Hyperion: Unveiling DApp Inconsistencies using LLM and Dataflow-Guided Symbolic Execution

Shuo Yang†, Xingwei Lin‡, Jiachi Chen†*, Qingyuan Zhong†, Lei Xiao†, Renke Huang†,
Yanlin Wang†, Zibin Zheng†
† Sun Yat-sen University, {yangsh233, zhongqy, xiaolei6, huangrk9}@mail2.sysu.edu.cn
{chenjch86, wangylin36, zhzibin}@mail.sysu.edu.cn
‡ Zhejiang University, xwlin.roy@zju.edu.cn

***Abstract*—The rapid advancement of blockchain platforms has significantly accelerated the growth of decentralized applications (DApps). Similar to traditional applications, DApps integrate front-end descriptions that showcase their features to attract users, and back-end smart contracts for executing their business logic. However, inconsistencies between the features promoted in front-end descriptions and those actually implemented in the contract can confuse users and undermine DApps's trustworthiness.**

In this paper, we first conducted an empirical study to identify seven types of inconsistencies, each exemplified by a real-world DApp. Furthermore, we introduce HYPERION, an approach designed to automatically identify inconsistencies between front-end descriptions and back-end code implementation in DApps. This method leverages a fine-tuned large language model LLaMA2 to analyze DApp descriptions and employs dataflow-guided symbolic execution for contract bytecode analysis. Finally, HYPERION reports the inconsistency based on predefined detection patterns. The experiment on our ground truth dataset consisting of 54 DApps shows that HYPERION reaches 84.06% overall recall and 92.06% overall precision in reporting DApp inconsistencies. We also implement HYPERION to analyze 835 real-world DApps. The experimental results show that HYPERION discovers 459 real-world DApps containing at least one inconsistency.

***Index Terms*—smart contract, LLM, inconsistency detection, dataflow analysis, symbolic execution**

I. INTRODUCTION

In recent years, the blockchain industry has witnessed a remarkable proliferation of various decentralized applications (DApps), which have gained substantial popularity and market capitalization [1]–[3]. Similar to traditional applications like web or Android apps, a DApp utilizes a user interface (UI) as its front-end to present feature descriptions and employs smart contracts for back-end logic execution.

However, inconsistency between the front-end descriptions and the actual implementation of their back-end contract code of DApps may have detrimental consequences [4]. For example, a DApp might advertise a 3% investment return in its description, but the actual return could be changed (lower than 3%) after investment. Similarly, in the case of some NFT DApps, the front-end description may claim that NFTs can "live forever", while in reality, the metadata of these NFTs is stored on centralized servers, which are susceptible to shutdown. These inconsistencies can pose threats to users' interests and undermine the trustworthiness of DApps.

* Jiachi Chen is the corresponding author.

Many research efforts have been made to detect vulnerabilities in smart contracts to ensure their safety [3], [5]–[8]. However, they usually overlook the importance of inconsistencies between the front-end descriptions and back-end implementations in DApps. For instance, consider a scenario where a DApp offers users a 2% interest rate and does not mention fees; traditional vulnerability detection tools may not flag this as an issue. However, when the promised rates in the DApp's description do not align with this or fees are charged secretly, it can indicate dishonest behavior of the DApps' owners and pose potential risks to DApp users.

The detection of DApp inconsistencies presents unique challenges. First, there is no systematic work aimed at uncovering the various types of inconsistencies within the DApp ecosystem. This absence makes it not easy to design rules to detect inconsistencies. Second, the diverse and intricate structures of DApp front-end descriptions present considerable difficulties when attempting to extract information related to user assets. Third, it is challenging to recover the semantics from the contract code effectively. Many malicious DApp developers may not disclose their source code on Ethereum [9]. Consequently, detecting inconsistencies from the bytecode level is crucial, but also increases the difficulty.

To address these challenges, we first conducted an empirical study employing an open card sorting approach [10] to find DApp inconsistencies. This involved analysis of both the front-end descriptions and the smart contracts of a dataset comprising 321 DApps (see Section III). Based on this approach, we define seven types of DApp inconsistencies that encompass both DeFi and NFT DApps, namely *Unguaranteed Reward*, *Hidden Fee*, *Adjustable Liquidity*, *Unconstrained Token Supply*, *Unclaimed Fund Flow*, *Changeable DApp Status*, and *Volatile NFT Accessibility*.

To automatically detect these inconsistencies, we propose a tool named HYPERION, which contains a description analyzer HYPERTEXT and a contract analyzer HYPERCODE. HYPERTEXT leverages LLaMA2 [11], a widely-recognized, public-available large language model [12] (LLM), to scrutinize the diverse natural language descriptions found on DApp websites. To optimize LLaMA2 for the specific domain task, i.e., extract critical attributes from the DApp front-end descriptions, we first design a prompt template based on the chain of thought [13] (CoT) prompting technique, tailored

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00015

to our identified inconsistency types. Then, we manually label a dataset containing 63 DApp inconsistencies to fine-tune LLaMA2 (see Section IV-B1), thereby enhancing its performance for our downstream task and obtaining our model HYPERTEXT. Finally, we employ the Natural Language Tool Kit [14] (NLTK) to extract inconsistency-related attributes from HYPERTEXT's output (see Section IV-B2).

For the analysis of DApp contract bytecode, HYPERCODE employs an approach that utilizes dataflow analysis to guide symbolic execution on the contract intermediate representation (IR), so as to obtain relevant program states and recover contract semantics (see Section IV-C). Specifically, HYPERCODE first decompiles the contract bytecode and performs dataflow analysis to recover low-level semantics such as contract call and state variable storage access operations. Then, it performs a graph analysis based on the recovered semantics to obtain the contract's fund transfer and state variable dependency relationship. To obtain contract semantics, HYPERCODE proposes a symbolic execution framework based on contract IR. This framework utilizes dataflow and graph analysis to optimize search paths and direct symbolic execution to check relevant program states, merging the strengths of both methods. Furthermore, we propose algorithms that compare the information extracted from DApp descriptions and contract semantics based on this framework to detect the seven defined inconsistencies, unveiling whether one DApp's *saying* is consistent with its *doing*.

To evaluate the performance of HYPERION, we first collect two datasets; one serves as the ground truth dataset, containing 54 labeled DApps used to define DApp inconsistencies and evaluate the effectiveness of the HYPERION. The other dataset is used to evaluate the performance of HYPERION on analyzing real-world wild DApps, which contains 835 real-world DApps collected from DappBay [15] and DappRadar [16]. HYPERION reaches an overall recall of 84.06% and an overall precision of 92.06% in the first ground truth dataset. Furthermore, HYPERION identifies 459 out of 835 DApps with at least one inconsistency that we define in the second large-scale dataset, with an overall precision of 92.10% in our sampled dataset. Among the true positives (268 DApps) in the sampled dataset, we find that 67 (25%) DApps' websites become inaccessible within only 3 months.

In summary, the contributions of our paper are as follows:

- We define seven common inconsistency types between DApp front-end description and back-end smart contract implementation. To enhance comprehension, we illustrate each inconsistency with a real-world DApp example and its consequences.
- We design and develop HYPERION, a tool that leverages LLM and dataflow-guided symbolic execution to automatically detect defined inconsistencies.
- We find 459 out of 835 DApps in our dataset contain at least one inconsistency, which demonstrates the prevalence of the defined inconsistencies in the DApp ecosystem.
- To promote further research and transparency, we have released the source code of HYPERION (both fine-tuned HYPERTEXT and HYPERCODE) as well as the DApp dataset resources and experimental results in the repository https://github.com/shuo-young/Hyperion.

II. BACKGROUND

A. DApp and Smart Contract

Decentralized applications (DApps) typically leverage smart contract technology to achieve autonomous and transparent operations [17], [18]. A DApp usually uses a user interface (UI) that showcases its functionalities to attract users. Some financial-related features, such as return on investment (ROI), token supply, and liquidity, are central to user interest and are implemented within the smart contract. Smart contracts are commonly written in high-level programming languages, e.g., Solidity [19]. Ethereum Virtual Machine (EVM) is a stack-based virtual machine that executes the contract EVM bytecode. The EVM executes transactions by splitting contract EVM bytecode into operation codes (opcodes), each with specific execution instructions.

B. Large Language Model

Recently, Large language models (LLMs) have exhibited remarkable proficiency in natural language understanding [20]. Notably, ChatGPT [12] is distinguished by its advanced performance. However, it faces limitations such as limited availability in some regions and high API usage fees. In contrast, LLaMA [11] offers advantages in terms of transparency, adaptability, cost-free access, and robust performance in natural language tasks. These features make LLaMA an ideal model for complicated or unstructured natural language understanding. Additionally, the ability of LLaMA in downstream tasks can be further enhanced by its fine-tuning capabilities, a process of training the model on specific task-related labeled data in a supervised manner [21], [22]. This adaptability is further amplified in the context of instruction-tuning [23]–[25], which trains models to interpret and act on explicit instructions in diverse tasks. Numerous models have been developed based on LLaMA, including Code LLaMA [26], optimized for understanding and generating programming code, and LLaMA2 [27], which offers enhanced capabilities for natural language processing. Given our specific objective of comprehending intricate natural language data in DApp descriptions, we select LLaMA2 as our base model.

III. INCONSISTENCIES DEFINITION

Figure 1 illustrates how we define the seven types of DApp inconsistencies. First, we collected DApp descriptions and corresponding smart contracts from two platforms. Then, we employed the open card sorting approach [10] to analyze the collected data manually. Finally, we defined seven DApp inconsistencies, each illustrated with a real-world DApp example.

A. Data Collection

We first crawled all 576 DApps labeled as *high-risk* or *red-alarm* by DappRadar [16] and DappBay [15], two main platforms that offer detailed information on DApps. We then removed 255 DApps whose websites are inaccessible for further

analysis and manually collected the HTML of each DApp by visiting their official websites. Next, we extracted related smart contract addresses from these websites and collected their source codes from Etherscan. We finally obtained descriptions and smart contracts of 321 DApps.

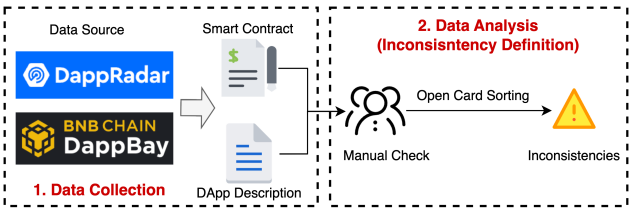


Fig. 1. Workflow of defining DApp inconsistencies.

B. Data Analysis

To find DApp inconsistencies, three authors use the open card sorting approach [10], which is widely adopted in problem-finding and definition in software engineering [3], [28]. In this process, we first created a card that contains the front-end description and the contract code for each DApp. We considered two aspects to ensure the significance of the inconsistency issues, namely user financial-related information in DApp descriptions (e.g., promised return on investment, claimed fees charged, and the stability of the token economy) and corresponding financial-related code logic in smart contract implementations. These aspects are highly related to users' profits or assets and are usually the major concern of users. Then, three authors worked together to determine the labels for each card. They followed the detailed steps illustrated in the previous problem definition works [3], [28].

There are three steps in the open card sorting process. In the first step, they randomly chose 40% of the cards and read the DApp descriptions. They extracted the claims that are related to users' funds or assets, e.g., *our DApp pays 8% daily for users*, or *we charge another 4% fee for marketing*, etc. Similarly, they read the contract code and focused on the implementations related to the fund transfer or tokens. They checked the transfer target, the transfer amount, and other attributes related to the assets. For those contracts without source code, they used an online decompiler [29] and analyzed the contract IR. Based on the two aspects of information extracted from the DApp description and the smart contract, they compared them and understood the inconsistencies.

In the second step of card sorting, the same three authors analyzed and categorized the remaining 60% cards independently, following the same steps as in the first round. Meanwhile, if they encountered new descriptions or contract implementation related to users' funds or assets other than what they found in the first round (e.g., daily reward), they kept the cards to the final discussion.

In the third step, they compared the results, discussed the differences, and finally identified seven types of inconsistencies across 54 DApps from all the collected 576 DApps. The remaining 267 DApps found no further inconsistencies or had no financial-related codes or descriptions during the open card sorting process. We established the mapping relationship between the filtered DApp and the defined inconsistencies, which is publicly available in our repository.

C. Definition of DApp Inconsistencies

Table I shows the definition of seven inconsistency types between the DApp front-end descriptions and smart contracts. The third column highlights the inconsistency types, and to clearly show each inconsistency, we list both the illustration of front-end descriptions and the implementation of smart contracts (columns fourth and fifth). Our defined inconsistencies cover NFT and DeFi DApps.

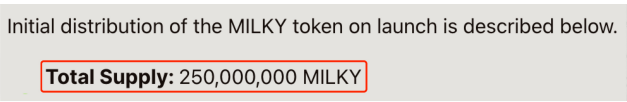
The following paragraphs provide the corresponding detailed definition and example for each inconsistency.

(1) Unguaranteed Reward (UR): Rewards are pivotal in incentivizing user investment in DApps. Some DApps advertise high rewards to attract investments, but inconsistencies between these advertised rewards and the contract's actual implementation can deceive users. Such inconsistencies might lead to users not receiving the guaranteed rewards or even resulting in financial loss.


```
1 function calculateEggSell(uint256 eggs) public view
      returns(uint256){
2   return calculateTrade(eggs,marketEggs,address(this
      ).balance);}
3 function sellEggs() public{
4   uint256 eggValue=calculateEggSell(hasEggs);
5   uint256 baseFee=devFee(eggValue);
6   mrkoneAddress.transfer(baseFee*25/10); // 2.5%
7   mrktwoAddress.transfer(baseFee*25/10); // 2.5%
8   devAddress.transfer(baseFee*1); // 1%
9   msg.sender.transfer(SafeMath.sub(eggValue,baseFee
      *6));}
```

Fig. 2. DApp descriptions (top) and contract snippet (bottom) of *BNB Ultra*.

Example: Descriptions on *BNB Ultra* [30] (Figure 2) might mislead users that they can obtain a daily profit of 3%. An examination of the contract [31] reveals that the reward is not guaranteed. Specifically, line 9 indicates that the user reward comprises two components, i.e., the value of *eggs* and *baseFee*. The value of *eggs* is determined by the function *calculateTrade()* (line 2), depending on variable `marketEggs` and contract balance. This implementation means that the promised 3% reward rate is subject to undetermined variable factors, contradicting the front-end claim.




```
1 function mint(uint256 amount) public onlyOwner
      returns (bool) {
2   _mint(_msgSender(), amount);
3   return true;}
4 function _mint(address account, uint256 amount)
      internal {...
5   _totalSupply = _totalSupply.add(amount);
6   _balances[account] = _balances[account].add(amount
      );}
```

Fig. 3. DApp description (top) and contract snippet (bottom) of *MILKY* token.

(2) Hidden Fee (HF): DApps often charge fees to maintain their operations, compensating for resources such as management efforts. However, issues arise when these fees are either

TABLE I
DEFINITIONS OF THE 7 DAPP INCONSISTENCIES.

General Type	DApp Type	Inconsistency Type	Front-end Description	Smart Contract Implementation
Mathematics	DeFi	*Unguaranteed Reward (UR)*	Claim a reward of x%	Reward rate is not a guaranteed constant value
		Hidden Fee (HF)	Not claim fee or claim fee as x%	Fees are charged with a rate of y% (y!=x)
Transparency		*Adjustable Liquidity (AL)*	Claim x years of liquidity lock time	The lock time are adjustable to be less than x
	DeFi/NFT	*Unconstrained Token Supply (UTS)*	Claim a token supply of x	The supply can be bigger than x
		Unclaimed Fund Flow (UFF)	Not claim the possible fund flow to the project owner	The contract fund can be withdrawn by specific users
		Changeable DApp Status (CDS)	Not claim the possible pause of the DApp	The DApp can be paused to impede trading
	NFT	*Volatile NFT Accessibility (VNA)*	Not claim the accessibility of NFTs or claim they are accessible	The NFTs can be inaccessible

undisclosed on the DApp's front-end description or differ from the actual contract implementation, which can unexpectedly impact user assets and compromise the DApp's trustworthiness.

Example: Returning to *BNB Ultra* [30], the final reward for users is reduced by fees directed to predetermined addresses (Figure 2, line 9). Specifically, lines 6-8 show fees being allocated to three different addresses, with the total fee amount determined by the *devFee()* function (line 5). However, this fee structure [31] is not disclosed in the DApp's front-end description (Figure 2), resulting in a *Hidden Fee* inconsistency.

(3) Unconstrained Token Supply (UTS): Many DApps have their native tokens, e.g., Aave [32] and BAYC [33]. Setting the maximum supply of tokens can uphold tokens scarcity [3], stabilizing the market. Inconsistencies between the stated token supply and actual contract code can detrimentally affect token holders by affecting the token's value and scarcity [34].

Example: Descriptions about the *MILKY* token [35] claims a total supply of 250M (Figure 3). However, the contract [36] (line 2) allows one to mint tokens without a supply cap.

(4) Adjustable Liquidity (AL): Liquidity lock time refers to a period during which some users (e.g., token project managers) cannot withdraw their tokens. The liquidity lock time in DeFi DApps is crucial for stability [37], offering a stable pricing environment and encouraging prolonged asset holding. Inconsistencies in this aspect, such as unaligned claims or adjustable lock times, may cause market instability and compromise the interests of investors.

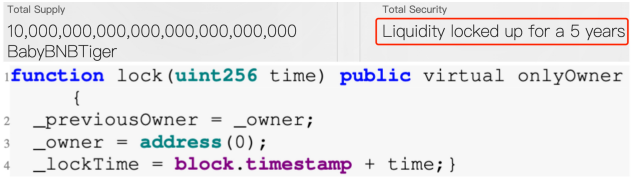


Fig. 4. Description (top) and contract snippet (bottom) of *Baby BNB Tiger*.

Example: *Baby BNB Tiger* claims a 5-year liquidity lock [38] (Figure 4). However, its contract [39] allows the owner to modify this time (line 4) through the *lock()* function arbitrarily.

(5) Unclaimed Fund Flow (UFF): Most DApps allow users to deposit or transfer funds to make profits. In some cases, DApp owners may have the capability for emergency withdrawals. However, transparency is important for users. Failure to disclose this capability can significantly risk user assets [40] and lead to a loss of trust in the DApp ecosystem.

Example: The DApp *Metarevo* [41] (Figure 5) does not disclose that the owners have the ability to transfer users' assets in front-end descriptions. However, its contract [42] allows the owner to withdraw all balances (line 8) via the *clearETH()* function (lines 5-8).

```
1 function authNum(uint256 num)public returns(bool){
2     require(_msgSender() == _auth, "Permission
          denied");
3     _authNum = num;
4     return true;}
5 function clearETH() public onlyOwner() {
6     require(_authNum==1000, "Permission denied");
7     _authNum=0;
8     msg.sender.transfer(address(this).balance);}
```

Fig. 5. Contract code snippet of *Metarevo*.

(6) Changeable DApp Status (CDS): The pause function in DApps provides developers with the capability to manage unexpected events (security issues) or implement updates [43]. However, the lack of information about the pause status for users may result in failed transfers, eroding trust in the DApp.

Example: In the *BalanceNetWork* DApp [44], the pause functionality is not publicly disclosed (Figure 6). The *pause()* function (lines 1-3) allows the owner to pause the DApp [45], which stops the trading system by restricting the *_transfer()* function (lines 4-7) by *whenNotPaused* modifier.

```
1 function pause() onlyOwner whenNotPaused external {
2     paused = true;
3     emit Pause();}
4 function _transfer(address from, address to, uint256
       value) internal whenNotPaused {
5     ...
6     _balances[from] -= value;
7     _balances[to] += value;}
```

Fig. 6. Contract code snippet of *BalanceNetWork*.

(7) Volatile NFT Accessibility (VNA): NFTs represent digital ownership of unique assets on the blockchain [3]. However, storing NFT metadata on centralized servers while claiming its permanence introduces inconsistencies. Centralized storage is susceptible to shutdowns or issues [46], conflicting with the promise of permanence and longevity. To ensure durability, decentralized solutions such as IPFS [47] and Arweave [48] are recommended for storing NFT metadata.

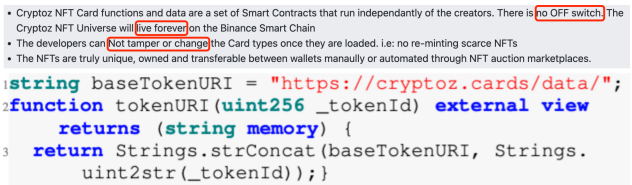


Fig. 7. Description (top) and contract snippet (bottom) of *Cryptoz Universe*.

Example: *Cryptoz Universe NFT* claims its NFTs can live forever on the blockchain [49] (Figure 7). However, its smart

contract [50] uses HTTPS for metadata storage (line 1). When this server was shut down, the NFTs became inaccessible, hurting user benefits and contradicting the claim of permanence.

IV. Methodology

In this section, we introduce our tool HYPERION, which is capable of detecting the seven types of inconsistencies defined above by analyzing DApp descriptions and smart contracts.

A. Overview

Figure 8 shows an overview of the HYPERION, which has three components, namely *Description Analysis*, *Contract Semantic Analysis*, and *Inconsistency Detection*.

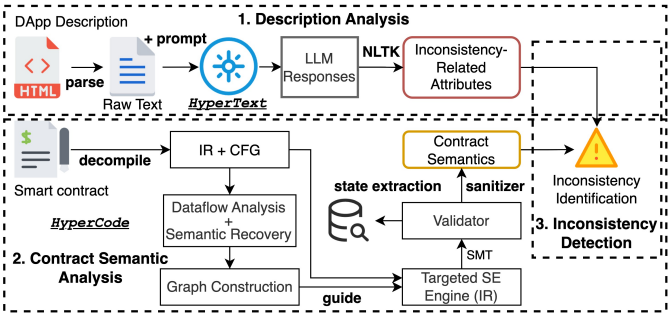


Fig. 8. Overview of HYPERION approach.

The input of HYPERION contains two parts, i.e., front-end description (can be a DApp URL, HTML, or text) and back-end contract code.

For *Description Analysis*, the DApp descriptions are first parsed as raw text, and then concatenated with our designed prompts through HYPERTEXT, which is obtained by LLaMA2 instruction-tuning (see Section IV-B1). Next, NLTK is employed to extract inconsistency-related attributes from the output of HYPERTEXT (LLM responses), which are used for further comparison with the contract semantics.

In *Contract Semantic Analysis*, HYPERCODE first decompiles the contract bytecode using Elipmoc [51], to recover the CFG and the contract IR. HYPERCODE then performs dataflow analysis on this IR to extract inconsistency-related semantics, such as fund transfers. This analysis includes constructing fund transfer and storage variable dependency graphs that guide targeted symbolic execution on the IR to trace inconsistency-related paths. Furthermore, the SMT solver validates program states collaborated with on-chain state extraction, thus identifying contract semantics.

Finally, in *Inconsistency Detection*, HYPERION incorporates the attributes extracted from DApp descriptions and contract semantics to identify inconsistencies based on defined rules.

B. LLM-based DApp Description Analysis

In this subsection, we give details of how we extract attributes related to inconsistencies from the DApp description.

1) Instruction-Tuning: In this part, We introduce the process of obtaining our HYPERTEXT model. We adopt LLaMA2 as our base model due to its adaptability, cost-free access, and excellent performance in natural language tasks (see Section II-B). To make LLaMA2 perform well in our downstream DApp description analysis task, we propose an instruction-tuning approach, as depicted in Figure 9. Specifically, after we obtain the raw text of the description, (1) we first design specific prompts to improve the model's efficacy in yielding the specific desired attributes we want to extract. (2) Next, we adopt a prompt segmentation method to fix issues caused by long input. (3) Then, we perform instruction tuning with manually labeled DApp descriptions and finally get our model HYPERTEXT.

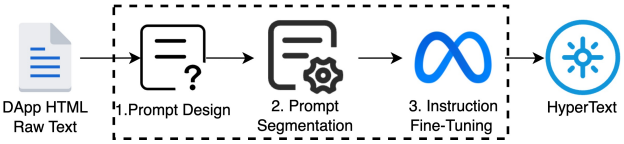


Fig. 9. Workflow of LLaMA2 Instruction-Tuning.

Prompt Template

System Prompt (SP): You are an expert in blockchain and smart contracts. Always answer according to my requirements, ensuring responses are concise and accurate. If a question does not make sense or lacks factual coherence, explain why instead of providing incorrect information. If you do not know the answer to a question, do not share false information.

User Prompt (UP)-Numeric

UP_{it}: Extract numerical information related to the { %INCONSISTENCY_NUMERIC_INFO } from the provided text.

UP_{CoT}: None

DApp Description (D): The text I provide is: { %TEXT }

Inconsistency Type	Inconsistency Numeric Info
Unguaranteed Reward (UR)	rate of reward or profit
Hidden Fee (HF)	rate of fee or tax
Adjustable Liquidity (AL)	lock time
Unconstrained Token Supply (UTS)	total amount or supply

Fig. 10. Prompt Template of Numeric Inconsistency-related Information.

Prompt Design. LLMs operate on a prompt-based learning approach [52], with prompt design crucially impacting performance [12]. We adopt LLaMA2's recommended prompt structure [53], comprising a system prompt (SP) and a user prompt (UP). SP defines the model's role R, e.g., requiring the model to act as a *"smart contract expert"* and includes general instructions GI to ensure accuracy, e.g., requiring the model to *"avoid sharing false information"*. The user prompt UP is divided into UP_{it} and chain of thought [13] (UP_{CoT}) components. Specifically, UP_{it} directs LLM to analyze and extract specific kinds of inconsistency-related information types, i.e., numeric and boolean. Figure 10 and Figure 11 show the prompt templates for extracting numeric and boolean inconsistency-related information, respectively. For generating numeric values like rewards or fee rates, the prompt directly instructs the LLM to 'extract numeric values from the provided DApp description'. However, when generating boolean values, which are not explicitly stated in the description, the prompt does more than just instruct the LLM to 'answer with yes or no.' It also utilizes CoT (Chain of Thought) patterns to construct UP_{CoT}, which guide the LLM in deducing the answer by providing key phrase explanations (e.g., DApp pause, the storage way of NFTs) and illustrative examples. This approach helps the LLM interpret

and infer boolean values from the context. The complete prompt template for DApp description analysis is formalized as $\{P = \{SP : R + GI\} + \{UP : UP_{it} + UP_{CoT}\} + D\}$, where D represents the raw text of the DApp description.

Prompt Template

System Prompt (SP): You are an expert in blockchain and smart contracts. Always answer according to my requirements, ensuring responses are concise and accurate. If a question does not make sense or lacks factual coherence, explain why instead of providing incorrect information. If you do not know the answer to a question, do not share false information.

User Prompt (UP)-Boolean

UP_{it}: Whether this text indicates { %INCONSISTENCY_BOOLEAN_INFO }. You need to answer 'yes' or 'no' and provide a concise explanation.

UP_{CoT}: Think step by step: First, the definition of { %KEY_PHRASE } is: { %DEFINITION }. Second, if { %SCENARIO }, there may be some descriptions like: { %EXAMPLE }. This is just an example to help you understand. The actual situation is not limited to such descriptions.

DApp Description (D): The text I provide is: { %TEXT }

Inconsistency Type	Information in Prompt Template
Unclaimed Fund Flow (UFF)	**Inconsistency boolean info:** The existence of a method for clearing or withdrawing all tokens/assets contained in the contract by someone. **Key phrase:** clearing assets **Definition:** An act of clearing contract assets by calling specific functions in the contract (usually only accessible to privileged users) to transfer all tokens or ethers. **Scenario:** DApp can clear the assets in the contract **Example:** includes a feature that allows for the transfer of all contract assets to the project owner for safety reasons.
Changeable DApp Status (CDS)	**Inconsistency boolean info:** the DApp can be paused **Key phrase:** DApp pause **Definition:** A DApp pause refers to a feature embedded within a smart contract of a DApp, allowing the contract owner to temporarily suspend specific functionalities, such as token minting. **Scenario:** DApp can be paused **Example:** for the safety and security of our users, in rare emergency cases or to address potential vulnerabilities, the DApp owner can temporarily suspend specific functionalities, such as token minting or transfers.
Volatile NFT Accessibility (VNA)	**Inconsistency boolean info:** the NFT is stored in centralized server **Key phrase:** stored in centralized server **Definition:** NFTs are stored using HTTPS, or suggests the possibility of the server shutting down. **Scenario:** NFT is not stored in a centralized server **Example:** NFTs can be accessed forever or are stored via IPFS.

Fig. 11. Prompt Template of Boolean Inconsistency-related Information.

Prompt Segmentation. The maximum number of tokens that LLaMA2 can process in a single prompt is 4096 [11]. However, the token length of some DApp descriptions in our dataset exceeds this limit, which yields incorrect answers and unwanted output. To address this issue, we segment the raw text of the DApp description D by setting a token limit. Every segmentation D_i is further concatenated with the design prompt SP and UP to construct the prompt P_i. Through experimentation, we find that a limit of 3000 tokens per segmentation yields the best results (compared to 500, 1000, 1500, 2000, and 4000). This choice effectively handles lengthy descriptions without significantly impacting LLM's comprehension abilities. Although segmentation can result in incomplete sentences, LLMs are skilled in contextual understanding, which allows them to interpret meaning and maintain coherence with fragmented input, preserving their overall comprehension and effectiveness.

Instruction-Tuning. To enhance LLaMA2's effectiveness for our specific task, we further instruction-tune the model using labeled data on 63 DApp inconsistencies [20]. Specifically, we first segment the raw text of the DApp descriptions, adhering to the 3000 tokens per segmentation, and then construct the LLaMA2 instruction according to our designed prompts P for every segmentation. However, this step yields some wrong responses and useless information, e.g., multiple lines of empty spaces without any text. To refine LLaMA2's responses, we remove redundant words while only retaining outputs that contain the relevant information, and correct any inaccuracies. The revised responses form the basis of our training dataset, which aligns with the structure of the *alpha* dataset, as defined by the LLaMA2 official fine-tuning project [54]. The format can be represented by the tuple (*instruction*, *input*, *output*), where *instruction* refers to the user prompt UP, *input* represents DApp description raw text D, and *output* denotes the adjusted LLaMA2 response. After that, we adopt LORA [55], a famous PEFT (Parameter-Efficient Fine-Tuning) [56] method, to train the learnable parameters for our inconsistency information extraction task based on our training dataset. The above process achieves an 84% precision on our test dataset comprising 54 labeled DApp descriptions (Section III-B), and we finally obtain our model HyperText. Due to page limitation, we provide a detailed fine-tuning process in our open repository.

2) Inconsistency-related Attributes Extraction: The output generated by HyperText comprises several sentences assessing the presence of inconsistency-related information within the DApp descriptions. However, these outputs vary in format, necessitating the extraction of uniformly formatted key-value attributes for further comparison with the contract semantics.

In our experiments, we find that directly extracting key-value attributes using LLM is inefficient and often inaccurate. To address this challenge, we employ the NLTK to extract inconsistency-related information from LLM answers and unify them into the key-value format attributes. For instance, (1) to find the reward rate from HyperText's answer, we first tokenize the sentences and use POS tagging to label the words. Then, we scan for keywords like "reward" and its synonyms extended by NLTK WordNet. The final result is located around the keyword, and we obtain the numeric value based on the digit word tag and symbol '%'. (2) It is straightforward to extract the 'yes' or 'no' boolean symbols from HypetText's answers, as shown in the template (Figure 11). We directly extract the boolean value from the LLM response.

C. Contract Semantic Analysis

This subsection details how HyperCode recovers high-level semantic features related to inconsistencies from low-level bytecode. The analysis can be divided into three parts, i.e., decompilation and dataflow analysis, graph analysis, and IR-based symbolic execution.

1) Decompilation and Dataflow Analysis: For contract bytecode analysis, we utilize Elipmoc for decompilation. Elipmoc converts EVM bytecode into a high-level IR, structured in static single assignment (SSA) form, and delineates function borders. Utilizing this IR, we perform dataflow analysis with datalog [57] rules to extract essential semantics for subsequent graph analysis and symbolic execution. This includes defining core IRs related to storage, external calls, and data flow.

Instruction :- $SSTORE(s, y, z)$ | $x := SLOAD(s, y)$
| $CALL(s, arg)$
| $x := CALLPRIVATE(pf, arg)$
| $RETURNPRIVATE(t, v)$
| $x := PHI(y, z)$

Letters (x,y,z,t,v) denote the variables declared in the IR. The variables pf and arg represent the private function and call arguments, respectively, and s denotes the SSA statement in which an instruction lies. The semantics of the first three instructions are the same as those of the EVM opcodes. The storage write instruction $SSTORE(s, y, z)$ signifies that the statement s writes variable z to the storage address y. And $x := SLOAD(s, y)$ represents the variable x loaded from the storage address y in statement s. Instruction $CALL(s, arg)$ denotes the statement s executes the external contract invocation with arguments arg.

The final three instructions - unique to Elipmoc IR - pertain to dataflow and control flow within the IR. Instructions $CALLPRIVATE$ and $RETURNPRIVATE$ are involved in private function calls: *x := CALLPRIVATE(pf,arg)* calls the private function pf with arguments arg, and x captures the return value. The *RETURNPRIVATE(t,v)* instruction facilitates returning variables v to the caller at target t. Lastly, the *x := PHI(y,z)* instruction indicates the flow of variables y and z to x, playing a pivotal role in dataflow within the IR.

Based on the instruction semantics above, we summarize the high-level semantics in Table II that we adopt to formulate our dataflow and semantic recovery rules. The first eight relations in the table are supported by Elipmoc, while the remaining three rules are induced on the basis of the eight relations, which recover a higher-level semantic of contracts. According to the definition of the seven DApp inconsistencies, there are three of them related to transfer funds (*UR*, *HF*, *UFF*), while the other four are about reading and writing storage (*AL*, *UTS*, *CDS*, and *VNA*). Therefore, our analysis primarily revolves around two types of operations: *fund transfer* and *specific storage access* (store and load), crucial for pinpointing inconsistencies in contract bytecode.

The first induced relationship is *Transfer*, *Transfer* operations are extracted by identifying call operations in the contract bytecode. From the collected dataset for finding inconsistencies, there are two types of transfers, i.e., Ether and ERC20 token transfer. ERC20 token transfers are identified through the function signatures mandated by the ERC20 standard: *transfer()* and *transferFrom()*, with respective function signatures 0xa9059cbb and 0x23b872dd [58]. In contrast, Ether transfers are characterized by those *CALL* operations, which uniquely do not utilize memory arguments for the call target and transfer amount, referring to the *CALL* instruction semantics [3]. HyperCode identifies these transfer operations and uses rules 1 to 3 to determine the critical call sites in the IR.

$$\frac{\frac{CA(cs,r,0)}{CA(cs,a,1)}\ EC(cs,*,fs)\ \ C(fs) = \text{"}0xa9059...\text{"}\ \ SF(cs,cf)}{Transf(cs,r,a,cf)} \tag{1}$$

$$\frac{\frac{CA(cs,r,1)}{CA(cs,a,2)}\ EC(cs,*,fs)\ \ C(fs) = \text{"}0x23b87...\text{"}\ \ SF(cs,cf)}{Transf(cs,r,a,cf)} \tag{2}$$

$$\frac{!CA(cs,*,*)\ \ CALL(cs,*,r,a,*,*,*,*)\ \ SF(cs,cf)}{Transf(cs,r,a,cf)} \tag{3}$$

The second induced relationship is *SenderGuard*, which helps us analyze whether some operations are restricted to specific users, e.g., check whether the function caller is the owner. This relationship is induced by the following rule 4. When a statement s retrieves variable x from storage slot y and subsequently compares x with the function's caller, we can deduce that function cf incorporates a sender verification mechanism. This process helps us to identify privileged users within the contract.

$$\frac{x = SLOAD(s,y)\ \ SF(s,cf)\ \ Comp(x, CALLER)}{SG(y,cf)} \tag{4}$$

The other key induction *StorageInfer* is to recover semantics about storage inference. Our approach involves deducing five distinct types of storage variables from the IR by identifying characteristic features of storage operations. These types include: owner, time, supply, pause, and token URI. Recognizing these variables is essential for pinpointing operations that interact with inconsistency-related variables. For example, to determine the storage slot for the *owner* variable that represents the contract owner or privileged users, we utilize rule 5. If a contract implements the function *F-owner()* with a specific signature $SHA3(F)$, we infer that the return variable loaded from slot y is indicative of the *owner* variable. In scenarios where *owner()* is not explicitly defined, we resort to pattern matching based on the *SenderGuard* (SG) induction. Specifically, a variable loaded from slot y and compared with the $CALLER$ indicates the *owner* variable is stored in slot y.

$$\frac{\frac{x=SLOAD(s,y)}{SF(s,cf)}\ [C(fs) = SHA3(F)||\frac{Controls(x,*)}{SG(y,cf)}]}{ST_{owner}(y,cf)} \tag{5}$$

In our approach, the identification of function signatures is a critical step. We refer to standard interfaces defined in Ethereum Improvement Proposals [59] (EIPs) and utilize widely recognized third-party libraries, such as OpenZeppelin [60]. This technique allows us to quickly locate critical variables in the contract's bytecode, which is also adopted by other works [58]. We use this method to find the storage of variables that represent the token supply (e.g., *totalSupply()*), the DApp pause status (e.g., *pause()*), and the token uri (e.g., *tokenURI()*) of the NFT from contract bytecode as shown in rule 6. For contracts lacking these standard function implementations, our strategy involves a detailed analysis of operational sequences and constraints in the bytecode. We have crafted specific

TABLE II
CONTRACT IR-LEVEL SEMANTIC RELATIONS

Relation	Notation	Description
$Constant$	$C(x) = v$	Variable x is inferred to denote a constant value v
$Externalcall$	$EC(cs, addr, fs)$	Call the function fs of the contract $addr$ at the call site cs
$Controls$	$Controls(x, s)$	Variable x determines whether the statement s will be executed
$MathOp$	$x = Math(op, y, z)$	Variable y and z are two elements of the math operation op
$CallArg$	$CA(cs, x, i)$	Variable x is the ith parameter of the external call whose call site is cs
$FuncArg$	$FA(fs, x)$	Variable x is the argument of the function whose signature is fs
$DataFlow$	$\downarrow DF(x, y)$	The value of the variable x will flow to the variable y
$StatementFunc$	$SF(x, y)$	The statement x is in the public function y
$Transfer$	$Transf(cs, r, a, cf)$	The variables r and a are the recipient and the transfer amount of call site cs respectively in the function whose signature is cf
$SenderGuard$	$SG(x, cf)$	The variable loaded from storage slot x is inferred to be compared with $CALLER$ in function whose signature is cf
$StorageInfer$	$ST_{var}(x, cf)$	Variable var is inferred to be stored in slot x in function with signature cf

patterns based on source code level features, derived from our ground truth dataset for defining inconsistencies. These patterns facilitate the identification of storage locations by extracting unique features and constraints within the bytecode.

The rules from 7 to 9 outline our variable semantic recovery patterns. For example, rule 7 is employed to identify the state variable storing the DApp's liquidity lock duration. We ascertain whether the arguments of a public function x, combined with the current timestamp (derived from *TIMESTAMP*), flow to a variable z, stored in slot y. Represented semantically as `lock=now+x`, where x is user-defined lock time, and now is the current timestamp, slot y is inferred to store the lock time. Likewise, we look for the *ADD* operations (rule 8) and the control semantics (rule 9) to infer the storage of the token supply and the pause variable, respectively. Due to the page limit, please refer to our repository for more details.

$$\frac{x = SLOAD(s, y)\ \ SF(s, cf)\ \ C(fs) = SHA3(F)}{ST_x(y, cf)\ \ x \in [supply, pause, token\_uri]} \tag{6}$$

$$\frac{SSTORE(s, y, z)\ \ \frac{TIMESTAMP(t)}{\downarrow DF(t,z)}\ \ \frac{FA(fs,x)}{\downarrow DF(x,z)}\ \ SF(s, cf)}{ST_{time}(y, cf)} \tag{7}$$

$$\frac{x = SLOAD(s, y)\ \ r = MathOp(+, x, *)\ \ \frac{SSTORE(a,y,R)}{\downarrow DF(r,R)}}{ST_{supply}(y, cf)} \tag{8}$$

$$\frac{x = SLOAD(s, y)\ \ SF(s, cf)\ \ \frac{Controls(x,a)}{SSTORE(a,y,T)}\ \ C(T) = True}{ST_{pause}(y, cf)} \tag{9}$$

2) Graph Analysis: After we recover the high-level semantics of the IR of the contract, we construct a graph to obtain the connection between the critical information that we extracted in the contract. This graph contains two subgraphs, i.e., the fund transfer graph (FTG) and the state variable dependency graph (SDG). These subgraphs facilitate our analysis of contract semantics, particularly in relation to inconsistency attributes.

The graph construction is grounded on three key relations: *Transfer*, *SenderGuard*, and *StorageInfer*. The $Transfer(cs, r, a, cf)$ relation is utilized to identify recipient nodes and the corresponding transfer amount edges for each transfer operation. We also identify whether the transfer amount to some nodes is part of the amount transferred to users to identify receiving fee operations, and whether there exists a recipient who can withdraw the contract balance.

For state variable dependencies, we start by identifying state variable nodes (e.g., ST_x) using $ST_{var}(x, cf)$. Then, through the *SenderGuard* relationship, we determine state variables that are conditionally manipulated based on sender verification (e.g., pause status ST_{pause} is controlled by the owner). This step is crucial for mapping the complex dependencies of state variables within the contract and finding those privileged operations.

In our graph analysis, we delve into the roles of recipients and their associated high-level features. Additionally, we organize the extracted information at the function level to guide our subsequent symbolic execution. Two essential features are extracted for this purpose: $\overrightarrow{FTG}(cs, r, a, cf, p)$ and $\overrightarrow{SDG}(cs, x, cf, d)$, where p denotes the inferred privileged owner, e.g., who can withdraw the contract balance, and d represents the dependency relationships among state variables.

3) IR-based Symbolic Execution: With the extraction of basic features completed, we proceed with an IR-based symbolic execution framework to obtain runtime states and validate critical attributes. The IR, distinct from EVM opcodes, features unique instructions (as detailed in section IV-C1). The interaction between identified public and extracted private functions is managed by *CALLPRIVATE* and *RETURNPRIVATE* instructions, which facilitate parameter and result passing in IR CFG. The *PHI* instruction is critical in merging variables from divergent control flows. The first element indicates the in-loop variable that determines the loop exit condition, while the second acts as an out-loop bound checker. Due to page limitation, we show an exemplary IR CFG in our repository.

Based on the IR CFG illustrated above, HYPERION builds an extensible symbolic execution framework from scratch that makes use of the completeness of IR CFG while incorporating the semantics of each instruction and the dataflow analysis results. To guide the symbolic execution process, we use dataflow information to monitor critical variables involved in key operations and their flows, e.g., variables flow from and to storage slots (via SSTORE and SLOAD), to formulate

induction rules. Graphs extracted based on contract semantics using dataflow analysis and datalog rules label key variables (e.g., transfer amounts) and statements (e.g., external calls) at the function level. They also highlight dependencies among core state variables, e.g., which state can be changed by the contract's privileged user. These graphs record information identical to the contract IR operated by symbolic execution, guiding which functions to test which variables to load, and at which program point to check the execution states. This framework can support our testing during symbolic execution and check critical states for contract semantics recovery, which is also extensible for programming more rules.

D. Inconsistency Detection

To detect the inconsistencies, the frontend analysis yields key-value information for critical attributes, e.g., fee rate. The backend analysis maps these attributes to key variables (e.g., transfer recipient and amount, token uri) and operations (e.g., ether/token transfer, contract states modification) that are identified using induction rules and graph analysis during symbolic execution. Specifically, the attributes extracted by HYPERTEXT are denoted as F, with each attribute A_f assigned a numeric (n) or boolean (b) value V, based on the inconsistency type. For contract semantics, HYPERCODE identifies attributes A_b and their values V, as well as expressions F_{expr} from symbolic execution (SE).

$$F : \{A_f : V\}, V \in n|b \qquad B : \{A_b : V, F_{expr}\}, \frac{V \in n|b}{F_{expr} \rightarrow SE}$$

To facilitate this detection, HYPERION utilizes public nodes via the Web3 API [61], employing methods like *getCode()* for bytecode retrieval and *getStorageAt()* for accessing specific storage data. In total, our HYPERION supports inconsistency detection across 13 blockchain platforms.

UR inconsistency is flagged when the DApp's description cites a fixed reward rate $A_{f(r)}$. Yet, the contract's semantic analysis reveals a transfer amount $F_{expr_{ta}}$ dependent on dynamic factors (like contract balance), with the transfer target $F_{expr_{tt}}$ being the user (*CALLER*). In *HF* inconsistency detection, HYPERION examines DApp descriptions for fee claims and analyzes contract semantics for transfer amounts to specific addresses. An example involves symbolically expressing the transfer amount as *bvudiv_i(Ia_store-1-*Iv, 100)*, with HYPERION then querying public nodes for corresponding storage values to calculate and compare fee rates *Ia_store-1/100* against those mentioned in the DApp description. The modifiable status of fee variables is also reported through SDG. *UFF* inconsistency is identified when the DApp description omits mention of fund withdrawal by specific or privileged users, yet contract semantics suggest otherwise.

For *AL* inconsistency, HYPERION assesses both the front-end lock time description and the contract's ability to modify this duration. An inconsistency is noted if the lock time is alterable, irrespective of whether users are informed or not. In *UTS* analysis, HYPERION contrasts the front-end description with the contract's token minting capabilities. Two scenarios are flagged as inconsistencies: (1) when a limited token supply is claimed, but the contract allows unconstrained minting; (2) when the front-end does not explicitly mention any limit on the number of tokens, but the contract is designed to support unconstrained minting.

For the boolean inconsistencies *CDS* and *VNA*, HYPERION extracts the boolean value inferred from the DApp description. The modifiability of the DApp status can be obtained from contract semantics by rule 9 when performing symbolic execution, so as to report the *CDS* inconsistency. To obtain the storage way of NFTs, we request the public node to obtain the base URI of DApp NFTs (considered as the prefix of stored metadata of NFTs), which is stored in the inferred variable from contract semantics. We judge whether the NFT is stored in decentralized storage services (such as IPFS [47] and Arweave [48]) or in centralized ways (HTTPS and Base64 [62]) from this prefix. HYPERION compares this information with the storage way inferred from the description to report *VNA*.

V. EVALUATION

A. Experiment Setup

The experiment was conducted on a server running Ubuntu 20.04.1 LTS and equipped with 128 Intel(R) Xeon(R) Platinum 8336C @ 2.30GHzE CPUs, and 2 NVIDIA A800 80GB PCIes.

Dataset. We used two datasets to evaluate HYPERION. The first one is the ground truth dataset, which contains 54 DApps we labeled for inconsistency definition in Section III-B. The second dataset contains 835 real-world DApps with their available HTML files, contract addresses, and platforms. Notably, HYPERION is compatible with contracts written in Solidity, regardless of whether they are deployed on Ethereum, e.g., BNB Chain [63], Polygon [64]. We crawled them from the DeFi and NFT categories on DappRadar and DAppBay.

Evaluation Metrics. We summarize the following research questions (RQs) to evaluate HYPERION.

- RQ1. How effective is HYPERION in detecting inconsistencies in our ground-truth DApp dataset?
- RQ2. How is the performance of HYPERION in detecting inconsistencies in the large-scale DApp dataset?
- RQ3. What is the efficacy of HYPERTEXT and HYPERCODE in analyzing DApp descriptions and contract semantics, respectively?

B. Answer to RQ1: Effectiveness in the Ground Truth Dataset

To answer RQ1, we run HYPERION in our ground truth dataset of 54 DApps. The detection result is shown in Table III, which outlines the number of each type of inconsistency (Incs) in the dataset, true positives (TP) correctly identified, false negatives (FN) not detected, and false positives (FP) wrongly identified by HYPERION. We use formulas $\frac{\#TP}{\#TP+\#FN} \times 100\%$ and $\frac{\#TP}{\#TP+\#FP} \times 100\%$ to calculate the recall (Rec) and precision (Prec) rate, respectively. Furthermore, we also calculate the overall precision and recall of HYPERION on the ground truth dataset. Using overall precision as an example, the overall result can be calculated by the formula $\frac{\sum_{i=1}^{n} p_{c_i} \times |c_i|}{\sum_{i=1}^{n} |c_i|}$, in which p_{c_i} represents the precision to detect inconsistencies i, and $|c_i|$ is the number of inconsistencies i. This method is also adopted by other works [3], [65].

The results show that the overall precision and recall of HYPERION are 92.06% and 84.06%, respectively. The detailed false negative and false positive analysis is discussed and illustrated in our answer to RQ3 (see Section V-D).

TABLE III
DETECTION RESULT IN GROUND TRUTH DATASET.

DApp Inconsistency	# Incs	# TP	# FN	# FP	Rec (%)	Prec (%)
Unguaranteed Reward	19	12	7	0	63.2	100.0
Hidden Fee	21	17	4	0	81.0	100.0
Adjustable Liquidity	4	4	0	0	100.0	100.0
Unconstrained Token Supply	8	8	0	3	100.0	72.7
Unclaimed Fund Flow	12	12	0	2	100.0	85.7
Changeable DApp Status	4	4	0	0	100.0	100.0
Volatile NFT Accessibility	1	1	0	0	100.0	100.0

C. Answer to RQ2: Detection in a Large-scale Dataset

To answer RQ2, we run HYPERION on 835 unlabeled DApps obtained from DAppRadar and DAppBay. The experimental results given in Table IV (the second and third columns) show the frequency of each DApp inconsistency in this dataset.

To evaluate the performance of HYPERION in finding inconsistencies in the large-scale dataset, we refer to a random sampling method based on the confidence interval [66] to generalize the population of the total number of problems found for this inconsistency, which is also adopted in the previous works [3]. Specifically, to establish the sample size of each inconsistency, we set a confidence interval of 10 and a confidence level of 95% and calculate the number of samples (S, the fourth columns in Table IV) that we need to collect [67]. The calculated results of the seven inconsistencies are 18, 44, 13, 60, 67, 34, and 49, respectively. The evaluation dataset is then randomly sampled according to the result and manually labeled by three authors of this paper. We analyzed all the reported samples of *UR* and *AL* as the total number reported is close to the calculated number that should be sampled for these two inconsistencies.

TABLE IV
DETECTION RESULT IN LARGE-SCALE DATASET.

DApp Inconsistency	# Incs	Per (%)	# S	# TP	# FP	Prec (%)
Unguaranteed Reward	22	2.63	22	20	2	90.9
Hidden Fee	77	9.22	44	38	6	86.4
Adjustable Liquidity	15	1.80	15	15	0	100.0
Unconstrained Token Supply	159	19.04	60	46	14	76.7
Unclaimed Fund Flow	223	26.71	67	66	1	98.5
Changeable DApp Status	51	6.12	34	34	0	100.0
Volatile NFT Accessibility	98	11.74	49	49	0	100.0

The fifth and sixth columns in Table IV show precision evaluation on the randomly sampled dataset, divided into TPs (268) and FPs (23) (we manually labeled all DApp with *UR*, and *AL* inconsistencies to make the results more reliable), which is also adopted by other related works [3], [5], [68]. The precision rate of HYPERION in the analysis of each inconsistency is shown in the seventh column.

The results show that for *UR*, *HF*, *UTS*, and *UFF* inconsistencies, HYPERION reports them with a precision of 90.9%, 86.4% 76.7%, and 98.5%, respectively, and reaches a precision of 100% when analyzing other types of inconsistency. Our tool HYPERION reaches an overall precision of 92.10%. In addition, we have further checked TPs (268 DApps) from the sampled dataset based on Hyperion's reports. We find that 67 (**25%**) of them are now **inaccessible** within just 3 months. Of the remaining accessible DApps, 41 (**15.3%**) DApps are labeled as *high risk* by DAppRadar. These numbers underscores the threats these DApps pose to users' assets.

D. Answer to RQ3: Evaluation of Respective Performance

We observe that HYPERION has some FP and FN in the former two RQs. However, we do not know whether these inaccuracies are stemmed from HYPERTEXT or HYPERCODE. Therefore, we propose RQ3, evaluating HYPERION's two analyzers' performance, respectively. Our evaluation utilizes the ground truth dataset in RQ1 and the randomly sampled dataset used in RQ2.

1) HyperText: The development of HYPERTEXT involved multiple evaluations and experiments mentioned in Section IV-B. Starting with the design of effective prompts, we progressed through prompt segmentation and instruction-tuning our model. Due to the page limit, detailed experimental results and related datasets are presented in our open repository.

For the performance evaluation, three authors manually reviewed the DApp descriptions and the corresponding outputs of our HYPERTEXT. This involved reading the website pages and verifying the accuracy of extracted attributes. In the ground truth dataset, we noted instances where HYPERTEXT failed to extract reward or fee information from DApp descriptions, leading to missed inconsistency. The precision and recall of HYPERTEXT are 100% and 92.8%. In the large-scale dataset, we find that HYPERTEXT exhibited misclassification errors due to the existence of prompt keywords. For example, in the DApp *Alchemix* [69], HYPERTEXT mistakenly categorized the text "total expected supply after 3 years: 100%" as token supply information, misled by the keywords 'total' and 'supply', despite the context indicating a time duration rather than token quantity. The overall precision of HYPERTEXT in the large-scale dataset is 95.9%. Additional examples of HYPERTEXT's incorrect responses are available in our repository.

2) HyperCode: To evaluate HYPERCODE, we compare its output against the contracts code to identify FNs and FPs. The precision and recall of HYPERCODE in the ground truth dataset are 92.06% and 84.06%, respectively, and the precision in the large-scale dataset achieves 92.1%.

False Negatives. To analyze false negatives, we refer to the ground-truth dataset used in RQ1. We find that there are some FNs of *UR* and *HF* inconsistencies due to path explosion in symbolic execution and missing detectable fee transfer operations. (1) HYPERCODE sets constraints such as path search depth and loop iteration limits to avoid the path explosion, which contributes to its inability to reach deep checkpoints in the IR CFG of some complicated contracts. (2) In detecting *HF* inconsistencies, HYPERCODE misses cases where state variables storing fees are not part of the transfer operation. Consequently, even if there is an inconsistency between the DApp description and the actual fee rate stored in the state variable, the absence of a detectable fee transfer operation results in false negatives.

False Positives. (1) To identify *UTS* inconsistencies, our HYPERCODE uses the ERC interface *totalSupply()* to locate the state variable for total token supply. However, some contracts do not directly return this variable when implementing the interface. For instance, contracts using *_currentIndex-_startTokenId()* [70] or *_allTokens.length* [71] for total supply deviate from our pattern. Moreover, an FP is noted where the condition check occurs after adding to the total supply state variable [72], contradicting our pattern of requiring a protective comparison before such an addition. (2) HYPERCODE incorrectly identifies split payments to preset addresses as fee transactions, which could actually be regular payments. For example, a contract's *payment()* function (lines 201 to 211) splits the user's payment between two trusted wallets [73]. In contrast to being a *Hidden Fee*, it is merely a division of a user's payment across two accounts. (3) For *UFF*, false positives exist when the contract logic actually stipulates transferring all remaining funds when the balance is insufficient for the user's earnings. (4) When detecting *UR*, HYPERCODE incorrectly views percentage-based transfers as a profit distribution, while the function's purpose is to manage funds stuck in the contract. Specific samples are provided in our repository.

Furthermore, the dataflow analysis on the decompiled IR reduces the 88.7% of functions to be tested on average (in our large-scale experiments) when HYPERCODE performing IR-based symbolic execution, improving the detection efficiency.

Comparison Experiment. We conducted a survey on works published in top journals/conferences on SE/Security and found that the proposed inconsistencies have not been covered before, while NFTGuard [3] and Pied-Piper [74] have similar detection patterns for contract semantics of UTS and UFF, respectively. We compare HYPERCODE with NFTGuard, as Piped-Piper is not fully open source (missing the fuzzing part). We randomly select 50 DApps from the large-scale dataset and collect their contract sourcecode (NFTGuard only supports sourcecode). The result shows that HYPERION finds 16 problematic *UTS* contract behaviors with a precision of 75%, while NFTGuard does not report any bugs. Details are provided in our repository.

VI. THREATS TO VALIDITY

External Validity. Our approach, employing the symbolic execution technique, faces challenges due to the increasing complexity of DApp contracts [3]. This complexity can lead to a path explosion issue, making it difficult to analyze deeper program points. Additionally, our dataflow analysis presents a limitation in accurately recovering dataflow through struct data types. In description analysis, despite instructing-tuning the LLaMA2, we encounter instances of information misidentification. This issue partly stems from the inherent nature of LLMs, where subtle nuances in data or prompts can influence output accuracy. However, identifying specific error causes or model misinterpretations remains a challenge. However, HYPERION can be improved by an instruction-tuning process with a larger labeled DApps dataset, and more rules to identify complicated data structures in the contract.

Internal Validity. Our manual labeling process for LLM construction and evaluation might introduce errors, particularly in differentiating between FNs and TPs. To mitigate this, we implement a double-check procedure and continuously update our dataset to ensure accuracy. All experimental results and evaluations are transparently shared in our repository.

VII. RELATED WORK

Inconsistency detection in DApps. Several previous works focus on unexpected behaviors in DApps. For instance, DAppHunter [17] assesses consistency among user intentions, blockchain wallet transactions, and contract behaviors, though it does not delve into contract bytecode or DApp descriptions analysis. VetSC [4] detects discrepancies between contract bytecode and DApp category rules, deduced from the text around DApp buttons or widgets. Our approach, HYPERION, introduces a broader scope of inconsistency, bridging the gap between DApp descriptions and contract behaviors using natural language understanding and program analysis.

Smart contract security issues detection. Recently, many program analysis tools have been developed to focus on detecting security problems in smart contracts. Static analysis tools like [5], [75], [76] are widely used for security issue detection. Dynamic testing tools [8], [74], [77], along with machine learning approaches [78], also contribute to this landscape. However, to the best of our knowledge, HYPERION is the first tool incorporating both description analysis and contract semantics extraction to report newly found inconsistencies.

Inconsistency detection of traditional applications. There are some related works focusing on detecting trustworthiness and inconsistency issues in mobile [79]–[81] and web applications [82]. However, the inexistence of a framework with rich semantics in the smart contract programming language poses unique challenges to contract bytecode analysis.

The immutable nature of blockchain transactions and financial attributes highlights the risks of DApp inconsistencies. Our work specifically addresses this challenge and bridges the gap to enhance the DApp ecosystem's security, assisting developers, users, and marketplaces in identifying inconsistencies.

VIII. CONCLUSION

In this paper, we define seven types of DApp inconsistencies from an empirical study and introduce HYPERION, a tool using LLM and dataflow-guided symbolic execution to identify inconsistencies between DApp descriptions and smart contract implementations automatically. HYPERION instruction-tunes LLaMA2 for DApp description analysis and utilizes dataflow-guided symbolic execution for contract bytecode analysis. The experimental results show our HYPERION's effectiveness in unveiling DApp inconsistencies with an overall precision of 92.06% and an overall recall of 84.06%.

ACKNOWLEDGMENT

This work is partially supported by fundings from the National Key R&D Program of China (2022YFB2702203), the National Natural Science Foundation of China (62302534, 62332004).

REFERENCES

[1] K. Wu, "An empirical study of blockchain-based decentralized applications," arXiv preprint arXiv:1902.04969, 2019.

[2] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," in Proceedings of the 4th ACM Conference on Advances in Financial Technologies, 2022, pp. 30–46.

[3] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in nft smart contracts," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 373–384. [Online]. Available: https://doi.org/10.1145/3597926.3598063

[4] Y. Duan, X. Zhao, Y. Pan, S. Li, M. Li, F. Xu, and M. Zhang, "Towards automated safety vetting of smart contracts in decentralized applications," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, 2022, pp. 921–935.

[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 254–269.

[6] 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.

[7] 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 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 227–239.

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

[9] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, "Smartdagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability," in Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 752–764.

[10] D. Spencer, Card sorting: Designing usable categories. Rosenfeld Media, 2009.

[11] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[12] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" arXiv preprint arXiv:2309.05520, 2023.

[13] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[14] "Natural language toolkit," 2023, https://www.nltk.org/.

[15] "Dappbay," 2023. [Online]. Available: https://dappbay.bnbchain.org/

[16] "The world's dapp store discover, track & trade everything defi, nft and gaming," 2023. [Online]. Available: https://dappradar.com/

[17] J. Zhou, T. Jiang, H. Wang, M. Wu, and T. Chen, "Dapphunter: Identifying inconsistent behaviors of blockchain-based decentralized applications," in 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 2023, pp. 24–35.

[18] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," Future Generation Computer Systems, vol. 105, pp. 475–491, 2020.

[19] "Documentation," 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.23/

[20] "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," 2023. [Online]. Available: https://shangwenwang.github.io/files/ICSE-24A.pdf

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

[22] P. Liu, X. Qiu, and X. Huang, "Recurrent neural network for text classification with multi-task learning," arXiv preprint arXiv:1605.05101, 2016.

[23] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, and G. Wang, "Instruction tuning for large language models: A survey," 2023.

[24] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray et al., "Training language models to follow instructions with human feedback," Advances in Neural Information Processing Systems, vol. 35, pp. 27 730–27 744, 2022.

[25] H. Li, Y. Liu, X. Zhang, W. Lu, and F. Wei, "Tuna: Instruction tuning using feedback from large language models," 2023.

[26] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin et al., "Code llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023.

[27] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al., "Llama 2: Open foundation and fine-tuned chat models," arXiv preprint arXiv:2307.09288, 2023.

[28] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," IEEE Transactions on Software Engineering, vol. 48, no. 1, pp. 327–345, 2020.

[29] "Dedaub," 2023. [Online]. Available: https://library.dedaub.com/

[30] "Website of bnb ultra," 2023. [Online]. Available: https://bnbultra.com/

[31] "Bnb ultra contract," 2023. [Online]. Available: https://bscscan.com/address/0x87577f0b6694be2e35d9b8ae852bbbdb3a0344d2#code

[32] "Aave liquidity protocol. earn interest, borrow assets, and build applications," 2023. [Online]. Available: https://aave.com/

[33] "Bayc-bored ape yacht club," 2023. [Online]. Available: https://boredapeyachtclub.com/

[34] J. K. Brekke and A. Fischer, "Digital scarcity," Internet Policy Review, vol. 10, no. 2, Apr. 2021. [Online]. Available: https://policyreview.info/glossary/digital-scarcity

[35] "Website of milky way," 2023. [Online]. Available: https://drive.google.com/drive/folders/1j7RjRD1OQ1IBnFMT0W-XxpyEshO5EMnl

[36] "Milky way token contract," 2023. [Online]. Available: https://bscscan.com/token/0x11F814bf948c1e0726c738c6d42fA7234f32b6E8#code

[37] "Liquidity locking in crypto explained simply," 2023. [Online]. Available: https://coinbrain.com/dictionary/liquidity-locking-in-crypto-explained-simply

[38] "Website of baby bnb tiger," 2023. [Online]. Available: https://drive.google.com/drive/folders/1GPTJ0s1R9pI-fAnuKcQqOxGd-ZpfZUK1

[39] "Baby bnb tiger contract on bscscan," 2023. [Online]. Available: https://bscscan.com/token/0x5a04565ee1c90c84061ad357ae9e2f1c32d57dc6#code

[40] "Backdoor flaw sees australian firm lose $6.6 million in cryptocurrency," 2023. [Online]. Available: https://finance.yahoo.com/news/backdoor-flaw-sees-australian-firm-115323212.html

[41] "Website of metarevo," 2023. [Online]. Available: https://drive.google.com/drive/folders/1sB1zoh-xc-VJzB5U3YHELa4JtegKC1mZ

[42] "Metarevo contract on bscscan," 2023. [Online]. Available: https://bscscan.com/token/0x41c0fff85f53da07b373237c5149d5a06b880701#code

[43] "Adding pause functionality to secure solidity smart contracts," 2023. [Online]. Available: https://blog.logrocket.com/pause-functionality-secure-solidity-smart-contracts/

[44] "Website of balancenetwork," 2023. [Online]. Available: https://drive.google.com/drive/folders/1EVvDRPwq1VxydrQnmAVl0RNJ3JWS3O6i

[45] "Balancenetwork contract on bscscan," 2023. [Online]. Available: https://bscscan.com/address/0x5Cf8eA4278f689B301C4a17DdCa9D5ec8b0B0511#code

[46] D. Das, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, "Understanding security issues in the nft ecosystem," in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, 2022, pp. 667–681.

[47] "Interplanetary file system (ipfs)," 2023. [Online]. Available: https://docs.ipfs.io/

[48] "Meet arweave: Permanent information storage." 2023. [Online]. Available: https://www.arweave.org/

[49] "Website of cryptoz nft universe," 2023. [Online]. Available: https://drive.google.com/drive/folders/1oh86AzGxJoLTlpa6EKjyIcyh9r6gzlZg

[50] "The cryptoz nft universe contract on bscscan," 2023. [Online]. Available: https://bscscan.com/token/0x8a0c542ba7bbbab7cf3551ffcc546cdc5362d2a1#code

[51] N. Grech, S. Lagouvardos, I. Tsatiris, and Y. Smaragdakis, "Elipmoc: Advanced decompilation of ethereum smart contracts," Proceedings of the ACM on Programming Languages, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[52] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," ACM Comput. Surv., vol. 55, no. 9, jan 2023. [Online]. Available: https://doi.org/10.1145/3560815

[53] "Llama 2 is here - get it on hugging face," 2023, https://huggingface.co/blog/llama2#how-to-prompt-llama-2.

[54] "Llama 2 fine-tuning / inference recipes, examples and demo apps," 2023. [Online]. Available: https://github.com/facebookresearch/llama-recipes

[55] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[56] "Parameter-efficient fine-tuning," 2023. [Online]. Available: https://huggingface.co/docs/peft/index

[57] S. Ceri, G. Gottlob, L. Tanca et al., "What you always wanted to know about datalog(and never dared to ask)," IEEE transactions on knowledge and data engineering, vol. 1, no. 1, pp. 146–166, 1989.

[58] Q. Kong, J. Chen, Y. Wang, Z. Jiang, and Z. Zheng, "Defitainter: Detecting price manipulation vulnerabilities in defi protocols," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1144–1156. [Online]. Available: https://doi.org/10.1145/3597926.3598124

[59] "Ethereum improvement proposals," 2023. [Online]. Available: https://eips.ethereum.org/

[60] "Securely code, deploy and operate your smart contracts," 2023. [Online]. Available: https://www.openzeppelin.com/

[61] "web3py," 2023, https://web3py.readthedocs.io/en/stable/web3.eth.html.

[62] "Base64," 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Base64

[63] "Harnessing decentralization to make the impossible possible," 2023. [Online]. Available: https://www.bnbchain.org/en/bnb-smart-chain

[64] "The value layer of the internet," 2023. [Online]. Available: https://polygon.technology/

[65] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," IEEE Transactions on Software Engineering, 2021.

[66] "Confidence interval," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Confidence_interval

[67] "Sample size calculator," 2023. [Online]. Available: https://www.surveysystem.com/sscalc.htm

[68] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in Ndss, 2018, pp. 1–12.

[69] "Alchemix self-repaying loans allow you to leverage a range of tokens without risk of liquidation." 2023. [Online]. Available: https://alchemix.fi/

[70] "$8liens contract code," 2023. [Online]. Available: https://etherscan.io/address/0x740c178e10662bbb050bde257bfa318defe3cabc#code

[71] "Mutantcats contract code," 2023. [Online]. Available: https://etherscan.io/address/0xaadba140ae5e4c8a9ef0cc86ea3124b446e3e46a#code

[72] "Mchcoin contract code," 2023. [Online]. Available: https://etherscan.io/address/0xD69F306549e9d96f183B1AecA30B8f4353c2ECC3#code

[73] "Huxleycomicsissue56 contract code," 2023. [Online]. Available: https://etherscan.io/address/0x42fe737749683595e4315b443eadcc9346a994d9#code

[74] F. Ma, M. Ren, L. Ouyang, Y. Chen, J. Zhu, T. Chen, Y. Zheng, X. Dai, Y. Jiang, and J. Sun, "Pied-piper: Revealing the backdoor threats in ethereum erc token contracts," ACM Trans. Softw. Eng. Methodol., vol. 32, no. 3, apr 2023. [Online]. Available: https://doi.org/10.1145/3560264

[75] "Mythril," 2023, https://mythril-classic.readthedocs.io/en/master/module-list.html.

[76] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in Proceedings of the Third ACM Symposium on Cloud Computing, 2012, pp. 1–14.

[77] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 778–788.

[78] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach," in Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–13.

[79] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. K. Leung, "Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies," IEEE Transactions on Software Engineering, vol. 47, no. 2, pp. 221–242, 2018.

[80] D. Bui, Y. Yao, K. G. Shin, J.-M. Choi, and J. Shin, "Consistency analysis of data-usage purposes in mobile apps," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2824–2843. [Online]. Available: https://doi.org/10.1145/3460120.3484536

[81] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017, pp. 308–318.

[82] F. S. Ocariza, K. Pattabiraman, and A. Mesbah, "Detecting unknown inconsistencies in web applications," in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017, pp. 566–577.