enables us to replace each instruction with its corresponding symbolic expression. During the analysis, we retain various types of edges to enrich our model. Specifically, we define four types of edge features based on different dependencies: register, branch, return, and call. Therefore, we expand each built snapshot graph with these edge features for further learning. After performing lightweight symbolic analysis across all contracts, we aggregate the resulting symbolic expressions and use them to pre-train a Doc2vec model [46]. As a result, we can obtain the embedding vectors for each expression, i.e., the initial node embeddings.

Fig. 3 elaborates an example of extracting node features using the lightweight symbolic analysis. Starting with the entry point, the initial input parameters are stored in register $r1$ and we refer to it as `REG1`. The program first reads the input from $r1$ and puts it into the register $r2$. Thus, both $r1$ and $r2$ hold the input value `REG1` (①). The value of $r2$ then undergoes a binary operation, i.e., `mul64`, where it is multiplied by 4. As a result, the corresponding symbolic value is updated to `REG1` $\times$ `4` (②). After that, a conditional branch `jne` is used to check whether the value of $r2$ is equal to `24` (③). If the condition is met, the return register `RET` is set to `1` and the program terminates execution. Otherwise, the program assigns the value `0` to register $r3$ and then writes it into the memory address `REG1_8` using the `stxdw` operation (⑥). After completing the symbolic analysis of variables and statements, the interpretation rules are applied to generate the symbolic expression for each instruction as outlined in the last column of the table in Fig. 3. For instance, the binary operation at ② will produce the symbolic expression `REG1` $\times$ `4 MUL64 REG1 4`, and the memory access operation at ⑥ is represented as `STXDW MEM [REG1_8] 0`. Our symbolic expressions preserve not only the opcodes and operands one instruction inherently carry but also incorporate the variable information passed during execution. Then, these expressions are used to train the Doc2vec model and generate the corresponding node features.

D. Prefix Extractor

After extracting the initial features for each graph node, we can leverage the powerful representation learning capabilities of graph neural networks to capture the semantics of the entire snapshot. This process will be detailed in the next subsection (§ III-E). Meanwhile, we aim to integrate additional vulnerability-specific domain knowledge into the model to improve its capacity for representing various types of vulnerabilities. Prefix learning is an efficient and lightweight model optimization technique designed to enhance a model's adaptability to specific domain tasks [47]. The key concept is to integrate a trainable prefix vector that serves as a complementary component to the model's inputs. This approach has been extensively applied in various fields such as natural language processing and multi-modal tasks [47], [48], [49].

Inspired by the success of prefix learning strategies, *Soleker* extracts particular prefixes for different types of vulnerabilities based on the rules outlined in Table IV. We next provide explanations of each rule.

POC: It first checks whether an instruction uses `ldxdw` operation to load data according to the offset of the public key of the *owner* into a register R_{opk}. A continuous sequence of instructions S_I then iteratively compares the value stored in this register with the value of another register R_{pid} containing the *program id*, processing eight bytes at a time. Before comparing the final eight bytes, a flag such as `0` is set (i.e., $S_I[\cdot]$). Listing 4 demonstrates an example of the `REGCMP` operation. It loads values from two given memory addresses with the same offset `off` into registers `r3` and `r4`, then compares the values to determine the jump target `lbb`.

```
ldxdw r3, [r1 + off] ;Load value from (r1 + off)
ldxdw r4, [r2 + off] ;Load value from (r2 + off)
jne r4, r3, lbb  ;Jump if r4 is not equal to r3
```

Listing 4. An example of the operation `REGCMP(r1,r2)`.

Once the comparison is complete, a jump operation (as indicated by $\lozenge_{\mathbb{C}}$) is executed to evaluate the satisfaction of

TABLE IV
RULES FOR EXTRACTING DIFFERENT PREFIXES

Prefix	Extraction Rules
POC	$\exists\, I, S_I\big(\big(I.\text{opcode} = \texttt{ldxdw} \land I.\texttt{offset} = \texttt{offset}_{opk} \land I.\text{dst\_reg} = \text{R}_{opk}\big) \land \big(S_I[\cdot].\text{opcode} = \texttt{mov64} \land S_I[\cdot].\text{imm} = 0\big) \land \big(\texttt{REGCMP}(\text{R}_{opk}, \text{R}_{pid}) \in S_I\big) \land \big(S_I[\text{-1}].\text{opcode} \in \Diamond_{\mathbb{C}}\big)\big)$
PSC	$\exists\, I_1, I_2\big(\big(I_1.\text{opcode} = \texttt{ldxb} \land I_1.\texttt{offset} = \texttt{offset}_{sig}\big) \land \big(I_2.\text{opcode} \in \Diamond_{\mathbb{C}} \land I_2.\text{imm} = 0\big) \land \big(I_2.\text{pc} = I_1.\text{pc} + 1\big)\big)$
PKC	$\exists\, I_1, I_2, S_I\big(\big(I_1.\text{opcode} = \texttt{ldxdw} \land I_1.\texttt{offset} = \texttt{offset}_{pbk} \land I_1.\text{dst\_reg} = \text{R}_{pbk}\big) \land \big(I_2.\text{opcode} = \texttt{lddw} \land I_2.\text{imm} = \texttt{offset}_{epk} \land I_2.\text{dst\_reg} = \text{R}_{epk}\big) \land \big(S_I[\cdot].\text{opcode} = \texttt{mov64} \land S_I[\cdot].\text{imm} = 0\big) \land \big(\texttt{REGCMP}(\text{R}_{pbk}, \text{R}_{epk}) \in S_I\big)\big)$
PCPI	$\exists\, I_1, I_2, I_3, S_I\big(\big(I_1.\text{opcode} = \texttt{ldxdw} \land I_1.\texttt{offset} = \texttt{offset}_{epk} \land I_1.\text{dst\_reg} = \text{R}_{epk}\big) \land \big(I_2.\text{opcode} = \texttt{ldxdw} \land I_2.\texttt{offset} = \texttt{offset}_{pbk} \land I_2.\text{dst\_reg} = \text{R}_{pbk}\big) \land \big(I_3.\text{opcode} = \texttt{jeq}\big) \land \big(S_I[\cdot].\text{opcode} = \texttt{mov64} \land S_I[\cdot].\text{imm} = 0\big) \land \big(\texttt{REGCMP}(\text{R}_{epk}, \text{R}_{pbk}) \in S_I\big) \land \big(\big(S_I[\text{-2}].\text{opcode} = \texttt{jne} \land S_I[\text{-2}].\text{imm} = 0\big) \lor \big(S_I[\text{-1}].\text{opcode} \in \Diamond_{\mathbb{C}}\big)\big)\big)$

the condition. This rule applies to the potential comparison between the *owner* of an account and the *program id*.

PSC: It verifies the existence of two consecutive instructions. The first instruction uses the `ldxb` operation to load a byte of data from the *is_signer* position $\texttt{offset}_{sig}$ into a register, while the second one checks whether the value in the register is equal to 1 (signed) or not (unsigned).

PKC: This rule describes potential comparison between a public key with an expected key in the SBPF bytecode. Specifically, the instruction sequence begins by accessing the public key of an account and storing it into a register R_{pbk} using the `ldxdw` operation. Next, the `lddw` instruction directly loads the expected key into another register R_{epk} based on its offset address $\texttt{offset}_{epk}$. Similarly, a series of iterative comparisons are then performed, followed by the conditional jumps determined by the comparative results.

PCPI: The rule encompasses two types of patterns to capture different scenarios. (1) If the expected key is a fixed value, it can be loaded directly using the `lddw` operation at its offset address $\texttt{offset}_{epk}$, similar to PKC; (2) Otherwise, registers are required for loading the value and performing the comparisons. In this case, the `ldxdw` operation is used instead of the `lddw`, while the remaining comparison process remains similar to the previous one. The various offsets mentioned above are derived from the input layout discussed in § II.

E. Vulnerability Detector

After extracting various features from the given SBPF bytecode, we next design a dedicated detector that equips with learnable models to adequately mine the implicit semantics in these features for vulnerability detection as shown in Fig. 4.

1) Snapshot Graph Learning: Based on the constructed snapshot and the runtime analyzer, the graph can be extended as $G = (V, E, A)$ where A represents the edge features as defined in § III-C. To capture the semantics of code within the snapshot graph, we leverage graph neural networks for their powerful graph learning capabilities [50]. Since the previous study [51] demonstrates the effectiveness of the gated graph neural network (GGNN) [52] in data flow analysis during graph representation learning, we are inspired to employ it for our objectives. Specifically, the update process of the vanilla GGNN model primarily involves two key phases: message propagation and node update operations. However, the edges in a graph encode additional semantic information that is critical for understanding the code structure and semantics [53]. To incorporate this information into the learning process, we modify the original propagation mechanism of the GGNN by explicitly integrating edge features. Specifically, for the edge e_{ij} between a pair of nodes (i, j), we introduce a transformation function $\mathcal{F}$ to linearly project the raw edge feature into a unified semantic space, i.e., $e'_{ij} = \mathcal{F}(e_{ij})$, where e'_{ij} denotes the transformed edge feature. Then, we can formalize the process of message propagation as follows.

$$m_i = \sum\nolimits_{j \in \mathcal{N}_i} (h_j + e'_{ij}) \tag{1}$$

where m_i represents the message propagation result that combines the features of neighboring nodes and their associated edges; $\mathcal{N}_i$ represents the set of all neighboring nodes of the central node i, not limited to its child nodes, enabling graph learning to capture information from all neighbors; h_j is the embeddings of node j.

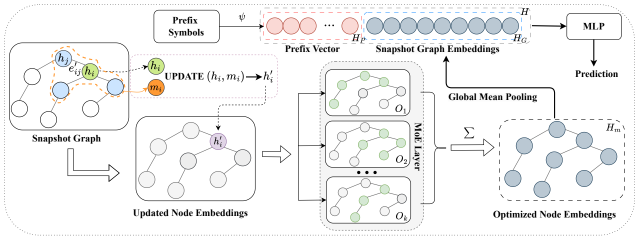


Fig. 4. The architecture of the vulnerability detector.

Then, a update function is employed to incorporate original node features with the message propagation features, which is defined as follows.

$$h'_i = \text{UPDATE}\ (m_i, h_i) \tag{2}$$

Here, we use the built-in update function of GGNN due to its ability to dynamically adapt to new node features [52]. As a result, we obtain a node set H that encompasses all the updated node features h'_i.

To further optimize the node semantics and enhance model adaptation, we introduce an additional adaptation layer before graph pooling operation. Due to the sparse activation and generalization abilities, we adopt the mixture of experts model [54] as our adaptation layer, which is defined as follows.

$$H_m = \sum\nolimits_{k \in \mathcal{S}_e} W_k(H) \cdot O_k(H), \tag{3}$$

where H_m denotes a feature set of optimized nodes and $\mathcal{S}_e$ represents the set of experts; W_k and O_k separately correspond to the weights assigned and the output of the expert k.

To generate graph-level embedding vectors for each snapshot graph, we implement a readout layer that aggregates all node features using a global mean pooling strategy. This operation is formulated as follows.

$$H_G = \frac{1}{|H_m|} \sum\nolimits_{\hat{h'_i} \in H_m} \hat{h'_i} \tag{4}$$

where H_G signifies the global feature vector for the snapshot graph and $\hat{h'_i}$ represents the embedding vector of the node i after the updating and optimizing process.

2) Prefix-guided Graph Learning: Although the snapshot encapsulates the overall semantics of the contract bytecode, most vulnerability-related manifestations in smart contracts are concentrated within a small subset of the instructions. To accurately identify these vulnerabilities, it is necessary to account for the specific localized features associated with each type of vulnerability. However, in the snapshot graph learning process, although the whole code semantics are captured, certain subtle features will be overlooked during network propagation [21]. Therefore, we design an extra prefix vector to guide the model in learning subtle vulnerability-specific semantic embeddings. More specifically, we can extract particular prefix tailored to different vulnerabilities according to the rules defined in § III-D. If no instructions match the specified rules, a default prefix identifier is assigned to ensure alignment with the overall data.

Subsequently, a projection function ψ is introduced to transform the prefix symbol p into a learnable vector representation, i.e., $H_P = \psi(p)$. Finally, the prefix vector H_P is combined with the snapshot graph representation H_G to derive the vulnerability-specific semantic embedding H, which can be formalized as follows.

$$H = [H_P \mid\mid H_G] \tag{5}$$

3) Classification: In the final stage, the vulnerability detector inputs the semantic representation produced by the prefix-guided graph learning process into a MLP layer for classification. To facilitate the model training, we employ cross-entropy loss function to guide the update of model parameters. We select the output with the highest probability as the final prediction during the detection process.

IV. Evaluation

A. Experimental Setup

1) Dataset: Given its data-driven nature as a learning-based model, *Soleker* requires adequate training data. The data need to encompass not only a variety of vulnerable contracts but also benign ones. However, no large-scale datasets specifically focused on Solana contract vulnerabilities are currently available. Therefore, we aim to create a suitable dataset ourselves. Because of the time-intensive process of manually constructing a vulnerability dataset, we leverage contextual learning capabilities of large language models (LLMs) [55], [56], [57] to automate the vulnerability injection.

More specifically, the dataset construction for each vulnerability type involves the following three steps: (1) Example preparation: A vulnerable contract from existing data [20] is selected to guide LLM's context learning, capturing real-world vulnerability patterns. Notably, while the vulnerability sample is relatively simple, it effectively reflects real-world vulnerability manifestation and serves as a representative example for context learning in LLMs. (2) Contract generation: The LLM is instructed with the provided example to generate contract code containing the specified vulnerability while enhancing functional diversity by adjusting the temperature parameter. The prompt used for this process is: *"Here is an example of [VUL]: [EXAMPLE]. Please replicate the vulnerability described above, understand its underlying causes, and generate a complex Solana smart contract in Rust containing a [VUL] vulnerability. Ensure that each generated contract is unique in structure and semantics"*. Here, *[VUL]* and *[EXAMPLE]* represent the vulnerability type and the selected example contract, respectively. We also prompt the LLM to generate a fixed version of the contracts with the following instruction: *"Provide a fixed version of the Solana smart contract that addresses the [VUL]"*. The generated contracts are compiled, and any non-compilable code is iteratively refined using error feedback until it becomes compilable. This process is guided by the prompt: *"Ensure the code includes all necessary module imports and compiles in Rust"*. We utilize OpenAI services [58] with GPT-4o as the backend model to generate the dataset. As a result, we can obtain diverse contracts with consistent vulnerability types and labels. (3) Manual verification: Two co-authors independently verify that all compilable contracts implement meaningful functionality and successfully include the specified vulnerability. Meanwhile, regular cross-checks and discussions are conducted to ensure consistency throughout the review process. Only contracts approved by both authors that meet these criteria are retained, ensuring realism and reducing bias.

As a result, we obtain a total of 1,756 labeled smart contracts. Table V presents the dataset statistics, where #LOC denotes the average lines of code and #Func. represents the average number of functions. To the best of our knowledge, this represents the largest Solana contract vulnerability dataset to date, exceeding existing benchmarks [7], [8] by over 100 times (1,756 vs. 15).

2) Implementation: Soleker is implemented in approximately 4,500 lines of code. We utilize the Soufflé framework [59] as the Datalog engine. Each instruction is embedded as a 300-

TABLE V
Statistics of Different Types of Vulnerabilities

Vulnerability	Vulnerable	Non-vulnerable	# LOC	# Func.
MSC	321	310	3,188	67
MOC	147	124	4,262	85
MKC	213	196	3,059	66
ACPI	227	218	4,874	81

dimensional vector while each prefix vector is 100-dimensional. Since the compiled smart contracts are stored as unreadable *.so* binary files, we convert them into SBPF bytecode using the *solana-ledger-tool* [60] with the flags *program disassemble*, to facilitate further analysis. We employ 5-fold cross-validation to minimize bias and report the average results. Our experimental materials can be found at *https://github.com/sepine/Soleker.*

3) Performance metrics: We evaluate the performance of *Soleker* and the baselines using Precision, Recall, and F1-score.

4) Research Questions: This paper aims to answer the following five research questions (RQs). RQ1: How accurate is *Soleker* in detecting vulnerabilities? RQ2: How does each designed component contribute to vulnerability detection? RQ3: How does the imbalance ratio impact *Soleker*? RQ4: What is the time overhead of *Soleker*? RQ5: Can *Soleker* identify vulnerabilities in real-world on-chain smart contracts?

B. RQ1: Effectiveness

1) Approach: The goal of *Soleker* is to detect vulnerabilities from the given Solana SBPF bytecode. To evaluate the effectiveness of *Soleker*, we compare it against the state-of-the-art vulnerability detection tool VRust [7] using our constructed dataset. In addition, we introduce a LLM-based method LLM_{SBPF}, which leverages GPT-4o for vulnerability detection. Specifically, we build on recent work [61] to define LLM's role and limit the scope of vulnerabilities with the following prompt: *"You are a vulnerability detector for a Solana smart contract. Here are four common vulnerabilities: [VULS]. Check the following smart contract bytecode for the above vulnerabilities"*. Here, *[VULS]* represents the four specified types of vulnerabilities. Additionally, we adopt the Chain-of-Thought strategy to enhance reasoning abilities by instructing the model to *"Think step by step, carefully"*. We apply *Soleker* and LLM_{SBPF} to analyze vulnerabilities directly on contract bytecode, while VRust is applied to source code due to its lack of support for bytecode analysis.

2) Results: Table VI demonstrates the evaluation results for *Soleker* and baselines in detecting the four types of vulnerabilities. We can see that *Soleker* achieves average improvements of 169.1%, 129.9%, and 189.3% compared to VRust in detecting MSC, MOC, and ACPI, respectively, across the three metrics. The reason is that VRust relies entirely on fixed static patterns, making it less adaptable to diverse vulnerability representations. In contrast, *Soleker* can capture various vulnerability semantics in the SBPF bytecode, enabling it to detect the vulnerabilities with better performance. Interestingly, VRust demonstrates better performance in identifying missing key checking vulnerabilities compared to detecting others. This is because such key checking operations are mainly associated with the handling of particular function calls in Solana system programs. The limited representation enables VRust's inference rules to identify this type of vulnerability. Nonetheless, our method still achieves an average improvement by 11.4%. Notably, *Soleker* has the capability to analyze the SBPF bytecode of Solana smart contracts, whereas VRust is restricted to merely dealing with the source code. This highlights the practicality of *Soleker* in real-world scenarios, as over 98% of Solana contracts are not open source [5]. Besides, *Soleker* demonstrates average improvements of 156.5%, 126.4%, 132.5%, and 128.6% compared to LLM_{SBPF} in detecting these vulnerabilities. This reveals that while LLMs excel in tasks like code comprehension and vulnerability detection, they struggle with low-level bytecode analysis. These results underscore the need to enhance LLMs' ability to understand bytecode effectively. Overall, *Soleker* achieves average improvements of 178.0%, 68.8%, and 132.4% across the three metrics, respectively.

TABLE VI
COMPARISON RESULTS FOR *Soleker* AND BASELINES

Method	VRust			LLM_{SBPF}			*Soleker*		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
MSC	24.21	49.21	32.46	25.89	50.88	34.32	95.24	94.82	94.86
MOC	39.79	46.41	32.70	29.06	53.90	37.76	91.79	90.61	90.92
MKC	81.86	79.01	78.92	27.12	52.08	35.67	90.20	88.35	88.57
ACPI	19.59	48.33	27.88	29.23	54.06	37.94	92.75	92.12	92.27

Answer to RQ1: Compared to the baseline methods, *Soleker* shows average improvements of 178.0%, 68.8%, and 132.4% in detecting different vulnerabilities.

C. RQ2: Ablation Study

1) Approach: As mentioned in § III, *Soleker* incorporates a runtime analyzer to capture runtime semantics of each instruction for generating node features and develops a prefix extractor to enhance the model with vulnerability-specific localized features. Therefore, we design the following variants to investigate the effectiveness of these components in improving detection performance. The variants comprise: excluding the runtime analyzer and using only the opcode as node features (w/o runtime); removing the prefix extractor and merely relying on snapshot graph embeddings for detection (w/o prefix); and directly utilizing the extracted prefix vectors for detection ($\textit{Soleker}_{wpo}$). We report the average performance in detecting the four types of vulnerabilities.

2) Results: Fig. 5 illustrates the average results in terms of the three metrics. We can observe that discarding the runtime analyzer prevents the model from capturing runtime contextual semantics, resulting in a significant performance drop of 51.2%. Likewise, excluding the extracted prefix vectors leads to a 26.8% performance decline due to the loss of guidance from vulnerability-specific localized features. Moreover, relying solely on prefix vectors fails to capture the rich semantic context of the bytecode, leading to less effective detection. On the whole, the average performance of *Soleker* improves by 31.0%, 31.1%, and 36.6% across the three metrics when all design components are integrated. This indicates that the features extracted from the runtime analyzer and prefix extractor can work in synergy, complementing each other to improve the detection capability of *Soleker*.

Answer to RQ2: Each component within *Soleker* contributes to enhancing detection performance.

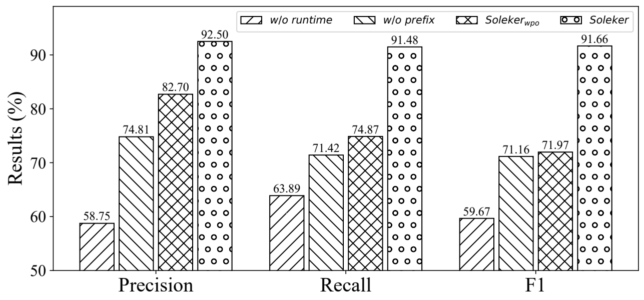


Fig. 5. Ablation results of *Soleker*.

D. RQ3: Impact of Imbalance Ratio

1) Approach: Since real-world vulnerabilities always exhibit highly imbalanced distributions, this question is designed to explore the effectiveness of *Soleker* in such scenarios. More specifically, we employ random sampling to systematically adjust the dataset, reflecting varying ratios of vulnerable to non-vulnerable samples. Starting with a highly imbalanced ratio of 0.1, we incrementally increase the ratio to 0.9, approaching a more balanced distribution. This sampling procedure is repeated five times for each ratio setting to minimize sampling bias. We report the average results for each vulnerability type.

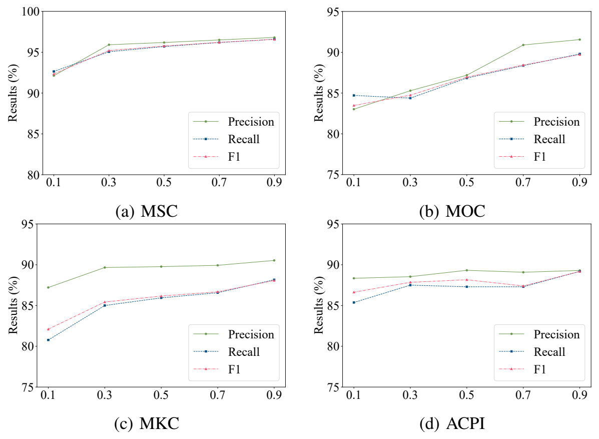


Fig. 6. Impacts of varying imbalance ratios.

2) Results: Fig. 6 presents the average results under different imbalance ratios. We can find that *Soleker* consistently demonstrates better detection performance across varying imbalance ratios, achieving average precision, recall, and F1 scores of 90.36%, 88.67%, and 88.85%, respectively, across all four types of vulnerabilities. This is because the design of *Soleker* successfully captures the distinguishable features between vulnerable and non-vulnerable samples, enabling the model to maintain its detection capability. It also demonstrates that *Soleker* remains effective even in real-world scenarios with highly imbalanced sample ratios.

Answer to RQ3: *Soleker* can effectively detect vulnerabilities under various imbalance ratios.

E. RQ4: Overhead

1) Approach: *Soleker* integrates program analysis with deep learning models to detect vulnerabilities in Solana contract bytecode. This leads us to explore the efficiency of *Soleker* in detecting vulnerabilities. Therefore, we record the execution time consumed by each component of *Soleker* during the experiments, namely $Soleker_{SB}$, $Soleker_{RA}$, $Soleker_{PE}$, and $Soleker_{VD}$, respectively. In addition, we also record the processing time of VRust for comparison. To ensure a fair comparison, all methods are executed on the same machine. We present their average time consumption in seconds.

2) Results: Table VII illustrates the average time overhead of each component within *Soleker* and VRust. The average process time is 0.125 seconds for the snapshot builder, 0.380 seconds for the runtime analyzer, 0.075 seconds for the prefix extractor, and 0.001 seconds for the vulnerability detector. Furthermore, compared to VRust, *Soleker* is on average over 335 times faster when analyzing a contract. While training the model requires more time, it is merely a one-time process. The rapid runtime analysis and swift prediction capabilities of *Soleker* enable it to provide near-instantaneous responses in real-world detection scenarios.

TABLE VII
TIME OVERHEAD

Component	$Soleker_{SB}$	$Soleker_{RA}$	$Soleker_{PE}$	$Soleker_{VD}$	*Soleker*	VRust
Time (s)	0.125	0.380	0.075	0.001	0.581	195.466

Answer to RQ4: *Soleker* can inspect contract bytecode in an average of 0.581 seconds, making it over 335 times faster than the baseline.

F. RQ5: Vulnerabilities in the Wild

1) Approach: To assess the capability of *Soleker* in identifying unknown vulnerabilities beyond the labeled dataset, we queried a RPC node on the Solana mainnet to identify programs that belong to the BPF Upgradeable Loader (BPFLoaderUpgradeab1e11111111111111111111111) as of November 2024. Due to Solana's rapid block generation and high transaction volume, data retrieval is limited by speed. Thus, we opt to randomly collect 2,000 on-chain contracts as a balanced trade-off. Since all these contracts are stored as the base64-encoded binaries, we first decode them into *.so* ELF files and then disassemble them into SBPF bytecode using the official Solana toolkit [60]. We then train *Soleker* on our labeled dataset and use the resulting model to inspect the bytecode. Additionally, we further conduct a manual analysis of the results.

2) Results: *Soleker* reports 118 vulnerable on-chain contracts, including 54 missing signer checking, 52 missing owner checking, and 43 arbitrary cross program invocations. Among the remarked vulnerable contracts, we discover that 15.3% of them are still active on the Solana mainnet, while the remaining contracts have either been abandoned or upgraded as recorded in the Solana explorer [62]. Although Solana supports the *security.txt* standard for embedding contact information in smart contracts, none of the analyzed contracts included it, leaving developers anonymous. Therefore, we are unable to directly disclose our findings to the respective developers.

Manual Analysis. As highlighted in the previous study, confirming vulnerabilities poses a significant challenge because of the low-level bytecode representation and the unavailability of source code [8]. Therefore, we randomly select fifteen vulnerable samples and fifteen benign samples flagged by *Soleker*, and try our best to manually analyze the corresponding disassembled SBPF bytecode. We find two false positives and zero false negatives among these samples. Listing 5 demonstrates an example that involves arbitrary cross program invocations and is successfully detected by *Soleker*. More detailed instructions are omitted for simplicity. *Soleker* analyzes semantic information captured from the instructions associated with the system call *sol_invoke_signed_rust* to identify this vulnerability. As for the false positives, we observe that the corresponding SBPF bytecode involves complex controls and many byte-by-byte checking with similar patterns. These factors prevent *Soleker* from understanding deeper semantics and lead to the misidentification.

```
entrypoint:
  ... ; Without checking the expected key
  call function_1558
function_1558: ...
  ldxdw r1, [r10-0x50]  ldxdw r2, [r10-0x48]
  ldxdw r3, [r10-0x40]  ldxdw r4, [r10-0x88]
  ldxdw r5, [r10-0x80]  ; Invocation parameters
  syscall sol_invoke_signed_rust
```

Listing 5. An example of ACPI detected by *Soleker*.

Answer to RQ5: *Soleker* can uncover vulnerabilities in on-chain smart contracts.

V. Discussion

First, to generate the initial node features, we develop a lightweight symbolic analyzer to capture the runtime semantics of each instruction. Since the low-level instructions will handle memory operations, some memory locations may be lost during symbolic analysis. However, since our primary goal is to capture the runtime semantics of each instruction, we utilize a merge operation to consolidate information from all possible flows during the simulation execution. In addition, to mitigate the issue of potential infinite loops during simulation execution, we set a maximum depth limit of 100 to balance infinite loop prevention with analytical capability.

Second, as the development language for Solana contracts, Rust supports multiple levels of compilation optimization, which may result in variations in the generated bytecode. Fortunately, contract compilation on Solana utilizes the official toolkit [60], which employs the `--release` flag to enable high-level optimizations (i.e., `opt-level=3`) by default [63]. This ensures that the resulting bytecode is consistently optimized to a standardized level. Accordingly, *Soleker* is evaluated on highly optimized contract bytecode.

Third, synthetic data is a common solution to address the scarcity of real-world datasets, therefore, we leverage LLMs for this purpose. However, this approach will introduce several potential biases. Source bias stems from the reliance on the LLM's training corpus, which may fail to fully capture the breadth and diversity of real-world Solana vulnerabilities. Prompting bias is another concern, as the phrasing and structure of prompts could inadvertently lead LLMs away from the intended generation objectives. To relieve these biases, we limit the scope of vulnerabilities and guide the LLM's generation process using real-world vulnerability examples. Moreover, contract diversity is improved by adjusting the temperature parameter. Additionally, all generated data undergoes manual validation to ensure it closely reflects real-world scenarios.

Fourth, *Soleker* currently focuses on four types of vulnerabilities because of their prevalence and significant impact on the Solana ecosystem. However, *Soleker* can be applied to other vulnerabilities that may become prevalent in the future, with additional effort to analyze vulnerability-specific prefixes.

VI. Related Work

There are a lot of studies that primarily focused on detecting vulnerabilities in Ethereum smart contracts by utilizing various analysis techniques, such as static code analysis [64], [65], symbolic execution [66], [67], [68], fuzzing [69], [70], [71], formal verification [16], [17], [72], and deep learning [11], [73], [74]. However, they cannot be applied to the vulnerability detection in the Solana ecosystem due to the different implementations and runtime semantics.

Apart from the work on vulnerability detection in Solana smart contracts discussed in § I, other related studies have investigated various aspects of the Solana ecosystem, such as the evolutionary process from Bitcoin to Solana [75], scalability challenges [76], code reuse analysis [77], transaction analysis [2], and security concerns from the developer's perspective [5]. Furthermore, some efforts have explored the Solana eBPF runtime through differential fuzzing [78] and formal semantics verification [79]. In contrast, our work focuses on detecting vulnerabilities in the bytecode of Solana smart contracts, serving as a complement to existing studies.

VII. Conclusion

This paper presents *Soleker*, an innovative learning-based solution for automated vulnerability detection in Solana smart contract bytecode. This approach first captures the runtime semantics of instructions related to blockchain interactions, extracts localized vulnerability-specific features, and then introduces a prefix-guided neural network for feature learning and vulnerability detection. We create the first labeled dataset of Solana smart contract vulnerabilities to evaluate the performance of *Soleker*. Extensive experiments demonstrate that *Soleker* outperforms the baseline methods in both detection effectiveness and efficiency, with each crafted component contributing to its success. In the future, we plan to explore other security threats within the Solana ecosystem. In addition, we will expand our tool to support vulnerability detection beyond the Solana platform.

Acknowledgment

We sincerely thank the anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong ITF Project (No. PRP/005/23FX) and Hong Kong RGC Projects (PolyU15224121, PolyU15231223).

References

[1] S. F. Homepage, "Solana," https://solana.com/, 2025.

[2] X. Zheng, Z. Wan, D. Lo, D. Xie, and X. Yang, "Why does my transaction fail? a first look at failed transactions on the solana blockchain," in *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2025.

[3] S. L. T. Fees, "Solana transaction fees," https://github.com/solana-labs/solana, 2025.

[4] E. Olaoluwa, "Comparing network fees," https://cwallet.com/blog/comparing-network-fees-ethereum-vs-bsc-vs-polygon-vs-solana/, 2023.

[5] S. Andreina, T. Cloosters, L. Davi, J.-R. Giesen, M. Gutfleisch, G. Karame, A. Naiakshina, and H. Naji, "Defying the odds: Solana's unexpected resilience in spite of the security challenges faced by developers," in *Proceedings of the 31st ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 4226–4240.

[6] S. F. Accounts, "Solana account model," https://solana.com/docs/core/accounts, 2025.

[7] S. Cui, G. Zhao, Y. Gao, T. Tavu, and J. Huang, "Vrust: Automated vulnerability detection for solana smart contracts," in *Proceedings of 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022, pp. 639–652.

[8] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, and K. Pohl, "Fuzz on the beach: Fuzzing solana smart contracts," in *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023, pp. 1197–1211.

[9] M. White, "Oracle attack on solend costs the project 1.26 million," https://www.web3isgoinggreat.com/?blockchain=solana&id=oracle-attack-on-solend-costs-the-project-1-26-million, 2022.

[10] H. Hofstadt, "Check instructions sysvar," https://github.com/wormhole-foundation/wormhole/commit/e8b91810a9bb35c3c139f86b4d0795432d647305, 2023.

[11] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[12] T. Wong, C. Zhang, Y. Ni, M. Luo, H. Chen, Y. Yu, W. Li, X. Luo, and H. Wang, "Confuzz: Towards large scale fuzz testing of smart contracts in ethereum," in *Proceedings of the 2024 IEEE Conference on Computer Communications (INFOCOM)*, 2024, pp. 1691–1700.

[13] R. Liang, J. Chen, C. Wu, K. He, Y. Wu, R. Cao, R. Du, Z. Zhao, and Y. Liu, "Vulseye: Detect smart contract vulnerabilities via stateful directed graybox fuzzing," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2025.

[14] Z. Ma, M. Jiang, X. Luo, H. Wang, and Y. Zhou, "Uncovering nft domain-specific defects on smart contract bytecode," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2025.

[15] W. Li, X. Li, Z. Li, and Y. Zhang, "Cobra: interaction-aware bytecode-level vulnerability detector for smart contracts," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2024, pp. 1358–1369.

[16] J. Frank, C. Aschermann, and T. Holz, "Ethbmc: A bounded model checker for smart contracts," in *Proceedings of the 29th USENIX Security Symposium (USENIX SEC)*, 2020, pp. 2757–2774.

[17] J. Stephens, K. Ferles, B. Mariano, S. Lahiri, and I. Dillig, "Smartpulse: automated checking of temporal properties in smart contracts," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, 2021, pp. 555–571.

[18] J. Jiang, X. Peng, J. Chu, and X. Luo, "Conscs: Effective and efficient verification of circom circuits," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 737–737.

[19] X. Peng, Z. Sun, K. Zhao, Z. Ma, Z. Li, J. Jiang, X. Luo, and Y. Zhang, "Automated soundness and completeness vetting of polygon zkevm," in *Proceedings of the 34th USENIX Security Symposium (USENIX SEC)*, 2025, pp. 4093–4108.

[20] Neodyme, "Solana security workshop," https://workshop.neodyme.io/, 2022.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[22] S. F. Programs, "Programs on solana," https://solana.com/docs/core/programs, 2025.

[23] Z. Li, Z. He, X. Luo, T. Chen, and X. Zhang, "Finding correctness issues on ethereum verkle tries via preimage-aware differential testing," *IEEE Transactions on Software Engineering (TSE)*, 2025.

[24] S. Labs, "rbpf," https://github.com/solana-labs/rbpf, 2025.

[25] S. Foundation, "Transfer hook," https://solana.com/developers/courses/token-extensions/transfer-hook#2-transfer-hook-instruction, 2025.

[26] S. F. PDA, "Program derived address (pda)," https://solana.com/docs/core/pda, 2025.

[27] S. F. CPI, "Cross program invocation (cpi)," https://solana.com/docs/core/cpi, 2025.

[28] Smartstate, "Solana common crypto project vulnerabilities," https://smartstate.tech/blog/6-solana-common-crypto-project-vulnerabilities.html, 2024.

[29] Helius, "A hitchhiker's guide to solana program security," https://www.helius.dev/blog/a-hitchhikers-guide-to-solana-program-security, 2024.

[30] Cyberscope, "How secure are solana smart contracts?" https://cyberscope.medium.com/how-secure-are-solana-smart-contracts-cbbc12be5aad, 2025.

[31] Auditfirst, "Common vulnerabilities in solana programs (smart contracts)," https://auditfirst.io/blog/common-vulnerabilities-in-solana-programs, 2025.

[32] h4rkl and E. Keddell, "Create a basic program, part 3 - basic security and validation," https://solana.com/developers/courses/native-onchain-development/program-security, 2025.

[33] E. Keddell and h4rkl, "Arbitrary cpi," https://solana.com/developers/courses/program-security/arbitrary-cpi#lesson, 2025.

[34] R. Behnke, "Explained: The cashio hack (march 2022)," https://www.halborn.com/blog/post/explained-the-cashio-hack-march-2022, 2022.

[35] Slowmint, "Crema finance exploit," https://www.certik.com/resources/blog/crema-finance-exploit, 2022.

[36] B. Newar, "Slope wallets blamed for solana-based wallet attack," https://cointelegraph.com/news/slope-wallets-blamed-for-solana-based-wallet-attack, 2022.

[37] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 239–248.

[38] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *Proceedings of the 41st IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1176–1186.

[39] Q. Shi, X. Xie, X. Fu, P. Di, H. Li, A. Zhou, and G. Fan, "Datalog-based language-agnostic change impact analysis for microservices," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2025, pp. 652–652.

[40] Z. Ma, M. Jiang, F. Luo, X. Luo, and Y. Zhou, "Surviving in dark forest: Towards evading the attacks from front-running bots in application layer," in *Proceedings of the 34th USENIX Security Symposium (USENIX SEC)*, 2025, pp. 1–18.

[41] H. Yi and W. Hickey, "Solana system calls," https://github.com/solana-labs/solana/tree/master/programs/bpf_loader, 2025.

[42] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "Deepinfer: Deep type inference from smart contract bytecode," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, pp. 745–757.

[43] L. Gao, Y. Qu, S. Yu, Y. Duan, and H. Yin, "Sigmadiff: Semantics-aware deep graph matching for pseudocode diffing," in *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS)*, 2024, pp. 1–19.

[44] K. Zhao, Z. Li, W. Chen, X. Luo, T. Chen, G. Meng, and Y. Zhou, "Recasting type hints from webassembly contracts," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE)*, 2025, pp. 2665–2688.

[45] Ghidra, "Automated struct identification with ghidra," https://blog.grimm-co.com/2020/11/automated-struct-identification-with.html, 2020.

[46] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 2nd International Conference on Machine Learning (ICLR)*, 2014, pp. 1188–1196.

[47] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL-IJCNLP)*, 2021, pp. 4582–4597.

[48] K. Zhou, J. Yang, C. C. Loy, and Z. Liu, "Learning to prompt for vision-language models." *International Journal of Computer Vision (IJCV)*, vol. 130, no. 9, pp. 2337–2349, 2022.

[49] H. Zhang, Z. Li, P. Wang, and H. Zhao, "Selective prefix tuning for pre-trained language models," in *Findings of the Association for Computational Linguistics (ACL-Findings)*, 2024, pp. 2806–2813.

[50] Z. Guo, J. Li, G. Li, C. Wang, S. Shi, and B. Ruan, "Lgmrec: local and global graph learning for multimodal recommendation," in *Proceedings of the 38th AAAI Conference on Artificial Intelligence (AAAI)*, vol. 38, no. 8, 2024, pp. 8454–8462.

[51] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[52] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," in *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016.

[53] H. Wang, R. Yang, K. Huang, and X. Xiao, "Efficient and effective edge-wise graph representation learning," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2023, pp. 2326–2336.

[54] Y. Zhou, T. Lei, H. Liu, N. Du, Y. Huang, V. Zhao, A. M. Dai, Q. V. Le, J. Laudon *et al.*, "Mixture-of-experts with expert choice routing," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 7103–7114, 2022.

[55] R. Agarwal, A. Singh, L. Zhang, B. Bohnet, L. Rosias, S. Chan, B. Zhang, A. Anand, Z. Abbas, A. Nova *et al.*, "Many-shot in-context learning," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 37, pp. 76 930–76 966, 2024.

[56] J. Park, J. Park, Z. Xiong, N. Lee, J. Cho, S. Oymak, K. Lee, and D. Papailiopoulos, "Can mamba learn how to learn? a comparative study on in-context learning tasks," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024, pp. 39 793–39 812.

[57] J. Jiang, Z. Li, H. Qin, M. Jiang, X. Luo, X. Wu, H. Wang, Y. Tang, C. Qian, and T. Chen, "Unearthing gas-wasting code smells in smart contracts with large language models," *IEEE Transactions on Software Engineering (TSE)*, 2024.

[58] Microsoft, "Azure openai service," https://azure.microsoft.com/en-us/products/ai-services/openai-service, 2025.

[59] T. S. Project, "Soufflé: Logic defined static analysis," https://souffle-lang.github.io/, 2025.

[60] S. Labs, "Solana," https://github.com/solana-labs/solana, 2025.

[61] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 34, no. 4, pp. 1–30, 2025.

[62] SolanaFM, "A friendly solana explorer." https://solana.fm/?cluster=mainnet-alpha, 2025.

[63] T. R. P. Language, "The cargo book," https://doc.rust-lang.org/cargo/reference/profiles.html#release, 2025.

[64] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2nd IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019, pp. 8–15.

[65] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[66] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications security (CCS)*, 2016, pp. 254–269.

[67] C. Diligence, "Mythril," https://github.com/ConsenSysDiligence/mythril, 2025.

[68] J. Huang, J. Jiang, W. You, and B. Liang, "Precise dynamic symbolic execution for nonuniform data access in smart contracts," *IEEE Transactions on Computers (ToC)*, vol. 71, no. 7, pp. 1551–1563, 2021.

[69] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269.

[70] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 227–239.

[71] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 322–333.

[72] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 67–82.

[73] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *Proceedings of the IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 2021, pp. 378–389.

[74] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2024, pp. 1–13.

[75] X. Li, X. Wang, T. Kong, J. Zheng, and M. Luo, "From bitcoin to solana–innovating blockchain towards enterprise applications," in *Proceedings of the International Conference on Blockchain*, 2021, pp. 74–100.

[76] G. A. Pierro and R. Tonelli, "Can solana be the solution to the blockchain scalability problem?" in *Proceedings of the 29th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1219–1226.

[77] C. Wang, Y. Li, J. Gao, K. Wang, J. Zhang, Z. Guan, and Z. Chen, "Solasim: Clone detection for solana smart contracts via program representation," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2024, pp. 258–269.

[78] C. Peng, M. Jiang, L. Wu, and Y. Zhou, "Toss a fault to bpfchecker: Revealing implementation flaws for ebpf runtimes with differential fuzzing," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024, pp. 3928–3942.

[79] S. Yuan, Z. Zhang, J. Lu, D. Sanan, R. Chang, and Y. Zhao, "A complete formal semantics of ebpf instruction set architecture for solana," *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 9, no. OOPSLA1, pp. 1–27, 2025.