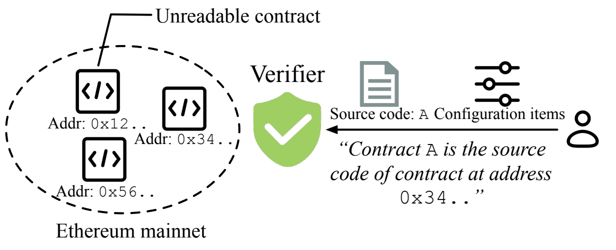
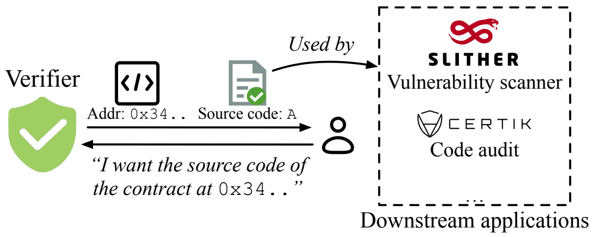
Abusing the Ethereum Smart Contract Verification Services for Fun and Profit

Pengxiang Ma*[1], Ningyu He*[2], Yuhua Huang[1], Haoyu Wang§[1] and Xiapu Luo[3]
[1] Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
[2] Key Lab of HCST (PKU), MOE; SCS, Peking University, China
[3] The Hong Kong Polytechnic University, China

***Abstract*—Smart contracts play a vital role in the Ethereum ecosystem. Due to the prevalence of kinds of security issues in smart contracts, the smart contract verification is urgently needed, which is the process of matching a smart contract's source code to its on-chain bytecode for gaining mutual trust between smart contract developers and users. Although smart contract verification services are embedded in both popular Ethereum browsers (e.g., Etherscan and Blockscout) and official platforms (i.e., Sourcify), and gain great popularity in the ecosystem, their security and trustworthiness remain unclear. To fill the void, we present the first comprehensive security analysis of smart contract verification services in the wild. By diving into the detailed workflow of existing verifiers, we have summarized the key security properties that should be met, and observed eight types of vulnerabilities that can break the verification. Further, we propose a series of detection and exploitation methods to reveal the presence of vulnerabilities in the most popular services, and uncover 19 exploitable vulnerabilities in total. All the studied smart contract verification services can be abused to help spread malicious smart contracts, and we have already observed the presence of using this kind of tricks for scamming by attackers. It is hence urgent for our community to take actions to detect and mitigate security issues related to smart contract verification, a key component of the Ethereum smart contract ecosystem.**



(a) Request source code verification



(b) Ask source code for a contract

Fig. 1: Source code verification in Ethereum ecosystem.

I. Introduction

Ethereum, as one of the representative blockchain platforms, is regarded as a medal contender of Satoshi's Bitcoin. Its market cap peaked at $540 billion in November 2021 [16]. The success of Ethereum cannot omit the existence of tens of millions of smart contracts deployed on it.

Specifically, smart contracts on Ethereum can be seen as scripts that will be executed once pre-defined conditions are met. Alongside the characteristics of *irreversibility* and *determinacy* of blockchain, developers start to compose decentralized applications (DApps) with smart contracts, e.g., gambling games [62], decentralized exchanges [66], and decentralized autonomous organizations that can propose and discuss proposals [22], where all participants are willing to and have to obey game rules that are coded in the smart contracts. Considering the efficiency and I/O issue, Ethereum only stores a compact format, i.e., bytecode, of smart contracts within its decentralized database.

However, the unreadability of bytecode severely hampers the development of the whole ecosystem. For example, on Ethereum, accounts are eligible to create and issue tokens by deploying their own token contracts, it is hard to tell scam tokens from official or real ones by only auditing the bytecode. For example, Xia et al. [72] identified over 10K scam token contracts, where scammers have gained a profit of at least $16 million. Such a gap between users' expectation and actual executed logic in unreadable bytecode urges the emergence of *smart contract verification*, i.e., *the process of matching a smart contract's source code to its on-chain bytecode*.

Smart contract verification have been integrated into Ethereum browsers, e.g., Etherscan [26] and Blockscout [6], and other official platforms, e.g., Sourcify [2]. As shown in Fig. 1(a), by providing a piece of source code, a set of configuration items, and an address, the verification service compiles the given source code according to the configuration and compares it with the on-chain bytecode in the designated address. For all source code that passes the verification, they

*Pengxiang Ma and Ningyu He are co-first authors.
§Haoyu Wang (haoyuwang@hust.edu.cn) is the corresponding author.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24992
www.ndss-symposium.org

will be stored by the service provider. Once someone asks the source code of a contract, the verification service returns it if it has been verified, as illustrated in Fig. 1(b). The retrieved source code can be used in various kinds of downstream applications, e.g., vulnerability detection and code auditing. Indeed, many research studies [28], [32] and industrial products [8] rely on the smart contract verification services. Intuitively, source code verification services will help gain mutual trust between smart contract developers and their users, and boost a series of applications.

Are the smart contract verification services trustworthy? Surprisingly, no prior studies have considered whether these verification services work as expected. Imagine such a situation where a malicious developer provides a seemingly harmless source code that can pass the verification of an on-chain contract, which however is embedded with backdoors. Moreover, what if one's smart contract can be verified by a source code that is elaborately constructed by malicious competitors? These severe consequences can happen once there are vulnerabilities or bugs under any of the modules of the verification services. These vulnerabilities can be abused by adversaries, leading to an extreme negative impact to the whole Ethereum ecosystem.

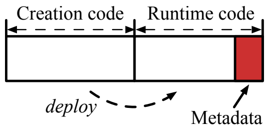
This Work. We take the first step to perform a comprehensive security analysis on Ethereum smart contract verification services in this paper. Based on the implementation of three widely-adopted verifiers, i.e., Etherscan, Sourcify, and Blockscout, we firstly distill the general workflow of them and identify key modules that make them up (see §IV). Then we propose the key security properties that should be satisfied in these services (see §V). Against the security properties, we summarize eight types of potential vulnerabilities that can exploit the verification service. Further, for each type of vulnerability, we propose a detection method, the corresponding proof of concept (PoC) to illustrate how they can be exploited, and concrete steps to avoid the abuse (see §VI). Our comprehensive analysis reveals 19 vulnerabilities in total on three services, each of which can be exploited by attackers. After a timely disclosure to service providers, 15 vulnerabilities have been confirmed by the corresponding official and 10 of them have been patched. We further measure the impact, and observe that tens of millions of deployed innocent contracts can be abused, and hundreds of contracts may have already committed fraud by exploiting these vulnerabilities (see §VII).

Our contributions can be summarized as follows:

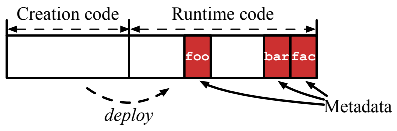
- To the best of our knowledge, we are the first to characterize the design and implementation of popular Ethereum source code verification services.
- We uncover eight types of overlooked but exploitable vulnerabilities hidden in verification services.
- We propose the concrete methods to detect, exploit and mitigate for all these types of vulnerabilities, and reveal 19 vulnerabilities hidden in three verifiers. By the time of writing, 15 vulnerabilities have been confirmed, and 10 of them have been patched in time.
- We show that the uncovered vulnerabilities can introduce an extreme negative impact on the Ethereum ecosystem, i.e., tens of millions of innocent contracts can be abused by attackers, and hundreds of contracts have already been manipulated.

```
contract foo {...}
contract bar {...}
contract fac {
    function f() public{
        foo c_foo = new foo();
        bar c_bar = new bar();
    }
}
```

(a) A simple factory contract.



(b) The structure of most smart contracts.



(c) The structure of the factory contract illustrated in Fig. 2a.

Fig. 2: The bytecode structure of Ethereum smart contracts.

We have released all artifacts at: link.

II. Background

A. Smart Contract

Most Ethereum smart contracts are written in Solidity [54], a high-level programming language specifically designed by the Ethereum official. Correspondingly, Ethereum provides a specific compiler, named *solc*, which takes Solidity files as input and generates a bytecode file, which can be executed by *Ethereum Virtual Machine* (*EVM*). An EVM bytecode consists of a series of opcodes, which can interact with the data structures maintained by EVM [23]. For example, the `PUSH` opcode pushes its operand onto an operand stack.

Typically, an EVM bytecode can be divided into three parts according to their functionalities, i.e., *creation code*, *runtime code*, and *metadata*, as shown in Fig. 2b. Specifically, the creation code can be executed only once. It is responsible for deploying the corresponding runtime code, where the concrete executing logic of a smart contract, e.g., implementations of functions, is encoded. Moreover, the fixed-length metadata is part of the runtime code and not executable. It is the hash result of the meta information during the compilation (like solc version) [55]. Additionally, the metadata can be used as a key to retrieve and index the corresponding smart contract in a decentralized database. For most smart contracts, there is only one piece of metadata, while things look different for *factory contracts* [70]. Specifically, a factory contract can deploy other contracts solely by itself, as shown in Fig. 2a, where the `fac` contract can deploy `foo` and `bar` whenever the function `f` is invoked. Its bytecode structure is shown in Fig. 2c. The most obvious distinction is that there are three pieces of metadata embedded in the runtime code, corresponding to `fac` and the contracts it can deploy. Note that, there exist explicit and fixed indicators in the head and tail of each metadata, which can thus be identified easily. Except for these three basic structures, we often use the term, *bytecode*, as a general one to refer all these three parts together. When a user tends to initiate a transaction with the locally compiled bytecode for deployment, the bytecode as well as the initial values of parameters are embedded into the *input* field of the transaction.

Functions and variables in Ethereum smart contracts can be specified by type specifiers [54]. Some of them specify the visibility, e.g., `private` and `external`, which can be used to differentiate access control on functions and variables. Another specifier, named `immutable`, is introduced to specify read-only variables. Different with the ordinary constants, immutable variables are calculated and initialized till the contract is deployed by the creation code.

B. Smart Contract Verification

Code Is Law, is one of the core principles of Ethereum. In other words, all smart contracts in finalized blocks are non-updatable and cannot be rolled back. As we mentioned in §II-A, smart contracts are stored in the form of EVM bytecode. Due to its unreadability, it requires huge efforts to identify developers' original intention.

Such a transparency issue urges the emergence of *smart contract verification*[1], which is either a feature offered by Ethereum browsers (e.g., Etherscan [26] and Blockscout [6]) or a service provided by the Ethereum official (e.g., Sourcify [2]). Specifically, anyone can upload a source code file and claim it is the implementation of an on-chain smart contract. The service is responsible for verifying if the bytecode compiled from the given source code matches the one deployed on-chain. Once matched, all users are able to access the source code file for further usage.

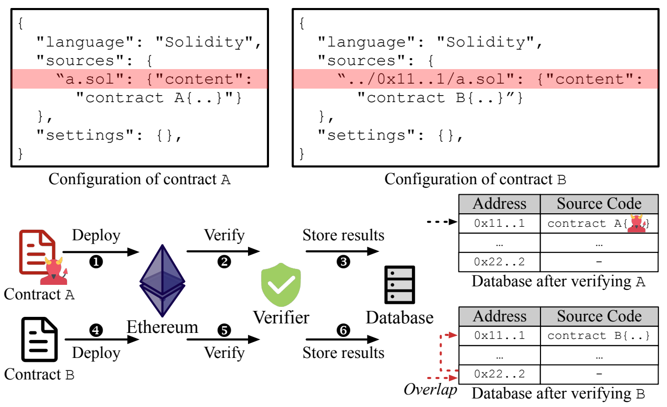
There are two specific results to further describe that the provided source code is matched to the on-chain contract, i.e., *partial match* and *exact match*. To be specific, if the given source code can generate an identical result with the on-chain one except for the metadata, it is described as a *partial match*. This is because some configuration items are inconsistent with the ones at the time of original compilation. Otherwise, it is an *exact match*. However, such a subtle difference is only considered by Sourcify, where an exact match can replace the existing partial match on the same contract [58]. In contrast, Blockscout and Etherscan do not allow replacements of any already manually verified contracts. According to the statistics [25], [2], [6], less than 1%, 0.7% and 0.3% of contracts have been manually verified in Etherscan, Sourcify, and Blockscout, respectively, indicating a huge attack surface for adversaries.

III. Adversary Model

A. Motivating Example

If the smart contract verification services are exploited, it is possible that the provides source code is inconsistent with the contract in the designated address. Fig. 3 illustrates a concrete example. As we can see, from **step 1** to **step 3**, attackers intentionally deploy an evil contract (`A`), request a source code verification by providing `A`. Thus, the source code of `A` is stored in the corresponding address, `0x11...1`, for

[1]Note that we use *smart contract verification* and *source code verification* interchangeably in this paper.



```
{
  "language": "Solidity",
  "sources": {
    "a.sol": {"content":
      "contract A{..}"}
  },
  "settings": {},
}
```

Configuration of contract `A`

```
{
  "language": "Solidity",
  "sources": {
    "../0x11..1/a.sol": {"content":
      "contract B{..}"}
  },
  "settings": {},
}
```

Configuration of contract `B`

Address	Source Code
0x11..1	contract A{..}
...	...
0x22..2	-

Database after verifying `A`

Address	Source Code
0x11..1	contract B{..}
...	...
0x22..2	-

Database after verifying `B`

Fig. 3: A motivating example that exploits verification services.

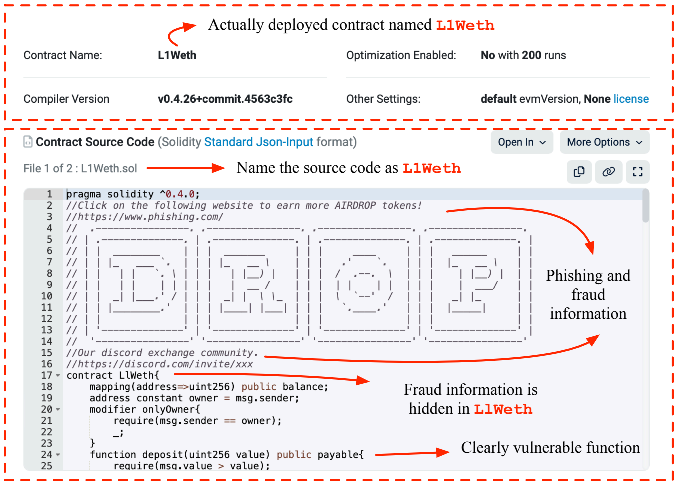
example. Then, to hide their malicious intention, they compose another harmless contract, named `B`. Based on the contract `B`, attackers repeat the above processes from **step 4** to **step 6**. However, at **step 5**, they slightly modify the configuration file. As the highlighted rows indicate, they claim the path of `B` as `../0x11...1/a.sol`. To this end, the source code of `B` overwrites the `A`'s in the back-end database. Users who visit `0x11...1` eventually obtain the source code of `B` without explicit warnings. In other words, the harmless contract covers the malicious intention of the actually deployed one. This vulnerability is uncovered in Sourcify by us (see §VI-B4).

B. Adversary Model

We assume that attackers can access both on-chain and off-chain data as normal users. Specifically, attackers can access all deployed smart contracts through a self-deployed node or Ethereum browsers, and arbitrarily request verification by providing necessary files. Thus, depending on whether attackers request verification on contracts deployed by themselves or other developers, we can divide the consequences of exploiting smart contract verification services into two categories, i.e., *adversarial verification* and *source scam*.

Adversarial Verification ($\mathcal{A}_1$). This consequence corresponds to verifying contracts deployed by other developers. Given an arbitrary contract bytecode, malicious users can forge a piece of source code that can successfully pass the verification process by exploiting vulnerabilities in the verifiers. Such a verification not only does not require authorizations from the actual developers, but also does not notify them that their contracts have been verified by others. Additionally, the verified source code can show malicious or scam information that discredits the victim. For example, Fig. 4 illustrates a real-world example[2]. Specifically, we can see the actual deployed contract is named as `L1Weth`, and its meta information is shown in the upper part. Attackers exploit a vulnerability (the one detailed in §VI-A1) and provide a source code file, as shown in the lower part. There are three interesting points: 1) attackers deliberately name the source code file as `L1Weth.sol`, though the most visible contract is named as `LlWeth`; 2) there are an ASCII art diagram, a phishing link and a fake discord channel; and 3) the `deposit` function

[2]Address: 0xc536...


Actually deployed contract named L1Weth
Name the source code as L1Weth
Phishing and fraud information
Fraud information is hidden in LlWeth
Clearly vulnerable function
Open In
More Options

Contract Name: **L1Weth**
Optimization Enabled: **No** with **200** runs
Compiler Version **v0.4.26+commit.4563c3fc**
Other Settings: **default** evmVersion, **None** license

Contract Source Code (Solidity Standard Json-Input format)
File 1 of 2 : L1Weth.sol

```
pragma solidity ^0.4.0;
//Click on the following website to earn more AIRDROP tokens!
//https://www.phishing.com/
//  .--------------. .--------------. .--------------. .--------------.
// | .------------. || .------------. || .------------. || .------------. |
// | |            | || |            | || |            | || |            | |
// | |  ________  | || |  _______   | || |     ____   | || |   ______   | |
// | | |_   ___ `.| || | |_   __ \  | || |   .'    `. | || |  |_   __ \ | |
// | |   | |   `. \| || |   | |__) | | || |  /  .--.  \| || |    | |__) || |
// | |   | |    | || || |   |  __ /  | || |  | |    | || || |    |  ___/ | |
// | |  _| |___.' /| || |  _| |  \ \_| || |  \  `--'  /| || |   _| |_    | |
// | | |________.' | || | |____| |___|| || |   `.____.' | || |  |_____|   | |
// | |            | || |            | || |            | || |            | |
// | '------------' || '------------' || '------------' || '------------' |
//  '--------------' '--------------' '--------------' '--------------'
//Our discord exchange community.
//https://discord.com/invite/xxx
contract LlWeth{
    mapping(address=>uint256) public balance;
    address constant owner = msg.sender;
    modifier onlyOwner{
        require(msg.sender == owner);
        _;
    }
    function deposit(uint256 value) public payable{
        require(msg.value > value);
```

Fig. 4: A real-world example of adversarial verification, where the upper and lower parts correspond to the meta information of the deployed contract and the one of the provided source code, respectively.

has an obvious vulnerability. Intuitively, the credibility of the original contract is significantly compromised. As for the *adversarial*, it can be explained in two ways. On the one hand, a successful source code verification, even not requested by the actual developers, has no chance to be replaced in Blockscout and Etherscan. On the other hand, in Sourcify, if a smart contract is only partially verified by its developers, attackers can construct an exact match to adversarially replace the original one (see §II-B).

Source Scam ($\mathcal{A}_2$). This consequence corresponds to the situation where attackers are exactly the developers of the verified contracts. As the example in §III-A, attackers can exploit vulnerabilities by providing a seemingly harmless contract to hide malicious intentions. Consequently, the fake contract gains users' trust, which is however a trap.

These two consequences are explicitly related to users who access Ethereum contracts. Further, the vulnerabilities can also bring in critical consequences to the whole ecosystem. For example, lots of downstream applications, e.g., source-code level vulnerability detectors like Slither [28] and Echidna [32] and smart contract auditing companies [8], rely on the source code retrieved from these verifiers. If they are compromised, the reliability of such services is severely impacted.

IV. Source Code Verifier

Against three mainstream verifiers, i.e., Etherscan, Sourcify, and Blockscout, we firstly overview their general workflow in §IV-A. Then, we delve deeper in their implementations in §IV-B and §IV-C. Last, against the proprietary Etherscan, we detail how a black-box testing is conducted in §IV-D.

A. Overview

Fig. 5 illustrates the general workflow and architecture of verifiers. Moreover, Table I illustrates the options adopted by them in each module. Generally speaking, a verifier takes source code, configuration items, and an address as inputs.

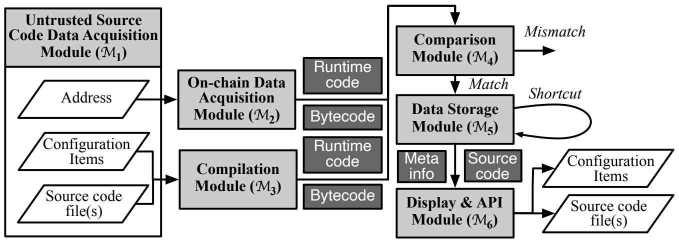

Untrusted Source Code Data Acquisition Module ($\mathcal{M}_1$)
Address
Configuration Items
Source code file(s)
On-chain Data Acquisition Module ($\mathcal{M}_2$)
Compilation Module ($\mathcal{M}_3$)
Runtime code
Bytecode
Runtime code
Bytecode
Comparison Module ($\mathcal{M}_4$)
Mismatch
Match
Shortcut
Data Storage Module ($\mathcal{M}_5$)
Meta info
Source code
Display & API Module ($\mathcal{M}_6$)
Configuration Items
Source code file(s)

Fig. 5: Architecture and workflow of the source code verifiers.

According to the configuration items, the verifier can compile the given source code in a deterministic way. Then, the verifier fetches the bytecode and the runtime code according to the given address, and compares if they are identical to the locally compiled ones[3]. Once matched, the provided source code would be labeled as *verified* and served to users who query it via APIs exposed by verifiers.

B. Components of Verifier

Smart contract verifiers can be abstracted into six modules with calling relations, denoted from $\mathcal{M}_1$ to $\mathcal{M}_6$, respectively.

1) Untrusted Source Code Data Acquisition Module ($\mathcal{M}_1$): $\mathcal{M}_1$ is responsible for obtaining untrusted data, i.e., source code file(s), configuration items, and an address, from the verifying requester. As for providing multiple source code files, it is because a DApp may require multiple files to achieve a complex functionality. Thus, to enable compiling a DApp with the hierarchical structure among source code files, solc allows users to pack source code files and their paths into a single file in JSON format. Additionally, once a contract is compiled by solc, a set of configuration items is generated automatically, corresponding to the meta information of the compilation adopted this time, e.g., solc version. Last, the contract stored in the given address is regarded as the verification target. Moreover, specific service providers may require extra information. For example, Blockscout requires a contract name as the entry when multiple contracts exist.

2) On-chain Data Acquisition Module ($\mathcal{M}_2$): From the given address, this module retrieves necessary on-chain data, i.e., the runtime code and the bytecode (i.e., both creation code and runtime code) on demand. Specifically, to obtain the on-chain runtime code, $\mathcal{M}_2$ queries a standard RPC, named `eth.getCode()`, which is exposed by client nodes. As for obtaining the bytecode, $\mathcal{M}_2$ firstly queries some third-party databases, e.g., Etherscan, to obtain the hash of the creation transaction to the address. Then, $\mathcal{M}_2$ queries `eth_getTransactionByHash()` and extracts the bytecode from its `input` field.

3) Compilation Module ($\mathcal{M}_3$): $\mathcal{M}_3$ invokes solc to compile and obtain both bytecode and runtime code for the subsequent comparison process. According to the source code files and configuration items obtained by $\mathcal{M}_1$, solc can perform compilation in a deterministic way. Even so, the compiled runtime code may be unusable. The reason lies in that there

[3]Note that the comparison works on demand. Specifically, Blockscout only compares the bytecode, while Sourcify compares bytecode only if the comparison on runtime code failed [61]. Etherscan compares both of them.

TABLE I: Adopted options in modules for different source code verifiers.

		$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$	**Shortcut**
Etherscan[1]	Runtime code	Fetch on-chain ones according to the given address	Compilation + Replacing immutable	Regex matching in tailing part	Centralized database	Inheritance across identical runtime code
	Bytecode		Compilation			
Sourcify	Runtime code		Compilation + Simulating	Regex matching in tailing part[2]	IPFS	–
	Bytecode		Compilation	Prefix matching + Regex matching in tailing part[2]		
Blockscout	Bytecode		Compilation	Differential analysis	Centralized database	Inheritance across identical runtime code / Inheritance across platforms

[1]All adopted options in Etherscan are speculated, please refer to §IV-D.
[2]Sourcify only perform the comparison on bytecode once the result of the comparison of runtime code is mismatched [61].

may exist *immutable* variables (see §II-A), whose values are dynamically determined by executing the creation code during the deployment. We find two ways are adopted by verifiers to resolve this issue, i.e., *directly replacing immutable variables* and *simulating creation code* [57].

As for the former method, during the compilation, it leaves all immutable variables as blanks and records all their offsets. Then, before comparing the compiled one and the on-chain one, it fetches the corresponding bytes from the on-chain runtime code according to the recorded offsets, and fills them into the blanks. Hence, the values of immutable variables in the local compiled contract are definitely identical to the on-chain ones, which *somehow does not achieve the purpose of verification*. As for the latter method, $\mathcal{M}_3$ invokes `eth_call()` exposed by client nodes to simulate the behavior of creation code. For example, suppose there is a statement in a constructor: `address immutable public owner = msg.sender;`, where `owner` is an immutable variable that can be dynamically assigned as the user's address who deploys this contract. The advantage of this statement is that the owner address does not need to be hard-coded, but is dynamically set as the transaction initiator during deployment. This avoids permission verification problems such as owner address misuse.

4) Comparison Module ($\mathcal{M}_4$): $\mathcal{M}_4$ is responsible for comparing the fetched on-chain data and the locally compiled ones. As we stated in §IV-B2, the bytecode is obtained from the `input` field of the creation transaction, which means that the bytecode may be followed by values of parameters if the constructor requires (see §II-A). Thus, instead of asking requesters to provide concrete values of parameters, another feasible strategy is determining if the locally compiled bytecode is the *prefix* of the one parsed from the creation transaction.

Except for that, comparing bytecode or runtime code is still challenging due to the existence of metadata. Though verification requesters are asked to provide configuration items to try to generate identical metadata, it is still difficult to ensure that the uploaded ones are identical to the ones at the actual compilation. Thus, removing the metadata in advance is the best choice to avoid verification failure. These three verifiers adopt distinct methods to identify and remove metadata. Specifically, because the metadata is wrapped by explicit indicators [54], Sourcify directly identifies such a pattern through regex matching at the tail of the runtime code. However, as Fig. 2c shows, factory contracts may have multiple metadata, where only removing the tailing one is insufficient. Thus, Blockscout adopts another method. It intentionally inserts additional configuration items, which only changes the metadata while keeping runtime logic unaltered. It can identify all metadata by observing which bytes are changed and wrapped in the metadata pattern.

5) Data Storage Module ($\mathcal{M}_5$): After completing the above comparison, all uploaded files need to be properly preserved. Thus, $\mathcal{M}_5$ is accountable for storing them in a permanent database. For Etherscan and Blockscout, they store all source code files in their owned and centralized back-end servers. Users may be concerned about such a centralization issue, like a single point of failure can cause a completely unavailable service. Moreover, such centralized back-end servers adopted by verifiers have been compromised in TRON, where 26 million TRX tokens are stolen [18]. Thus, Sourcify adopts IPFS [36], a decentralized file system that can be accessed by anyone who maintains an IPFS node, to enhance its confidence.

6) Display & API Module ($\mathcal{M}_6$): Through $\mathcal{M}_6$, users can access the corresponding source code files when asking for a specific address. Additionally, the displayed information also includes some meta information as illustrated in Fig. 4. Furthermore, Sourcify directly offers a metadata file consisting of all uploaded configuration items to users.

C. Shortcuts

All these three verifiers generally follow the design and workflow we mentioned in §IV-A and §IV-B. However, we found some *shortcuts*, i.e., skipping some modules when a condition meets, exist in their implementations. The purpose of these shortcuts is to alleviate the workload or save computing resources. We detail the found two shortcuts in the following.

Source Code Inheritance across Identical Runtime Code. Intuitively, if two on-chain runtime code are identical, we can conclude that the specific logic of the non-constructor part of these two contracts is equivalent. Therefore, in Etherscan and Blockscout, if a contract is verified, all other contracts whose runtime code are identical to this one's will also be marked as verified. Such a shortcut provides an automatic verification for factory contracts. Take a famous decentralized finance application, Uniswap [66], as an example, which allows token exchanges at an interest rate calculated by supply and demand.

TABLE II: Security properties for source code verifiers.

	Security Property	Related Modules
$\mathcal{P}_1$	Unrestorability	$\mathcal{M}_3$, $\mathcal{M}_4$
$\mathcal{P}_2$	Consistency	$\mathcal{M}_2$, $\mathcal{M}_3$, $\mathcal{M}_4$, $\mathcal{M}_5$, $\mathcal{M}_6$

Users can create and deploy a contract of an exchangeable token pair by invoking `create_pair` in the factory contract. Because the concrete steps of how to perform exchanges are coded in the factory contract, all newly deployed contracts are identical in terms of runtime code. Consequently, the previously verified source code files can be directly inherited to the newly created ones without further verification requests.

Source Code Inheritance across Platforms. To avoid resource consumption, Blockscout recognizes the verification results of Sourcify without manual interventions. In other words, against an address, users can only request verification on Sourcify, Blockscout automatically inherits the source code files uploaded to the address. Note that, there is no automatic results inheritance *from* or *to* Etherscan, which still requires manual verification requests from users for each contract.

D. Blackbox Testing in Etherscan

Etherscan is not open-source, thus we use a black-box testing based approach to investigate what options are adopted by each module. Specifically, the process can be roughly divided into: 1) constructing and deploying a contract, 2) constructing the corresponding source code, 3) trying to bypass or utilize specific options in modules, and 4) observing whether the verification is passed.

Take determining how metadata is identified in Etherscan as an example. The biggest challenge for identifying metadata lies in factory contracts, who have multiple ones. Thus, we firstly deploy several factory contracts with different solc versions. Then, we ask verifications for them by providing the corresponding source code files. The results show that the verification cannot pass due to the existence of unidentified metadata when solc is less than or equal to 0.4, but plays opposite when solc is greater than 0.4. Thus, we infer that when solc $\leq$ `0.4`, Etherscan adopts the same method as Sourcify's, i.e., regex matching. However, we cannot conclude which option is adopted when solc is greater than 0.4. All speculated options of these modules are reflected in Table I. Detailed processes and specific contracts used for black-box testing are released in our artifact: link.

V. Security Properties

According to the two possible consequences (see §III-B), two security properties ($\mathcal{P}$) should be guaranteed to protect the normal functionalities of source code verification services, which are shown in Table II.

Unrestorability ($\mathcal{P}_1$). *Restoration* refers to a kind behavior, where a user somehow obtains source code via reverse engineering or exploiting compiler features. However, this source code is definitely not the original one composed by developers, but can bypass the source code verification. Such a behavior is intuitively unexpected for both users and developers as it

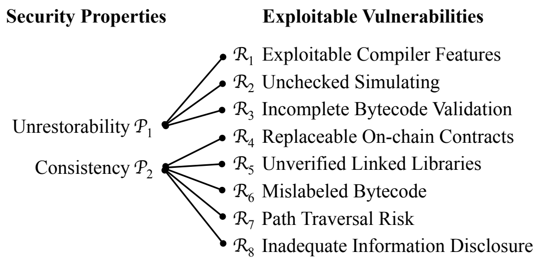

Security Properties
Exploitable Vulnerabilities
$\mathcal{R}_1$ Exploitable Compiler Features
$\mathcal{R}_2$ Unchecked Simulating
$\mathcal{R}_3$ Incomplete Bytecode Validation
$\mathcal{R}_4$ Replaceable On-chain Contracts
$\mathcal{R}_5$ Unverified Linked Libraries
$\mathcal{R}_6$ Mislabeled Bytecode
$\mathcal{R}_7$ Path Traversal Risk
$\mathcal{R}_8$ Inadequate Information Disclosure
Unrestorability $\mathcal{P}_1$
Consistency $\mathcal{P}_2$

Fig. 6: Relations between security properties ($\mathcal{P}$) and exploitable vulnerabilities ($\mathcal{R}$) in source code verifiers.

cannot reflect the original intention of the deployed contract. Analogous to the *preimage resistance* of hash functions [68], we propose *unrestorability*, trying to avoid such behaviors. To meet this property, verifiers should be robust to the following threats. First, *threats to the compilation module*. Features adopted by $\mathcal{M}_3$ may violate this security property. For example, both verbatim function [51] and loose inline assembly [53] in solc can assist developers to designate the compiled opcode sequence. Second, *threats to the comparison module*. During the comparison stage, as we mentioned in §IV-B4, some options are enabled to enhance the user-friendliness, like only considering the prefix of the retrieved bytecode. Such options can be utilized by attackers to hamper the unrestorability.

Consistency ($\mathcal{P}_2$). *Consistency* refers to a property that the semantics between the source code files that passed the verification and the targeted on-chain contract should be identical. Thus, comparing to the semantics of the targeted on-chain contract, there should not be redundancy or absence in the semantics of the uploaded source code files. Intuitively, two types of inconsistency can emerge. On the one hand, *the semantics of the provided source code files is absent*. For example, in $\mathcal{M}_3$, the implementation of functions in linked libraries is not included in the caller. In addition, before the comparison in $\mathcal{M}_4$, all metadata should be identified and removed. However, if a piece of runtime logic is mistakenly marked as metadata, the corresponding semantics will be overlooked. Consequently, the absence in the semantics of source code files results in more code being executed than users think. On the other hand, *the semantics of the provided source code files is redundant*. In $\mathcal{M}_6$, the consistency is reflected by the unambiguity of the displayed information. For example, if attackers can display redundant or misleading information, like multiple contracts with an identical name, the unambiguity is violated. The downstream applications based on the data exposed by $\mathcal{M}_6$ will be influenced. Moreover, such an inconsistency can also emerge after the verification. Specifically, after a successful verification, the uploaded source code files are stored in $\mathcal{M}_5$. The stored ones may be overwritten unexpectedly or intentionally by tampering file systems.

VI. Vulnerability Details

Against two security properties, we found eight types of vulnerabilities, whose relations are shown in Fig. 6. In this section, we firstly organize them according to their consequences ($\mathcal{A}_1$ and $\mathcal{A}_2$). Then, we present the technical details of detection, exploitation, and mitigation against each of them.

A. *Vulnerabilities Leading to $\mathcal{A}_1$*

Three types of vulnerabilities ($\mathcal{R}_1$ to $\mathcal{R}_3$) are categorized to this consequence. The impact of exploiting them depends on the scale of unverified contracts and the practicality of the exploits. As we introduced in §II-B, less than 1%, 0.7% and 0.3% of contracts have been manually verified in Etherscan, Sourcify, and Blockscout, respectively. Thus, if the corresponding exploits are used extensively, the impact can be summarized in twofold: 1) preventing original developers from providing actual original source code files, and 2) embedding malicious or scam information within the provided source code files to discredit victims. Specifically, as $\mathcal{R}_1$ to $\mathcal{R}_3$ violate $\mathcal{P}_1$, attackers are able to construct a contract according to the victim's bytecode to bypass the source code verification. In the same source code file, attackers can compose another contract, within which embeds phishing or scam information as illustrated in Fig. 4 to discredit the victim contract. The latter contract can even be placed in a more visible position. In the following, we mainly focus on how to forge source code files, instead of the concrete way to spread malicious information and commit fraud.

1) Exploitable Compiler Features ($\mathcal{R}_1$): In $\mathcal{M}_3$, attackers can abuse features of solc to break $\mathcal{P}_1$. Specifically, such features include *YUL* [56] (an intermediate language) and *loose inline assembly* [53], both of which can be inlined in Solidity. For example, taking advantage of the `verbatim_0i_0o` function in YUL, users can directly designate the compiled opcode sequence. Though according to the implementation of solc, an invalid opcode `fe` will be appended after the opcode sequence, it still reflects how users can flexibly control the deployed bytecode via abusing compiler features. Likewise, when solc $\leq$ `0.4`, users can also adopt loose inline assembly to generate arbitrary opcode sequences.

Detection. To detect if $\mathcal{R}_1$ is exploitable in three verifiers, we construct a set of contracts. Each of them is composed of a single fallback function, where it embeds a verbatim function or a piece of loose inline assembly code. Fallback function has an empty function signature, meaning that the compiled bytecode is not wrapped by additional opcodes, like function identifiers [52]. Through observing the difference between the opcode sequence declared in source code and the consequently compiled one, we can conclude if $\mathcal{R}_1$ is exploitable in verifiers.

```
1 contract A_ {
2  //target bytecode '6080604052600436106101 33..'
3  function() external payable{
4   assembly{       //6080604052
5     0x4           //6004
6     calldatasize  //36
7     lt            //10
8     tag1          //610133
9     ...
```

Listing 1: The PoC of $\mathcal{R}_1$.

PoC. Consider a deployed contract named `A`, whose runtime code is `0x608060....` We can compose a contract `A_`, as shown in Listing 1, which only contains a fallback function (L3). To exploit $\mathcal{R}_1$ in Etherscan, we firstly translate the victim's, i.e., `A`'s, runtime code (without metadata) into a piece of loose inline assembly in the fallback function of `A_`, as shown from L4 to L9. Then, we compile `A_` locally and manually replace the metadata of `A_` to the `A`'s. In this way, `A_` and `A` are identical in terms of runtime code. Thus, we deploy `A_` and verify it by providing Listing 1, which should be an exact match certainly[4]. Consequently, due to the shortcut in Etherscan (see §IV-C), `A` is automatically labeled as verified with the source code of `A_`. As Sourcify allows only verifying the runtime code (see §IV-B4), it is easier to exploit it. We can directly request verification for `A` with `A_` in Listing 1. After the metadata is removed, they certainly have an identical runtime code to pass the verification. Since Blockscout inherits the results of Sourcify, Blockscout is also exploitable.

Exploitation Conditions. Successfully abusing $\mathcal{R}_1$ is limited by some conditions. First, only valid opcodes can be written in loose inline assembly, resulting in the PoC has to be compiled with solc $\leq$ `0.4` and the target contract cannot have multiple pieces of metadata. Specifically, if solc is greater than `0.4`, it automatically appends an invalid opcode `fe` in front of the tailing metadata. And if there are multiple pieces of metadata, it cannot guarantee that each byte of such a random hash string can always correspond to a valid opcode. Second, operands of `PUSH` cannot be led by zero. In Ethereum, there are 32 types of `PUSH` which can push 1 to 32 bytes, respectively. However, if we intend to encode opcodes like `PUSH2 0001` in loose inline assembly, solc will automatically optimize it as `PUSH1 01`, which changes the final opcode sequence.

Mitigation. It is impractical to prohibit the use of such features in the provided source code, because they can provide several advantages, like supporting functionalities unavailable in Solidity and saving gas, which are proven by Liao et al. [38]. Therefore, to avoid the abuse, here are two possible solutions for verifiers. On the one hand, raise the threshold for requesting contract verification, e.g., all requesters must follow the know-your-customer (i.e., KYC) policy [67] or being authorized in advance. On the other hand, if these features are found to be heavily used by solc, verifiers should allow source code replacement of such contracts or give users a clear warning.

2) Unchecked Simulating ($\mathcal{R}_2$): Simulating is an option in $\mathcal{M}_3$, which is adopted by Sourcify in verifying runtime code to handle immutable variables. Specifically, the simulator takes whatever it receives as a creation code, executes it locally, and regards the returned bytecode as the corresponding runtime code. Such a process may break $\mathcal{P}_1$ if attackers construct a wrapper (not a valid creation code), which returns the victim's runtime code directly. Generally, the pointer of the runtime code is acquired by `codecopy`, an opcode located at the end of the creation code. Then, the pointed data, i.e., runtime code, is returned and processed. However, solc does not check if the pointer is actually returned by `codecopy` and the validity of the returned data. Thus, attackers can construct a piece of code, within which explicitly returns a pointer that points to the victim's runtime code before the actual `codecopy`.

Detection. To detect if $\mathcal{R}_2$ is exploitable, we deliberately insert some functions that can explicitly change the control flow in the constructor, like `call` and `return`. By asking conducting source code verification on these contracts and observing if the control flow of the compiled bytecode changes, we can conclude if simulating is adopted in $\mathcal{M}_3$.

```
contract A_ {
 constructor() public {
```

[4]Note that, the modified metadata has no influence on the verification result because it is removed before comparing, see §IV-B4

```
3     // Assign victim A's runtime code to bytecode
4     bytes memory bytecode = hex'608060...';
5     assembly {
6      return (add(bytecode, 0x20), mload(bytecode))
7     }
8    }
9   }
```

Listing 2: The PoC of $\mathcal{R}_2$.

PoC. Suppose a contract `A` is deployed on-chain, and we can construct a contract `A_` as shown in Listing 2. At L4, it assigns the runtime code of `A` to the variable `bytecode`. Then, at L6, a return is explicitly declared in YUL, which is located before the actual `codecopy` opcode. Thus, the simulator will execute the `return` at L6 before running into the actual `codecopy`, leading to an *early return* of a pointer, which points to the `bytecode`. Considering that such an attack can accomplish the *exact match* (as metadata can also be directly encoded in `bytecode`), we can verify arbitrary contracts that are not exactly matched yet.

Mitigation. To mitigate the issue, we highly recommend verifiers to not adopt simulating during the compilation stage. This is because simulating has to rely on EVM, which inevitably introduces new features that may be improperly utilized or abused. After we made the suggestion, Sourcify officially abandoned simulating and started to adopt the direct replacing strategy that we mentioned in §IV-B3 [60].

3) Incomplete Bytecode Validation ($\mathcal{R}_3$): Considering the user-friendliness, verifiers may not ask users to provide the concrete values of parameters that are appended after the bytecode. Such a strategy can be abused by attackers, which breaks $\mathcal{P}_1$. Specifically, assume a constructor requires a parameter, whose value is p. It is always a mismatch if directly comparing locally compiled bytecode, denoted as $byte_{loc}$, to the `input` field of the creation transaction, denoted as $byte_{chain}||p$, due to the absence of p. Thus, verifiers may adopt *prefix matching*, i.e., verifying if $byte_{loc}$ is the prefix of $byte_{chain}||p$. If it is, they consider the verification passes. However, this strategy can be abused if a comprehensive examination misses. For example, a prefix of $byte_{loc}$, which is even not a valid bytecode, may also be the prefix of $byte_{chain}||p$. Furthermore, if verifiers do not take an empty string into consideration, an empty string can also be the prefix of $byte_{chain}||p$.

Detection. To detect if $\mathcal{R}_3$ is exploitable in verifiers, we intentionally design a set of contracts, which can generate a prefix of the victim contract. This process can be completed through utilizing loose inline assembly (see §VI-A1) or by composing *abstract contracts*. Specifically, abstract contracts can be compiled to an empty bytecode. In other words, an abstract contract can be regarded as a prefix of any contract. If such contracts can pass the verification, $\mathcal{R}_3$ is exploitable.

```
1   abstract contract WrappedToken{
2     mapping (address=>uint256) public balance;
3     function withdraw (uint256 value) public {
4       balance[msg.sender] -= value;
5       payable(msg.sender).transfer(value);
6     }
7   }
```

Listing 3: The PoC of $\mathcal{R}_3$.

PoC. Firstly, we choose a victim contract that is unverified. We claim the code snippet in Listing 3 is the corresponding source code, which is wrapped by an abstract contract. Abstract contracts have the following two characteristics: 1) the compiled bytecode is empty, and 2) functions within them will be recognized as ABI functions [50], just like the functions in normal smart contracts. Because of the problematic implementation of verifiers (e.g., Sourcify [59]), such an empty bytecode can be regarded as the prefix of any bytecode. To this end, users who browse this contract will notice that a function called `withdraw` is listed as an ABI function, but it has an apparent integer overflow vulnerability at L4. The actual deployed contract is discredited.

Mitigation. To avoid this exploitation, verifiers should firstly always perform a non-null check on $byte_{loc}$, i.e., an empty bytecode compiled locally is unacceptable. Further, verifiers should ensure that the arguments part can always be successfully identified. It can be done by verifying whether the suffix bytes after removing the matched prefix can be parsed as legal parameters according to the corresponding ABI file. If it is possible, it means that $byte_{loc}$ equals to $byte_{chain}$.

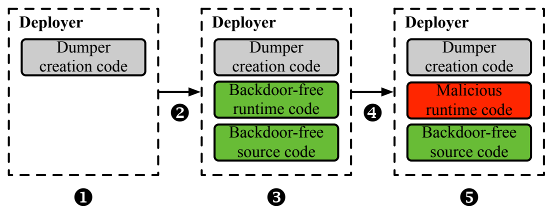
B. Vulnerabilities Leading to $\mathcal{A}_2$

Several types of exploitable vulnerabilities can lead to $\mathcal{A}_2$, i.e., conducting source scam for inexperienced users. They can cause extreme severe impact.

Specifically, the verification requesters who exploit these vulnerabilities ($\mathcal{R}_4$ to $\mathcal{R}_8$) are malicious developers themselves. They firstly deploy a malicious contract, and verify it by another piece of seemingly harmless source code. The verification between these two mismatched contracts can be bypassed by exploiting these vulnerabilities, where a concrete instance is illustrated as the motivating example (see §III-A). Similarly to the introduction on vulnerabilities leading to $\mathcal{A}_1$ (see §VI-A), in this section, we deliberately ignore details about how to construct and deploy malicious contracts. In the following, we mainly focus on how to bypass the verification, i.e., importing inconsistency to break $\mathcal{P}_2$.

1) Replaceable On-chain Contracts ($\mathcal{R}_4$): The status and enabled features of the client node that verifiers depend on can be exploited by attackers to replace already on-chain contracts, which breaks $\mathcal{P}_2$. For instance, client nodes may suffer the chain reorganization issue [41], which is a normal phenomenon for blockchains as forking is inevitable. According to the statistics [27], this issue happens around 5 times an hour in Ethereum. It requires service providers on blockchain platforms to proactively avoid the imported negative impact, e.g., decentralized exchanges only consider the transactions located under certain block heights as valid ones [15]. For verifiers, if unconfirmed on-chain smart contracts are taken as targets for the source code verification, the validity of their results may be hampered. Additionally, the EVM version adopted by the verifier-dependent client node may also lead to abuses. For example, Ethereum has introduced a new opcode, named `create2` [21]. It was originally designed to deploy a smart contract on a predetermined address for developers. However, this opcode can be abused to deploy another piece of runtime code on an existing address, which may invalidate the original source code verification result on the same address [24].

Detection. Against above two examples, to detect the former one, a reasonable method is to deploy multiple client nodes and



❶ Assemble a deployer with `Dumper`'s creation code
❷ Deploy `A`'s runtime code
❸ Conduct source code verification with `A`
❹ Self-destruct the runtime code
❺ Re-deploy malicious `A_`'s runtime code

Fig. 7: Overview of the PoC of $\mathcal{R}_4$.

initiate multiple transactions simultaneously to urge state forking. However, such a testing on consensus algorithm is out of scope of this work. Thus, we mainly focus on the exploitability of the later one, i.e., if verifiers can be cheated by redeploying contracts on an existing address through `create2`. Specifically, if the creation code is unchanged, the address of the deployed contract also keeps unchanged through `create2`. Taking advantage of this feature, we compose a contract as a deployer, which has a fixed creation code and takes the to-be-deployed runtime code as an argument. Intuitively, if an address is available and can be redeployed after the original one is self-destructed, the verified source code may correspond to the original one, indicating verifiers are exploitable to $\mathcal{R}_4$.

```
1  contract Deployer {
2   bytes public deployBytecode;
3   address public deployedAddr;
4   function deploy(bytes memory code) public {
5    deployBytecode = code;
6    address target;
7    bytes memory dumperBytecode = hex"{Dumper
         contract's creation code}";
8    assembly {
9     target := create2(callvalue, add(0x20,
          dumperBytecode), mload(dumperBytecode), 0
          x11)
10   }
11   deployedAddr = target;}
12 }
13 contract Dumper {
14  constructor() public {
15   Deployer dp = Deployer(msg.sender);
16   bytes memory bytecode = dp.deployBytecode();
17   assembly {
18    return (add(bytecode, 0x20),mload(bytecode))
19     }}
20 }
```

Listing 4: The toolchain of conducting PoC of $\mathcal{R}_4$.

PoC. Fig. 7 and Listing 4 illustrate how a fraud can be committed through exploiting $\mathcal{R}_4$. In the preparation stage, i.e., **step 1**, we first compile the `Dumper` declared at L13 in Listing 4, extract its creation code and assign it to `dumperBytecode` at L7. Thus, the contract `Deployer` at L1 can be seen as a general deployer. Any runtime code passed through `code` at L4 can be deployed at an unchanged address. At **step 2**, we compile a harmless contract `A`, and call the function `deploy` of `Deployer` at L4 with `code` as `A`'s runtime code. The `create2` at L9 deploys this contract on an address. Then, at **step 3**, we provide the corresponding source code files of `A` and ask for a verification. After the verification completes, we call the function `selfdesctruct` in `A` to make this address available, as shown in **step 4**. We again compose another evil contract `A_`, and pass its runtime code in `deployer`. Due to the characteristic of `create2`, such an evil contract is also deployed on the identical address to `A`. Consequently, the verification results of `A` are mistakenly bound on the actual executed contract, i.e., `A_`.

Mitigation. To prevent such an attack, verifiers should ensure the immutability of the contracts being verified and make necessary updates simultaneously. On the one hand, for those contracts that can still be rolled back, e.g., through a chain reorganization, it is best for verifiers not to provide services to them. On the other hand, for those contracts in finalized blocks, when verifiers provide the source code file, they should also highlight the difference between the current bytecode and the one at the time the verification is requested. If they are different, it means that the deployed contracts are self-destructed or re-inited by `create2` on the same address.

2) Unverified Linked Libraries ($\mathcal{R}_5$): If the implementation of invoked linked libraries is not considered by verifiers, it intuitively violates $\mathcal{P}_2$. Specifically, in order to support modular design when developing DApps, developers can embed part of the contract logic into a *library contract*. Two kinds of libraries exist, i.e., *linked library* and *embedded library*, whose distinctions are subtle [17]. On syntactic level, the only distinction is that they are specified by `public` and `internal`, respectively. However, they are encoded in different ways. Specifically, a call to functions in linked libraries is encoded as an opcode `delegatecall`, which takes the invoked library's address and the function signature of the callee as arguments. This means that the callee linked library is actually located at another address, and its logic does not appear in the caller's bytecode at all. Conversely, as for embedded libraries, the logic of callee is directly included into the caller's. Such subtle differences may not be taken into account by users. In other words, though verification requesters provide the source code of the caller contract and the invoked linked libraries together, verifiers only consider the function signatures instead of the implementation of the functions in the provided linked libraries. Because there is no clear warnings about this issue, users may be deceived as the implementation of invoked linked library is not verified at all.

Detection. To detecting if a verifier is exploitable to $\mathcal{R}_5$, we firstly deploy a contract that invokes a linked library where it poses malicious behaviors. Then, we verify the contract while uploading a library with identical function signature but different implementation. If the verification can pass, it means malicious users can commit fraud through exploiting $\mathcal{R}_5$.

```
pragma solidity ^0.8.0;
contract A{ // caller
  uint totalsupply = 0;
  function is_zero() public view returns(bool){
    L.check(totalsupply);
    // compiled to: L.delegatte(abi("check(uint)
        ", totalsupply))
    return true;
  }
}
library L{ // linked library
  function check(uint balance) public view{
    require(balance == 0);
  }
}
```

Listing 5: The PoC of $\mathcal{R}_5$.

PoC. Listing 5 illustrates a call to a linked library. We can see from L5 that `L` is a linked library, and `check` is the callee function. This is equivalent to the statement at L6, i.e., only the address of `L` and the signature of `check` is considered by the caller contract. To this end, we can commit fraud by utilizing this feature. Specifically, we firstly update Listing 5 by replacing L12 to `selfdestruct(msg.caller)` that can destroy the contract in callers. Then, we compile and deploy this malicious contract on-chain. To conduct exploitation, we provide the seemingly normal Listing 5 at the source code verification, which should pass the verification because the contract `A` is unchanged, and the signature of the callee function `check` in the linked library also keeps unaltered. Consequently, the malicious intention is covered.

Mitigation. If the invoked linked libraries cannot be verified recursively, we urge verifiers to explicitly warn users that the implementation of linked libraries is unreliable. Moreover, users need to carry out further checks themselves. In other words, only when the main contract and the invoked linked libraries are both verified and there are no security issues in their implementations, users can trust the main contract.

3) Mislabeled Bytecode ($\mathcal{R}_6$): When performing the comparison in $\mathcal{M}_4$, verifiers flag and intentionally skip certain fields in order to avoid unnecessary verification failures. However, such an intentional skip may lead to the inconsistency issue. Two types of fields lie in this scope, i.e., *metadata* and *linked library placeholders*.

The necessity of removing all metadata before $\mathcal{M}_4$ is detailed in §IV-B4. However, problematic metadata labeling and extracting may leave metadata behind or extract innocent bytes as metadata, leading to a failed verification or unexpected verification bypass, respectively. Specifically, the current pattern of metadata is 53-bytes long, including a 34-bytes IPFS hash, a 3-bytes solc version, and a 16-bytes magic number [54]. To extract all metadata, as we mentioned in $\mathcal{M}_4$, two ways are adopted by current verifiers, i.e., *regex matching* and *differential extracting*. Specifically, no matter where a metadata locates, regex matching is supposed to be effective and efficient. However, in 2021, a white hat exploited the buggy regex matching to intentionally mark a piece of runtime code as metadata, where he hid a backdoor [49]. It proves the unreliability of this strategy. Thus, Blockscout has proposed the idea of differential extracting. According to its implementation [5], it can be summarized as follows:

- According to configuration items, it compiles the given source code files, which is denoted as c.
- Blockscout intentionally adds a pre-defined and useless configuration item to make the compiled bytecode changed as c'. The newly added item is to import a linked library located in a file named `SOME_TEXT_USED_AS_FILENAME` [7].
- By comparing c and c', Blockscout can identify an index, denoted as i, where the first difference occurs. Based on i, it looks backward and forward to find a byte string that follows the metadata pattern.
- The above indexing and metadata identifying process repeat till all metadata are identified.

Linked library placeholders should also be removed before conducting comparison. Specifically, during the compilation, solc firstly replaces each invoked linked library address with a 40-bytes placeholder. If users require verification on Etherscan or Blockscout, both verifiers ask users to provide concrete addresses. Unlike them, Sourcify provides a more user-friendly solution, which is detailed as follows:

- Sourcify identifies the first appeared placeholder, records its offset, and extracts the following 40-bytes string as a *regex pattern*.
- Sourcify then locates the bytes from on-chain bytecode according to the recorded offset, regarding them as the actual invoked library address.
- Among all placeholders, it matches the ones according to the regex pattern (1st step), and replaces them with the on-chain bytes (2nd step).
- The above process repeats till no placeholders exist.

This implementation is safe when solc is greater than `0.4`, where the placeholder is a 34 bytes hash with fixed prefix and suffix. However, the situation turns opposite when solc is `0.4`, where the placeholder is a string like:

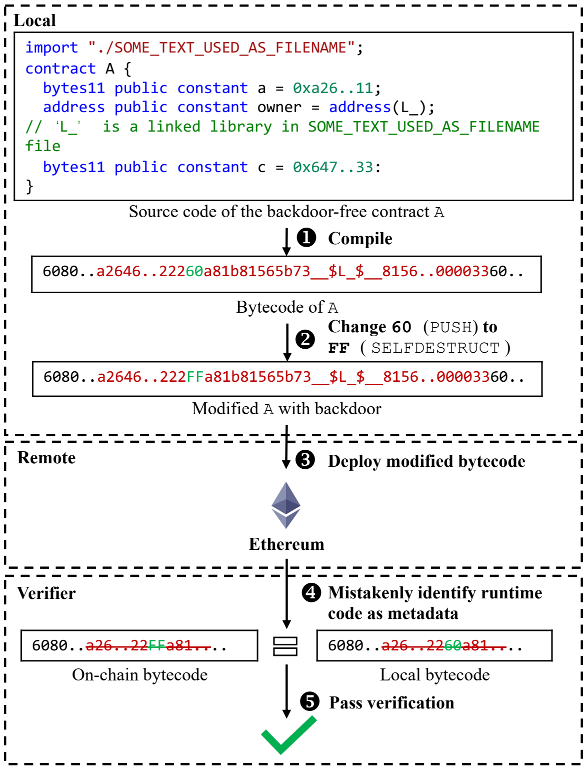
$$\_\ \_\ \ \textit{filePath}\ :\ \textit{libName}\ \_\ \_$$

, where *filePath* and *libName* are strings explicitly declared in the source code. Because solc does not require the format of *filePath*, e.g., ended by `.sol`, attackers can construct a value to abuse Sourcify's regex matching mechanism.

Detection. Instead of exploiting the regex matching mechanism in identifying metadata like the previous attack [49], we focus on if the differential labeling proposed by Blockscout is vulnerable. To this end, we compose a contract, in which we deliberately invoke the library imported by Blockscout. Then, between $\mathcal{M}_3$ and $\mathcal{M}_4$, we compare the compilation results of the Blockscout generated one and the normal one. If some runtime logic is missed, it means that part of runtime logic is mislabeled, violating $\mathcal{P}_2$. **PoC #1** illustrates an example.

As for detecting the vulnerability on identifying linked library placeholders, we construct a contract whose *filePath* and *libName* are deliberately set as strings following regex grammar. Then, after $\mathcal{M}_3$ of Sourcify, we observe if the forged *filePath* and *libName* can hit any valid runtime logic and replace it with on-chain bytecode. If it is, it means attackers can construct malicious contracts to arbitrarily label runtime logic to intentionally remove it before the comparison in $\mathcal{M}_4$. We will demonstrate is in the following **PoC #2**.

PoC #1. Fig. 8 illustrates how to exploit the differential extracting. Specifically, we compose a piece of source code, as shown in the top of Fig. 8. We deliberately import a file named `SOME_TEXT_USED_AS_FILENAME`, and assign the library (`L_`), which is defined in the imported file, to a variable. At **step 1 & 2**, we compile the source code, and locally update an opcode near the library placeholder to `FF`, i.e., `selfdestruct`. Then, we deploy such a modified bytecode on-chain. After that, we request a source code verification through Blockscout. Because the library `L_` is defined in the imported strange file, which is inserted by Blockscout when conducting differential extracting, the bytes at that location



```
import "./SOME_TEXT_USED_AS_FILENAME";
contract A {
  bytes11 public constant a = 0xa26..11;
  address public constant owner = address(L_);
// 'L_'  is a linked library in SOME_TEXT_USED_AS_FILENAME
file
  bytes11 public constant c = 0x647..33:
}
```

Local
Source code of the backdoor-free contract A
❶ Compile
6080..a2646..22260a81b81565b73__$L_$__8156..00003360..
Bytecode of A
❷ Change 60 (PUSH) to FF (SELFDESTRUCT)
6080..a2646..222FFa81b81565b73__$L_$__8156..00003360..
Modified A with backdoor
Remote
❸ Deploy modified bytecode
Ethereum
Verifier
❹ Mistakenly identify runtime code as metadata
6080..~~a26..22FFa81..~~..
On-chain bytecode
6080..~~a26..2260a81..~~..
Local bytecode
❺ Pass verification

Fig. 8: Overview of the PoC #1 of $\mathcal{R}_6$.

are unexpectedly updated. According to the implementation adopted by Blockscout we mentioned before, it should mark the `L_` nearby area, including the malicious `FF`, as metadata and remove it, which is normal runtime logic actually. Because other parts of bytecode are still identical to the on-chain ones, it can pass the verification, while users are unaware of there is a dangerous `selfdestruct` hidden in the deployed contract.

```
1  pragma solidity ^0.4.0;
2  import "./$.{37}|2{40}|"; // file path
3  contract A {
4   address constant public owner = address(0x222
        ..22);
5   uint public b;
6   function c() public{
7     b = foo.bar();
8  }}
9
10 library foo{ // lib name
11  function bar() public pure returns(uint) {
12    return 1;
13 }}
```

Listing 6: The PoC #2 of $\mathcal{R}_6$.

PoC #2. Listing 6 illustrates a malicious smart contract. The L2 and L10 correspond to *filePath* and *libName*, respectively. Thus, a placeholder is generated as[5]:

`__$.{37}|2{40}|:foo_..._`

To this end, Sourcify takes the above placeholder as a regex pattern. Specifically, the `__$.{37}` can match the current placeholder itself, but the `2{40}` can match the address declared at L4. Suppose this placeholder locates at the offset o. We can place any 40-byte sequences at o on the on-chain contract, like an address or even a backdoor. Therefore, through the constructed regex pattern, Sourcify mistakenly replaces the address at L4 to a predefined byte sequence.

Mitigation. Generally speaking, when dealing with such non-naive fields, like metadata and linked library placeholders, extra operations need to be conducted to avoid unnecessary verification failures. According to Occam's razor, verifiers should prioritize the reliability instead of introducing unnecessary steps, which can be reflected on the mitigations against these two exploitations.

Specifically, when identifying metadata, the differential extracting is better because it is conducted on semantic level while the regex matching is only on syntactic level and proven unsafe [49]. However, Blockscout unnecessarily introduces a linked library that can be exploited. Therefore, we urge the verifiers to adopt the simplest way of differential extracting, like adding a meaningless space character after the source code, which can update the metadata while keeping runtime logic unchanged. As for identifying linked library placeholders, Sourcify can avoid this attack by enumerating all placeholders one by one instead of adopting regex matching. Because verifying a contract is basically a one-shot process, underlining the efficiency barely introduces extra advantages.

4) Path Traversal Risk ($\mathcal{R}_7$): Similar to $\mathcal{R}_4$, where on-chain contracts can be replaced by exploiting features in EVM, by exploiting $\mathcal{R}_7$, existing contracts can also be overwritten. Specifically, solc allows users to pack source code files and their corresponding paths in a single JSON file, where the paths can be designated arbitrarily. To this end, if verifiers do not strictly validate the paths, it is very likely that attackers can abuse this feature to conduct an arbitrary path traversal.

Detection. To detect if a verifier is vulnerable to $\mathcal{R}_7$, we construct several JSON files, within which the paths of source code files are declared in a malicious path traversal way, like deliberately adding `../` to refer to its parent directory. Then, we try to access the directory to see if the source code files are uploaded to that invalid directory.

PoC. Suppose there is a contract with backdoors named `A`. After the deployment and source code verification, let us assume its source code is stored in `0x12..fe/source/A.sol`. To fool users, we can deploy another contract without backdoors, named `A_` for instance. However, during performing the source code verification, we claim the path of `A_` is: `../../0x12..fe/source/A.sol`. To this end, the verifier stores the source code of `A_` at the place where it should be used to store `A`. When users examine the source code files of `A`, they are fooled by the implementation of `A_`.

Mitigation. Path traversal is a well-studied vulnerability in traditional scenarios, e.g., web applications [30]. Therefore, we strongly suggest verifiers to adopt a comprehensive sanity check on the given paths of source code files by following the best practices that are widely-adopted [43], [46].

5) Inadequate Information Disclosure ($\mathcal{R}_8$): The inconsistency can be introduced by not only really existing issues (like $\mathcal{R}_4$ to $\mathcal{R}_7$), but also users misunderstanding. Specifically, if a source code verification is passed, the verifier should display

[5]The ... refers to underline characters to make the whole placeholder as 40 bytes long.

the verified information to the public. Take the Etherscan as an example (see Fig. 4), it illustrates lots of meta information, but the contract name does not contain the directory it belongs to. Such an inadequate information disclosure may mislead users and be abused. For example, attackers can put the main contract and a fake one with the identical name under different paths. Because only the contract name is displayed, users may be misled to take the fake one as the verified contract, which can negatively impact the credit of the project.

Detection. To detect $\mathcal{R}_8$, we can upload multiple contracts with an identical name to verifiers. Then, we observe if users can distinguish them. If they cannot, we conclude that this verifier is exploitable to $\mathcal{R}_8$.

PoC. We develop two contracts, both named `erc721.sol`, following a simplified ERC-721 standard [20] to issue tokens. They are put under different directories. e.g., `test/erc721.sol` and `main/erc721.sol`. The only difference between them is the initial value of `totalSupply`, i.e., the field refers to the maximum amount of tokens that is allowed to be minted, where the value of `totalSupply` under the `test` one is much higher than the one under the `main` directory. We deploy the `test/erc721.sol` firstly, and verify it by uploading both files in a json format. To fool users further, we can even put another `config.sol` under the `main` directory. Consequently, users may be fooled by a seemingly limited edition of the one shown in `main/erc721.sol`.

Mitigation. Verifiers should ensure that the disclosed information is consistent to the ones adopted by the compiler. If some fields cannot be displayed at all or cannot be displayed completely, verifiers should at least guarantee the displayed information is unambiguous.

VII. Evaluation

A. Experimental Setup & Ethical Considerations

In this work, we target the most popular Ethereum smart contract verification services, including Etherscan, Sourcify, and Blockscout. For all these three verifiers, they provide the verification services on both Ethereum mainnet and testnet. For ethical considerations, all evaluations are conducted on the *goerli* testnet, which is independent of the Ethereum mainnet.

We have carefully designed methods to detect the exploitable vulnerabilities. Specifically, for the $\mathcal{A}_1$-related vulnerabilities, i.e., $\mathcal{R}_1$ to $\mathcal{R}_3$, we randomly sample 10 widely-used contracts with source code from mainnet as victim contracts, and deploy them on the testnet. After that, according to the victim's bytecode, we construct PoCs to try to pass the verification. Note that a successful verification does not imply the $\mathcal{A}_1$ consequence, as we need to ensure that the forged source code cannot be replaced. Therefore, we further request verifications against these victim contracts with their original source code. If the re-verification fails, we conclude that the testing verifier is exploitable. For the $\mathcal{A}_2$-related vulnerabilities, we firstly construct some contracts with backdoors, like conducting `selfdestruct` or transferring to other users, compile and deploy them on the testnet. Then, according to PoCs raised from $\mathcal{R}_4$ to $\mathcal{R}_8$, we try to construct source code without such malicious behaviors, and bypass the source code verification. If it succeeds and no warnings raise, we can conclude that the verifier is exploitable in terms of the corresponding vulnerability.

TABLE III: Overall vulnerability detection results. For each vulnerability, – and ✗ refer to infeasible (safe) and exploitable, respectively. * refers to the official teams have confirmed our reported vulnerabilities, and ✓ indicates it has been patched after our timely disclosure.

Consequence	Exploitable Vulnerailities	Etherscan	Sourcify	Blockscout
$\mathcal{A}_1$	$\mathcal{R}_1$	✗	✗*	✗*
	$\mathcal{R}_2$	–	✗* (✓)	✗* (✓)
	$\mathcal{R}_3$	–	✗* (✓)	✗* (✓)
$\mathcal{A}_2$	$\mathcal{R}_4$	✗	✗*	✗*
	$\mathcal{R}_5$	✗	✗*	✗* (✓)
	$\mathcal{R}_6$	–	✗* (✓)	✗* (✓)
	$\mathcal{R}_7$	–	✗* (✓)	✗* (✓)
	$\mathcal{R}_8$	✗	–	✗* (✓)

Finally, after confirming the exploitability of vulnerabilities, we conduct timely disclosure to impacted verifiers within 30 minutes. On the one hand, it benefits a timely fixup on exploitable vulnerabilities. On the other hand, it gives little chance for malicious users to replay exploitations by observing our uploaded source code files and initiated transactions.

B. Detecting Results

Overall Result. Table III presents the overall results. Surprisingly, all these popular verifiers are vulnerable, which can be abused by attackers. In total, we have uncovered 19 exploitable vulnerabilities. Etherscan is vulnerable to 4 kinds of vulnerabilities, while Sourcify poses the risks of 7 types of vulnerabilities, and Blockscout can be exploited by all types of vulnerabilities.

Further Exploration. As Sourcify and Blockscout are vulnerable to most kinds of vulnerabilities, we next deep dive into them. For Sourcify, some of its designs for user-friendliness introduce security issues. For example, Sourcify does not require users to provide the addresses of invoked linked libraries, it tries to replace placeholders according to a regex pattern, which is also embedded in the source code. Such an adoption of untrusted data harms its initial goodwill on usability. For Blockscout, it is vulnerable to all kinds of attacks. Recall the shortcut we mentioned in §IV-C, which can be seen as an amplifier to expand the exploitable scope of PoCs. For example, if a vulnerability is exploitable in Sourcify, we also consider it can be exploited in Blockscout because the latter one recognizes the results of the former one. Through the shortcut, Blockscout inherits the malicious verification results of exploiting $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$ and $\mathcal{R}_7$ from Sourcify. But then again, the transparency of Blockscout and Sourcify enables a more insightful and timely security analysis.

Vulnerability Patching. We disclose the vulnerabilities to verifiers timely. Among the 19 uncovered vulnerabilities, 15 of them have been confirmed by the official teams, and 10 vulnerabilities have been fixed. For Sourcify and Blockscout, we observed that the vulnerabilities of $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_6$, and $\mathcal{R}_7$ were fixed within 12 hours. However, as suggested in Table III, we find that four kinds of vulnerabilities are almost exploitable in all verifiers by the time of this writing ($\mathcal{R}_1$, $\mathcal{R}_4$, $\mathcal{R}_5$, and

TABLE IV: The statistic of affected contracts.

Consequence	Exploitable Vulnerailities	# Afftected Contracts
$\mathcal{A}_1$	$\mathcal{R}_1$	49,598
	$\mathcal{R}_2$	partial verified / unverified contracts (∼58.9M)
	$\mathcal{R}_3$	unverified contracts (∼58.9M)
$\mathcal{A}_2$	$\mathcal{R}_4$	2
	$\mathcal{R}_5$	244

$\mathcal{R}_8$). By communicating with the official teams, we summarize the following challenges. Specifically, as for $\mathcal{R}_1$, loose inline assembly is a necessary feature that is widely adopted by normal contracts to improve its runtime efficiency [44]. Directly disabling it shall affect the usability of verifiers. As for the other three kinds of vulnerabilities, the verification services thought that users should be aware of the scams, e.g., paying attentions to whether a smart contract invokes a malicious linked library or has a suspicious accompanying file with identical name. Thus, they are considering adding explicit warnings for users in recent updates.

C. *Contracts Affected by $\mathcal{A}_1$-related Vulnerabilities*

1) Method: Among all three $\mathcal{A}_1$-related vulnerabilities, exploiting $\mathcal{R}_2$ and $\mathcal{R}_3$ have no prerequisites. Thus the number of influenced smart contracts are all smart contracts that are not exactly matched yet and unverified, respectively. As for $\mathcal{R}_1$, due to the restriction we mentioned in §VI-A1, we compose a simple and effective SQL query statement to filter out all possible victim contracts. It consists of three sets of `like` operators based on the patterns we summarized. Specifically, first, to narrow down the scope to all contracts compiled from solc with `0.4` version, we heuristically use the magic number `a165627a7a7230`, which is located in metadata that is used by solc `0.4`, as the pattern. Then, we restrict the operand of each `PUSH` operators has no leading zeros. Last, because the exploit can only be conducted on contracts with at most a single piece of metadata, we filter the contract with multiple pieces of metadata out by adopting the magic number `0029a165`.

2) Result: As shown in Table IV, We finally identify more than 49K contracts that are influenced by $\mathcal{R}_1$. Among them, by identifying the function signatures, 25K and 2,440 contracts are suspected to follow ERC20 and ERC721, respectively, which may lead to huge financial and ecological impact if they are adversarially verified. In addition, all unverified solidity contracts in Sourcify, around 58.9M, are affected by $\mathcal{R}_2$ and $\mathcal{R}_3$, where the former one even includes the partial verified ones (around to 9K contracts). Moreover, as we clarified in §I, lots of downstream applications rely on source code verification services. If millions of contracts can be adversarially verified by forged smart contracts, the effectiveness and even safety of these applications are severely impacted. We illustrate a case study in our artifact [1], where Slither [28], a well-known vulnerability detector on Ethereum smart contracts on source-code level, is exploited by a piece of forged source code, resulting in file overwriting in its hosting environment.

D. *Contracts Affected by $\mathcal{A}_2$-related Vulnerabilities*

Five types of vulnerabilities are related to $\mathcal{A}_2$. Therefore, we try to measure if there exists any contracts that have already performed source scam on users.

1) Method: To the best of our knowledge, among all these five types of vulnerabilities, $\mathcal{R}_6$ to $\mathcal{R}_8$ are zero-day vulnerabilities uncovered by us. No existing instances are found on Ethereum mainnet. Thus, we mainly focus on the existence of instances that exploit $\mathcal{R}_4$ and $\mathcal{R}_5$. Based on tintinweb [42], a dataset consisting of all verified contracts on Ethereum, we performed a series of rule-based detection to filter such contracts. Specifically, as for identifying contracts abusing $\mathcal{R}_4$, for each verified contract, we iterate its transactions to identify if there are more than one successful contract deployment. To identify contracts abusing $\mathcal{R}_5$, we first filter out all verified contracts that have a call relationship to linked libraries. Then, we perform a difference checking between the provided linked library and the one invoked. Consequently, we can identify if attackers uploaded a forged linked library.

2) Result: As shown in Table IV, we finally find 2 and 244 contracts whose developers are suspected to abuse $\mathcal{R}_4$ and $\mathcal{R}_5$ to commit fraud. For the 2 cases under $\mathcal{R}_4$, after a manual recheck, we confirm that one case has performed a fake proposal attack against Tornado.cash [69]. Specifically, by exploiting $\mathcal{R}_4$, the attackers associate a piece of source code that has gained the user's trust to a malicious proposal. The proposal ultimately resulted in a \$750K financial loss for Tornado.cash [48]. As for the 244 cases that are related to $\mathcal{R}_5$, we confirm that at least 6 of them invoked malicious libraries but uploaded with benign ones. For example, there is an attack targeting the Saddle Finance [47], a well-known decentralized exchange. Interestingly, its actually invoked linked library is a deprecated version, which has a vulnerability in the price calculation that can be utilized by malicious users to exchange a small number of token for another bunch of token at a rate deviated from the market. However, the developer of Saddle Finance has uploaded the updated and bug-free version of the invoked library, which covers the vulnerability hidden in the library. Eventually, this vulnerability has been exploited and led to a financial loss of \$7.2 million.

VIII. Related Work

Blockchain Security. Currently, a great deal of research work is focusing on the security of the whole blockchain ecosystem [45], [10], [37], [35], [34], [71], [75], [4], [13], [12]. While lots of them are delving into the security risks of various modules in the blockchain ecosystem, there is still a void when targeting source code verification services.

Ethereum Scam. In recent years, with the increasing financial value embedded in the Ethereum contracts, the research on the scam in the Ethereum ecosystem has become a hot spot [73], [74], [11], [65], [14], [31], [72]. For example, Chen et al. [11] propose a heuristic-guided symbolic execution technique to identify Ponzi scheme contracts in Ethereum. However, there is little discussion about if scams can be performed by a mismatched source code and deployed bytecode.

Ethereum Smart Contract Analysis. In terms of Solidity contract security, there is a lot of work [32], [19], [40], [28],

[29], [64], [9], [3], [33], [39] focus on contract vulnerability detection based on source code. For instance, Feist et al. [28] build a static analysis framework for solidity source code, called Slither, to provide support for bug detection in Ethereum smart contracts. However, all of them assume the fetched source code files are correct and not consider the risks brought by source code verifiers.

IX. Threats to Validity

We have identified three threats to validity in this work. First, there might be more kinds of vulnerabilities related to smart contract verification services, which are not covered in this work. Etherscan is close-sourced and we can only conduct a black-box analysis against it. Nevertheless, we argue that we have comprehensively performed a code audit on Blockscout and Sourcify against the intuitive security properties, and summarized the vulnerabilities related to almost all the modules of verifiers. We believe this work can remind developers and security researchers of this previously overlooked direction. Second, beyond the detection principles for vulnerabilities, we also illustrate the corresponding PoCs for them. However, we need to mention that these exploitations are not the only way to tamper verifiers. For example, in addition to adopting `return` in YUL to exploit the simulating mechanism in $\mathcal{R}_2$, attackers could also utilize the jump statement in loose inline assembly [76]. Moreover, exploits can even be combined to perform a more covert attack. In a nutshell, we have presented these vulnerabilities in the most understandable way, but developers should never take the risks hidden in verifiers lightly. Last, we have considered the three mainstream verifiers in Ethereum in this work, and identified several exploitable vulnerabilities. There do exist other verifiers in Ethereum, e.g., Tenderly [63]. We argue that our detecting principles can be directly applied to them, and even verifiers on other EVM blockchain platforms.

X. Conclusion

We explored the security issues of smart contract verification services in this work, an uncharted direction of the Ethereum ecosystem. We have depicted the design and implementation of smart contract verification services, summarized their security properties, observed eight kinds of vulnerabilities, and proposed effective detection and attack methods. Our exploration has uncovered 19 exploitable vulnerabilities in popular smart contract verification services, posing a great impact to millions of smart contracts in the ecosystem. Our results encourage our research community to invest more efforts into the under-studied directions.

Acknowledgement

We would like to express our sincere thanks for all anonymous reviewers and the shepherd for their valuable suggestions on improving this paper. Additionally, we really appreciate the official teams of Blockscout and Sourcify for their timely responding, and samczsun for his insightful comments on this paper. This work was supported in part by National Key R&D Program of China (2021YFB2701000), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the Knowledge Innovation Program of Wuhan-Basic Research, HUST CSE-HongXin Joint Institute for Cyber Security, Hong Kong RGC Project (No. PolyU15224121) and research grants from Huawei.

References

[1] "Github repository for displaying each attack," june 2023. [Online]. Available: https://github.com/source-code-scam-paper/source-scam-all-in-one/

[2] "Sourcify—the ethereum source code verifier," june 2023. [Online]. Available: https://sourcify.dev/

[3] S. Akca, A. Rajan, and C. Peng, "Solanalyser: A framework for analysing and testing smart contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 482–489.

[4] A. Biryukov and S. Tikhomirov, "Security and privacy of mobile wallet users in bitcoin, dash, monero, and zcash," *Pervasive and Mobile Computing*, vol. 59, p. 101030, 2019.

[5] Blockscout, "Blockscout—implementation of differential extraction metadata," 2023. [Online]. Available: https://github.com/blockscout/blockscout-rs/blob/stats/v1.0.0/smart-contract-verifier/smart-contract-verifier/src/verifier/contract_verifier.rs#L95-L121

[6] ——, "Blockscout—the ethereum explorer," May 2023. [Online]. Available: https://eth.blockscout.com/stats

[7] ——, "Implementation of blockscout for differential metadata extraction," 2023. [Online]. Available: https://github.com/blockscout/blockscout-rs/blob/stats/v1.0.0/smart-contract-verifier/smart-contract-verifier/src/verifier/contract_verifier.rs#L100-L121

[8] BlockSec, "Smart contract audit service," june 2023. [Online]. Available: https://blocksec.com/audit

[9] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He *et al.*, "Soda: A generic online detection framework for smart contracts." in *NDSS*, 2020.

[10] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks," in *Information Security Practice and Experience: 13th International Conference, ISPEC 2017, Melbourne, VIC, Australia, December 13–15, 2017, Proceedings 13*. Springer, 2017, pp. 3–24.

[11] W. Chen, X. Li, Y. Sui, N. He, H. Wang, L. Wu, and X. Luo, "Sadponzi: Detecting and characterizing ponzi schemes in ethereum smart contracts," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 2, pp. 1–30, 2021.

[12] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 703–715.

[13] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1186–1201.

[14] Z. Cheng, X. Hou, R. Li, Y. Zhou, X. Luo, J. Li, and K. Ren, "Towards a first step to understand the cryptocurrency stealing attack on ethereum." in *RAID*, vol. 2019, 2019, pp. 47–60.

[15] Circle, "Circle— reorgs and associated risks," 2023. [Online]. Available: https://developers.circle.com/developer/reference/confirmations

[16] Coinmarketcap, "Coinmarketcap—market capitalization of ethereum," 2023. [Online]. Available: https://coinmarketcap.com/currencies/ethereum/

[17] J. Cvllr, "Solidity tutorial: all about libraries," 2023. [Online]. Available: https://jeancvllr.medium.com/solidity-tutorial-all-about-libraries-762e5a3692f9

[18] DappReview, "26 million trx stolen behind the rashomon," 2023. [Online]. Available: https://blocking.net/3832/26-million-trx-stolen-behind-the-rashomon-episode-2/

[19] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[20] Eip-721, "Non-fungible token standard," june 2023. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[21] Ethereum, "Constantinople, where create2 was added," 2023. [Online]. Available: https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement

[22] ——, "Decentralized autonomous organizations," 2023. [Online]. Available: https://ethereum.org/en/dao/#dao-governance

[23] ethereum, "Evm opcode introduction." june 2023. [Online]. Available: https://ethereum.org/en/developers/docs/evm/opcodes/

[24] Ethereum-Magicians, "Potential security implications of create2 eip 1014," 2023. [Online]. Available: https://ethereum-magicians.org/t/potential-security-implications-of-create2-eip-1014/2614/2

[25] etherscan, "Ethereum daily verified contracts chart," 2023. [Online]. Available: https://etherscan.io/chart/verified-contracts

[26] Etherscan, "Etherscan—the ethereum blockchain explorer," May 2023. [Online]. Available: https://etherscan.io

[27] ——, "Forked blocks excluded blocks as a result of chain reorganizations," 2023. [Online]. Available: https://etherscan.io/blocks_forked

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

[29] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[30] M. Flanders, "A simple and intuitive algorithm for preventing directory traversal attacks," *arXiv preprint arXiv:1908.04502*, 2019.

[31] B. Gao, H. Wang, P. Xia, S. Wu, Y. Zhou, X. Luo, and G. Tyson, "Tracking counterfeit cryptocurrency end-to-end," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–28, 2020.

[32] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[33] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*. Springer, 2020, pp. 161–179.

[34] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "Eosafe: Security analysis of eosio smart contracts." in *USENIX Security Symposium*, 2021, pp. 1271–1288.

[35] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 129–144.

[36] IPFS, "Ipfs—the decentralised file system," May 2023. [Online]. Available: https://docs.ipfs.tech

[37] K. Li, J. Chen, X. Liu, Y. R. Tang, X. Wang, and X. Luo, "As strong as its weakest link: How to break blockchain dapps at rpc service." in *NDSS*, 2021.

[38] Z. Liao, S. Song, H. Zhu, X. Luo, Z. He, R. Jiang, T. Chen, J. Chen, T. Zhang, and X. Zhang, "Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 777–801, 2022.

[39] S.-W. Lin, P. Tolmach, Y. Liu, and Y. Li, "Solsee: a source-level symbolic execution engine for solidity," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1687–1691.

[40] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[41] M. Neuder, D. J. Moroz, R. Rao, and D. C. Parkes, "Low-cost attacks on ethereum 2.0 by sub-1/3 stakeholders," *arXiv preprint arXiv:2102.02247*, 2021.

[42] M. Ortner and S. Eskandari, "Smart contract sanctuary," june 2023. [Online]. Available: https://github.com/tintinweb/smart-contract-sanctuary

[43] owasp, "owasp—path traversal," 2023. [Online]. Available: https://owasp.org/www-community/attacks/Path_Traversal

[44] R. Park, "Inline assembly, good or bad practice," 2023. [Online]. Available: https://ethereum.stackexchange.com/questions/72895/inline-assembly-good-or-bad-practice

[45] Z. Peng and Y. Chen, "All roads lead to rome: Many ways to double spend your cryptocurrency," *arXiv preprint arXiv:1811.06751*, 2018.

[46] Rayne, "Github—virtual path," 2023. [Online]. Available: https://github.com/Rayne/virtual-path

[47] REKT, "Saddle finance attack," june 2023. [Online]. Available: https://rekt.news/saddle-finance-rekt2/

[48] rekt, "Tornado cash governance - rekt," 2023. [Online]. Available: https://rekt.news/tornado-gov-rekt/

[49] Samczsun, "Hiding in plain sight," May 2023. [Online]. Available: https://www.paradigm.xyz/2021/11/hiding-in-plain-sight

[50] Solidity, "Abstract contracts," 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.20/contracts.html#abstract-contracts

[51] ——, "Allow verbatim in solidity assembly blocks," june 2023. [Online]. Available: https://github.com/ethereum/solidity/issues/12067

[52] ——, "Function selector," 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.20/abi-spec.html#function-selector

[53] ——, "Return back loose assembly and forbid optimizer to touch its output," june 2023. [Online]. Available: https://github.com/ethereum/solidity/issues/6517

[54] ——, "Solidity—the solidity doc," june 2023. [Online]. Available: https://solidity.readthedocs.io/

[55] ——, "Solidity documentation—contract metadata," 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.19/metadata.html

[56] ——, "Yul — an intermediate language in solc," june 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.20/yul.html#verbatim

[57] Sourcify, "Sourcify— matchwithsimulation implementation in sourcify," 2023. [Online]. Available: https://github.com/ethereum/sourcify/blob/v2.0.0/packages/lib-sourcify/src/lib/verification.ts#L186-L244

[58] ——, "Sourcify—full vs partial match," 2023. [Online]. Available: https://docs.sourcify.dev/docs/full-vs-partial-match/

[59] ——, "Sourcify—prefix checking implementation in sourcify," 2023. [Online]. Available: https://github.com/ethereum/sourcify/blob/v2.1.1/packages/lib-sourcify/src/lib/verification.ts#L287

[60] ——, "Sourcify—replace immutable references," 2023. [Online]. Available: https://github.com/ethereum/sourcify/blob/v2.6.0/packages/lib-sourcify/src/lib/verification.ts#L273-L277

[61] ——, "Sourcify—verify deployed contract in sourcify," 2023. [Online]. Available: https://github.com/ethereum/sourcify/blob/v2.6.0/packages/lib-sourcify/src/lib/verification.ts#L71-L158

[62] M. Stephenson, "Medium—ethereum, fomo3d, and dangerous game theory," 2023. [Online]. Available: https://medium.com/hackernoon/fomo3d-and-dangerous-game-theory-97bd5f47ab3b

[63] Tenderly, "Smart contract verification in tenderly," june 2023. [Online]. Available: https://docs.tenderly.co/monitoring/smart-contract-verification

[64] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[65] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," *arXiv preprint arXiv:1902.06976*, 2019.

[66] Uniswap, "Uniswap—the ethereum decentralised exchange," june 2023. [Online]. Available: https://github.com/Uniswap

[67] Wiki, "Wiki—know your customer," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Know_your_customer

[68] ——, "Wiki—preimage attack," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Preimage_attack

[69] Wikipedia, "Tornado cash," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Tornado_Cash

[70] wissal haji, “Learn solidity: The factory pattern,” 2023. [Online]. Available: https://betterprogramming.pub/learn-solidity-the-factory-pattern-75d11c3e7d29

[71] K. Wüst and A. Gervais, “Ethereum eclipse attacks,” ETH Zurich, Tech. Rep., 2016.

[72] P. Xia, H. Wang, B. Gao, W. Su, Z. Yu, X. Luo, C. Zhang, X. Xiao, and G. Xu, “Trade or trick? detecting and characterizing scam tokens on uniswap decentralized exchange,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 3, pp. 1–26, 2021.

[73] P. Xia, H. Wang, Z. Yu, X. Liu, X. Luo, G. Xu, and G. Tyson, “Challenges in decentralized name management: the case of ens,” in *Proceedings of the 22nd ACM Internet Measurement Conference*, 2022, pp. 65–82.

[74] P. Xia, H. Wang, B. Zhang, R. Ji, B. Gao, L. Wu, X. Luo, and G. Xu, “Characterizing cryptocurrency exchange scams,” *Computers & Security*, vol. 98, p. 101993, 2020.

[75] Y. Yang, T. Kim, and B.-G. Chun, “Finding consensus bugs in ethereum via multi-transaction differential fuzzing.” in *OSDI*, 2021, pp. 349–365.

[76] X. Yu, “Evm opcode jop,” 2023. [Online]. Available: https://github.com/xhyumiracle/defcon27/