Uncovering NFT Domain-Specific Defects on Smart Contract Bytecode

Zuchao Ma, Muhui Jiang, Xiapu Luo, Haoyu Wang, and Yajin Zhou

***Abstract*—The peak of monthly trade volume of NFT (non-fungible token) has reached $4.95 billion USD in August 2023, which shows the hot trend and the potential significance of NFT. However, the smart contract responsible for managing NFT may contain defects, which can be exploited by attackers to cause severe damage to victims. We take the first step to systematically analyze three kinds of defects on NFT contracts, namely fragile NFT binding, non-compliant implementation, and implanted backdoor. In particular, we propose *Emerium*, the first extensible detection framework for capturing these defects by inspecting the bytecode of smart contracts. We conduct extensive experiments to evaluate *Emerium*, and the experimental results show that it can detect the aforementioned defects with 0.83 and 0.89 F-measure for ERC-721 contracts and ERC-1155 contracts, respectively. Applying *Emerium* to 87,839 ERC-721 and 9,808 ERC1155 NFT contracts of real world, we uncover 44,863,255 defects of fragile NFT binding, 1,373 defects of non-compliant implementation, and 105 defects of backdoor (also with a new CVE).**

***Index Terms*—Defect, smart contract, NFT, blockchain.**

I. Introduction

REPRESENTING digital assets, Non-fungible tokens (NFT) are recorded on blockchain and usually bound with other assets like pictures, videos, postal stamps, etc., stored in other places [1]. Till August 2023, the peak of monthly NFT trading volume of OpenSea, the largest NFT market, has reached $4.95 billion USD [2]. The life cycle of NFTs is managed by smart contracts [3] (or NFT contracts), which run on blockchain to allow users to mint, buy, and sell NFTs on chain with cryptocurrency. The design of NFT contracts should follow token standards proposed by Ethereum, i.e., ERC-721 [4] and ERC-1155 [5], in which contract functions are defined with specified formats and semantics. These functions involve token ownership, transfer, approval, and metadata retrieval, which provide interfaces for platforms, e.g., NFT markets, to operate NFTs.

Received 30 September 2023; revised 10 March 2025; accepted 14 March 2025. Date of publication 28 March 2025; date of current version 4 September 2025. This work was supported in part by Hong Kong RGC Projects under Grant PolyU15224121 and Grant PolyU15231223, and in part by the National Natural Science Foundation of China under Grant 62172301. *(Corresponding author: Xiapu Luo.)*

Zuchao Ma, Muhui Jiang, and Xiapu Luo are with the Department of computing, The Hong Kong Polytechnic University, Hong Kong, China (e-mail: csxluo@comp.polyu.edu.hk).

Haoyu Wang is with the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China.

Yajin Zhou is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China.

Digital Object Identifier 10.1109/TDSC.2025.3556285

Motivation: Due to the hot trend and significant value of NFTs, NFT contracts under attacks can cause serious losses to users, which makes examining NFT contract security emerging and important. While existing detection tools for smart contracts (e.g., Oyente [6] and Mythril [7]) have been effective in detecting common vulnerabilities, such as reentrancy, integer overflow, transaction order dependency, etc., identifying NFT-domain-specific contract defects has not been well explored. By inspecting the security issues reported by famous security auditors, such as SlowMist [8] and BlockSec [9], we systematically summarize critical security defects of NFT contracts that deserve deep attention.

Defect 1 (D1). Fragile NFT binding: NFT contracts maintain on-chain metadata in their storage to record the off-chain location of asset information represented by NFTs (c.f. Fig. 2). If an NFT contract is exploited to tamper on-chain metadata, then corresponding assets will be replaced, which makes the NFTs in the contract lose their significance. In addition, asset information (c.f. Fig. 2) is stored off-chain, e.g., on specified servers, which implies the information can be tampered with or unavailable, and finally leads to the loss of NFTs. For instance, an NFT worth $11 million USD is lost, because its on-chain metadata points to an invalid location [10].

Defect 2 (D2). Non-compliant implementation: Being the most widely used standards for NFT, ERC-721 and ERC-1155 define multiple functions and events with clear definitions and expect all NFT contracts to support them so that other smart contracts and off-chain applications can interact with NFT contracts correctly. Unfortunately, not all NFT contracts strictly follow the specification defined in ERC-721 and ERC-1155. Consequently, the non-compliant implementation can lead to vulnerable contracts being exploited, resulting in great loss. For instance, we find the contract of NFT project *etheria* [11] whose NFT price once met 80 ETH ($236,187.20 USD) [12], contains a vulnerability [13] that can be exploited to steal an NFT, which is caused by the incorrect implementation of an ERC-721 function (c.f. Section V-C).

Defect 3 (D3). Implanted backdoor: NFT contracts are expected to manage NFTs with standard functions specified in ERC-721 and ERC-1155. However, developers can leave backdoor functions in smart contracts intentionally [14] or accidentally [15] to tamper token ownership. For example, a backdoor method in the Sandbox LAND contract [15] whose traded volume has met 157.5K ETH ($464.57 million USD), can be exploited by an attacker to burn other users' NFTs (c.f. List. 4 and Section II-C). Besides, since off-chain applications listens

standard events [16] of NFT contracts to act accordingly, backdoor functions can emit fake events to mislead the applications. For example, Sleepminting attack [17] emits `Transfer` event to inform NFT market Opensea [18] to display meaningless trade.

Challenges: Automatically detecting these defects is not trivial as there are two main challenges.

C1. Requiring NFT domain knowledge: These defects concerning the design of NFT management, which involves storage layout maintaining NFT information, and function specification that regulates NFT trade. Since the layout and specification are NFT-domain-specific, a detection tool needs to extract patterns from NFT contracts for detection, by taking advantage of NFT-domain knowledge, which is more than traditional program analysis. Designing and distilling the patterns to well-reflect various defects is challenging.

C2. Lacking semantics in binary analysis: NFT contracts, especially malicious ones, may not provide source code (c.f. Section V). It is not trivial to extract patterns from contract bytecode for detecting defects, as bytecode cannot intuitively reflect NFT-domain-specific semantics, e.g., a seller of an NFT or an owner of an NFT. Therefore, it is difficult for existing works targeting smart contract analysis to detect these defects, as they fail to recover NFT-domain-specific semantics from bytecode.

Existing approaches: Das et al. adopt the API [19] provided by NFT market OpenSea [18] to collect asset data for detecting partial D1 defects [1], which makes their work limited in detecting the contracts that are not collected by OpenSea, and the asset replacement risk within contract logic. Besides, Yang et al. [20] propose NFTGuard, a source code analyzer to detect five defects in ERC-721 contracts, concerning mutable proxy, reentrancy, unlimited minting, and so on. Four of them are general (not NFT-domain-specific) defects, and one of them focuses on a subcategory of D2 defects. Xiao et al. design WakeMint to identify sleepminting risk existing in contract source code [21], which is caused by two subcategories of D2 defects and one D3 defect. Therefore, existing approaches have not tackled the challenges of this work, due to their scopes and capabilities.

Our approach: To address the aforementioned challenges, we propose the first NFT-contract-oriented binary detection framework named *Emerium*, which is shown in Fig. 1. Given the bytecode, the ABI (Application Binary Interface), and transactions (if available) of a smart contract, *Emerium* detects hidden defects in an end-to-end manner, without relying on source code. For solving challenge **C1**, *Emerium* combines runtime information with path reachability to construct the patterns that reflect defects (Section II-E). For solving challenge **C2**, *Emerium* recovers NFT-domain-specific semantics of contract variables, by referring to ERC-721/ERC-1155 specifications (Section III-B).

Emerium is an extensible framework, and it empowers users to develop plugins with Python running on the framework to detect various defects. Specifically, *Emerium* provides on-demand semantic runtime information retrieval (Section III-B), and supports plugins to leverage the provided information for identifying potential defects. To improve the efficiency of developing plugins for users, *Emerium* also empowers a plugin to adopt rules to customize its capability (Section III-C). In this paper, we develop three plugins named NFT Binding Analyzer (Section IV-A), Functionality Analyzer (Section IV-B), and Backdoor Analyzer (Section IV-C) that target on detecting D1, D2, and D3 individually.

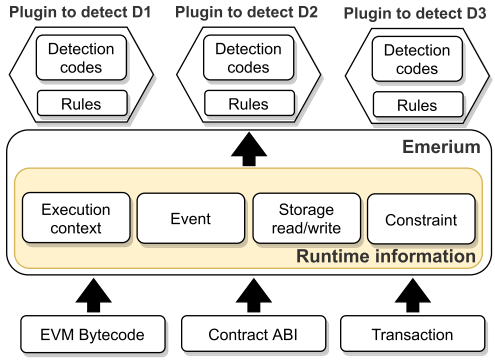

Fig. 1. Overview of *Emerium*. It takes bytecode, ABI, and transaction as input, providing on-demand runtime information retrieval to support plugins for detection.

This work makes the following contributions.

- *Comprehensive study:* We conduct a systematic study on three major defects in NFT contracts, namely fragile NFT binding, non-compliant implementation, and implanted backdoor, and design new methods to detect them.
- *Extensible framework:* We propose *Emerium*, the first NFT-contract bytecode detection framework. *Emerium* tackles the issues caused by bytecode analysis, and provides semantically-rich runtime information for retrieval (Section III). Based on *Emerium*, diverse plugins can be designed to detect various defects (Section IV). We provide a replication package in https://github.com/MacherCS/emerium.
- *Well-designed evaluation:* Experimental results show that *Emerium* can detect defects with the F-measure of 0.83 (for ERC-721) and 0.89 (for ERC-1155). Applying *Emerium* to 87,839 ERC-721 and 9,808 ERC1155 NFT contracts deployed to Ethereum, we uncover 44,863,255 defects of fragile NFT binding, 1,373 defects of non-compliant implementation, and 105 defects of implanted backdoor. (Section V)
- *Impactful discovery:Emerium* uncovers the defects of popular NFT contracts (Section V-C - Section V-D), some of which possess the traded volume worth several millions dollars. *Emerium* also discovers a new CVE (CVE-2022-35621) of a project worth 39 ETH ($115,074.18 USD).

II. Preliminary

In this section, we further explain the defects, and introduce the security issues caused by them. After that, we define the threat model of the work, and provide a motivating example to illustrate how *Emerium* detects a defect.

A. Security Issues Caused by Fragile NFT Binding (D1)

Fig. 2 shows that an NFT contract stores on-chain metadata for each minted NFT, which is a URL pointing to off-chain data in a JSON file (`nft.json`). Off-chain data, generated using

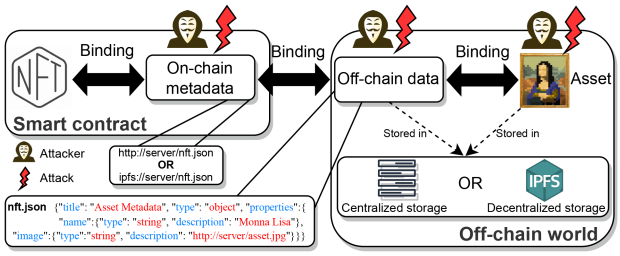

Fig. 2. Security issues caused by fragile NFT binding.

the ERC-721 or ERC-1155 Metadata JSON Schema [4], [5], contains information about the asset linked to the NFT, such as its name, description, and an image URL pointing to the asset. For example, `nft.json` includes an image field with the URL `http//server/asset.jpg` for an asset `Mona Lisa`. The off-chain data and the asset can be stored on centralized servers (e.g., AWS) or decentralized systems (e.g., Inter Planetary File System, IPFS [22]).

Incorrect binding: If an NFT contract is exploited to tamper the on-chain metadata in the contract storage, then the assets recorded by the metadata will be changed, which makes the NFTs in the contract represent incorrect assets. Besides, storing off-chain data or an asset in centralized storage (e.g., http server) risks data integrity, as it lacks the design for verification, allowing potential tampering. **Unavailable binding.** Employing centralized storage suffers from the data unavailability due to potential node failure. In both cases, once the on-chain metadata, off-chain data, or the asset is tampered with or unavailable, the NFT loses its value.

B. Security Issues Caused by Non-Compliant Implementation (D2)

ERC-721 (Section II-F) and ERC-1155 (Section II-G) define the standard ABIs (Application Binary Interface) of multiple functions and events, and expect developers to implement them accordingly. However, non-compliant implementation still exists due to the limited developing experience or malicious attempts. We summarize 3 types of non-compliant implementation that can cause security issues.

Disabled functionality: Decentralized applications (DApps), such as NFT markets [1] and Decentralized Exchanges (DEX) [23], use standard ABIs to interact with NFT contracts. For example, OpenSea [18], the largest NFT marketplace, calls the `safeTransferFrom` function of an ERC-721 contract to transfer an NFT from seller to buyer. Incorrectly implemented ABIs can make an NFT contract's functionality unavailable, disrupting trades on major markets and causing economic losses. Because users often mint NFTs with expensive cost [24], hoping to profit from future value increases by selling them.

Incorrect check: An NFT smart contract must implement intact checks in its standard functions to prevent malicious exploits. For example, the ERC-721 `transferFrom` function, shown in List. 1, transfers an NFT (`_tokenId`) from the seller (`_from`) to the buyer (`_to`). It must verify that the seller is the NFT's owner (line 3 of List. 1), ensuring correct parameter semantics. If this check fails, the Ethereum Virtual Machine (EVM) reverts the execution to prevent exploits. *Emerium* identified a flaw in the Evoh Llama Frens contract [25], with a trading volume of 39 ETH ($115,074.18 USD), which improperly implements this check (List. 2), allowing an attacker to fake NFT trades on OpenSea market (CVE-2022-35621).

```
1 function transferFrom(address _from, address _to,
      uint256 _tokenId) public {
2     ...
3     require(ownerOf(_tokenId) == _from);
4     require(_to != address(0));
5     ...
6 }
```

Listing 1: Checking the semantics of parameters

```
1 function transferFrom(address _from, address _to,
      uint256 _tokenId) external {
2     _transfer(_from, _to, _tokenId);
3 }
4 function _transfer(address _from, address _to,
      uint256 _tokenId) internal {
5     ...
6     address owner = ownerOf(_tokenId);
7     if (msg.sender == owner || getApproved(_tokenId)
          == msg.sender || isApprovedForAll(owner,
          msg.sender)
8     ){...}
9     revert("Caller is not owner nor approved");
10 }
```

Listing 2: Evoh Llama Frens that fails to check parameter `_from`: 0xf883ab97ed3d5a9af062a65b6d4437ea015efd8a

Additionally, `transferFrom` must verify that the buyer is not the zero address (line 4 of List. 1). If the buyer is zero, the NFT will be transferred to an invalid account and locked permanently. Failing to implement this check correctly can result in significant economic losses for users.

Incorrect notification: Ethereum smart contracts emit events to inform other off-chain entities to act accordingly [16]. Off-chain entities connect to Ethereum JSON-RPC API and subscribe the standard events to perceive the activity of a smart contract. For example, as shown in List. 3, `transferFrom` function of ERC-721 emits `Transfer` event to inform other entities that the NFT `_tokenId` has been transferred from the seller `_from` to the buyer `_to` (line 3).

```
1 function transferFrom(address _from, address _to,
      uint256 _tokenId) public {
2     ...
3     emit Transfer(_from, _to, _tokenId);
4     ...
5 }
```

Listing 3: Event notification

OpenSea [18], the largest NFT marketplace, subscribes to each `Transfer` event to track the transfer history of NFTs, which is displayed on the selling page to help users assess the NFT's value. NFTs sold by well-known artists or with more transaction history are often valued higher. If `Transfer`

event is missed in `transferFrom` function, OpenSea cannot capture the full transfer history, causing buyers to undervalue the NFT, leading to potential financial losses for sellers. Incorrectly setting `Transfer` event parameters can also result in off-chain entities, such as cross-chain relay servers, acting on false NFT trades, causing inconsistencies in cross-chain transfers [26]. Additionally, the "fake deposit" attack in 2018 [27], [28] exploited missing events to deceive exchange markets.

C. Security Issues Caused by Implanted Backdoor (D3)

Tampering storage: NFT contracts maintain crucial information in storage, and backdoor code left by developers can be exploited to manipulate this data, causing significant losses to users. For example, List. 4 shows a backdoor in the Sandbox NFT contract, which has a trade volume of 157.5 K ETH ($464.57 million USD) [29]. This backdoor allows an attacker to burn any user's NFTs. Specifically, `_owners[id]` (line 3) records the owner of the NFT `id`. The developer intended for `_burn` function (line 1) to be internal and delete NFT ownership, but mistakenly made it a public function. As a result, an attacker could set `from` parameter to `owner` and call `_burn`, tampering with the NFT ownership.

```
1 function _burn(address from, address owner, uint256
      id) public {
2     require(from == owner, "not owner");
3     _owners[id] = 2**160; // cannot mint it again
4     ...
5 }
```

Listing 4: A backdoor of Sandbox NFT: 0x50f5474724e0Ee42D9a4e711ccFB275809Fd6d4a

Faking event: Since off-chain entities subscribe to standard ERC-721 or ERC-1155 events to act accordingly, a backdoor can emit misleading events to deceive these entities. For instance of List. 5, *Emerium* uncovered a backdoor in the Su Squares NFT contract, which has a trade volume of 57 ETH ($87,515.52 USD) [30]. This backdoor allows the contract owner to emit arbitrary `Transfer` events, falsely appearing as legitimate trade history on OpenSea. Specifically, the owner can call the `grantToken` function with any `_tokenId` and `_newOwner` (line 1) to emit a `Transfer` event with arbitrary values for `_to` and `_tokenId` (line 7), faking a trade for any buyer and NFT.

```
1 function grantToken(uint256 _tokenId, address
      _newOwner) external onlyOperatingOfficer ... {
2     ...
3     _transfer(_tokenId, _newOwner);
4 }
5 function _transfer(uint256 _tokenId, address _to)
      internal {
6     ...
7     emit Transfer(from, _to, _tokenId);
8 }
```

Listing 5: A backdoor of Su Squares NFT: 0xe9e3f9cfc1a64dfca53614a0182cfad56c10624f

```
1 contract ERC721{
2 ...
3 function transferFrom(address _from, address _to,
      uint256 _tokenId) public
4 {
5     if (msg.sender != ownerOf(_tokenId) &&
6     msg.sender != getApproved(_tokenId) &&
7     !IsApprovedForAll(ownerOf(_tokenId),msg.sender))
8     {
9         revert();
10    }
11    ...
12 }
13 function ownerOf(uint256 _tokenId) public view
      returns (address) {
14    ...
15    return ownerMapping[_tokenId];
16 }
17 function getApproved(uint256 _tokenId) public view
      returns (address) {
18    ...
19    return approveMapping[_tokenId];
20 }
21 function IsApprovedForAll(address _owner, address
      _operator) public view returns (bool) {
22    return operatorMapping[_owner][_operator];
23 }}
```

C1 Not(==(msg.sender,ownerMapping[_tokenId]))
C2 Not(==(msg.sender,approveMapping[_tokenId]))
C3 Not(operatorMapping[ownerMapping[_tokenId]][msg.sender])

Fig. 3. A motivating example to explain detecting a defect.

D. Threat Model

Threats targeting NFT contracts: An adversary inspects the bytecode or source code of an NFT contract to exploit vulnerabilities and tamper with its storage, altering critical information. This involves modifying on-chain metadata to diminish the value of NFTs or tampering with ownership and authority management to steal them. Additionally, the adversary manipulates the contract to emit incorrect events, misleading off-chain entities into taking improper actions.

Threats targeting off-chain data bound with NFT contracts: An adversary attempts to tamper with or disable NFT data stored in off-chain storage, damaging the NFT's value. If the data is stored on a centralized server, the adversary tries to hack the server to alter the data. For example, the server owner can modify the stored information. Additionally, the adversary could launch network attacks on the server, such as Denial of Service (DoS), to render the off-chain data unavailable.

Defending goal: Emerium tries to identify potential defects of NFT contracts before and after their deployment, even though their source codes are unavailable. The input of *Emerium* contains the bytecode and ABI of a contract. If the detected contract is deployed to blockchain, its existing transactions can be used for enhancing detection performance.

E. Motivating Example

We use an ERC-721 contract example of Fig. 3 to explain **1.** what is a defect? **2.** how to detect such a defect? **3.** what are the challenges? **4.** how does *Emerium* solve them?

What is a defect? ERC-721 specifies that in `transferFrom` function, *only an owner, a controller, and an operator are authorized to transfer an NFT*. In NFT-domain-specific knowledge, owner is the account that holds and can manage (transfer/burn) the NFT. A controller can manage an NFT on behalf of the owner, while an operator can manage all of the owner's NFTs. Therefore, `transferFrom` function must verify whether the caller is an owner, controller, or operator, and if not, it should

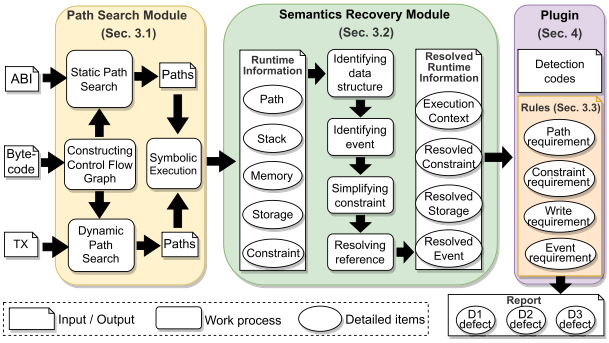

Fig. 4. Design and workflow of *Emerium*.

revert the transaction. Failing to implement this check correctly is considered as a defect.

How to detect a defect? We first present the correct implementation of the check, followed by how to determine if it exists by only analyzing the contract bytecode. The example checks if the caller is the owner of the NFT `_tokenId` (line 5 of Fig. 3) using `ownerOf` function, as specified by ERC-721 (line 13). Line 6 verifies if the caller is the controller of NFT `_tokenId` using the `getApproved` function of ERC-721 (line 17). Line 7 checks if the caller is the operator of the owner via `isApprovedForAll` function (line 21). In line 22, if `isApprovedForAll` returns true, then `_operator` is the operator of `_owner`. If line 5-7 are satisfied, line 9 triggers a revert. Therefore, if the path line 5→line 6→line 7→line 9 (denoted by p) exists in `transferFrom` function, the check is correctly implemented, indicating no defect. Thus, detecting the defect is reduced to proving the existence of p.

We prove the existence of p by searching for a path, p_b, consisting of bytecode basic blocks that can be mapped to p. Specifically, if p_b shares the same execution context and path constraints as p, it can be mapped to p. The execution context of a path describes the function that the path belongs to, and the end of the path. For example, p belongs to the `transferFrom` function and ends with `revert()`. Therefore, p_b should also belong to `transferFrom` and end with the EVM `REVERT` instruction. A path constraint defines the condition of variables that trigger a specific execution path [31]. As shown in Fig. 3, p has three path constraints: c_1, c_2, and c_3. We express these constraints using Z3 expressions [32].

c_1 indicates that `msg.sender` is not equal to `ownerMapping [_tokenId]`, meaning the caller is not the owner of NFT `_tokenId`. This is because the one-dimensional mapping `ownerMapping` (line 15) stores owner information. For example, `ownerMapping[_tokenId]` records the owner of the NFT `_tokenId`. c_2 shows that the caller is not the controller of NFT `_tokenId`, as `approveMapping` (line 19) records controller information. For instance, `approveMapping[_tokenId]` records the controller of NFT `_tokenId`. c_3 indicates that the caller is not the operator of the owner of NFT `_tokenId`. Differently from owner and controller, operator information is recorded in a two-dimensional mapping `operatorMapping` (line 22). `operatorMapping[owner][user]` records whether `user` is an operator of `owner`. If `operatorMapping[owner][user]` is true, it means `user` is an operator of `owner`. Therefore, p_b must contain constraints c_1, c_2, and c_3.

Challenges: Unfortunately, it is not trivial to find p_b due to the following issues: **a.** bytecode does not specify function context explicitly, i.e., the scope that includes all instructions belonging to a function, which requires us to distinguish the paths belonging to `transferFrom` function, and ending with `REVERT` instruction; **b.** we cannot generate c_1, c_2 and c_3 directly, because there is no concept about `ownerMapping`, `approveMapping`, `operatorMapping`, or `_tokenId` in bytecode.

Solutions: To solve challenge **a.**, we propose Path Search Module (Section III-A) that is capable of searching the paths belonging to a specific function and ending with a specific instruction (e.g., belonging to `transferFrom` function, and ending with `REVERT` instruction). We adopt symbolic execution to execute paths to generate constraints. To solve challenge **b.**, we design Semantics Recovery Module (SRM) (Section III-B), which tries to recover data structures (e.g., `ownerMapping`) and function parameters (e.g., `_tokenId`) during the symbolic execution.

Optimization: The motivating example fundamentally leverages an assertion "*at least one path belonging to* `transferFrom`*function, and ending with* `REVERT`*instruction, have the constraints* c_1, c_2, *and* c_3" to detect the defect. Namely, if the assertion holds, the defect does not exist. We design unified rule syntax that allows users of *Emerium* to define assertions (Section III-C), which enables the users to customize the detection capability more efficiently.

F. ERC-721 Functions & Events

We focus on the following ERC-721 functions and events for defect detection, since they fully cover the functionality of ERC-721 specification. (**F**: function, **E**: event)

F1:`safeTransferFrom(address _from,address _to, uint256 _tokenId)`, which is responsible for transferring an NFT `_tokenId` from seller `_from` to buyer `_to`.
F2:`safeTransferFrom(address _from,address _to, uint256 _tokenId,bytes data)`, which transfers an NFT `_tokenId` from seller `_from` to buyer `_to`.
F3:`transferFrom(address _from,address _to, uint256 _tokenId)`, which transfers an NFT `_tokenId` from seller `_from` to buyer `_to`.
F4:`approve(address _approved, uint256 _tokenId)`, which is used for assigning controller of an NFT `_tokenId` to `_approved`, by the owner or the operator.
F5:`setApprovalForAll(address _operator, bool _approved)`, which enables `_operator` to be the operator of function caller when `_approved` is 1, and disables the operator when `_approved` is 0.
E1:`Transfer(address _from, address _to, uint256 _tokenId)`, which announces the an NFT `_tokenId` is transferred from `_from` to `_to`.
E2:`Approval(address _owner, address _approved, uint256 _tokenId)`, which announces `_owner` assigns the controller authority about an NFT `_tokenId` to `_approved`.
E3:`ApprovalForAll(address _owner, address _operator, bool _approved)`, which announces `_owner` assigns or revokes the operator authority of `_operator`. When `_approved` is one, it means assigned; else, it means revoked.

G. ERC-1155 Functions & Events

We focus on the following ERC-1155 functions and events for defect detection. (**F**: function, **E**: event)

F6:`safeTransferFrom(address _from, address _to, uint256 _id, uint256 _value, bytes calldata _data)`, which transfers NFT `_id` from seller `_from` to buyer `_to`.
F7:`safeBatchTransferFrom(address _from, address _to, uint256[] _ids, uint256[] _values, bytes calldata _data)`, which transfers NFT `_ids` from seller `_from` to buyer `_to`.
F8:`setApprovalForAll(address _operator, bool _approved)`, which enables `_operator` to be the operator of function caller when `_approved` is 1, and disables the operator when `_approved` is 0.
E4:`TransferSingle(address _operator, address _from, address _to, uint256 _id, uint256 _value)`, which announces that the NFT `_id` is transferred from `_from` to `_to`. `_operator` is the account that makes the transfer.
E5:`TransferBatch(address _operator, address _from, address _to, uint256[] _ids, uint256[] _values)`, which announces that the NFT `_ids` are transferred from `_from` to `_to`. `_operator` is the account/contract that makes the transfer.
E6:`ApprovalForAll(address _owner, address _operator, bool _approved)`, which announces `_owner` assigns or revokes the operator authority of `_operator`. When `_approved` is one, it means assigned; else, it means revoked.

III. Design

In this section, we introduce the architecture of *Emerium*, which includes Path Search Module (PSM), Semantics Recovery Module (SRM), Rule Module (RM), as shown in Fig. 4. According to the motivating example (Section II-E), detecting defects rely on analyzing the existence of specific paths, so *Emerium* first adopts PSM to search paths (Section III-A). After obtaining the paths, *Emerium* executes paths symbolically to collect runtime information, including executed path, EVM stack, memory read/write, storage read/write, and path constraint.

Unfortunately, runtime information lacks NFT-domain-specific semantics, as bytecode does not explicitly provide concepts about data structures, events, or function parameters in an NFT contract. To address this, *Emerium* uses the SRM to recover these concepts from runtime information, and resolve NFT-domain-specific semantics of them (Section III-B). The resolved information is then used by the plugins (Section IV) created by users to detect defects. The plugins can directly execute detection codes or, more efficiently, by defining precise rules (Section III-C) for identifying defects. These rules synthesizes assertions based on four dimensions: path requirements, constraint requirements, write requirements, and event requirements. These assertions are matched with resolved runtime information to determine whether a defect exists. The following sections will detail each module of *Emerium*.

A. Path Search Module (PSM)

Each path consists of basic blocks that are executed sequentially. A basic block is a continuous sequence of instructions without any branching, as shown in Fig. 5. Within each block, a `JUMP` or `JUMPI` instruction (e.g., at address 42) guides the execution from one block (e.g., block 0) to the next (e.g., block 4) by setting the program counter to the address of the first instruction of the next block (e.g., address 113). Given the *path specification* that defines the function to which a path belongs

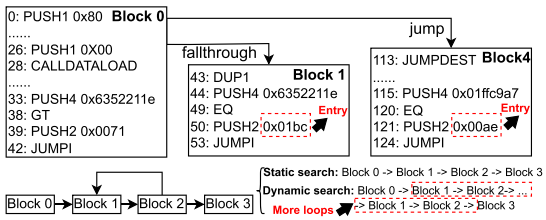

Fig. 5. The example of basic block, path, static path search, and dynamic path search.

and its last instruction, PSM searches for the path within the EVM runtime bytecode. PSM employs two mechanisms: static path search and dynamic path search.

Static path search: In this mechanism, PSM first constructs a control flow graph from the runtime bytecode and searches for paths within the graph. PSM supports function-level path searches, allowing for finer analysis. To achieve this, PSM must identify the entry point of a function, which is the address of the first instruction executed by that function. Following the approach of DEFECTCHECKER [33], PSM scans for a basic block containing the instruction `PUSH4` *funcSign*, followed by `EQ` and `PUSH2` *funcEntry* instruction, where *funcSign* is the function's signature and *funcEntry* is its entry. For example, in Fig. 5, the address 50 contains the entry point `0x01bc`. The ending instruction (e.g., `REVERT`) is located by scanning basic blocks. Given a specification to search for paths belonging to function α and ending with instruction β, PSM outputs paths starting from address 0, passing through the entry of α, and ending with β.

Dynamic path search: Static path search struggles to find all feasible paths using a control flow graph. Since paths may contain loops, limiting loop counts during the search is necessary to avoid path explosion, but this compromises the soundness of the search. To address this, when static path search fails to find a path under a given specification, *Emerium* uses dynamic path search, which extracts paths from the execution of existing transactions. Because each transaction execution corresponds to the execution of a specific path. Given a path specification, PSM queries the contract's transactions using the `txlist` API provided by Etherscan and filters transactions based on the first 4 bytes (function signature) of the transaction data. Once the transaction is identified, PSM replays it to obtain the instruction execution trace. PSM then splits this trace into basic blocks and connects them to form the path. As shown in Fig. 5, dynamic path search can identify paths with more loops.

B. Semantics Recovery Module (SRM)

SRM recovers data structures, events, and function parameters from the runtime information collected through symbolic execution. It identifies data structures and events through access patterns in the execution. Additionally, since constraints generated by symbolic execution may contain noise that obscures semantics (c.f. Section III-B-2), SRM simplifies these constraints to remove noise. Finally, SRM resolves references pointing to data

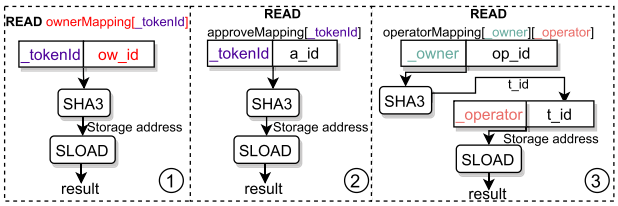

Fig. 6. The pattern for accessing role-related data.

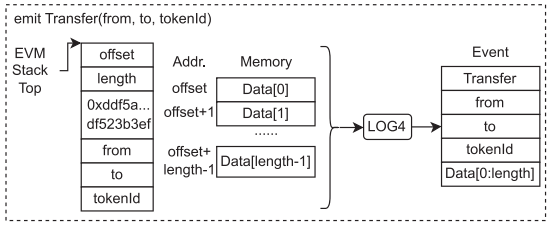

Fig. 7. The pattern for emitting `Transfer` event.

structures or function parameters, in events, storage read/write operations, and simplified constraints. The result is resolved runtime information enriched with NFT-domain-specific semantics for on-demand retrieval. In the following paragraphs, we will explain SRM's processes in detail.

1) Recovering Data Structure and Event: Identifying data structure patterns: An NFT contract uses mappings to store information. We define role-related data as the structure that records the owner, controller, or operator information in contract storage. For example, `ownerMapping` maps an NFT to its owner, as seen in line 15 of Fig. 3. EVM accesses (reads/writes) role-related data following specific patterns, known as *data structure pattern*. If SRM detects these patterns, it can recover role-related data during symbolic execution.

Fig. 6 provides three examples of patterns when EVM reads role-related data. In ①, when EVM reads `ownerMapping[_tokenId]`, it concatenates `_tokenId` with `ow_id` (the index of `ownerMapping` in contract storage) and computes its hash using `SHA3` instruction. This hash value serves as the storage address where the result is located. Since EVM storage operates as a key-value store, the hash is used as a key to retrieve the value via the `SLOAD` instruction. Similarly, ② and ③ show how the EVM reads controller and operator data, respectively. In pattern ①, if SRM knows the concrete value of `ow_id` in advance and identifies this pattern during symbolic execution, it can recognize the read operation on `ownerMapping` and recover it.

To obtain the concrete value of `ow_id`, *Emerium* symbolically executes `ownerOf` function of ERC-721 and monitors the storage reads, extracting `ow_id`. This works because `ownerOf` must read `ownerMapping` to query an owner, as seen in line 15 of Fig. 3. Similarly, `a_id` (the index of `approveMapping` for controllers) is extracted from the `getApproved` function, and `op_id` (the index of `operatorMapping` for operators) is identified through the `isApprovedForAll` function of ERC-721.

Identifying event: EVM executes `LOG` instruction to emit an event, with its parameters saved in EVM stack and memory. Therefore, SRM identifies an event by analyzing the content of EVM stack and memory when `LOG` is executed in symbolic execution. For instance of Fig. 7, when EVM emits an `Transfer` event of ERC-721, the top-6 elements of EVM stack are `offset`, `length`, `0xddf252ad...ef`, `_from`, `_to`, and `_tokenId` individually. `Data[0]-Data[length-1]` stored in memory constitutes the data field of the event.

2) Simplifying Constraint: Constraints generated by symbolic execution often contain two types of noise: discrete symbols and redundant logic. Since this noise obscures the underlying semantics, we address it through symbol recombination and redundancy elimination to simplify the constraints.

Discrete symbols and symbol recombination: Discrete symbols are generated when a constraint on an array is split into multiple sub-constraints on each item of the array. For example, assume a parameter representing a seller is stored in an array `CALLDATA[4,24]`, where `4` and `24` are the starting offset and ending offset. Then `==(CALLDATA[4,24],0x000...0)` is a constraint that describes `CALLDATA[4,24]` is equal to `0x000...0`, which implies the semantics that the seller is equal to zero account. However, in practical scenario, Z3 [32] organizes this constraint with the form of multiple sub-constraints, `And(==(CALLDATA[24],0x0),...,==(CALLDATA[4],0x0))`, where each `==(CALLDATA[_],0x0)` is a sub-constraint, and these sub-constraints are connected by a `And` operator (`And` represents its operands are all true). Discrete symbols, i.e., `And` operator and split `==(CALLDATA[_],0x0)` expression, make it difficult to reflect the semantics that the seller is equal to zero account, as split `CALLDATA[_]` cannot represent the seller. To remove this kind of noise, SRM recombines the symbols within split sub-constraints to create a new symbol that reflects semantics. For instance, split `CALLDATA[_]` is recombined to be a new symbol `CALLDATA[4,24]` that represents the buyer.

Redundant logic and redundancy elimination: Redundant logic is caused by using symbolic expressions containing `==` operator, as intermediate results in EVM stack or memory. For example, EVM compares α and 0 with `EQ` instruction. The symbolic result `==(`α`,0x0)` is stored in stack as a intermediate result. If another `EQ` instruction compares this intermediate result and 0 again, `==(==(`α`,0x0),0x0)` will be generated as a new intermediate result. When the new intermediate result is compared with 0 to generate a constraint, i.e., `==(==(==(`α`,0x0),`**`0x0)`**`,`**`0x0)`**, then this constraint contains the redundant logic `==(==(_,`**`0x0)`**`,`**`0x0)`**. This is because the constraint is logically equivalent to `==(`α`,0x0)` (a more simplified form) reflecting the semantics that α is equal to zero. Due to the redundant logic, such semantics cannot be reflected. On the other hand, redundant logic can also be caused by `If` operator. Specifically, **If(**α**, 0x1,0x0)** represents if α is `0x1`, then the result of this whole expression is `0x1`, else it will be `0x0`. In this expression, **If(_, 0x1,0x0)** is redundant logic, as

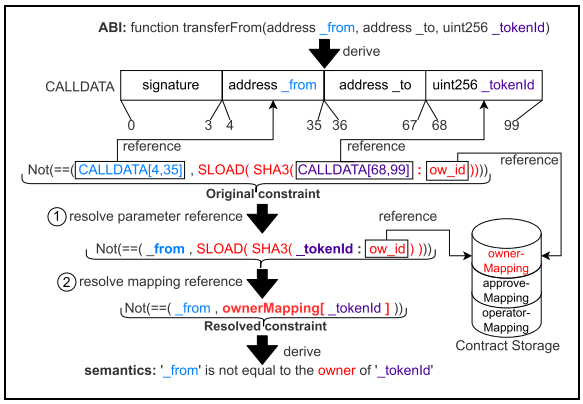

Fig. 8. Resolving parameter and mapping reference.

the expression is logically equivalent to α. The redundant logic prevents the expression from reflecting the semantics that the expression is equal to α. To remove this noise, SRM eliminates redundant $==$ and `If` operators in expressions.

3) Resolving Reference: Resolving reference refers to recovering the function parameters or data structures in simplified constraint, storage, and event. This process includes resolving parameter reference and resolving mapping reference.

Resolving parameter reference: We use the example of Fig 8 to explain resolving parameter/mapping reference. There is an original constraint generated in the execution context of `transferFrom` function of ERC-721. The target of SRM is to resolve the parameter/mapping references within the original constraint, to generate the resolved constraint that reflects the semantics "`_from` *is not equal to the owner of* `_tokenId`". According to the contract ABI, `_from` is stored in `CALLDATA[4,35]` (i.e., the reference of `CALLDATA[4,35]` is `_from`); `_tokenId` is stored in `CALLDATA[68,99]` (i.e., the reference of `CALLDATA[68,99]` is `_tokenId`). After resolving parameter reference, the constraint becomes `Not(==(from,SLOAD(SHA3(_tokenId:ow_id))))`.

Resolving mapping reference: In the above constraint, `SLOAD(SHA3 (_tokenId:ow_id))` is generated by two steps of EVM. The first step is, EVM concatenates `_tokenId` and `ow_id` (generate `_tokenId:ow_id`), and the result is used for calculating a hash value by `SHA3` instruction (generate `SHA3(_tokenId:ow_id)`). The second step is, EVM reads the storage by `SLOAD` instruction with the address `SHA3(_tokenId:ow_id)`. Since these two steps are the pattern to read `ownerMapping` (see ① of Fig. 6), SRM directly replaces `SLOAD(SHA3(_tokenId:ow_id))` with `ownerMapping [_tokenId]`. Finally, the constraint becomes `Not(==(_from,ownerMapping [_tokenId]))` that reflects the semantics intuitively.

C. Rule Module (RM)

RM adopts rules to precisely define the assertion that resolved runtime information from SRM should satisfy. An assertion can by synthesized by 4 dimensions: path requirement, constraint requirement, write requirement, and event requirement that characterize execution context, resolved constraint, resolved storage, and resolved event individually. The assertion is formalized as $< \mathbb{P}, \mathbb{C}, \mathbb{W}, \mathbb{E} >$. $\mathbb{P} = \{p_1, p_2, \ldots\}$, representing the set of paths (also named path specification), and p_i means a path. $\mathbb{C}$ is the set of constraints, and $\mathbb{C} = \{Constraint(p_1), Constraint(p_2), \ldots\}$, where $Constraint(p_i)$ represents the set of constraints generated by executing path p_i. $\mathbb{W}$ is the set of storage write operations, and $\mathbb{W} = \{Write(p_1), Write(p_2), \ldots\}$, where $Write(p_i)$ is the set of the write operations that happen during the execution of p_i. $\mathbb{E}$ is the set of events, and $\mathbb{E} = \{Event(p_1), Event(p_2), \ldots\}$, where $Event(p_i)$ is the set of the events that are emitted during the execution of p_i.

To describe the elements of $\mathbb{P}$, we define $Search(start, end)$, where $start$ is the set of entries (defined as $start = \{e_1, e_2, \ldots\}$, e_i is entry, i.e., the first EVM instruction of function or contract), and end is the set of EVM instructions. $Search(start, end)$ represents the set of the paths searched by PSM with the specification "passing the entry of $start$, ending with the instructions of end". Note that all paths start from the entry of a contract (the instruction whose address is 0). Here we use the signature of a function to represent the entry of this function. A function signature is a 4-byte string to represent a smart contract function [34]. For example, $Search(\{0{\times}42842e0e, 0{\times}23b872dd0\}, \{STOP\})$ represents the set of paths coming from the two groups: 1. the paths passing the entry of F1 function (whose signature is `0x42842e0e`), ending with `STOP` instruction; 2. the paths passing the entry of F3 function (whose signature is `0x23b872dd0`), ending with `STOP` instruction. Specially, we use $ROOT$ to represent the entry of a contract (i.e., 0 address).

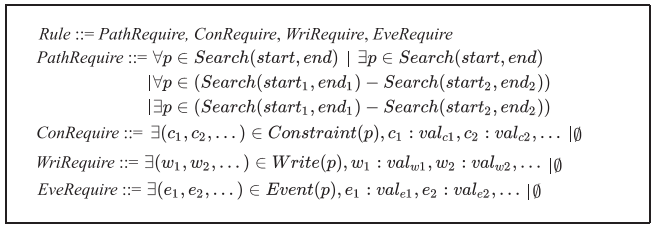

Fig. 9. Rule syntax.

Finally, rule can be formalized with the syntax shown in Fig. 9. Rule consists of four parts: *PathRequire*, *ConRequire*, *WriRequire*, and *EveRequire*. *PathRequire* describes the path requirement of rule, i.e., the elements of $\mathbb{P}$. '$\forall p \in Search(start, end)$' means 'for all paths of $Search(start, end)$'. '$\exists p \in Search(start, end)$' means 'there exists a path of each group of $Search(start, end)$ at least'. '$(Search(start_1, end_1) - Search(start_2, end_2))$' means excluding the paths of $Search(start_2, end_2)$ from $Search(start_1, end_1)$. '|' means either its left side or right side can be derived. *ConRequire* describes the element $Constraint(p)$ of $\mathbb{C}$, and c_i means the element of $Constraint(p)$. '$c_i : val_{ci}$' represents the value of c_i is val_{ci}. '$\emptyset$' is a terminal symbol representing empty. *ConRequire* can derive empty, which means there is no assertion

about $\mathbb{C}$. Similarly, *WriRequire* describes the element of $\mathbb{W}$, and *EveRequire* describes the element of $\mathbb{E}$. An example of a rule:
$\exists p \in Search(\{0{\times}23b872dd\}, \{REVERT\}), \exists c{\in}Constraint(p),$
$c : Not(== (\_from, ownerMapping[\_tokenId])).$

The rule asserts that there is a path for executing the F3 function (with the function signature $0{\times}23b872dd$) that ends with the `REVERT` instruction. Additionally, there is a constraint during the execution: `Not(==(_from,ownerMapping[_tokenId]))`.

IV. Plugins to Detect Defect

In this section, based on the extensible framework, we design plugins named NFT Binding Analyzer, Functionality Analyzer, and Backdoor Analyzer to detect D1, D2, and D3 individually.

A. Plugin 1: NFT Binding Analyzer (NBA)

NBA detects three kinds of D1:

D1.1. tampering on-chain metadata: On-chain metadata is a URL stored in contract storage that points to a JSON file describing the asset linked to an NFT. If functions modify the on-chain metadata, the associated asset is replaced, leading to significant loss. To detect the presence of D1.1, NBA checks for write operations on the on-chain metadata outside of mint or burn functions. Since metadata is updated during the minting or burning process, these specific write operations are excluded from detection.

To identify minting or burning functions, we use a heuristic approach since there is no standard for their signatures. Based on the contract ABI [35], we consider functions with the keyword 'mint' in their name as minting functions (denoted by `M`) and those with 'burn' as burning functions (denoted by `B`). Since ERC-721 provides `tokenURI` function to query on-chain metadata (ERC-1155 uses `uri`), we treat the mapping identified in these functions (denoted as `metaMapping`) as the storage for on-chain metadata. We then define the following rule 1 to detect D1.1.

$\neg\exists p{\in}(Search(\{ROOT\}, \{RETURN, STOP\}) - Search(\{M, B\}, \{RETURN, STOP\})),$
$\exists w{\in}Write(p), w : metaMapping[*] : *$

where * represents any values. The rule describes the assertion: there is no such a path that satisfies **a.** the path does not belong to the functions for minting and burning an NFT, and **b.** there is a writing operation on `metaMapping` during the execution of the path.

D1.2. off-chain data is invalid: Off-chain data includes the JSON file pointed to by on-chain metadata and the asset referenced by the image URL within the JSON file. Since this data is stored off-chain, it can become invalid due to unexpected issues, such as server crashes. For example, an $11 million NFT album went missing [10]. This defect cannot be detected through contract code analysis alone, so NBA collects and verifies off-chain data. Specifically, given an NFT contract address, NBA queries E1 events from the blockchain and filters events where the first parameter is 0, as these indicate an NFT has been minted. The third parameter of such events provides the ID of the minted NFT. Using this ID, NBA calls `tokenURI` function to retrieve the on-chain metadata and locate the off-chain data. NBA accesses off-chain data by sending HTTP or IPFS requests [22] to check whether it is valid. This example also demonstrates how plugins can integrate additional functionality into detection, beyond simply using rules.

D1.3. off-chain data is stored in a centralized server: If off-chain data is stored on a centralized server (e.g., an HTTP server), it is vulnerable to tampering by the server owner, who can modify the stored data. For example, the NFT collection provider 'neitherconfirm' replaced 18 NFT assets stored on a centralized server [36]. Additionally, centralized servers are more susceptible to Denial-of-Service attacks and node failures, increasing the risk of losing off-chain data. To detect this defect, NBA uses regular expression matching to identify the type of link pointing to the off-chain data and determine if the defect is present.

B. Plugin 2: Functionality Analyzer (FA)

FA systematically detects 17 kinds of D2 defects that are defined by negating ERC-721 and ERC-1155 specification [4], [5]. For better readability, we use F1-E3 (c.f. Section II-F) to represent the functions and events specified by ERC-721. We use F6-E6 (c.f. Section II-G) to represent the functions and events specified by ERC-1155.

D2.0. disabled functionality: If an ERC-721 contract lacks the functions or events specified by ERC-721, or an ERC-1155 contract lacks the functions or events specified by ERC-1155, then this is a D2.0 defect. FA uncovers such a defect by finding the inconsistency between contract ABIs and the corresponding standard, in terms of function/event names and function/event parameter types.

1) Non-Compliant ERC-721 Implementation. D2.1. incorrect caller check: If F1, F2, and F3 do not revert when the caller of the function is not owner, controller, nor operator, then this is a D2.1 defect. For instance, motivating example (Section II-E) illustrates the case that F3 reverts when the caller of the function is not owner, controller nor operator. We design the following rule 2 to detect a D2.1 defect.

$\exists p{\in}Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde\}, \{REVERT\}), \exists(c_1, c_2, c_3){\in}Constraint(p),$
$c_1 : Not(== (msg.sender, ownerMapping[\_tokenId]))\vee Not(== (ownerMapping[\_tokenId], msg.sender)),$
$c_2 : Not(== (msg.sender, approveMapping[\_tokenId]))\vee Not(== (approveMapping[\_tokenId], msg.sender)),$
$c_3 : Not(operatorMapping[ownerMapping[\_tokenId]][msg.sender])$

In this rule, $Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde\}, \{REV\ \ ERT\})$ means selecting the paths starting from the entry of F1-F3 functions (`0x42842e0e` is the signature of F1, `0xb88d4fde` is the signature of F2, `0x23b872dd` is the signature of F3), ending with `REVERT` instruction. This guarantees these paths are in F1, F2, or F3, and they end with `REVERT` instruction. The rule describes, for the paths of Fi (i.e., F1, F2, and F3), at least one path can be executed to generate c_1, c_2 and c_3. $c_i : x{\vee}y$ means c_i is x or y, so c_1, c_2, and c_3 of the rule not only consider the shapes mentioned in the motivating example, but also other possible constraint shapes.

To maintain the paper readability, lengthy rules to detect D2 defects related to ERC-721 are shown in Fig. 10.

D2.2. incorrect seller check: If F1, F2, and F3 do not revert when their parameter `_from` is not the owner of the NFT

Rule for detecting D1

$$\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{M,B\},\{RETURN,STOP\})), \exists w{\in}Write(p), w : metaMapping[*] : * \quad (1)$$

Rules for detecting D2

$$\begin{aligned}&\exists p{\in}Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde\},\{REVERT\}), \exists(c_1,c_2,c_3){\in}Constraint(p),\\ &c_1 : Not(==(msg.sender, ownerMapping[\_tokenId])){\vee}Not(==(ownerMapping[\_tokenId], msg.sender)),\\ &c_2 : Not(==(msg.sender, approveMapping[\_tokenId])){\vee}Not(==(approveMapping[\_tokenId], msg.sender)),\\ &c_3 : Not(operatorMapping[ownerMapping[\_tokenId]][msg.sender])\end{aligned} \quad (2)$$

$$\begin{aligned}&\exists p{\in}Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde\},\{REVERT\}), \exists c{\in}Constraint(p),\\ &c : Not(==(ownerMapping[\_tokenId], \_from)){\vee}Not(==(\_from, ownerMapping[\_tokenId]))\end{aligned} \quad (3)$$

$$\exists p{\in}Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde\},\{REVERT\}), \exists c{\in}Constraint(p), c : Not(\_to) \quad (4)$$

$$\exists p{\in}Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde\},\{REVERT\}), \exists c{\in}Constraint(p), c : Not(ownerMapping[\_tokenId]) \quad (5)$$

$$\begin{aligned}&\exists p{\in}Search(\{0{\times}42842e0e, 0{\times}b88d4fde\},\{REVERT\}), \exists(c_1,c_2){\in}Constraint(p), c_1 : UGT(EXTCODESIZE(\_to), 0),\\ &c_2 : Not(==(RETURNDATA, 0{\times}150b7a02)){\vee}Not(==(0{\times}150b7a02, RETURNDATA))\end{aligned} \quad (6)$$

$$\begin{aligned}&\exists p{\in}Search(\{0{\times}095ea7b3\},\{REVERT\}), \exists(c_1,c_2){\in}Constraint(p),\\ &c_1 : Not(==(msg.sender, ownerMapping[\_tokenId])){\vee}Not(==(ownerMapping[\_tokenId], msg.sender)),\\ &c_2 : Not(operatorMapping[ownerMapping[\_tokenId]][msg.sender])\end{aligned} \quad (7)$$

$$\forall p{\in}Search(\{0{\times}42842e0e, 0{\times}23b872dd\},\{STOP\}), \exists e{\in}Event(p), e : Transfer(\_from, \_to, \_tokenId) \quad (8)$$

$$\forall p{\in}Search(\{0{\times}095ea7b3\},\{STOP\}), \exists e{\in}Event(p), e : Approval(ownerMapping[\_tokenId], \_approved, \_tokenId) \quad (9)$$

$$\forall p{\in}Search(\{0{\times}a22cb465\},\{STOP\}), \exists e{\in}Event(p), e : ApprovalForAll(msg.sender, \_operator, \_approved) \quad (10)$$

Rules for detecting D3

$$\begin{aligned}&\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde, M, B\},\\ &\{RETURN,STOP\})), \exists w{\in}Write(p), w : ownerMapping[*] : *\end{aligned} \quad (11)$$

$$\begin{aligned}&\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{0{\times}095ea7b3, 0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde, B\},\\ &\{RETURN,STOP\})), \exists w{\in}Write(p), w : approveMapping[*] : *\end{aligned} \quad (12)$$

$$\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{0{\times}a22cb465\},\{STOP\})), \exists w{\in}Write(p), w : operatorMapping[*][*] : * \quad (13)$$

$$\begin{aligned}&\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{0{\times}42842e0e, 0{\times}23b872dd, 0{\times}b88d4fde, M, B\},\{STOP\})),\\ &\exists e{\in}Event(p), e : Transfer(*,*,*)\end{aligned} \quad (14)$$

$$\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{0{\times}095ea7b3\},\{STOP\})), \exists e{\in}Event(p), e : Approval(*,*,*) \quad (15)$$

$$\neg\exists p{\in}(Search(\{ROOT\},\{RETURN,STOP\}) - Search(\{0{\times}a22cb465\},\{STOP\})), \exists e{\in}Event(p), e : ApprovalForAll(*,*,*) \quad (16)$$

Fig. 10. Rules for detecting D1-D3 of ERC-721 contracts.

`_tokenId`, then this is a D2.2 defect. `_from` represents the seller of NFT `_tokenId`. If `_from` is not the owner of the NFT `_tokenId`, then the semantics of `_from` or `_tokenId` is not correct, which could lead to potential vulnerabilities. For example, the vulnerability CVE-2022-35621 uncovered by *Emerium*, is caused by D2.2 and can be exploited to fake an NFT trade in OpenSea market. We adopt rule 2 of Fig. 10 to detect such a defect.

D2.3. incorrect buyer check: If F1, F2, and F3 do not revert when parameter `_to` is zero address, then this is a D2.3 defect. `_to` represents the address of the buyer. If the NFT is transferred to zero address, then the NFT is owned by an invalid address. Namely, the NFT will be locked forever. We adopt rule 4 of Fig. 10 to detect such a defect.

D2.4. incorrect token check: If F1, F2, and F3 do not revert when parameter `_tokenId` is not a valid NFT, then this is a D2.4 defect. `_tokenId` is the NFT that is sold, which means it should be valid, i.e., its owner should not be zero (an invalid address). If this check is not implemented correctly, an invalid NFT can be transferred to a buyer. We adopt rule 5 of Fig. 10 to detect such a defect.

D2.5. incorrect buyer contract check: A D2.5 defect occurs if functions F1 and F2 do not revert when an NFT is transferred to a contract that cannot manage it. Since the buyer of an NFT can be a smart contract, the NFT will be permanently locked in the contract if it lacks functions to manage the NFT. To prevent this, ERC-721 requires F1 and F2 to call the `onERC721Received` function of the buyer contract and verify that the returned value matches the first 4 bytes of keccak256(onERC721Received(address,address,uint256,bytes)), ensuring the contract can manage NFTs. Failing to implement this check can result in unexpected NFT locking. For example, OpenSea accidentally sent 42 NFTs to addresses that no one controls, resulting in losses exceeding $100,000 [37]. We use rule 6 in Fig. 10 to detect this defect.

D2.6. unauthorized controller assignment: If F4 does not revert when the caller is neither the owner of NFT nor the operator of the NFT owner, then this is a D2.7 defect. ERC-721 assigns the controller role for an NFT to an account by calling F4. ERC-721 requires F4 to check the caller needs to be the owner of `_tokenId`, or the operator of the owner of `_tokenId`. Since the controller `_approved` can transfer `_tokenId` on behalf of its owner, unauthorized controller assignment can lead to stealing `_tokenId`. We adopt rule 7 of Fig. 10 to detect such a defect.

D2.7. incorrect transfer notification: If F1, F2, and F3 do not emit E1 correctly during their execution, then this is a D2.9 defect. ERC-721 specifies F1, F2, and F3 need to emit E1 during their execution, and the parameters of E1 should be the same as the parameters of F1, F2, and F3. If E1 is missed or emitted with incorrect parameters, then it informs off-chain entities of incorrect token transfer behaviors. We use rule 8 of Fig. 10 to detect such a defect.

D2.8. incorrect controller notification: If F4 does not emit E2 correctly during its execution, then this is a D2.10 defect. ERC-721 specifies F4 needs to emit E2 during its execution, and the

Rule for detecting D2

$$\exists p{\in}Search(\{0{\times}f242432a, 0{\times}2eb2c2d6\}, \{REVERT\}), \exists(c_1, c_2){\in}Constraint(p), c_1 : Not(==(msg.sender, \_from)){\vee}Not(==(\_from, msg.sender)), c_2 : Not(operatorMapping[\_from][msg.sender]) \quad (17)$$

$$\exists p{\in}Search(\{0{\times}f242432a, 0{\times}2eb2c2d6\}, \{REVERT\}), \exists c{\in}Constraint(p), c : Not(\_to) \quad (18)$$

$$\exists p{\in}Search(\{0{\times}f242432a\}, \{REVERT\}), \exists c{\in}Constraint(p), c : UGT(\_value, balanceMapping[\_id][\_from]) \quad (19)$$

$$\exists p{\in}Search(\{0{\times}f242432a\}, \{REVERT\}), \exists(c_1, c_2){\in}Constraint(p), c_1 : UGT(EXTCODESIZE(\_to), 0), c_2 : Not(==(RETURNDATA, 0{\times}f23a6e61)){\vee}Not(==(0{\times}f23a6e61, RETURNDATA)) \quad (20)$$

$$\exists p{\in}Search(\{0{\times}2eb2c2d6\}, \{REVERT\}), \exists(c_1, c_2){\in}Constraint(p), c_1 : UGT(EXTCODESIZE(\_to), 0), c_2 : Not(==(RETURNDATA, 0{\times}bc197c81)){\vee}Not(==(0{\times}bc197c81, RETURNDATA)) \quad (21)$$

$$\forall p{\in}Search(\{0{\times}2eb2c2d6\}, \{STOP\}), \exists e{\in}Event(p), e : TransferBatch(msg.sender, \_from, \_to, *, *) \quad (22)$$

$$\forall p{\in}Search(\{0{\times}a22cb465\}, \{STOP\}), \exists e{\in}Event(p), e : ApprovalForAll(msg.sender, \_operator, \_approved) \quad (23)$$

Rule for detecting D3

$$\neg\exists p{\in}(Search(\{ROOT\}, \{RETURN, STOP\}) - Search(\{0{\times}f242432a, 0{\times}2eb2c2d6, M, B\}, \{RETURN, STOP\})), \exists w{\in}Write(p), w : balanceMapping[*] : * \quad (24)$$

$$\neg\exists p{\in}(Search(\{ROOT\}, \{RETURN, STOP\}) - Search(\{0{\times}a22cb465\}, \{STOP\})), \exists w{\in}Write(p), w : operatorMapping[*][*] : * \quad (25)$$

$$\neg\exists p{\in}(Search(\{ROOT\}, \{RETURN, STOP\}) - Search(\{0{\times}f242432a, M, B\}, \{STOP\})), \exists e{\in}Event(p), e : TransferSingle(*, *, *, *, *) \quad (26)$$

$$\neg\exists p{\in}(Search(\{ROOT\}, \{RETURN, STOP\}) - Search(\{0{\times}2eb2c2d6, M, B\}, \{STOP\})), \exists e{\in}Event(p), e : TransferBatch(*, *, *, *, *) \quad (27)$$

$$\neg\exists p{\in}(Search(\{ROOT\}, \{RETURN, STOP\}) - Search(\{0{\times}a22cb465\}, \{STOP\})), \exists e{\in}Event(p), e : ApprovalForAll(*, *, *) \quad (28)$$

Fig. 11. Rules for detecting D2-D3 of ERC-1155 contracts.

parameter `_owner` of E2 should be the same as the owner of the parameter `_tokenId` of F4. Also, the parameters `_approved` and `_tokenId` of E2 should be the same as the parameters of F4. If E2 is missed or emitted with incorrect parameters, then it informs off-chain entities of incorrect controller information. We use rule 9 of Fig. 10 to detect such a defect.

D2.9. incorrect operator notification: If F5 does not emit E3 correctly during its execution, then this is a D2.11 defect. ERC-721 specifies F5 needs to emit E3 during its execution, and the parameter `_owner` of E3 should be the caller of F5; the parameters `_operator` and `_approved` of E3 should be the same as the parameters of F5. If E3 is missed or emitted with incorrect parameters, then it informs off-chain entities of incorrect operator information. We use rule 10 of Fig. 10 to detect such a defect.

2) Non-Compliant ERC-1155 Implementation. D2.10. incorrect caller check: Similarly to D2.1 defect of Section IV-B-1, if F6 and F7 of ERC-1155 do not revert when the caller of the function is neither equal to parameter `_from` nor the operator of `_from`, then this is a D2.10 defect (detected by rule 17 of Fig. 11).

D2.11. incorrect buyer check: Similarly to D2.3 defect of Section IV-B-1, if F6 and F7 do not revert when parameter `_to` is zero address, then this is a D2.11 defect (detected by rule 18 of Fig. 11).

D2.12. incorrect token check: Similarly to D2.4 defect of Section IV-B-1, if F6 does not revert when the balance of owner for NFT `_id` is lower than parameter `_value`, then this is a D2.12 defect. `_value` is the count of how many NFT `_id` is transferred. Therefore, the balance of the owner for NFT `_id` should be lower than `_value`. We adopt rule 19 of Fig. 11 to detect such a defect.

D2.13. incorrect buyer contract check: Similarly to D2.5 defect of Section IV-B-1, if F6 does not revert when the returned value is not the first-4 bytes of `keccak256 (onERC1155Received(address,address,uint256, uint256, bytes))`, then this is a D2.13 defect (detected by rule 20 of Fig. 11).

D2.14. incorrect batch-transfer check: Similarly to D2.5 defect of Section IV-B-1, if F7 does not revert when the returned balue is not the first-4 bytes of `keccak256 (onERC1155BatchReceived(address,address, uint256[], uint256[],bytes))`, then this is a D2.14 defect (detected by rule 21 of Fig. 11).

D2.15. incorrect batch-transfer notification: If F7 does not emit E5 correctly during its execution, then this is a D2.15 defect. ERC-1155 specifies F7 to emit E5 during its execution, the parameter `_operator` of E5 should be the caller, and other parameters of E5 must be the same as the parameters of F7. We adopt rule 22 of Fig. 11 to detect such a defect.

D2.16. incorrect operator notification: If F8 does not emit E6 correctly during its execution, then this is a D2.16 defect. ERC-1155 specifies F8 to emit E6 during its execution, the parameter `_owner` of E6 should be the caller, and other parameters of E6 must be the same as the parameters of F8. We adopt rule 23 of Fig. 11 to detect such a defect.

C. Plugin 3: Backdoor Analyzer (BA)

BA detects 4 kinds of D3 defects for ERC-721 contracts, and 3 kinds of D3 defects for ERC-1155 contracts. We use F1-E3 to represent the functions and events specified by ERC-721 (Section II-F), and F6-E6 to represent the functions and events specified by ERC-1155 (Section II-G).

1) Implanted Backdoors in ERC-721 Contracts. D3.1. Tampering ownership: A D3.1 defect occurs if an external or public function modifies the storage that records NFT ownership, but does not belong to F1-F3 or the minting and burning functions.

In ERC-721, F1-F3 are the three functions that transfer NFT ownership, each equipped with access control, allowing only the owner, controller, or operator to transfer the NFT. Additionally, while the mint (M) and burn (B) functions also modify ownership, they are not specified by ERC-721. If any other function in an ERC-721 contract can change NFT ownership, the NFTs could be vulnerable to theft. To detect this, BA excludes F1-F3, M, and B during PSM's search and checks for any path that includes a write operation to the storage recording ownership (`ownerMapping`). We use rule 11 in Fig. 10 to identify this defect.

D3.2. Tampering controller: A D3.2 defect occurs if an external or public function modifies the storage that records controller information but does not belong to F1-F4 or the burn function. ERC-721 specifies that a controller can only be assigned or revoked by F1-F4 functions, which include access control, allowing only the owner or operator to manage controllers. Additionally, the burn function (B) can revoke a controller. If any other function outside of F1-F4 and B modifies the controller storage, it could assign a controller role to an attacker. With this role, the attacker could transfer NFTs on behalf of the owner, leading to theft. To detect this defect, BA excludes F1-F4 and B in PSM's search and checks for any write operations to the storage recording controller information (`approveMapping`). We use rule 12 in Fig. 10 to identify this defect.

D3.3. Tampering operator: A D3.3 defect occurs if an external or public function modifies the storage that records operator information, and this function is not F5. In ERC-721, operators are assigned by F5 function. If any other function modifies the operator storage, an attacker could be assigned the operator role. With this role, the attacker could steal NFTs, as operators can transfer all NFTs on behalf of their owner. To detect this defect, BA excludes the F5 function during the search and checks for write operations to the storage that records operator information (`operatorMapping`). We use rule 13 of Fig. 10 to identify this defect.

D3.4. Faking event:

-Fake transfer notification. A fake transfer notification occurs if an external or public function emits E1 but does not belong to F1-F3 or mint/burn functions. Because F1-F3, along with the mint/burn functions, are designed to transfer NFTs and notify off-chain entities of transfer activities. BA uses rule 14 of Fig. 10 to detect this defect.

-Fake controller notification. If there is an external/public function that can emit E2, and this function is not F4, then this is a fake controller notification. Because F4 is the function for setting controller information of an NFT contract, and it needs to inform off-chain entities of the controller assignment. BA adopts rule 15 of Fig. 10 to detect this defect.

-Fake operator notification. If there is an external/public function that can emit E3, and this function is not F5, then this is a fake operator notification. Because F5 is the function for setting operator information of an NFT contract, and it needs to inform off-chain entities of the operator assignment. BA adopts rule 16 of Fig. 10 to detect this defect.

2) Implanted Backdoors in ERC-1155 Contracts: D3.1. Tampering ownership: Similarly to D3.1 defect of Section IV-C-1, if there is an external/public function that can modify the storage recording ownership, and this function belongs neither to F6-F7 nor to mint/burn functions, then this is a D3.1 defect. We design rule 24 of Fig. 11 to detect this defect.

D3.3. Tampering operator: Similarly to D3.3 defect of Section IV-C-1, if there is an external/public function that can modify the storage recording operator information, and this function is not F8, then this is a D3.3 defect. We design rule 25 of Fig. 11 to detect this defect.

D3.4. Faking event: ERC-1155 contracts have similar D3.4 defects of Section IV-C-1.

-Fake transfer notification. If there is an external/public function that can emit E4, and this function is not F6, nor the functions for minting and burning NFTs, then this is a fake transfer notification (detected by rule 26 of Fig. 11).

-Fake batch-transfer notification. If there is an external/public function that can emit E5, and this function is not F7, then this is a fake batch-transfer notification (detected by rule 27 of Fig. 11).

-Fake operator notification. If there is an external/public function that can emit E6, and this function is not F8, then this is a fake operator notification (detected by rule 28 of Fig. 11).

V. Evaluation

In this section, we evaluate the effectiveness of *Emerium* in detecting defects, and introduce the insight from uncovered defects.

Research questions: We answer the following research questions:

· *RQ1:* How is the effectiveness of *Emerium* in detecting defects?

· *RQ2:* How is the effectiveness of *Emerium* in recovering role-related data?

· *RQ3:* What defects does *Emerium* uncover?

· *RQ4:* What is the scalability of *Emerium*?

· *RQ5:* Does *Emerium* outperform state-of-the-art works?

Experimental setup: We conduct the experiment on the server equipped with AMD EPYC 9654 CPU, 256 GB memory, Ubuntu OS 22.04, and Z3 solver v4.13. **Implementation.** Path search module of *Emerium* adopts EtherSolve [38] to construct the control flow graph (CFG) of contract bytecode. We develop *Emerium*'s symbolic execution engine based on teEther [39], as teEther searches paths before conducting symbolic execution, which enables *Emerium* to filter the paths that are meaningless for detecting defects in advance to reduce time cost.

Dataset: Since there is no existing dataset related to the defects in this paper, we collected contracts labeled 'ERC-721' and 'ERC-1155' from Etherscan [37] to create our dataset. The initial collection was done in March 2022 and continuously updated until August 2022. The dataset contains 99,584 ERC-721 contracts and 13,682 ERC-1155 contracts. Of these, 88.2% (87,838/99,584) of ERC-721 contracts and 71.7% (9,808/13,682) of ERC-1155 contracts are verified and provide ABIs. For unverified contracts, we retrieved their ABIs by querying the ABI database [40]. Ultimately, we obtained ABIs for 94.3% (93,931/99,584) of ERC-721 contracts and 96.3% (13,181/13,682) of ERC-1155 contracts. After deduplicating

TABLE I
THE PERFORMANCE ABOUT DETECTING DEFECT OF ERC-721 CONTRACTS

	D1.1	D1.2	D1.3	D2.0	D2.1	D2.2	D2.3	D2.4	D2.5	D2.6	D2.7	D2.8	D2.9	D3.1	D3.2	D3.3	D3.4
Accuracy	0.57	1.00	1.00	1.00	0.94	0.97	0.99	0.99	1.00	0.98	0.99	0.98	0.99	0.97	0.99	1.00	0.97
Precision	0.92	1.00	1.00	1.00	0.49	0.62	0.99	0.93	1.00	0.70	1.00	0.64	1.00	0.583	0.71	1.00	0.57
Recall	0.39	1.00	1.00	1.00	1.00	1.00	0.80	1.00	1.00	1.00	0.71	0.93	0.57	0.93	1.00	1.00	0.89
F-measure	0.55	1.00	1.00	1.00	0.66	0.76	0.89	0.96	1.00	0.82	0.83	0.76	0.73	0.72	0.83	1.00	0.70
Confirm	333	Fig. 13	Fig. 13	Tab. 3	56	53	10	54	250	23	14	15	7	15	10	9	19

· Confirm represents the count of defects after manual review.

TABLE II
THE PERFORMANCE ABOUT DETECTING DEFECT OF ERC-1155 CONTRACTS

	D1.1	D1.2	D1.3	D2.0	D2.10	D2.11	D2.12	D2.13	D2.14	D2.15	D2.16	D3.1	D3.3	D3.4
Accuracy	0.48	1.00	1.00	1.00	0.96	1.00	0.97	1.00	1.00	1.00	1.00	0.93	0.99	0.94
Precision	1.00	1.00	1.00	1.00	0.46	1.00	0.79	1.00	1.00	1.00	1.00	0.70	0.86	0.72
Recall	0.44	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.57	1.00	0.88
F-measure	0.61	1.00	1.00	1.00	0.63	1.00	0.88	1.00	1.00	1.00	1.00	0.63	0.92	0.79
Confirm	186	Fig. 14	Fig. 14	Tab. 3	13	6	23	1	4	0	1	21	6	26

· Confirm represents the count of defects after manual review.

contracts based on bytecode hashes, we identified 46,447 distinct ERC-721 contracts and 4,913 distinct ERC-1155 contracts. With a symbolic execution timeout of 3 hours, we obtained results for 28,641 ERC-721 contracts and 1,852 ERC-1155 contracts.

A. *Effectiveness of Detecting Defects (RQ1)*

Metrics: We adopt Accuracy, Precision, Recall, and F-measure to evaluate the effectiveness of *Emerium* on detecting defects. We regard containing a defect as Positive (e.g., True Positive (TP) means a defect is detected). The higher false positives means the lower precision; the higher false negatives means the lower recall.

Ground truth: We selected 500 ERC-721 contracts and 200 ERC-1155 contracts from our results using stratified sampling [41] for manual review as ground truth. Specifically, we divided all samples into different groups based on result distribution, then randomly selected samples from each group to form a representative verification set [41]. For open-source contracts, we downloaded source code from Etherscan [42] for verification. For closed-source contracts, we used state-of-the-art decompilers, Gigahorse [43] and Panoramix [42], to assist in verifying the results. The manual analysis took 13 days to complete. The effectiveness of detecting ERC-721 contracts is shown in Table I, and the effectiveness for ERC-1155 contracts is shown in Table II.

The FP/FN analysis for detecting D1.1: The FP is caused by incorrectly identifying mint/burn functions. Since mint/burn will add/delete on-chain metadata, and they are not specified by ERC-721/1155, we do not have a perfect methodology to identify them in bytecode. Although we use the heuristic approach (matching 'mint' and 'burn' keywords in ABI) to distinguish them, there are still functions out of our consideration. Therefore, these functions are regarded as tampering on-chain metadata and cause FP.

The FN is caused by path searching limitation. *Emerium* detects defects by finding an appropriate path whose execution satisfies the assertion of a rule. This means if such a path cannot be found during the search, the detection fails. In static search, PSM sets the loop count to be 3 to avoid path explosion, and this makes it fail to find the paths containing more than 3 loops. In this case, *Emerium* misses some paths that tamper on-chain metadata, which causes FN.

The FP analysis for detecting D2.1: False positives (FPs) are caused by limitations in path search and failures in data structure recovery. We observe instances where *Emerium* miss paths needed to verify the correct caller check, leading to FPs. Additionally, we also find cases where *Emerium* is unable to recover role-related data (c.f. Sec V-B).

The FP analysis for detecting D2.2: The FP is caused by path searching limitation, and *Emerium* cannot find the path to reflect the check that the parameter `_from` is the owner of the NFT.

The FP/FN analysis for detecting D2.3: The FP is caused by path searching limitation, and *Emerium* cannot find the path to reflect the check that the buyer is not zero. The FN raises because *Emerium* finds an unfeasible path that reflects such a check. This can be solved by using SMT to check whether the path is feasible.

The FP analysis for detecting D2.4: The FP is caused by path searching limitation, and *Emerium* cannot find the path to reflect the check that the NFT is valid.

The FP analysis for detecting D2.6: The FP is caused by path search limitation. We observe *Emerium* misses the path that checks caller is owner or operator.

The FN analysis for detecting D2.7: The FN is caused by path search limitation. In the case of FN, we find *Emerium* cannot find a path belonging to `transferFrom` function and ending with `STOP` instruction. Therefore, the detection fails, and the result is regarded as negative.

The FP analysis for detecting D2.8: The FP is caused by the failure of recovering the storage recording ownership. Since the first parameter of E2 event is the owner of an NFT, and *Emerium* fails to recover the storage of owner, then the event emission is regarded as incorrect. Specifically, we observe in these contracts

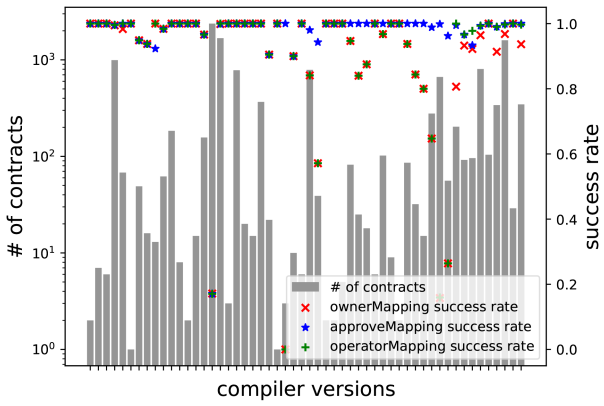

Fig. 12. Role-related data recovery rate. The histogram describes the distribution of the count(#) of contracts from different compiler versions (0.4.18 to 0.8.9 from left to right).

causing FP, the codes of accessing owner data are embedded in loop statements, which is hard for *Emerium* to capture the access pattern for owner data.

The FN analysis for detecting D2.9: The FN is caused by path search limitation. In the case of FN, we find *Emerium* cannot find a path belonging to `setApprovalForAll` function and ending with `STOP` instruction. Therefore, the detection fails, and the result is regarded as negative.

The FP analysis for detecting D2.10: The FP is caused by path searching limitation, and *Emerium* cannot find the path to reflect the check that the caller is the owner or the operator of the NFT.

The FP analysis for detecting D2.12: The FP is caused by path searching limitation, and *Emerium* cannot find the path to reflect the check that the NFT is valid.

The FP analysis for detecting D3.1: The FP is caused by incorrectly identifying mint/burn functions. Since these functions modify the storage that records ownership, *Emerium* may not precisely identify them and results in mint/burn functions being mistakenly flagged as tampering with ownership.

The FP analysis for detecting D3.2: The FP is caused by incorrectly identifying burn functions. Since burn function deletes the storage that records the controller of burned NFTs, *Emerium* may fail to recognize this function. As a result, unidentified burn functions are mistakenly flagged as tampering with the controller, leading to FP.

The FP analysis for detecting D3.4: The FP is caused by incorrectly identifying the functions for minting and burning NFTs. Since these functions emit events like E1, E4, or E5, and *Emerium* may not accurately identify them, events emitted by unrecognized mint/burn functions are mistakenly flagged as faking events.

The profit of dynamic path search: We randomly select 50 ERC-721 contracts containing existing transactions of executing `safeTransferFrom` function. After enabling dynamic path search, PSM can find the paths that are missed by static path search in 33 contracts.

The time cost: The average time to detect D1, D2, and D3 defects in ERC-721 contracts is 120.79 s, 412.23 s, and 29.44 s, respectively. For ERC-1155 contracts, the average detection times for D1, D2, and D3 defects are 34.10 s, 708.99 s, and 27.71 s, respectively. The time required to detect D1 and D3 defects in ERC-721 contracts is higher than in ERC-1155 contracts because Emerium analyzes more paths. However, detecting D2 defects takes longer in ERC-1155 contracts due to the increased complexity of the constraints (caused by their multi-dimensional structure), which requires more time to simplify.

Answer to RQ1: *Emerium* is effective, with the average F-measure of 0.83 on detecting ERC-721 defects, and 0.89 on detecting ERC-1155 defects.

B. Effectiveness of Recovering Role-Related Data (RQ2)

To evaluate how is the effectiveness of *Emerium* in recovering role-related data (RQ2), we use SRM to recover the role-related data of 12,700 ERC-721 contract bytecodes that are compiled by solidity compilers with 54 versions (from 0.4.18 to 0.8.9). To verify the effectiveness, we develop a script to extract the index positions (denoted by $index_{src}$) of `ownerMapping`, `approveMapping`, and `operatorMapping` in contract source codes. Then we compare the index positions (denoted by $index_{bin}$) of the structures recovered in bytecode. If $index_{src}$ is equal to $index_{bin}$, this implies the structure recovered in bytecode is exactly the structure in source code, which is labelled as correctly recovered.

The result is shown in Fig. 12, which indicates the success rate of recovering all role-related data reaches 88.8% for all compiler versions. The failed cases are caused by a. proxy-based function implementation; b. complex data structure; c. loop limitation in path search. Proxy-based function implementation refers to the role-related data is stored in an external contract. Without cross-contract analysis, *Emerium* cannot recover role-related data. Complex data structure refers to that role-related data is stored with an non-mainstream manner. For example, we observe contracts use the mapping embedded in a structure of library to store role-related data. Considering all possible data storage implementation is an open problem for bytecode analysis tools [44]. Loop limitation refers to that the path containing the accessing pattern cannot be found under the setting of loop limitation, which is also a fundamental challenge in symbolic execution.

Answer to RQ2: For 54 compiler versions from 0.4.18-0.8.9, *Emerium* can identify all role-related data with the average success rate of 88.8%.

C. Uncovering Defects (RQ3)

The confirmed defects are shown in Table I for ERC-721 contracts, and Table II for ERC-1155 contracts. We select some defects with the great impact to analyze.

Uncovering D1.1: We confirm 333 D1.1 defects of ERC-721 contracts, and 186 D1.1 defects of ERC-1155 contracts. We

TABLE III
CONFIRMED D2.0 DEFECT OF ERC-721 & ERC-1155 CONTRACTS

	D2.0 of ERC-721												D2.0 of ERC-1155									
	I	II	III	IV	V	VI	VII	VIII	IX	X	XI	XII	XIII	XIV	XV	XVI	XVII	XVIII	XIX	XX	XXI	XXII
Missing	68	44	37	90	15	19	73	90	68	2	32	63	0	4	1	4	2	1	1	4	0	9
Inconsistent	11	8	22	11	1	3	7	26	4	61	57	6	1	0	0	0	1	0	0	0	2	0

Missing represents the corresponding standard function or event is not found in the contract ABIs.
Inconsistent represents the standard function or event is found in the contract ABIs, but corresponding parameter types is incorrect.
Column name ([F] means function, [E] means event):
I: getApproved[F], II: approve[F], III: transferFrom[F], IV: safeTransferFrom[F], V: ownerOf[F], VI: balanceOf[F], VII: setApprovalForAll[F], VIII: safeTransferFrom(Bytes)[F], IX: isApprovedForAll[F], X: Transfer[E], XI: Approval[E], XII: ApprovalForAll[E], XIII: balanceOf[F], XIV: balanceOfBatch[F], XV: isApprovedForAll[F], XVI: safeBatchTransferFrom[F], XVII: safeTransferFrom[F], XVIII: setApprovalForAll[F], XIX: ApprovalForAll[E], XX: TransferBatch[E], XXI: TransferSingle[E], XXII: URI[E].

illustrate a case study of D1.1 in List. 6. GenArt721 is the contract of project Art Blocks [45], whose traded volume has met **2.2 k ETH** (**$6.27 million USD**) [46]. The artist of an NFT in this contract is allowed to change the on-chain metadata (line 2 and line 5) by the functions of line 1 and line 4, even after the NFT is bought by a buyer. Once the on-chain metadata is modified, the NFT could be invalid.

```
1 function updateProjectBaseURI(uint256 _projectId,
      string memory _newBaseURI) onlyArtist(_projectId
      ) public {
2     projects[_projectId].projectBaseURI =
          _newBaseURI;
3 }
4 function updateProjectBaseIpfsURI(uint256 _projectId
      , string memory _projectBaseIpfsURI) onlyArtist(
      _projectId) public {
5     projects[_projectId].projectBaseIpfsURI =
          _projectBaseIpfsURI;
6 }
```

Listing 6: Code snippet of Art Blocks Contract : 0x059edd72cd353df5106d2b9cc5ab83a52287ac3a

Uncovering D1.2&D1.3: 66,639 out of 87,839 ERC-721 contracts contain on-chain metadata samples pointing to off-chain data. We use the format *Type-Location* to describe the off-chain data sample (e.g., `JSON-HTTP` represents the JSON file pointed by on-chain metadata is stored in HTTP server, and this is a **D1.3**; `Asset-HTTP` represents the asset pointed by a JSON file is stored in HTTP server, also a **D1.3**). The results are shown in Fig. 13. 2,309 out of 9,808 ERC-1155 contracts contain on-chain metadata samples pointing to off-chain data. The results are shown in Fig. 14. *Emerium* uncovers 24,192,723 D1.2, and

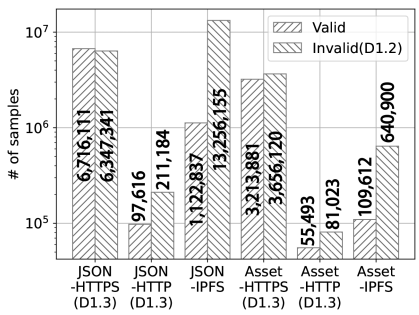

Fig. 13. Uncovered D1.2&D1.3 of ERC-721 contracts.

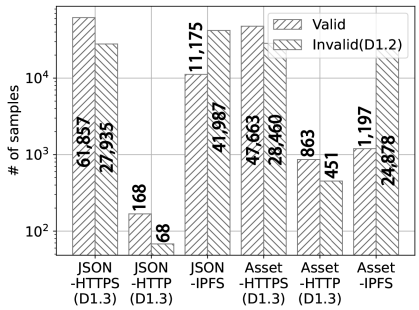

Fig. 14. Uncovered D1.2&D1.3 of ERC-1155 contracts.

20,378,769 D1.3 of ERC-721 contracts; 123,779 D1.2, and 167,465 D1.3 of ERC-1155 contracts. We found 13,963,920 off-chain data stored in IPFS is invalid. This implies IPFS is not as reliable as public thought. Although IPFS can protect data against tampering attacks and DoS attacks, the data stored on it can be deleted by its owner.

Uncovering D2.0: For D2.0, We observe two types - *Missing* and *Inconsistent*. *Missing* represents the corresponding standard function or event is not found in contract ABIs. *Inconsistent* represents the standard function or event is found in contract ABIs, but corresponding parameter types are incorrect. The result is shown in Table III. We found 818 D2.0 for ERC-721 contracts, 30 D2.0 for ERC-1155 contracts.

Uncovering D2.1: We confirm 56 D2.1 defects of ERC-721 contracts. We illustrate a case study of D2.1 in List. 7. Interfaced-EtheriaV12 is a contract of *etheria* [11], and the price of its NFT once met **80 ETH** (**$236,187.20 USD**) [12]. The function `transferFrom` responsible for transferring NFT is equipped with incorrect access control, by checking the authority of `tx.origin` instead of `msg.sender` (line 3), which causes the vulnerability [13]. Using `tx.origin` for authorization could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check, since `tx.origin` returns the authorized account.

Uncovering D2.2: We confirm 53 D2.2 defects of ERC-721 contracts. We illustrate a case study of D2.2 in List. 2. EvohClaimable is the contract of project Evoh Llama Frens [25] whose traded volume has met **39 ETH** (**$115,074.18 USD**) [25].

```
1 function transferFrom(address from, address to,
      uint256 tokenId) public virtual override {
2     ...
3     require(tx.origin == ownerOf(tokenId), "ERC721:
          transfer caller is not owner");
4     ...
5 }
```

Listing 7: Code snippet - InterfacedEtheriaV12 Contract: 0x974dEAc1597575fB77c991EB7506dc751b58489b

The contract implements incorrect access control in `transferFrom` function, which enables emitting an arbitrary NFT transfer event. Specifically, line 5 does not check `_from` should be the owner of `_tokenId`, which enables an attacker to set an arbitrary `_from` value. Then line 7 can be exploited by the attacker to emit an NFT transfer behavior of arbitrary buyer, which will be displayed in OpenSea to mislead users. This case is confirmed as CVE-2022-35621.

Uncovering D3.1: We confirm 15 D3.1 defects for ERC-721 contracts, and 21 D3.1 defects for ERC-1155 contracts. A case study of D3.1 of the project, Sandbox [29] whose traded volume has met **157.5 K ETH (**$**464.57 million USD)**, has been introduced in Section II-C. This backdoor can be exploited to burn users' NFTs.

Uncovering D3.4: We confirm 19 D3.4 defects for ERC-721 contracts, and 26 D3.4 defects for ERC-1155 contracts. A case study of D3.4 of the project, Su Squares NFT whose trade volume has met 57 ETH ($87,515.52 USD) [30], has been introduced in Section II-C. This backdoor can be exploited by an attacker to emit an arbitrary `Transfer` event.

Answer to RQ3: Emerium uncovers 44,863,255 defects of fragile NFT binding, 1,373 defects of non-compliant implementation, 105 defects of implanted backdoor and a new CVE.

D. *Extending* Emerium *to Detect New Defects (RQ4)*

Since *Emerium* can be customized to extend its capability, we wonder whether it can identify new defects by setting new detection rules. List. 8 displays the contract snippet of the The National Basketball Association NFT [47].

```
1 function mint_approved(vData memory info, uint256
      number_of_items_requested, uint16 _batchNumber)
      external {
2     ...
3     require(verify(info), "Unauthorised access
          secret");
4     ...
5 }
```

Listing 8: Code snippet - Association-sales Contract: 0xdd5a649fc076886dfd4b9ad6acfc9b5eb882e83c

It contains a vulnerability that enables an attacker to mint a large number NFTs, without paying any tokens, which is reported in [48]. When function `mint_approved` (line 1) mints an NFT, it lacks checking whether the caller of the function is authorized. Therefore, an attacker can use a `info` of a historical transaction to pass the check in line 3, so as to mint tokens freely. This defect is out of our consideration in advance, and we develop a new plugin and append a new rule $\forall p \in Search(\{M\}, \{STOP\}), \exists c \in Constraint(p), c :== (msg.sender, *) \vee == (*, msg.sender)$. where `M` means the signature of the function responsible for minting an NFT, and `msg.sender` represents the caller of function, and * means any value. This rule describes, the function for minting an NFT needs to check the caller of function. Our experiment proves this rule is capable of detecting this serious defect.

TABLE IV
COMPARISON WITH STATE-OF-THE-ART WORKS

Defects	Emerium		NFTGuard		WakeMint	
	F1	Recall	F1	Recall	F1	Recall
D2.6	0.82	1.00	0.00	0.00	N/A	N/A
D2.1	0.66	1.00	N/A	N/A	0.07	0.04
D2.2	0.76	1.00	N/A	N/A	0.07	0.04
D3.4	0.70	0.89	N/A	N/A	0.11	0.06

Answer to RQ4: *Emerium* can be customized to enhance its capability by designing rules to describe the oracles of defects.

E. *Comparing* Emerium *With Existing Works (RQ5)*

Dipanjan et al. [1] adopt the `assets` API [19] provided by NFT market OpenSea [18] to query off-chain data, and check D1.2 and D1.3. Differently from [1], NFT binding analyzer (Section IV-A) of *Emerium* proposes a general methodology to query the off-chain data of an NFT. This implies *Emerium* can detect D1.2 and D1.3 of the contracts that even are not collected by OpenSea, i.e., providing a **wider detection range** compared to [1]. Furthermore, *Emerium* uncovers the new finding concerning the off-chain data stored in IPFS. We find numerous (13,963,920) off-chain data stored in IPFS is invalid, which is not observed in [1]. This implies that, although IPFS is robust against tampering NFT data, it cannot promise data availability because data owners can delete the data.

As NFTGuard [20] is a source code detector that can identify D2.6, we conduct the comparison with NFTGuard in our verified dataset, which is illustrated in Table IV. NFTGuard fails to detect 23 D2.6 defects (low recall) in our dataset, which is caused by two reasons. 10 samples cannot be detected due to the failure of constructing solidity source mappings. 13 samples are missed because NFTGuard cannot handle the corresponding source code patterns. Besides, some of the patterns limit NFTGuard's effectiveness and lead to the failure of solving constraints. Differently, *Emerium* handles this issue by novel constraint simplification on binary.

WakeMint [21] is developed based on NFTGuard, and focuses on the detection of D2.1, D2.2, and D3.4. As shown in Table IV,

WakeMint fails to identify 27 D2.1 defects, 26 D2.2 defects, and 16 D3.4 defects, which leads to low recall. Like NFTGuard, WakeMint also meets the failure of constructing solidity source mappings and pattern identification.

Answer to RQ5: *Emerium* provides a wider detection scope and capability compared to existing works.

VI. Discussion and Limitations

In this section, we discuss the soundness and completeness of *Emerium*. For soundness, we discuss the limitations that lead to inaccurate detection. For completeness, we explore the extent to which *Emerium* can cover the scope of NFT defects.

Path explosion: Due to avoiding path explosion, *Emerium* limits the loop count to be 3 during its path search, which leads to inaccurate detection. To mitigate this fundamental challenge, we propose dynamic path search by replaying existing transactions to provide more paths. However, dynamic path search may fail because the contracts lack existing transactions to provide complicated paths. In future work, we consider to use fuzzing technique to search complicated paths to improve the performance further.

Identifying minting/burning function: Emerium may not always accurately identify the minting and burning functions of an NFT, leading to detection inaccuracies. Since ERC-721 and ERC-1155 do not define these functions, it is challenging to precisely identify them from bytecode. To mitigate this, we use a heuristic approach (keyword search in ABIs). To evaluate its effectiveness, we collected ABIs for 78,305 mint functions and 64,651 burn functions from ERC-721 contracts, and 6,915 mint functions and 2,632 burn functions from ERC-1155 contracts by analyzing function comments in the source code. Our approach identified 72,284/78,305 (92.3%) mint functions and 43,792/64,651 (67.7%) burn functions in ERC-721 contracts, and 5,690/6,915 (82.3%) mint functions and 2,143/2,632 (81.4%) burn functions in ERC-1155 contracts. In future work, we plan to explore methods for distinguishing mint/burn functions in bytecode.

Data recovery: The data structure considered in this paper follows the widely adopted OpenZeppelin specification [49]. In our dataset, 95.7% (84,026/87,838) of ERC-721 contracts and 91% (8,922/9,808) of ERC-1155 contracts use this specification. However, *Emerium* cannot perfectly recover the data structure of all NFT contracts, as some may have more complex designs. We conducted a post-hoc study on implementations that deviate from the OpenZeppelin storage structure. We identified 9,184 contracts using proxy-based implementations, where external contracts are called to access data structures. Additionally, 2,516 contracts use complex structures, such as arrays of structure variables, to store role-related data, and 2,064 contracts encode specific keys (e.g., subtracting a variable) to query role-related data. Since these implementations are customized based on contract-specific logic, designing a general methodology to identify and recover all such edge cases is highly challenging. We leave this for future work.

Extending Emerium : The rule syntax in Section III-C defines a defect using four dimensions: path, path constraint, storage write, and event. This means the rule's scope covers defects that can be described using these four dimensions. For instance, timestamp dependency, a defect outside the scope of this paper, can be described by this syntax and thus detected by *Emerium*. Timestamp dependency occurs when a smart contract uses the block's timestamp as a path constraint to control its execution [50]. Since this defect involves a path constraint, it can be defined using the rule syntax in that dimension. However, for another example, a reentrancy attack cannot be represented by this syntax, as it stems from updating the contract state after an external call, and the current syntax cannot capture order dimension. Extending *Emerium* to handle such cases is beyond the scope of this paper, but we plan to enhance the rule syntax in future work to improve its extensibility.

VII. Related Work

NFT security: Wang et al. [51] provide the first overview on the security evolution of state-of-the-art NFT solutions, in terms of technical components, protocols, standards, and desired proprieties. Dipanjan et al. [1] provide a study on the security issues of NFT market, including the interactions between user and market, the metadata validity, counterfeit NFT creation, wash trading, etc.. Since previous works do not target analyzing the security of contract codes, Yang et al. [20] propose NFTGuard, the first source code analyzer concerning 5 defects of ERC-721 contracts, including risky mutable proxy, reentrancy, unlimited minting, etc., which covers four general defects and a D2.6 defect of this paper. WakeMint is proposed by Xiao et al. to detect sleepminting risk caused by D2.1, D2.2, and D3.4 defects on ERC-721 contract source code [21].

Tools for detecting security issues of smart contracts: Several works detect invariants in smart contracts to identify potential security issues. TokenScope [52] detects inconsistent behaviors in ERC-20 token contracts by instrumenting the EVM and replaying transactions to check if contract execution results meet the defined invariants. Solythesis [53] allows users to manually instrument contract source code to validate invariant violations. SOLAR [54] verifies whether a contract properly checks authority before interacting with an ERC-721 contract. However, these tools cannot be applied to detect the defects discussed in this paper because the invariants they target do not encompass the specifications of ERC-721 and ERC-1155 contracts. For other tools designed to detect vulnerabilities in smart contracts, we classify them by detection type and scope in Table V.

Novelty: Compared to existing tools, *Emerium* is novel in its detection scope, capability, and extensibility. In terms of scope, *Emerium* addresses NFT-domain-specific defects, filling an important research gap. Regarding capability, *Emerium* detects defects without relying on source code, giving it broader

TABLE V
THE DIFFERENCES OF EXISTING WORKS

	Type	Scope									
		RE	UE	LE	TO	IO	UA	ERC-20	D1	D2	D3
Oyente [6]	Bytecode	●	●	○	●	●	○	○	○	○	○
Mythril [7]	Bytecode	●	●	○	○	●	◐	○	○	○	○
Osiris [55]	Bytecode	○	○	○	○	●	○	○	○	○	○
teEther [39]	Bytecode	○	○	○	○	○	◐	○	○	○	○
Maian [56]	Bytecode	○	○	●	○	○	◐	○	○	○	○
Securify [57]	Bytecode	●	●	●	●	○	◐	○	○	○	○
Zeus [58]	Source code	●	●	●	●	●	○	○	○	○	○
Contractfuzzer [59]	Bytecode	●	●	○	○	○	○	○	○	○	○
SmartCheck [60]	Source code	●	●	●	○	●	○	○	○	○	○
TokenScope [52]	Bytecode	○	○	○	○	○	○	●	○	○	○
Solythesis [53]	Source code	○	○	○	○	○	◐	○	○	○	○
Vandal [61]	Bytecode	●	●	○	○	○	○	○	○	○	○
MadMax [62]	Bytecode	○	○	●	○	●	○	○	○	○	○
Pied-Piper [63]	Source code	○	○	○	○	○	◐	○	○	○	○
DEFECTCHECKER [33]	Bytecode	●	○	○	●	○	◐	○	○	○	○
Dipanjan et al. [1]	Web link	○	○	○	○	○	○	○	◐	○	○
SOLAR [54]	Bytecode	○	○	○	○	○	◐	○	○	○	○
NFTGuard [20]	Source code	●	○	○	○	○	◐	○	○	◐	○
WakeMint [21]	Source code	○	○	○	○	○	○	○	○	◐	◐
Emerium	Bytecode	○	○	○	○	○	○	○	●	●	●

●:Fully cover; ◐:Partly cover; ○:Not cover; RE: Re-Entrancy; UE: Unhandled Exceptions; LE: Locked Ether; TO: Transaction Order Dependency; IO: Integer Overflow; UA: Unrestricted Action; ERC-20: the violation about ERC-20 standard.

applicability. For extensibility, on-demand plugins can be developed to expand *Emerium*'s detection range, allowing it to evolve and address new defects as they emerge.

VIII. CONCLUSION

In this work, we systematically analyze NFT contract defects in terms of fragile NFT binding, non-compliant implementation, as well as implanted backdoor. To detect these defects, we propose *Emerium*, the first bytecode-oriented NFT contract detection framework that uncovers the defects from real world. We believe *Emerium* plays a pioneer in guarding NFT contract security.

REFERENCES

[1] D. Das, P. Bose, N. Ruaro, C. Kruegel, and G. Vigna, "Understanding security issues in the NFT ecosystem," in *Proc. 2022 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 667–681.

[2] Dune analytics - opensea, 2022. Accessed: Feb. 23, 2022. [Online]. Available: https://dune.xyz/rchen8/opensea

[3] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "Deepinfer: Deep type inference from smart contract bytecode," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 745–757.

[4] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "Eip-721: Erc-721 non-fungible token standard, ethereum improvement proposals, no. 721," 2021. Accessed: Dec. 01, 2021. [Online]. Available: https://eips:ethereum:org/EIPS/eip-721

[5] W. Radomski, A. Cooke, P. Castonguay, J. Therien, E. Binet, and R. Sandford, "Eip-1155: Multi token standard," 2021. Accessed: Dec. 01, 2021. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. 2016 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[7] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," in *Proc. Eur. Symp. Res. Comput. Secur.*, Springer, 2019, pp. 747–766.

[8] SlowMist: Focusing on blockchain ecosystem security, 2023. Accessed: Apr. 17, 2023. [Online]. Available: https://www.slowmist.com/

[9] Blocksec: Building blockchain security infrastructure, 2023. Accessed: Apr. 17, 2023. [Online]. Available: https://blocksec.com/

[10] Yes, your NFTs can go missing–here's what you can do about it, 2022. Accessed: Mar. 18, 2022. [Online]. Available: https://decrypt.co/62037/missing-or-stolen-nfts-how-to-protect

[11] Etheria NFT, main contract address: 0xb21f8684f23dbb1008508b4de91a0aaedebdb7e4, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://etheria.world/howto.html

[12] Price of etheria NFT, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://opensea.io/assets/0x629a493a94b611138d4bee231f94f5c08ab6570a/804

[13] SWC-115 - authorization through tx.origin, 2022. Accessed: Nov. 23, 2022. [Online]. Available: https://swcregistry.io/docs/SWC-115

[14] T. Beery, "Unicats go phishing," 2021. Accessed: Dec. 01, 2021. [Online]. Available: https://zengo.com/unicats-go-phishing/

[15] The vulnerability behind the sandbox land migration, 2022. Accessed: Mar. 01, 2022. [Online]. Available: https://slowmist.medium.com/the-vulnerability-behind-the-sandbox-land-migration-2abf68933170

[16] Logging data from smart contracts with events, 2022. Accessed: Jul. 25, 2022. [Online]. Available: https://ethereum.org/ig/developers/tutorials/logging-events-smart-contracts/

[17] How to sleepmint NFT tokens, 2021. Accessed: Jun. 25, 2021. [Online]. Available: https://www.linkedin.com/pulse/how-sleepmint-nft-tokens-ke%C3%ADr-finlow-bates

[18] Opensea, 2021. Accessed: Dec. 01, 2021. [Online]. Available: https://opensea.io/

[19] Retrieve assets api of opensea, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://docs.opensea.io/reference/getting-assets

[20] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in NFT smart contracts," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 373–384.

[21] L. Xiao, S. Yang, W. Chen, and Z. Zheng, "WakeMint: Detecting sleep-minting vulnerabilities in NFT smart contracts," 2025, *arXiv:2502.19032*.

[22] Interplanetary file system (IPFS), 2021, Accessed: Dec. 01, 2021. [Online]. Available: https://ipfs:io

[23] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "SoK: Decentralized exchanges (DEX) with automated market maker (AMM) protocols," *ACM Comput. Surv.*, vol. 55, no. 11, pp. 1–50, 2023.

[24] How much does it cost to create an NFT: Detailed analysis, 2022. Accessed: May 21, 2023. [Online]. Available: https://medium.datadriveninvestor.com/how-much-does-it-cost-to-create-an-nft-detailed-analysis-8fa6d22315fe

[25] Trade volume of evohclaimable NFT, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://opensea.io/collection/evoh-llama-frens

[26] What are cross-chain NFTS?, 2023. Accessed: Aug. 23, 2023. [Online]. Available: https://blog.chain.link/cross-chain-nft/

[27] T. Chen et al., "SODA: A generic online detection framework for smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.

[28] Slowmist: 3,619 eth-based tokens are affected by fake deposit vulnerability, 2018. Accessed: May 21, 2023. [Online]. Available: https://news.8btc.com/slowmist-3619-eth-based-tokens-are-affected-by-fake-deposit-vulnerability

[29] Trade volume of sandbox NFT, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://opensea.io/collection/sandbox

[30] SU squares NFT, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://opensea.io/collection/su-squares

[31] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[32] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, Springer, 2008, pp. 337–340.

[33] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022.

[34] Function signature of ethereum, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://ethereum-blockchain-developer.com/031-understanding-abi-and-gas/01-abi-solidity/

[35] Contract ABI specification, 2022. Accessed: May 17, 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.13/abi-spec.html

[36] Replacing NFT assets stored in a HTTP server, 2021. Accessed: May 21, 2023. [Online]. Available: https://x.com/neitherconfirm/status/1369285946198396928

[37] Opensea bug destroys $100,000 worth of NFTs, 2021. Accessed: May 21, 2023. [Online]. Available: https://www.theblock.co/post/116924/opensea-bug-destroys-100000-worth-of-nfts-including-historical-ens-name

[38] F. Contro, M. Crosara, M. Ceccato, and M. D. Preda, "EtherSolve: Computing an accurate control-flow graph from ethereum bytecode," in *Proc. IEEE/ACM 29th Int. Conf. Prog. Comprehension*, 2021, pp. 127–137.
[39] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proc. 27th {USENIX} Secur. Symp.*, 2018, pp. 1317–1333.
[40] Ethereum signature database, 2021, Accessed: Dec. 01, 2023. [Online]. Available: https://www.4byte.directory/
[41] R. Singh and N. S. Mangat, in *Elements of Survey Sampling*. Berlin, Germany: Springer Science & Business Media, 2013, vol. 15.
[42] Etherscan, 2021. Accessed: Dec. 01, 2021. [Online]. Available: https://etherscan.io/
[43] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 1176–1186.
[44] T. Chen et al., "SigRec: Automatic recovery of function signatures in smart contracts," *IEEE Trans. Softw. Eng.*, vol. 48, no. 8, pp. 3066–3086, Aug. 2022.
[45] Art blocks, 2021. Accessed: Dec. 01, 2021. [Online]. Available: https://www.artblocks.io/
[46] Trade volume of art blocks NFT, 2022. Accessed: Apr. 17, 2022. [Online]. Available: https://opensea.io/collection/genesis-by-dca
[47] NBA announces 'dynamic' ethereum NFTs for playoffs, 2022. Accessed: Apr. 25, 2022. [Online]. Available: https://decrypt.co/98231/nba-announces-dynamic-ethereum-nfts-playoffs
[48] How to verify a signature in a wrong way – the association NFT case, 2022. Accessed: Apr. 25, 2022. [Online]. Available: https://blocksecteam.medium.com/how-to-verify-a-signature-in-a-wrong-way-the-associationnft-case-5a913e9b8a1d
[49] Openzeppelin, 2023. Accessed: Feb. 23, 2023. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts
[50] TimeStamp dependency in smart contracts, 2022. Accessed: Apr. 17, 2023. [Online]. Available: https://www.geeksforgeeks.org/timestamp-dependency-in-smart-contracts/
[51] Q. Wang, R. Li, Q. Wang, and S. Chen, "Non-fungible token (NFT): Overview, evaluation, opportunities and challenges," 2021, *arXiv:2105.07447*.
[52] T. Chen et al., "TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1503–1520.
[53] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 438–453.
[54] A. Li and F. Long, "Detecting standard violation errors in smart contracts," 2018, *arXiv: 1812.07702*.
[55] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.
[56] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.
[57] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. 2018 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[58] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12.
[59] B. Jiang, Y. Liu, and W. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2018, pp. 259–269.
[60] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.
[61] L. Brent et al., "Vandal: A scalable security analysis framework for smart contracts," 2018, *arXiv: 1809.03981*.
[62] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in ethereum smart contracts," in *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, 2018.
[63] F. Ma et al., "Pied-piper: Revealing the backdoor threats in ethereum ERC token contracts," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, pp. 1–24, 2022.

Zuchao Ma is currently working toward the PhD degree with The Hong Kong Polytechnic University, Hong Kong. His research interests include smart contract security and IoT network security.

Muhui Jiang received the PhD degree from the Department of Computing, Hong Kong Polytechnic University. His current research interests include blockchain security, network security, system security, and IoT security.

Xiapu Luo is currently a professor with the Department of Computing, The Hong Kong Polytechnic University. His research focuses on blockchain/smart contracts, mobile/IoT security and privacy, network/web security and privacy, software engineering and internet measurement with papers published in top conferences and journals. His research led to eight best/distinguished paper awards, including ACM SIGSOFT Distinguished Paper Award in ICSE'21, Best Paper Award in INFOCOM'18, etc. and several awards from the industry. He regularly serves in the program committee of major security conferences and is currently an editor for *IEEE/ACM Transactions on Networking*.

Haoyu Wang is currently a full professor with the School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST). He is leading the SECURITY PRIDE Research Group. His research covers a wide range of topics in Software Analysis, Privacy and Security, eCrime, Internet/ System Measurement, and AI Security. He has been awarded three best/distinguished paper awards, including WWW 2020 Best Student Paper Award (the first award from China), and ACM OOPSLA 2020 Distinguished Paper Award.

Yajin Zhou received the PhD degree in computer science from North Carolina State University, in 2015. He is a ZJU 100-Young Professor (since 2018), with the School of Cyber Science and Technology, Zhejiang University, China. He has published more than 40 papers, with more than 7,500 citations (Google Scholar). Two of his papers have been selected to the list of normalized Top-100 security papers since 1981. He was recognized as the Most Influential Scholar Award Honorable Mention for his contributions to the field of Security and Privacy. His current research spans decentralized finance (DeFi) security, software security, operating systems security.