Published: 19 June 2025
Accepted: 14 January 2025
Received: 12 September 2024

RESEARCH-ARTICLE

Automated and Accurate Token Transfer Identification and Its Applications in Cryptocurrency Security

SHUWEI SONG, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

TING CHEN, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

AO QIAO, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

XIAPU LUO, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

LEQING WANG, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

ZHEYUAN HE, University of Electronic Science and Technology of China, Chengdu, Sichuan, China

View all

Automated and Accurate Token Transfer Identification and Its Applications in Cryptocurrency Security

SHUWEI SONG, University of Electronic Science and Technology of China, China and The Hong Kong Polytechnic University, China
TING CHEN*, University of Electronic Science and Technology of China, China
AO QIAO, University of Electronic Science and Technology of China, China
XIAPU LUO*, The Hong Kong Polytechnic University, China
LEQING WANG, University of Electronic Science and Technology of China, China
ZHEYUAN HE, University of Electronic Science and Technology of China, China
TING WANG, Stony Brook University, USA
XIAODONG LIN, University of Guelph, Canada
PENG HE, University of Electronic Science and Technology of China, China
WENSHENG ZHANG, University of Electronic Science and Technology of China, China
XIAOSONG ZHANG, University of Electronic Science and Technology of China, China

Cryptocurrency tokens, implemented by smart contracts, are prime targets for attackers due to their substantial monetary value. To illicitly gain profit, attackers often embed malicious code or exploit vulnerabilities within token contracts. Token transfer identification is crucial for detecting malicious and vulnerable token contracts. However, existing methods suffer from high false positives or false negatives due to invalid assumptions or reliance on limited patterns. This paper introduces a novel approach that captures the essential principles of token contracts, which are independent of programming languages and token standards, and presents a new tool, CRYPTO-SCOUT. CRYPTO-SCOUT automatically and accurately identifies token transfers, enabling the detection of various malicious and vulnerable token contracts. CRYPTO-SCOUT's core innovation is its capability to automatically identify complex container-type variables used by token contracts for storing holder information. It processes the bytecode of smart contracts written in the two most widely-used languages, Solidity and Vyper, and supports the three most popular token standards, ERC20, ERC721, and ERC1155. Furthermore, CRYPTO-SCOUT detects four types of malicious and vulnerable token contracts and is designed to be extensible. Extensive experiments show that CRYPTO-SCOUT outperforms existing approaches and uncovers over 21,000 malicious/vulnerable token contracts and more than 12,000 transactions triggering them.

CCS Concepts: • **Security and privacy** → **Software and application security**.

*Corresponding authors.

Authors' Contact Information: Shuwei Song, University of Electronic Science and Technology of China, Chengdu, China and The Hong Kong Polytechnic University, Hong Kong, China, shuwei@std.uestc.edu.cn; Ting Chen, University of Electronic Science and Technology of China, Chengdu, China, brokendragon@uestc.edu.cn; Ao Qiao, University of Electronic Science and Technology of China, Chengdu, China, 202222080625@std.uestc.edu.cn; Xiapu Luo, The Hong Kong Polytechnic University, Hong Kong, China, csxluo@comp.polyu.edu.hk; Leqing Wang, University of Electronic Science and Technology of China, Chengdu, China, 202222080626@std.uestc.edu.cn; Zheyuan He, University of Electronic Science and Technology of China, Chengdu, China, 202211081309@std.uestc.edu.cn; Ting Wang, Stony Brook University, New York, USA, twang@cs.stonybrook.edu; Xiaodong Lin, University of Guelph, Guelph, Canada, xlin08@uoguelph.ca; Peng He, University of Electronic Science and Technology of China, Chengdu, China, 201822080439@std.uestc.edu.cn; Wensheng Zhang, University of Electronic Science and Technology of China, Chengdu, China, 201821080421@std.uestc.edu.cn; Xiaosong Zhang, University of Electronic Science and Technology of China, Chengdu, China, johnsonzxs@uestc.edu.cn.

ACM 2994-970X/2025/7-ARTFSE049
https://doi.org/10.1145/3715766

Additional Key Words and Phrases: Cryptocurrency Tokens, Token Transfer, Smart Contract

ACM Reference Format:
Shuwei Song, Ting Chen, Ao Qiao, Xiapu Luo, Leqing Wang, Zheyuan He, Ting Wang, Xiaodong Lin, Peng He, Wensheng Zhang, and Xiaosong Zhang. 2025. Automated and Accurate Token Transfer Identification and Its Applications in Cryptocurrency Security. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE049 (July 2025), 23 pages. https://doi.org/10.1145/3715766

1 Introduction

The aggregate market capitalization of cryptocurrencies has recently surpassed $2.55 trillion [10]. A few cryptocurrencies, such as ETH [17], are native to the blockchain and issued by it. In contrast, most cryptocurrencies, also referred to as *tokens*, are implemented by smart contracts, which are programs that run on the blockchain. In other words, the properties and activities of the tokens are managed by the code in the smart contract (a.k.a. token contract). Since Ethereum is the blockchain hosting the majority of token contracts, we focus on the tokens on Ethereum.

Various token contracts exhibit different characteristics but share two crucial functions: maintaining holders' balances and facilitating token transfers. The former is straightforward, as tokens typically have numerous holders, each with their own balance. For example, more than five million holders hold USDT [22]. The latter, token transfers, represents the movement of money and value. When a holder initiates a transfer request via the token contract's interface, the contract executes the transfer by updating the balances of the sender and receiver. We use the term *token transfer identification* to refer to the process of identifying how a token contract maintains holders' balances and conducts token transfers. Since token transfers reflect the flow of funds, their identification is crucial for numerous applications in cryptocurrency security [5, 27, 51]. For instance, it can facilitate the detection of malicious token contracts that cause financial loss, such as transferring fewer tokens than expected (§3) or generating fake notifications without actually transferring tokens (§6.2), among other issues. As another example, it can be leveraged to uncover the exploitable vulnerabilities in token contracts, such as no exception being thrown when a token transfer fails (§6.1), and more tokens than authorized are used from the holder's account (§6.4), to name a few.

Unfortunately, achieving automated and accurate token transfer identification is challenging. Current techniques fall into two categories. One considers the invocation/emission of standard interfaces/events as indications of token transfers [13, 26, 27], assuming that token contracts adhere strictly to the standards (e.g., ERC20 [47]). However, this assumption is not universally valid [7], leading to errors in token transfer identification. Another category of methods leverages the observation that token contracts store information about token holders (e.g., IDs, shares) in container-type variables (e.g., mapping) [7, 30]. Specifically, they first identify such variables, denoted as $\mathfrak{R}$s, and then monitor the manipulations of $\mathfrak{R}$s to capture transfers. While this concept is promising, designing and developing a practical solution is non-trivial due to two challenges:
C1: Identifying $\mathfrak{R}$s from smart contract bytecode is difficult because the bytecode lacks variable type information, and the types of $\mathfrak{R}$s are not standardized. Developers can store information about token holders in any data structure rather than being restricted to specific variable types.
C2: Recognizing token transfers effectively and efficiently is not straightforward due to several factors: the diversity of programming languages, which introduces unique bytecode characteristics that complicate the identification of $\mathfrak{R}$s, the complexity of handling inter-contract invocations, and the different interfaces introduced by various token standards to manipulate $\mathfrak{R}$s.

None of the existing methods can tackle these challenges. Instead, they rely on simple or a few manually-defined patterns to identify $\mathfrak{R}$s or only focus on specific languages and token standards, thus suffering from false positives (FPs) and false negatives (FNs). For instance, TokenScope relies on manually-defined patterns extracted from the bytecode of ERC20 contracts developed in Solidity [7].

As a result, it cannot recognize token contracts developed in other languages, $\mathfrak{R}$s with unknown patterns, and contracts following other token standards. Fröwis et al. use a simple pattern (i.e., two storage writes in a program path) to recognize ERC20 token contracts [26]. This approach suffers from high FNs and FPs due to its oversimplified pattern. TokenAware can recognize more types of $\mathfrak{R}$s [30], but its limitations include support only for Solidity, restriction to ERC20, and the inability to analyze undeployed contracts. §7 details the differences between our work and others.

In this paper, we propose a novel approach to automatically and accurately identify token transfers, effectively addressing the technical challenges previously mentioned. We develop a new and extensible tool, CRYPTO-SCOUT, and equip it with four plugins to detect malicious and vulnerable token contracts. It is worth noting that CRYPTO-SCOUT operates in two modes: (1) analyzing token contract bytecode before deployment and (2) examining transactions that invoke deployed contracts.

To tackle **C1**, we propose a novel approach that first learns the *accessing patterns* of basic container types and then identifies how $\mathfrak{R}$s are constructed according to the instructions used to access them (§4). Here, the *accessing pattern* of a basic container is defined as a sequence of instructions for computing the location of each item in the container.

To address **C2**, we first develop an inter-contract symbolic analysis (ICSA) approach to support the identification of token transfers involving multiple contract interactions (§4.3). Then, by inspecting the bytecode generated from the two most popular smart contract programming languages (i.e., Solidity [43] and Vyper [44]), we identify and address the unique issues in their bytecode (§4). Moreover, to handle different token standards, we exploit a fundamental principle of token transfers that remain stable despite variations in token standards: Within a transfer, the balances of two accounts (i.e., the sender and receiver) should be modified, and the standard interface/event should be invoked/emitted. CRYPTO-SCOUT currently supports the three most prevalent standards that are recognized by the community [36]: ERC20 for fungible tokens, as well as ERC721 and ERC1155 for non-fungible tokens [48]. Furthermore, since the capability of automated and accurate token transfer identification can empower many applications, CRYPTO-SCOUT provides interfaces for users to develop plugins to achieve various purposes. To illustrate, we develop four plugins to detect two types of malicious token contracts and two types of vulnerabilities in token contracts (§6).

Our evaluation (§5) shows that CRYPTO-SCOUT achieves high accuracy with no FP and only one (0.06%) FN while maintaining efficiency with an average analysis time of 1.75 seconds per smart contract. It discovers ten accessing patterns of $\mathfrak{R}$s, including *five* that state-of-the-art tools cannot recognize. Furthermore, the experimental results indicate that 6.4% of the token contracts implement ERC721 or ERC1155 standards, and 4.5% of the token contracts rely on inter-contract calls to execute transfers. Therefore, our design choices that support multiple token standards and inter-contract analysis are beneficial. In summary, we make the following contributions.

• We propose a novel approach that can automatically and accurately recognize the core variables (i.e., $\mathfrak{R}$s) in token contracts, which are used to store the information of token holders, and identify token transfers by monitoring the manipulations of $\mathfrak{R}$s.
• We develop CRYPTO-SCOUT, a prototype of our approach that supports the two most widely-used smart contract languages and the three most popular token standards. We also develop four plugins for CRYPTO-SCOUT to detect malicious token contracts and vulnerabilities in token contracts.
• We carry out extensive experiments to evaluate CRYPTO-SCOUT. The experimental results demonstrate its effectiveness and efficiency. Moreover, our plugins uncover over 21K token contracts with malicious code or vulnerabilities and more than 12K transactions triggering them in the dataset covering 13.7 million blocks (spanning 76 months).

2 Background

Ethereum has two kinds of accounts: smart contract and externally-owned account (EOA) [17]. Only the former contains the bytecode of smart contracts compiled from their source code, usually developed in high-level languages like Solidity [43] and Vyper [44]. Once deployed and invoked, the Ethereum virtual machine (EVM) executes the bytecode of smart contracts.

Transactions are used to transfer ETH (the native cryptocurrency in Ethereum, which is not a token), deploy and invoke smart contracts. Transactions include external transactions that are sent by EOAs and internal transactions that are sent by smart contracts. Only external transactions are recorded in blocks. The transaction invoking a smart contract should specify the callee's address, the ID of the invoked function, and arguments. When a smart contract invokes a function in the same contract, this intra-contract invocation is compiled into a JUMP instruction [55]. In contrast, inter-contract invocations are compiled into internal transactions.

There are many token standards (e.g., ERC20[47], ERC721[16], ERC1155[41]), which define standard interfaces and events and their semantics. Besides implementing these interfaces and emitting the events, token contracts may include customized functions and events. Although token standards differ, transferring tokens from one holder to another is a typical essential operation. We define a *token transfer* as a set of tuples $<addr, \Delta>$, where *addr* is the identifier (i.e., address) of a holder and Δ denotes the changed token amount. If $\Delta > 0$, the holder receives tokens.

Since a kind of token may be held by many holders who would join or leave, token contracts use container-type variables to record holders' information, including holders' identifiers and the number of tokens they possess. As token transfers are the outcomes of manipulating such variables, we call them core variables and use $\mathfrak{R}$s to denote them. For example, given a token contract that uses an $\mathfrak{R}$ to record the address and the number of tokens of each account, to transfer Δ tokens from one account to another account, the token contract will first locate the items in $\mathfrak{R}$ representing these two accounts according to their addresses and then change their balances by Δ respectively.

Ethereum provides three memory spaces: *stack* is used for temporarily keeping the operands and results of instructions [17]; *memory* is a temporary space for storing some local variable, arguments and return values [17]; *storage* is a permanent space for recording <key, value> pairs [17]. $\mathfrak{R}$s must be recorded in storage because all modifications of it, which result in token transfers, should be persistent. Note that SSTORE (SLOAD) is the only instruction that can write (read) storage.

3 A Motivating Example

```solidity
function transfer(address sender, address recipient, uint256 amount) {
  ......
  uint256 senderBalance = _balances[sender];
  if (sender == ownerA || sender == ownerB) {
    _balances[sender] = senderBalance - amount;
    _balances[recipient] += amount;
  } else {
    _balances[sender] = senderBalance - amount;
    uint256 trapAmount = (amount * 10) / 100;
    _balances[recipient] += trapAmount;
  }
  emit Transfer(sender, recipient, amount);
}
```

Fig. 1. The transfer function of a Salmonella ERC20 contract

The sandwich attack is a prevalent arbitrage strategy wherein an attacker exploits a pending victim transaction by placing front-run and back-run transactions around it [1]. In this scenario, the front-run transaction inflates the price of the target token, forcing the victim to purchase at an inflated price, while the back-run transaction capitalizes on the price increase, allowing the attacker to profit. However, sandwich attacks are not without risk. The Salmonella ERC20 token contract [4] is designed to exploit and steal funds from sandwich attackers. As shown in Fig. 1, the Salmonella contract performs transfers after determining if the sender is in the whitelist. If so, tokens are transferred; otherwise, only 10% of tokens are transferred. In both cases, a standard event that matches the full amount is emitted. The Salmonella contract developer steals money through the following steps. First, it submits a transaction to exchange ETH for Salmonella tokens. The sandwich attacker notices this transaction and then submits a front-run transaction, also exchanging ETH for Salmonella tokens. However, the sandwich attacker is not on the whitelist, so it only gets 10% of the expected number. Then, the sandwich attacker can only exchange these Salmonella tokens for a minimal amount of ETH in the back-run transaction and thus loses money. Consequently, the developer stole 103 ETH from the sandwich attacker within a few hours [4].

CRYPTO-SCOUT can determine whether a contract is a Salmonella ERC20 contract by the following steps. First, given the bytecode of a target contract, CRYPTO-SCOUT identifies the core variables used to store the information of token holders (i.e., *_balances*) by searching the accessing patterns of container-type data structures. CRYPTO-SCOUT then explore all possible execution paths of the contract. For each path, CRYPTO-SCOUT identifies token transfers by monitoring the manipulations of *_balances*. CRYPTO-SCOUT marks contracts as Salmonella if both paths exist in the contract: (1) the standard interface is executed, the standard event is emitted, and the tokens are transferred normally; (2) the standard interface is executed, the standard event is emitted, but the actual number of tokens transferred is smaller. Through the steps outlined above, we identified 16 cases of Salmonella contracts. The list can be found at https://github.com/xxki-workstation/CRYPTO-SCOUT.

4 CRYPTO-SCOUT

4.1 Overview

Anomalies in token transfers often signal underlying issues in token contracts. Therefore, accurate identification of token transfers is critical for detecting malicious and vulnerable token contracts. Token transfers can be inferred from modifications of $\mathfrak{R}$s. However, it is challenging to automatically identify $\mathfrak{R}$s and their modifications without source code because the variable types of $\mathfrak{R}$s are not standardized, and bytecode lacks type information. Fortunately, we observe that $\mathfrak{R}$s are composed of basic containers whose types are fixed. Hence, our approach first learns basic containers' accessing patterns, which refer to instruction sequences for computing the location of the container item, and then uses them to identify $\mathfrak{R}$s' accessing patterns. With $\mathfrak{R}$s' accessing pattern, we can identify accesses to the account balance from the execution traces of a token contract.

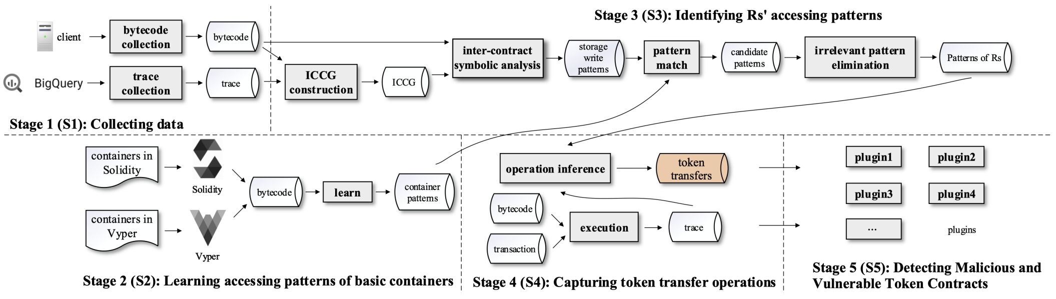


Fig. 2. The architecture of CRYPTO-SCOUT

As shown in Fig. 2, CRYPTO-SCOUT, comprising five stages, takes bytecode and historical transaction traces as input and outputs identified token transfers for plugins to detect malicious and vulnerable token contracts. Stage 1 utilizes blockchain clients and public databases to collect bytecode and traces to be analyzed. The methods are in accordance with previous studies [6, 54]. To identify $\mathfrak{R}$s' accessing patterns, it is first necessary to ascertain basic containers' accessing patterns. Therefore, stage 2 (**S2**) learns the accessing patterns of basic containers by performing symbolic analysis on seven contracts, each corresponding to one type of basic container (§4.2). Given a contract, Stage 3 (**S3**) identifies the accessing patterns of its $\mathfrak{R}$s, if any (§4.3). Since token transfers may involve inter-contract calls, **S3** first constructs the inter-contract call graph (ICCG). Following this, recognizing that token transfers require storage writes, **S3** conducts inter-contract symbolic analysis on the ICCG to collect *storage write patterns*, which are instruction sequences that compute the storage location to be written. Finally, the $\mathfrak{R}$s' accessing patterns are selected from the storage write patterns using two important insights: $\mathfrak{R}$s are composed of basic containers, and a token transfer requires two accesses to $\mathfrak{R}$s and the invocation/emission of the standard interface/event.

By monitoring the execution of $\mathfrak{R}$s' accessing patterns, Stage 4 (**S4**) detects modifications of $\mathfrak{R}$s. The details of these modifications, namely token transfers, are recorded. **S4** supports two distinct modes for capturing token transfers within the program paths of a non-deployed token contract, as well as within transactions that have already occurred (§4.4). In stage 5 (**S5**), plugins analyze token transfers to detect malicious and vulnerable token contracts.

Remark. By reverse engineering bytecode compiled from Solidity and Vyper codes, we discover two undocumented differences between Solidity and Vyper and equip CRYPTO-SCOUT with the ability to handle them. The differences will affect the bytecode analysis, but none of the existing tools can handle them. First, the accessing patterns of basic containers in different languages are different. CRYPTO-SCOUT handles it in **S2** (§4.2). Second, function calls in the bytecode compiled from Solidity and Vyper are different. CRYPTO-SCOUT copes with it when constructing ICCG in **S3** (§4.3).

4.2 S2: Learning Accessing Patterns of Basic Containers

Since $\mathfrak{R}$s are basic containers or combinations thereof, this stage learns the accessing patterns of basic containers to facilitate **S3** in recognizing how complex $\mathfrak{R}$s are composed. Different languages provide varying basic containers. This paper focuses on the two most popular languages recognized by the community: Solidity and Vyper. Solidity supports four basic containers: mapping, one-dimensional static array, one-dimensional dynamic array, and struct [43]. Vyper supports three basic containers: one-dimensional static array, mapping, and struct [44]. We do not consider a multi-dimensional array as a basic container because it can be composed of one-dimensional arrays.

To learn accessing patterns, we develop four smart contracts in Solidity and three in Vyper, each declaring and using one of the basic containers. After compiling them into bytecode, CRYPTO-SCOUT automatically extracts the accessing patterns of these basic containers by conducting symbolic execution. More specifically, each of these seven contracts modifies only the basic container, meaning there is only one storage write pattern in each contract. CRYPTO-SCOUT learns the accessing pattern of a basic container by recording this storage write pattern. Analyzing these seven contracts is sufficient, as the compiler dictates the accessing patterns of basic containers. These contracts are created once and can be reused to infer new accessing patterns if future compiler upgrades introduce changes.

We observe that the accessing pattern of a one-dimensional static array is the same as that of a struct because a static array can be considered as a struct whose items are of the same type, and hence, we do not introduce it below.

• *Mapping* in Solidity. Its accessing pattern is shown in Fig. 3(a). Each oval represents an EVM instruction, and each rectangle represents an operand. An edge from a rectangle A to an oval B

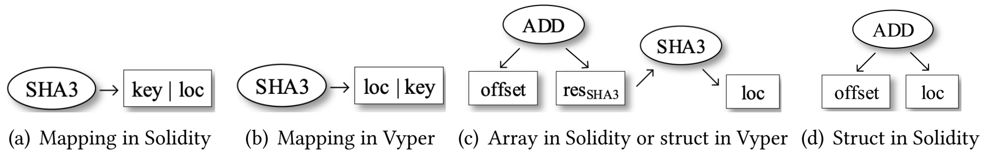


Fig. 3. Accessing patterns of basic containers

indicates A is the computation result of B. An edge from an oval A to a rectangle B suggests B is an operand of A. To access an item in a mapping, the location of the item is computed by hashing a piece of data by SHA3 instruction. The data is the concatenation of the *key* and the location of the mapping (i.e., *loc*) as shown in Fig. 3(a).
• *One-dimensional dynamic array* in Solidity. It has a location *loc* in the storage, and all array items are stored in sequence after the hashing of *loc*, i.e., SHA3(*loc*) in the storage. As shown in Fig. 3(c), to access an array item, the location of the array *loc* is needed to locate the first array item, and then an *offset* is added to locate the array item.
• *Struct* in Solidity. It has a location *loc* in the storage. To access an element of a struct, its *offset* from the beginning of the struct, which should be a constant, is added to *loc* to compute the location, as shown in Fig. 3(d).
• *Mapping* in Vyper. It differs from the mapping in Solidity only in that *loc* precedes *key* (Fig. 3(b)).
• *Struct* in Vyper. The accessing pattern of a one-dimensional dynamic array in Solidity is the same as that of a struct in Vyper (Fig. 3(c)). But CRYPTO-SCOUT can still differentiate these two data structures because CRYPTO-SCOUT first determines whether the bytecode is compiled from Solidity or Vyper contracts according to our observations of specific code features. More specifically, we observe that the bytecode of Vyper contracts uses a CALLDATALOAD instruction to read the function ID into the stack and then uses a MSTORE instruction to move the function ID from the stack to the memory. Differently, the bytecode of Solidity contracts just uses a CALLDATALOAD instruction to read the function ID into the stack without moving it to the memory.

4.3 S3: Identifying $\mathfrak{R}$s' Accessing Patterns

Identifying $\mathfrak{R}$s' accessing patterns is a prerequisite for capturing token transfers, as transfers require accessing $\mathfrak{R}$s. This stage comprises four steps. Step 1 constructs ICCG since transfers can be performed by inter-contract calls. Step 2 identifies storage write patterns within the ICCG, as transfers necessitate storage writes. In Step 3, candidate patterns are filtered from the storage write patterns using the insight that $\mathfrak{R}$s is composed of basic containers. Step 4 selects $\mathfrak{R}$s' accessing patterns from candidate patterns, employing a common feature of various token contracts (§4.3.4).

4.3.1 Step 1: Constructing ICCG. Token transfers may involve inter-contract calls, such as one contract implementing standard interfaces while another maintains $\mathfrak{R}$s (§5.6). Therefore, methods that analyze only a single contract are inadequate (e.g., TokenAware). This step constructs ICCG for each contract to enable subsequent inter-contract analysis. A study on ICCG construction assumes the call relationship of smart contracts is known [52]. However, such an assumption may not hold in practice because the callee's address may be given at runtime. Without depending on this assumption, our approach consists of three sub-steps. Sub-step 1 identifies inter-contract invocations by combining symbolic execution and historical traces (optional). Sub-step 2 identifies the intra-contract call graph of each contract according to the code features of the bytecode

compiled from Solidity contracts and/or Vyper contracts. Sub-step 3 constructs ICCG by connecting the intra-contract call graphs of the smart contracts according to their inter-contract invocations.
– *Sub-step* 1 aims to discover the call relationship of smart contracts. Given a contract bytecode, we perform symbolic execution on it to compute the addresses of its call targets. However, symbolic execution will be hindered if the call target depends on transaction inputs (e.g., function arguments). This is because symbolic execution, as a static analysis technique, cannot determine call targets that can only be determined at runtime. To solve this, we search for the call target using historical traces because they record information on all inter-contract calls that have occurred. This design choice of using historical traces to obtain inter-contract calls is evaluated as effective (§5.6). In summary, we integrate symbolic execution with historical traces to obtain more comprehensive inter-contract call information. Traces are optional; in their absence, the FN rate increases from 0.06% to 4.12% (§5.6).
– *Sub-step* 2 aims to build the call graph of each contract by recognizing intra-contract function invocations. It is challenging to accomplish this task due to two reasons. First, an intra-contract function invocation is compiled into a JUMP instruction rather than a dedicated instruction [49]. Thus we need to distinguish such JUMP instruction from others compiled from branch statements (e.g., *if*{...}*else*{...}). Second, there are differences in how Solidity and Vyper implement an intra-contract function invocation. Our solution described below solves these issues.

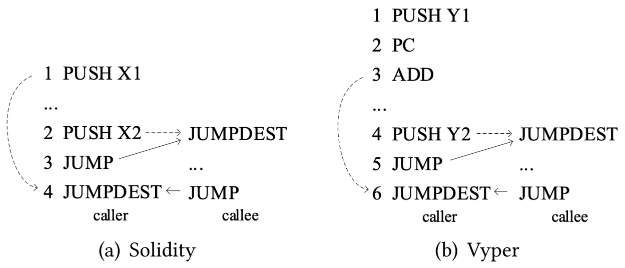


Fig. 4. Intra-contract function invocation

• *Solidity*. Fig. 4(a) illustrates the bytecode compiled from an intra-contract function invocation in Solidity. In this bytecode sequence, Line 1 pushes the return address of the function invocation, which points to Line 4. Line 2 pushes the address of the callee, and Line 3 performs a jump to the callee. To identify that the JUMP at Line 3 corresponds to a function invocation, we examine the operand of each PUSH instruction to determine if it points to the instruction immediately following a JUMP. If this condition is met, the JUMP is identified as an intra-contract function invocation. To validate this identification method, we compile intra-contract function invocations using various compiler versions (ranging from 0.1.1 to 0.8.19) with optimization enabled and verify whether the operand of PUSH consistently points to the instruction immediately after a JUMP. The results indicate that this characteristic is invariant across different compiler versions and optimization settings. Additionally, based on extensive experience in analyzing smart contracts and referencing the Solidity documentation [43], we confirm that the operand of PUSH points to the instruction immediately following a JUMP exclusively in cases of intra-contract function invocations.

• *Vyper*. Fig. 4(b) presents the bytecode generated from an intra-contract function invocation in Vyper. Unlike Solidity, Vyper does not directly push the return address of a function invocation. Instead, it computes the return address by adding an offset (Line 3) to the program counter of the executing instruction (Line 2). To identify the function invocation at Line 5, we collect the execution

results of the three instructions at Lines 1-3 (i.e., PUSH, PC, ADD) and check if the result points to the instruction immediately following a JUMP. If this condition is met, the JUMP is recognized as an intra-contract function invocation. Through experimentation with various versions of Vyper compilers, we confirm that the result of these three instructions consistently points to the instruction immediately after a JUMP in cases of intra-contract function invocations.
– *Sub-step* 3 constructs ICCG for each contract by linking individual call graphs according to the information of inter-contract calls.

4.3.2 Step 2: Identifying Storage Write Patterns. Token transfers necessitate modifying $\mathfrak{R}$s within storage, implying that $\mathfrak{R}$s' accessing patterns are inherently a subset of storage write patterns. This step aims to identify all storage write patterns within the contract under analysis. Our proposed approach involves exploring the contract's program paths, including inter-contract paths, to identify paths that execute storage writes (i.e., SSTORE). Subsequently, we collect the storage write patterns (i.e., instruction sequences that compute the storage location to be written) from the path.

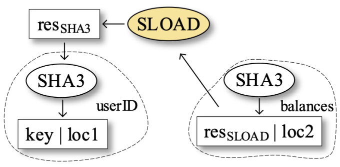
To explore paths, CRYPTO-SCOUT performs inter-contract symbolic analysis (ICSA) on the ICCG of the contract under test. Different from analyzing individual contracts, ICSA requires (1) identifying the invocation relationship between contracts (e.g., caller, callee, called function); (2) processing the instructions involved in inter-contract calls; (3) checking reachability of the inter-contract path. While the previous step addresses the first, this step addresses the remaining two and implements ICSA. Specifically, ICSA starts from each root node of the given contract's ICCG, corresponding to a public function. Initially, CRYPTO-SCOUT analyzes the contract in a manner analogous to traditional symbolic execution techniques, as the instructions being executed are confined within a single contract. This continues until an instruction that calls an external contract is encountered. At this point, CRYPTO-SCOUT infers the callee contract based on the ICCG, subsequently backing up the caller's context and initializing a new stack with the call arguments. The callee is then executed based on this new stack until it returns. Next, the backed-up context is restored, and the caller's subsequent instructions are executed. Throughout the execution of both the caller and callee contracts, CRYPTO-SCOUT maintains all path constraints and utilizes the constraint solver to determine the reachability of each path. The prototype of CRYPTO-SCOUT employs Z3 [40] as its solver and implements ICSA by improving Oyente [32]. Inspired by previous work, we employ strategies such as reusing constraints [46], pruning paths [3], and reducing constraint solving [53] to mitigate the adverse effects of path explosion and enhance execution efficiency.

For each reachable path, CRYPTO-SCOUT determines whether the storage is written and, if so, records the storage write pattern. To this end, when identifying an SSTORE in the path, CRYPTO-SCOUT employs def-use analysis [28] to infer the computation process of the first operand of SSTORE, which represents the storage location to be written. This computation process encompasses a series of instructions and symbolic values representing variable locations and account addresses (e.g., *loc* and *key* in Fig. 3), forming the storage write pattern that CRYPTO-SCOUT needs to record.

4.3.3 Step 3: Collecting Candidate Patterns of $\mathfrak{R}$s. After collecting a storage write pattern, we check whether it is composed of basic containers' accessing patterns. If not, the storage write pattern does not access $\mathfrak{R}$s, as $\mathfrak{R}$s are either basic containers or combinations thereof. To ease pattern matching, we organize all patterns as trees and then conduct pattern matching from the root to the leaves. If the tree of the storage write pattern can be divided into multiple ($\geq$ 1) parts and each part is a basic container's accessing pattern, the storage write pattern might be the accessing pattern of $\mathfrak{R}$s. For example, CRYPTO-SCOUT tries to match the tree in Fig. 5(b) with all basic containers' accessing patterns, and it finds that Fig. 3(a) matches because a SHA3 hashes a piece of data which is a concatenation of two values. Then, CRYPTO-SCOUT checks the remaining part of Fig. 5(b), and Fig. 3(a) matches again. Therefore, it knows this storage write involves two Solidity mappings. After

```
mapping(address => uint256) userID;
mapping(uint256 => uint256) balances;
function _transferFrom(address _from, address _to,
 uint256 _amount) public {
  updateBalance(_from, balancOf(_from) - _amount);
  updateBalance(_to, balanceOf(_to) + _amount);
}
function updateBalance(address _addr, uint256 _amount) {
  uint256 id = userID[_addr];
  balances[id] = _amount;
}
```

(a) Source code



(b) Pattern of the 𝔑s

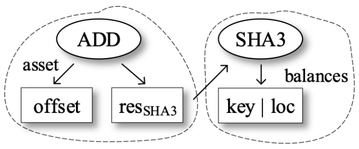
Fig. 5. An example of the variable composition

determining the basic containers involved in the storage write pattern, CRYPTO-SCOUT figures out how these containers are composed. We identify two composition modes as follows:

● *Variable composition.* It means that one container variable reads another container variable. Fig. 5(a) shows an example of this mode. 𝔑s in this contract include two mapping variables: *userID* that maps the addresses to holder IDs and *balances* maps the IDs to holder balances. Fig. 5(b) shows the accessing pattern of 𝔑s. An SLOAD, the only instruction that can read storage, is used to connect two mapping variables. They are composed in the way that one variable (i.e., *balances*) reads the result of SLOAD, which reads the result of SHA3 for accessing the second variable (i.e., *userID*). Consequently, if CRYPTO-SCOUT finds that SLOAD links the accessing patterns of two basic containers, it knows they are combined by this mode.

```
1  mapping(address => Asset)
2    public balances;
3  struct asset {
4      string name;
5      uint amount;
6  }
```

(a) Source code



(b) Pattern

Fig. 6. A nested container in Solidity

● *Nested container.* CRYPTO-SCOUT regards two containers as a nested container if their patterns are composed without using SLOAD. Fig. 6(a) shows an example of this mode, where 𝔑 is a mapping that maps an address to a struct. Fig. 6(b) shows the pattern to access *amount* (Line 5, Fig. 6(a)). As it does not contain a SLOAD, CRYPTO-SCOUT knows the two containers form a nested container.

4.3.4 Step 4: Removing Irrelevant Patterns. 𝔑s represent a subset of the numerous variables present in smart contracts. Many variables are unrelated to tokens yet share the same variable type as 𝔑s. Therefore, the candidate patterns identified in the previous step may encompass irrelevant patterns used to access other data. CRYPTO-SCOUT distinguishes between 𝔑s' accessing patterns and irrelevant patterns by exploiting a common feature of various token contracts: a single token transfer requires at least two modifications of 𝔑s (one for the sender and one for the receiver) and the invocation/emission of a standard interface/event. For each candidate pattern, CRYPTO-SCOUT checks if there is a program path that executes the accessing pattern at least twice and invokes a standard interface or emits a standard event. If such a path exists, CRYPTO-SCOUT regards the candidate pattern as the accessing pattern of 𝔑s. Otherwise, the candidate pattern is irrelevant since the pattern *only* appears in custom interfaces, and *no* standard event is emitted. Once CRYPTO-SCOUT finds an accessing pattern of $\mathfrak{R}$s from a smart contract, it is regarded as a token contract.

4.4 S4: Capturing Token Transfer Operations

By matching the execution trace of a token contract with the accessing pattern of its $\mathfrak{R}$s, this stage determines whether the trace is used to modify account balances and record the details of the modifications, namely token transfers. CRYPTO-SCOUT collects the execution traces in two ways. If the token contract has been invoked by transactions, CRYPTO-SCOUT executes the contract by replaying these transactions and then lists the token transfer triggered by them. Otherwise, CRYPTO-SCOUT symbolically executes a token contract and exposes token transfers in each program path.

Whether in concrete or symbolic terms, CRYPTO-SCOUT infers token transfers through the following steps. First, CRYPTO-SCOUT finds the address that is an operand of $\mathfrak{R}$s' accessing pattern, termed by *addr*, indicating the sender or receiver. Next, CRYPTO-SCOUT checks whether an SSTORE is executed after the pattern. If confirmed, it then checks whether the computation result of the pattern serves as the first operand of SSTORE, indicating the storage location to be updated. In this case, CRYPTO-SCOUT confirms that the SSTORE is used for modifying the $\mathfrak{R}$s. Subsequently, CRYPTO-SCOUT reads 32 bytes from the storage location, termed by val_{old}, standing for the token balance before transfer. In the following step, CRYPTO-SCOUT obtains the second operand of SSTORE, which is the new token balance after the transfer, termed by val_{new}. Consequently, the tuple $<addr, \Delta = val_{new} - val_{old}>$ is obtained. Finally, CRYPTO-SCOUT outputs token transfers as a set of such tuples.

4.5 S5: Detecting Malicious and Vulnerable Token Contracts

This stage provides users with interfaces to develop plugins that detect malicious and vulnerable token contracts. The plugin requests two pieces of data from CRYPTO-SCOUT: (1) the execution traces of the token contract, which contain executed instructions, called functions, raised events, and function arguments, and (2) token transfers within the traces. To demonstrate, we develop four plugins to detect two types of malicious token contracts and two types of vulnerabilities (§6). Our modular design ensures that the development effort remains manageable. The four plugins average just 96 lines of code.

5 Evaluation

5.1 Research Questions

The evaluation of CRYPTO-SCOUT focuses on its ability to recognize $\mathfrak{R}$s and token transfers since these are essential prerequisites for detecting malicious and vulnerable token contracts. Extensive evaluations are conducted to answer the following research questions.
RQ1: How is the accuracy of CRYPTO-SCOUT?
RQ2: How is the efficiency of CRYPTO-SCOUT?
RQ3: How many new accessing patterns of $\mathfrak{R}$s are discovered by CRYPTO-SCOUT?
RQ4: What if the inter-contract analysis is disabled?
RQ5: What if irrelevant patterns are not removed?

5.2 Datasets

We construct four datasets by collecting data from 13.7 million blocks (spanning 76 months).
- $D1$ contains 506,803 unique (duplicated bytecode copies are eliminated) bytecode. This dataset is used to evaluate the effectiveness and performance of CRYPTO-SCOUT and its competitors.
- $D2$ contains the historical traces in 13.7 million blocks. Applying CRYPTO-SCOUT to $D2$ allows assessing its ability to identify real-world token transfers.

- *D*3 includes 906 ERC20 [22], 594 ERC721 [19] and 150 ERC1155 token contracts [39] obtained from Etherscan. This dataset is used to gauge the FN rate.
- *D*4 contains 78,642 unique smart contracts obtained from Etherscan [20]. It is employed in assessing the FP rate. Obviously, *D*3 is a subset of *D*4, and all the bytecode compiled from *D*4 is the subset of *D*1.

5.3 RQ1: Accuracy of CRYPTO-SCOUT

For a contract under test, if CRYPTO-SCOUT correctly identifies the language used, the token standard followed, and the accessing pattern of $\mathfrak{R}$s, the report is a true positive. CRYPTO-SCOUT reports 46,412 token contracts out of *D*4. As inspecting all source code is time-consuming and labor-intensive, we manually check 1,041 randomly selected contracts and confirm that *no* FP is produced. According to the sample size determination theory [34], 1,041 samples can meet a confidence level of 99% and a precision of ±4%. The absence of FPs is attributed to removing irrelevant patterns (§4.3.4). The ablation study demonstrates that omitting this step would have escalated the FP rate to 1.3% (§5.7).

To evaluate the FN rate, we check how many token contracts in *D*3 are not detected by CRYPTO-SCOUT. Manual investigation confirms that it produces one FN (i.e., the FN rate is 0.06%) because CRYPTO-SCOUT fails to discover an inter-contract call. Fig. 7 shows the token contract that CRYPTO-SCOUT does not recognize. The token transfer involves multiple contracts, but the information of inter-contract calls is neither encoded in bytecode nor historical traces. Instead, the callee contract *externalContract* (Line 9) is read from the storage variable *uidToExternalNft* (Line 8), which depends on the input *_tokenId* (Line 6). Therefore, CRYPTO-SCOUT fails to construct a complete ICCG. In future work, we will consider utilizing fuzzing techniques [31] to explore call targets that cannot be revealed by either symbolic execution or traces.

Answer to RQ1: CRYPTO-SCOUT is accurate with 0.06% FN rate and produces no FP.

```
1   struct ExternalNft {
2       address nftContractAddress;
3       uint nftId;
4   }
5   mapping (uint => ExternalNft) uidToExternalNft;
6   function transferFrom(address _from, address _to, uint256 _tokenId) {
7   ......
8       ExternalNft memory externalNft = uidToExternalNft[_tokenId];
9       VIP181 externalContract = VIP181(externalNft.nftContractAddress);
10  ......
11       externalContract.transferFrom(_from, _to, externalNft.nftId);
12       emit Transfer(_from, _to, _tokenId);
13  }
```

Fig. 7. The only false negative of CRYPTO-SCOUT

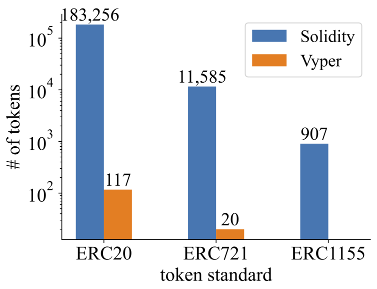


Fig. 8. Number of token contracts

We apply CRYPTO-SCOUT to *D*1, and it recognizes 195,885 token contracts. Fig. 8 shows that 93.6% are ERC20, 5.9% are ERC721, and 0.5% are ERC1155. Among them, 137 are developed in Vyper, while the rest are in Solidity.

• Comparison with TokenScope. By applying TokenScope to *D*1, we find that TokenScope fails to recognize 44 ERC20 contracts developed in Solidity, 117 ERC20 contracts developed in Vyper, all 11,605 ERC721 and all 907 ERC1155 contracts. The root cause is that TokenScope relies on four manually-defined patterns extracted from ERC20 contracts developed in Solidity [7]. We also experiment to compare the ability of CRYPTO-SCOUT and TokenScope in coping with the evolving of token contracts. Specifically, given seven basic containers (§4.2) and two composition modes (§4.3), we generate 50 $\mathfrak{R}$s by randomly combining these basic containers according to composition modes. Note that if a combined container is selected, it will further be combined with other basic/combined

containers to generate complicated $\mathfrak{R}$s. Fig. 6(a) shows an example of the generated $\mathfrak{R}$. Then, we write 50 token contracts, each implementing a function to transfer tokens by modifying a $\mathfrak{R}$. By applying CRYPTO-SCOUT and TokenScope to these contracts, we observe that CRYPTO-SCOUT identifies all $\mathfrak{R}$s, whereas TokenScope identifies none because these 50 randomly-generated $\mathfrak{R}$s do not include the patterns that can be recognized by it. Note that TokenScope cannot be directly applied to bytecode, as it is designed to analyze traces. Therefore, to evaluate TokenScope, we first deploy the bytecode to our private blockchain and then initiate a transaction to execute the function used to transfer tokens. After that, TokenScope can infer token transfers by analyzing traces. Since it does not identify any of the token transfers, we state that it does not identify the $\mathfrak{R}$s.
• Comparison with TokenAware. By applying TokenAware to $D1$, we find that it fails to recognize 117 ERC20 contracts, 11,605 ERC721 contracts, and 907 ERC1155 contracts. This limitation arises from its lack of consideration for inter-contract calls and the diversity of languages and token standards. In addition, we also use those 50 randomly generated token contracts containing complex $\mathfrak{R}$s to evaluate the ability of TokenAware in coping with the evolving of token contracts. We observe that TokenAware can only recognize 29 of them. The reason is that it only supports ERC20 token contracts developed in Solidity.

5.4 RQ2: Efficiency of CRYPTO-SCOUT

Using CRYPTO-SCOUT to analyze 506,803 unique smart contracts on a desktop equipped with an Intel i5-10400 CPU and 16GB memory, we observe that the average analysis time per contract is 1.75 seconds. Fig. 9 displays the cumulative distribution function, which indicates that 90% of contracts are analyzed within 2.13 seconds. This minimal time overhead equips CRYPTO-SCOUT with the capacity to handle a vast and continually increasing number of token contracts.

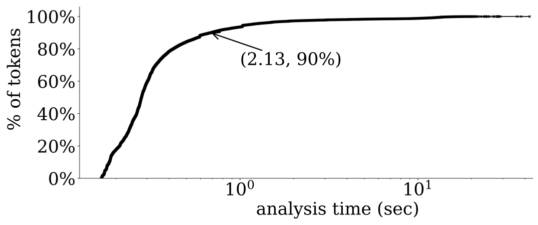


Fig. 9. CDF of analysis time per contract

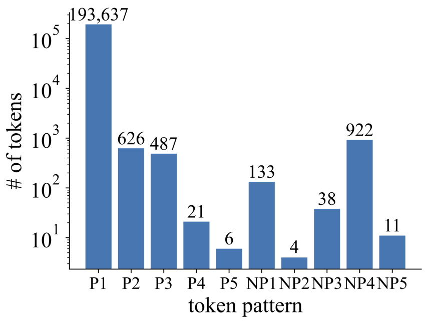


Fig. 10. Number of patterns

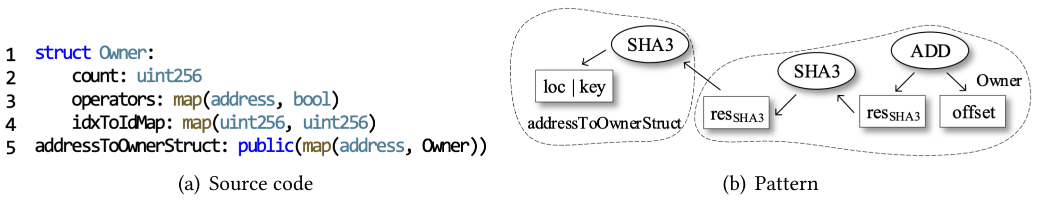
Answer to RQ2: CRYPTO-SCOUT is efficient, taking an average of 1.75 seconds to analyze a contract.

• Comparison with TokenScope. We evaluate the efficiency of TokenScope by applying it to $D1$. The experiment shows that TokenScope consumes 236.11 seconds on average to analyze a contract, 134.92 times the overhead incurred by CRYPTO-SCOUT. This high overhead is caused by TokenScope's requirement to maintain an Ethereum node for executing transactions and collecting traces.
• Comparison with TokenAware. By applying TokenAware to $D1$, we observe that it takes an average of 0.73 seconds to analyze a contract. Its efficiency comes at the cost of accuracy, as it is deficient in some essential capabilities (e.g., inter-contract analysis) compared to CRYPTO-SCOUT (§7).

5.5 RQ3: Patterns Discovered by CRYPTO-SCOUT

Through analyzing token contracts in $D1$, CRYPTO-SCOUT reveals *ten* accessing patterns of $\mathfrak{R}$s, where *five* (NP1 - NP5) are never reported before. Fig. 10 shows the number of unique token contracts using each pattern. Seven patterns (P1 – P5, NP1, NP3) appear in ERC20 token contracts; four (P1, P2, NP1, NP2) are used in ERC721 contracts; two (NP4 and NP5) are employed in ERC1155 contracts. Two (NP1 and NP2) are in Vyper contracts, and the other eight appear in Solidity contracts. The five new patterns are described below.

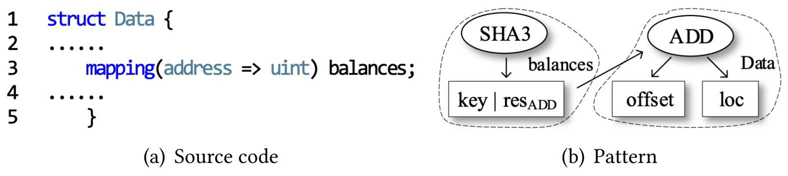
- **NP1** accesses a mapping variable that maps the account address to the account balance (Fig. 3(b)).
- **NP2** accesses a nested container that maps an address to a struct. Fig. 11(a) shows the code snippet of a real token contract using NP2. The variable *addressToOwnerStruct* maps an address to a struct *Owner* (Line 5), and *Owner* records the balance (Lines 2). Fig. 11(b) illustrates NP2, where the left part corresponds to the *addressToOwnerStruct* and the right part corresponds to the *Owner*.



```
1 struct Owner:
2     count: uint256
3     operators: map(address, bool)
4     idxToIdMap: map(uint256, uint256)
5 addressToOwnerStruct: public(map(address, Owner))
```

Fig. 11. NP2 appears in token contracts developed in Vyper.

- **NP3** accesses a struct, in which a mapping records the balance. Fig. 12(a) shows the code snippet of a real token contract using NP3. It defines a struct *Data* (Line 1) containing the mapping *balances* (Line 3). Fig. 12(b) shows NP3. The right part accesses the *Data*, and the left accesses *balances*.



```
1 struct Data {
2 ......
3     mapping(address => uint) balances;
4 ......
5     }
```

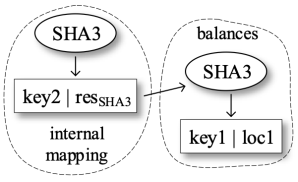
Fig. 12. NP3 appears in token contracts developed in Solidity.

- **NP4** accesses a nested mapping. It maps an address to another mapping, which maps a token ID to the balance. Fig. 13(a) shows the code snippet of a real ERC1155 token using NP4. NP4 is shown in Fig. 13(b). The right part accesses outer mapping, and the left accesses the inner mapping.
- **NP5** accesses a three-level nested container. Fig. 14(a) presents the code snippet of a real ERC1155 token contract using NP5. Each token type has a token ID, and thus a mapping *items* (Line 6) is used to map each token ID to a struct *Items* (Lines 1–5), which contains another mapping *balances* (Line 4). This inner mapping maps the account address to its balance. Fig. 14(b) depicts NP5. The right, middle, and left parts correspond to *items*, *Items*, and *balances*, respectively.

Answer to RQ3: Ten accessing patterns of $\mathfrak{R}$s are discovered. Five are previously unknown.

```
mapping (address => mapping(uint256 => uint256)) internal balances;
```

(a) Source code

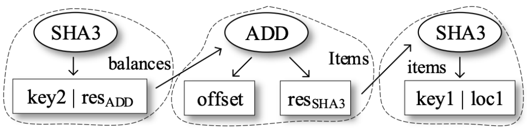


(b) Pattern

Fig. 13. NP4 appears in token contracts developed in Solidity.

```
struct Items {
    string name;
    uint256 totalSupply;
    mapping (address => uint256) balances;
}
mapping (uint256 => Items) public items;
```

(a) Source code



(b) Pattern

Fig. 14. NP5 appears in token contracts developed in Solidity.

5.6 RQ4: Effect of Inter-Contract Analysis

In **S3** (§4.3.1), CRYPTO-SCOUT constructs the ICCG utilizing both bytecode and historical traces and then performs ICSA. To assess the effect of traces, we modify the ICCG construction process to use only bytecode as input and apply this version of CRYPTO-SCOUT to analyze *D*3. Results show that 68 token contracts are missed (i.e., the FN rate increases to 4.12%). By analyzing the source code of these contracts, we find that token transfers necessitate inter-contract calls. However, the call targets are not encoded within the contracts themselves; instead, they must be retrieved at runtime from parameters or the blockchain's state. Therefore, it is effective to employ traces to infer call targets that are unavailable from static analysis of bytecode.

Furthermore, we assess the effect of ICSA by analyzing each contract in isolation and not accounting for inter-contract calls. This modification results in 74 token contracts in *D*3 being missed (i.e., the FN rate increases to 4.5%). Contract *C* in Fig. 15 is an example of the FN, with other FNs exhibiting similar characteristics. Fig. 15 presents a token transfer composed of three inter-contract calls. To transfer tokens from *msg.sender* to *_to*, transfer() in contract *A* is invoked, which calls transfer() in contract *B*. Then, *B* calls setBalance() in contract *C* twice to update balances of *msg.sender* and *_to*. Without inter-contract analysis, CRYPTO-SCOUT cannot recognize $\mathfrak{R}$s and their changes.

Answer to RQ4: Without inter-contract analysis, the FN rate will increase from 0.06% to 4.5%.

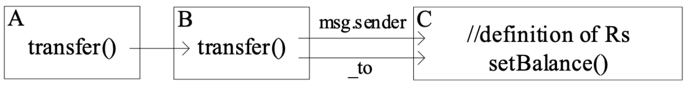


Fig. 15. A token transfer consisting of three inter-contract calls

5.7 RQ5: Effect of Removing Irrelevant Patterns

In order to differentiate $\mathfrak{R}$s from other variables, CRYPTO-SCOUT eliminates irrelevant patterns (§4.3.4). After disabling this step, CRYPTO-SCOUT detects 2,788 more token contracts from *D*4. As 623 of them are open-source, we read their source code and confirm that they are all FPs. Hence, the FP rate increases from 0% to at least 1.3% $(623/(46,412+623))$. We find that their candidate patterns do not access the $\mathfrak{R}$s; instead, they access other information, e.g., points used in game applications, tickets, workloads, etc. We find that no token contracts are missed, and therefore, FNs are not affected.

Answer to RQ5: Without eliminating irrelevant patterns, the FP rate increases from 0% to 1.3%.

6 Plugins of CRYPTO-SCOUT

CRYPTO-SCOUT works regardless of whether the target token contract has been invoked. We first develop two plugins, P1 and P2, without transactions (i.e., checking token contracts before deployment). Since thousands of token contracts with malicious code or vulnerabilities are discovered by these two plugins, we build a test bed to automate the validation of the results as much as possible. That is, we construct transactions in a private Ethereum blockchain to try to trigger malicious code or vulnerabilities in token contracts. If they are successfully triggered, we consider our detection accurate. Next, we develop two plugins, P3 and P4, in the presence of transactions (i.e., detecting malicious behavior and vulnerability exploitation that have occurred). Since these two plugins detect attacks that have actually occurred, we assess their accuracy by manually checking the transactions and source code (if available).

These four plugins focus on issues that are closely tied to the real world. P1 detects a type of vulnerable token contract, while P2 detects a type of malicious token contract, both of which can potentially be used as honeypot traps [27] by attackers. P3 detects stolen or frozen funds caused by malicious code. It focuses on exchanges that collectively hold more than 3,000 token types valued at $31.02 billion [21]. P4 detects attacks that exploit authorization vulnerabilities [29].

6.1 P1: Fake Deposit

• Description. The token standard specifies that the transfer interface *must* throw an exception if the sender's balance is insufficient [47]. As a result, many applications interpret a successfully executed interface without an exception as an indication that the sender has sufficient funds and that the token transfer is successful. However, these applications are susceptible to fake deposit attacks, as token contracts may not strictly adhere to the standard. This attack exploits a vulnerability where the standard interface does not throw an exception when the sender's balance is insufficient. Fig. 16 shows a real-world interface with such a vulnerability detected by P1. Due to developer oversight or a lack of understanding of the token standard, this interface returns false when the sender's balance is insufficient (Line 7) instead of throwing an exception. Consequently, an attacker can initiate an attack by specifying a huge transfer amount (*_value* on Line 1). Since no exception is triggered, the victim mistakenly believes the attacker has successfully transferred the tokens.

```
1  function transfer(address _to, uint256 _value) returns (bool success) {
2      if(balances[msg.sender] >= _value && _value > 0) {
3        balances[msg.sender] -= _value;
4        balances[_to] += _value;
5        Transfer(msg.sender, _to, _value);
6        return true;
7      } else {return false;}
8  }
```

Fig. 16. A vulnerable standard interface that can be exploited to launch a fake deposit attack

• Plugin design. With CRYPTO-SCOUT, P1 knows the token transfer on each program path of a token contract. For each path that does not transfer tokens, CRYPTO-SCOUT checks whether the standard interface (i.e., transfer() or transferFrom()) is invoked. Note that the transaction invoking a smart

contract should specify the function ID, and the function IDs of these two standard interfaces are known. Hence, by matching the function ID specified by the transaction with the two known function IDs, P1 knows whether a standard interface is invoked. If so, P1 obtains the arguments from the transaction and checks whether (1) the number of tokens to be transferred is not zero and (2) the sender and receiver are not the same accounts. If the two requirements are satisfied, the token contract can be exploited to launch a fake deposit attack.
• Experiment. This plugin is implemented in 54 lines of Python code. Using P1 to analyze all 195,885 token contracts recognized by CRYPTO-SCOUT, we uncover 20,220 vulnerable ones. Of these, 9,624 are open-source, and we attempt to exploit their vulnerabilities on our testbed to evaluate the accuracy of P1. 9,258 (96.2%) vulnerable contracts are successfully exploited, while the remaining 366 are manually verified due to the need for specific storage states from the Ethereum mainnet, which are missed in our private chain. The results show that none of them is an FP.

6.2 P2: Fake Notification

• Description. The token standard specifies that a standard event (i.e., Transfer()) *must* be emitted during a token transfer [47]. Consequently, many applications, such as exchanges and blockchain explorers, monitor this event to track token transfers. However, some malicious token contracts emit the event *without* actually transferring any tokens, creating false notifications to deceive applications. Fig. 17 shows a real-world malicious token contract detected by P2. The function mint() emits $\_to.length$ standard events (Line 3), but no tokens are transferred to $\_to[i]$.

```
1 function mint(address[] _to) public returns (bool) {
2     for(uint256 i = 0; i < _to.length; i++) {
3         Transfer(address(0), _to[i], 16....);
4     }
5     return true;
6 }
```

Fig. 17. A malicious token contract that can emit fake notifications

• Plugin design. By leveraging CRYPTO-SCOUT, P2 knows the token transfer on each program path. For each path that does not transfer tokens, P2 detects the emission of Transfer() by parsing the operands of the EVM instructions, LOG0, LOG1, LOG2, LOG3, LOG4, because only they can emit events [49]. P2 uncovers a program path that can produce a fake notification if (1) the third parameter of Transfer(), which denotes the number of transferred tokens, is not zero, and (2) the first parameter (i.e., sender) is not equal to the second parameter (i.e., receiver). If there is such a path, the token contract can produce fake notifications.
• Experiment. This plugin is implemented in 113 lines of Python code. Using P2 to analyze 195,885 unique token contracts, we discover that 1,485 can emit fake notifications. Of these, 755 are open-source, and we attempt to execute their malicious code on our testbed to evaluate the accuracy of P2. Fake notifications in 254 (33.7%) contracts can be validated automatically on our test bed. The remaining 501 are manually verified since their automated testing requires specific storage states on the Ethereum mainnet. The results show that none of them is an FP.

6.3 P3: Misleading DEX Attacks

• Description. Decentralized exchanges (DEXs) are a crucial type of smart contract that enable users to trade tokens. A typical DEX operation, as seen in platforms like EtherDelta [21], consists of three main phases: users first deposit tokens into the DEX, then trade those tokens within the platform, and finally withdraw their tokens. Fig. 18 illustrates two essential interfaces of the DEX. To deposit tokens, users call depositToken(), which then invokes the token contract's standard interface, transferFrom(), to process the transfer. For withdrawals, users invoke withdrawToken(), which then calls the standard interface, transfer(), to return the tokens.

The amount of tokens to be transferred is specified by passing parameters (i.e., *amount* in Lines 3 and 11) to standard interfaces. However, malicious token contracts can exploit this mechanism by manipulating the actual transfer amount, leading to discrepancies between the specified and transferred token quantities. For example, transferFrom() of a malicious contract can transfer fewer tokens than specified, allowing an attacker to deposit a smaller amount of tokens while the DEX registers a larger deposit. Similarly, during withdrawals, transfer() might transfer more tokens than the specified amount, enabling the attacker to withdraw more tokens than they initially deposited. These types of manipulations result in tokens being stolen. Moreover, malicious contracts can also cause tokens to become frozen within the DEX. This could happen if the contract transfers more tokens than specified during deposits or fewer tokens than specified during withdrawals, ultimately trapping user funds within the DEX.

```
1  function depositToken(address token, uint amount) {
2      if (token==0) throw;
3      if (!Token(token).transferFrom(msg.sender, this, amount)) throw;
4      tokens[token][msg.sender] = safeAdd(tokens[token][msg.sender], amount);
5      Deposit(token, msg.sender, amount, tokens[token][msg.sender]);
6  }
7  function withdrawToken(address token, uint amount) {
8      if (token==0) throw;
9      if (tokens[token][msg.sender] < amount) throw;
10     tokens[token][msg.sender] = safeSub(tokens[token][msg.sender], amount);
11     if (!Token(token).transfer(msg.sender, amount)) throw;
12     Withdraw(token, msg.sender, amount, tokens[token][msg.sender]);
13 }
```

Fig. 18. Interfaces of a DEX contract for depositing and withdrawing tokens

• Plugin design. For each transaction, P3 first checks if depositToken() or withdrawToken() of a DEX is invoked. If so, P3 records the token deposit or withdraw behavior (termed by $b1$) from the arguments of depositToken() or withdrawToken(). By leveraging CRYPTO-SCOUT, P3 knows the token transfer within each transaction (termed by $b2$). Finally, P3 identifies a misleading DEX attack by finding inconsistencies between $b1$ and $b2$.

• Experiment. This plugin is implemented in 110 lines of Python code. Then, we collect ten open-source DEXs that have a similar implementation as EtherDelta (i.e., they have the same withdrawToken() and depositToken() interfaces in Fig. 18). Next, we replay all transactions in 13.7 million blocks and use CRYPTO-SCOUT to identify token transfers. P3 reveals that 8,284 transactions mislead 8 DEXs, which are related to 127 token contracts with problematic implementations. By inspecting these 8,284 transactions, we discover that 120 kinds of tokens were stolen, with a total number of 1.2×10^{66} and a total value of 1,344.40 ETH. Besides, we reveal that 87 kinds of tokens were frozen, with a total number of 10^{33} and a total value of 135.02 ETH.

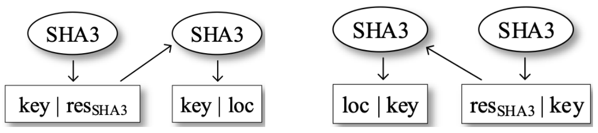
6.4 P4: Inconsistent transferFrom()

• Description. The standard interface transferFrom(address *_from*, address *_to*, uint256 *_value*) transfers *_value* tokens from *_from* to *_to*, if the transaction initiator is authorized by *_from* [47]. A token contract maintains a variable, termed by L, to record the token amount authorized by an account that another account can spend. If some tokens are transferred by transferFrom(), L should be modified accordingly. For ease of explanation, we use Q_L to denote the change of the number of authorized tokens, which is recorded in L, and $Q_{\mathfrak{R}}$ to indicate the number of transferred tokens, which is recorded in $\Re$s. If an account *spender* spends Q_L tokens held by the account *owner*, then the number of tokens that *owner* authorizes to *spender* will be reduced by Q_L and the number of tokens held by *owner* will be reduced by $Q_\Re$. In normal situations, Q_L should be equal to $Q_\Re$. If $Q_L \neq Q_\Re$, an inconsistent transferFrom() occurs. Fig. 19 shows a real-world vulnerable token contract detected by P4. Line 1 defines the L, which is a nested mapping. The inconsistency is caused by Line 6, where the arithmetic operation '$-$' should be applied rather than '+'. Consequently, each time the *spender* (e.g., an attacker) calls transferFrom(), the authorized amount does not decrease but increases. Therefore, the attacker can deplete the victim's tokens by repeatedly calling transferFrom().

```
1  mapping (address => mapping (address => uint256)) private _allowances;
2  function transferFrom(adddress _from, address _to, uint _value)...{
3  ......
4      balanceOf[_from] = balanceOf[_from] - _value;
5      balanceOf[_to] = balanceOf[_to] + _value;
6      allowance[_from][msg.sender] = allowance[_from][msg.sender] + _value;
7  ......
8  }
```

Fig. 19. The inconsistent transferFrom() of a real-world token contract

• Plugin design. By analyzing the bytecode in $D3$, we discover two accessing patterns of L, one for Solidity and the other for Vyper (Fig. 20). P4 detects the inconsistency by four steps. First, P4 checks if transferFrom() is invoked. If so, in the second step, P4 obtains the token transfer through CRYPTO-SCOUT. Third, P4 detects the modifications of L by monitoring the execution of the patterns shown in Fig. 20. This step is similar to monitoring the modifications of $\Re$s (§4.4). Finally, P4 detects the inconsistency by comparing the modifications of L with token transfers.



(a) Solidity (b) Vyper

Fig. 20. Accessing patterns of L

• Experiment. This plugin is implemented in 105 lines of code. By replaying all transactions in 13.7 million blocks, P4 detects 4,597 transactions that trigger inconsistency. These transactions were sent to 75 token contracts. By examining the identified inconsistency, we find two kinds of transactions. First, for 2,021 transactions, Q_L is smaller than $Q_\Re$. This indicates that the number of tokens that were actually transferred exceeds the authorized amount. Second, for 2,576 transactions, Q_L is larger than $Q_\Re$. In this case, a user cannot transfer as many tokens as the authorized amount. We compute the number of tokens involved by $\sum_{i=1}^{n} |Q_\Re - Q_L|$, where n is the number of transactions triggering the inconsistency. The result is about 1.52×10^8 tokens, valued at 75,435.88 ETH.

7 Related Work

Existing approaches can be divided into two categories.

• Techniques based on standard interface/event. This kind of approach has been widely applied in practice. By studying the source code of some popular projects, we find three exchange markets [11, 15, 24] rely on standard interface, and one blockchain explorer [50], two exchange markets [18, 37],

two wallets [23, 35] and one data collection tool [15] monitor both standard interfaces and events. This approach is also commonly utilized in academic research. Some studies employ standard interfaces and events to obtain token transfers and thus assess the token ecology and security [2, 5, 8, 9, 12, 13, 38, 42, 45]. McLaughlin et al. design an arbitrage identification method [33]. Dyson et al. design a digital forensic tool to recognize and track token transfers [14]. HORUS captures the flow of tokens from and to the attacker's account [25]. However, this kind of approach suffers from low accuracy since the implementation of token contracts may not strictly follow the standards.
• Techniques based on patterns. Fröwis et al. propose a simple pattern to recognize token contracts [26]. Their method suffers from high FPs and FNs due to the limitation of the oversimplified pattern. TokenScope applies four manually-extracted patterns to infer token transfers [7]. Hence, it cannot automatically recognize new types of $\Re$s, which can be caused by new combinations of basic containers or different languages. Furthermore, its scalability is limited by its reliance on manual bytecode analysis for pattern extraction. TokenAware, the closest work, identifies complex $\Re$s and then instruments an Ethereum node to identify token transfers in historical traces [30]. Five fundamental differences exist between CRYPTO-SCOUT and TokenAware. (1) CRYPTO-SCOUT can automatically learn the accessing patterns of basic containers (§4.2). This approach ensures flexibility and adaptability to future changes (e.g., compiler updates). In contrast, TokenAware relies on manual reverse engineering of bytecode. (2) CRYPTO-SCOUT can identify malicious and vulnerable contracts before their deployment and invocation (§4.4). TokenAware is only capable of detecting token transfers triggered by transactions. (3) CRYPTO-SCOUT can identify intra-contract and inter-contract calls and construct ICCG. After this, ICSA is used to explore the execution paths of the contract (§4.3). In contrast, TokenAware can only cover the paths triggered by transactions. (4) CRYPTO-SCOUT can recognize various $\Re$s that are implemented in different languages (§4.1), while TokenAware only supports Solidity. (5) TokenAware focuses on recognizing $\Re$s, while CRYPTO-SCOUT offers interfaces for users to develop plugins to detect various malicious and vulnerable contracts (§4.5).

8 Conclusion

This work proposes a novel approach to automatically recognize token transfers and implement it in CRYPTO-SCOUT that supports two languages and three token standards. We develop four plugins on top of CRYPTO-SCOUT to detect four types of malicious or vulnerable token contracts. Extensive experiments show that CRYPTO-SCOUT is effective and efficient, and the four plugins discover more than 21,000 malicious or vulnerable token contracts and more than 12,000 transactions triggering them.

9 Data Availability

The executable and experimental results are available at https://github.com/xxki-workstation. Since the detected malicious and vulnerable contracts may cause financial losses, we also include them in the repository to alert token holders. The DEXs detected by P3 and the first kind of token contracts detected by P4 are excluded because they are vulnerable to attacks.

Acknowledgments

The authors thank the anonymous reviewers for their constructive comments. This work is partially supported by the National Natural Science Foundation of China (62332004), the Sichuan Provincial Natural Science Foundation for Distinguished Young Scholars (2023NSFSC1963), and the Hong Kong RGC Projects (PolyU15224121, PolyU15231223).

References

[1] achinta das. 2021. DEFI Sandwich Attack Explain. https://medium.com/coinmonks/defi-sandwich-attack-explain-776f6f43b2fd.

[2] Monika Di Angelo and Gernot Salzer. 2023. Identification of token contracts on Ethereum: standard compliance and beyond. *Int. J. Data Sci. Anal.* 16, 3 (2023), 333–352. doi:10.1007/S41060-021-00281-1

[3] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 351–366. doi:10.1007/978-3-540-78800-3_27

[4] Defi Cartel. 2023. Wrecking sandwich traders for fun and profit. https://github.com/Defi-Cartel/salmonella.

[5] Federico Cernera, Massimo La Morgia, Alessandro Mei, and Francesco Sassi. 2023. Token Spammers, Rug Pulls, and Sniper Bots: An Analysis of the Ecosystem of Tokens in Ethereum and in the Binance Smart Chain (BNB). In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3349–3366. https://www.usenix.org/conference/usenixsecurity23/presentation/cernera

[6] Ting Chen, Zihao Li, Yufei Zhang, Xiapu Luo, Ang Chen, Kun Yang, Bin Hu, Tong Zhu, Shifang Deng, Teng Hu, Jiachi Chen, and Xiaosong Zhang. 2019. DataEther: Data Exploration Framework For Ethereum. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1369–1380. doi:10.1109/ICDCS.2019.00137

[7] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1503–1520. doi:10.1145/3319535.3345664

[8] Weili Chen, Mingdong Tang, and Zibin Zheng. 2023. Exploring and Analyzing the Token Ecosystem: A Complex Network Analysis Perspective. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 13, 3 (2023), 720–733. doi:10.1109/JETCAS.2023.3266396

[9] Weili Chen, Tuo Zhang, Zhiguang Chen, Zibin Zheng, and Yutong Lu. 2020. Traveling the token world: A graph analysis of Ethereum ERC20 token ecosystem. In *Proceedings of The Web Conference 2020* (Taipei, Taiwan) *(WWW '20)*. Association for Computing Machinery, New York, NY, USA, 1411–1421. doi:10.1145/3366423.3380215

[10] CoinMarketCap. 2024. Global Cryptocurrency Charts. https://coinmarketcap.com/charts/.

[11] Curvegrid. 2018. toy-block-explorer. https://github.com/curvegrid/toy-block-explorer.

[12] Monika di Angelo and Gernot Salzer. 2020. Characteristics of Wallet Contracts on Ethereum. In *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. 232–239. doi:10.1109/BRAINS49436.2020.9223287

[13] Monika di Angelo and Gernot Salzer. 2020. Characterizing Types of Smart Contracts in the Ethereum Landscape. In *Financial Cryptography and Data Security*, Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala (Eds.). Springer International Publishing, Cham, 389–404. doi:10.1007/978-3-030-54455-3_28

[14] Simon F. Dyson, William J. Buchanan, and Liam Bell. 2020. Scenario-based creation and digital investigation of ethereum ERC20 tokens. *Forensic Science International: Digital Investigation* 32 (2020), 200894. doi:10.1016/j.fsidi.2019.200894

[15] Ellaism. 2018. EthereumClassic Block Explorer. https://github.com/ellaism/etc-explorer.

[16] W. Entriken, D. Shirley, J. Evans, and N. Sachs. 2018. ERC-721 Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721.

[17] ethereum.org. 2025. Ethereum development documentation. https://ethereum.org/en/developers/docs/.

[18] EtherEx. 2016. EtherEx: decentralized exchange built on Ethereum. https://github.com/etherex/etherex.

[19] Etherscan. [n. d.]. Non-Fungible Token Tracker. https://etherscan.io/tokens-nft. Accessed in April 2020. The website has been updated, and the content may no longer be available.

[20] Etherscan. 2024. Etherscan APIs documentation. https://docs.etherscan.io/api-endpoints/contracts.

[21] Etherscan. 2024. Token Holdings 0x8d12A197cB00D4747a1fe03395095ce2A5CC6819. https://etherscan.io/tokenholdings?a=0x8d12A197cB00D4747a1fe03395095ce2A5CC6819.

[22] Etherscan. n.d.. Token Tracker (ERC-20). https://etherscan.io/tokens. Accessed in April 2020. The website has been updated, and the content may have changed.

[23] Etherwall. 2022. The first Ethereum desktop wallet. https://www.etherwall.com/.

[24] EthVM. 2023. EthVM: Open Source Processing Engine and Block Explorer for Ethereum. https://github.com/EthVM/EthVM.

[25] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I*. Springer-Verlag, Berlin, Heidelberg, 33–52. doi:10.1007/978-3-662-64322-8_2

[26] Michael Fröwis, Andreas Fuchs, and Rainer Böhme. 2019. Detecting Token Systems on Ethereum. In *Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers* (St. Kitts, Saint Kitts and Nevis). Springer-Verlag, Berlin, Heidelberg, 93–112. doi:10.1007/978-3-030-32101-7_7

[27] Rundong Gan, Le Wang, and Xiaodong Lin. 2023. Why Trick Me: The Honeypot Traps on Decentralized Exchanges. In *Proceedings of the 2023 Workshop on Decentralized Finance and Security* (Copenhagen, Denmark) *(DeFi '23)*. Association for Computing Machinery, New York, NY, USA, 17–23. doi:10.1145/3605768.3623546

[28] Mary Jean Harrold and Mary Lou Soffa. 1994. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.* 16, 2 (March 1994), 175–204. doi:10.1145/174662.174663

[29] Zheyuan He, Zhou Liao, Feng Luo, Dijun Liu, Ting Chen, and Zihao Li. 2022. TokenCat: Detect Flaw of Authentication on ERC20 Tokens. In *ICC 2022 - IEEE International Conference on Communications*. 4999–5004. doi:10.1109/ICC45855.2022.9839252

[30] Zheyuan He, Shuwei Song, Yang Bai, Xiapu Luo, Ting Chen, Wensheng Zhang, Peng He, Hongwei Li, Xiaodong Lin, and Xiaosong Zhang. 2023. TokenAware: Accurate and Efficient Bookkeeping Recognition for Token Smart Contracts. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 26 (Feb. 2023), 35 pages. doi:10.1145/3560263

[31] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/3238147.3238177

[32] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/2976749.2978309

[33] Robert McLaughlin, Christopher Kruegel, and Giovanni Vigna. 2023. A Large Scale Study of the Ethereum Arbitrage Ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3295–3312. https://www.usenix.org/conference/usenixsecurity23/presentation/mclaughlin

[34] Janice M. Morse. 2000. Determining Sample Size. *Qualitative Health Research* 10, 1 (2000), 3–5. doi:10.1177/104973200129118183

[35] MyEtherWallet. 2024. Ethereum's Original Wallet. https://www.myetherwallet.com/.

[36] Nico. 2023. Token standards. https://ethereum.org/en/developers/docs/standards/tokens.

[37] OAX. 2018. Decentralized Exchange Proof of Concept. https://gitlab.com/oax/dex-poc.

[38] Gustavo Ansaldi Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. 2020. An exploratory study of smart contracts in the Ethereum blockchain platform. *Empir. Softw. Eng.* 25, 3 (2020), 1864–1904. doi:10.1007/S10664-019-09796-5

[39] Optimism. 2023. Multi-Token Token Tracker. https://optimistic.etherscan.io/tokens-nft1155.

[40] Z3 Theorem Prover. 2024. Z3 Prover. https://github.com/Z3Prover/z3.

[41] W. Radomski, A. Cooke, P. Castonguay, J. Therien, E. Binet, and R. Sandford. 2018. EIP-1155: ERC-1155 Multi Token Standard. https://eips.ethereum.org/EIPS/eip-1155.

[42] Shahar Somin, Goren Gordon, Alex Pentland, Erez Shmueli, and Yaniv Altshuler. 2020. ERC20 Transactions over Ethereum Blockchain: Network Analysis and Predictions. arXiv:2004.08201 [physics.soc-ph] https://arxiv.org/abs/2004.08201

[43] Solidity Team. 2024. Solidity documentation. https://solidity.readthedocs.io/en/latest/.

[44] Vyper Team. 2024. Vyper documentation. https://docs.vyperlang.org/en/latest/.

[45] Friedhelm Victor and Bianca Katharina Lüders. 2019. Measuring Ethereum-Based ERC20 Token Networks. In *Financial Cryptography and Data Security*, Ian Goldberg and Tyler Moore (Eds.). Springer International Publishing, Cham, 113–129. doi:10.1007/978-3-030-32101-7_8

[46] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. doi:10.1145/2393596.2393665

[47] F. Vogelsteller and V. Buterin. 2015. EIP-20: ERC-20 Token Standard. https://eips.ethereum.org/EIPS/eip-20.

[48] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-Fungible Token (NFT): Overview, Evaluation, Opportunities and Challenges. arXiv:2105.07447 [cs.CR] https://arxiv.org/abs/2105.07447

[49] Gavin Wood. 2020. ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. https://ethereum.github.io/yellowpaper/paper.pdf.

[50] xDai. 2024. xDai Explorer. https://blockscout.com/poa/xdai/.

[51] Rui Xi, Zehua Wang, and Karthik Pattabiraman. 2024. POMABuster: Detecting Price Oracle Manipulation Attacks in Decentralized Finance . In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 3923–3942. doi:10.1109/SP54263.2024.00257

[52] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2021. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1029–1040. doi:10.1145/3324884.3416553

[53] Yufeng Zhang, Zhenbang Chen, and Ji Wang. 2012. Speculative Symbolic Execution. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 101–110. doi:10.1109/ISSRE.2012.8

[54] Lin Zhao, Sourav Sen Gupta, Arijit Khan, and Robby Luo. 2021. Temporal Analysis of the Entire Ethereum Blockchain Network. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2258–2269. doi:10.1145/3442381.3449916

[55] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1371–1385. https://www.usenix.org/conference/usenixsecurity18/presentation/zhou

Received 2024-09-12; accepted 2025-01-14