International Journal of Data Science and Analytics
https://doi.org/10.1007/s41060-021-00281-1

REGULAR PAPER

Identification of token contracts on Ethereum: standard compliance and beyond

Monika Di Angelo[1] · **Gernot Salzer**[1]

Received: 3 November 2020 / Accepted: 19 August 2021

Abstract
Next to cryptocurrencies, tokens are a widespread application area of blockchains. Tokens are digital assets implemented as small programs on a blockchain. Being programmable makes them versatile and an innovative means for various purposes. Tokens can be used as investment, as a local currency in a decentralized application, or as a tool for building an ecosystem or a community. A high-level categorization of tokens differentiates between payment, security, and utility tokens. In most jurisdictions, security tokens are regulated, and hence, the distinction is of relevance. In this work, we discuss the identification of tokens on Ethereum, the most widely used token platform. The programs on Ethereum are called smart contracts, which—for the sake of interoperability—may provide standardized interfaces. In our approach, we evaluate the publicly available transaction data by first reconstructing interfaces in the low-level code of the smart contracts. Then, we not only check the compliance of a smart contract with an established interface standard for tokens, but also aim at identifying tokens that are not fully compliant. Thus, we discuss various heuristics for token identification in combination with possible definitions of a token. More specifically, we propose indicators for tokens and evaluate them on a large set of token and non-token contracts. Finally, we present first steps toward an automated classification of tokens regarding their purpose.

Keywords EVM bytecode · Heuristic · Interface · Smart contracts · Token standards · Transaction data

1 Introduction

Tokens (more specifically crypto tokens) are similar to the coins of a cryptocurrency, with two main differences. First, they do not have a blockchain or distributed ledger of their own. Rather, they are a digital asset on top of a cryptocurrency or blockchain, representing the right to something. As a medium of exchange, tokens can act as a currency themselves.

Secondly, tokens are programmable and can be used beyond the mere exchange of value. In this respect, tokens are part of an application, often a decentralized application (DApp). DApps are applications on a P2P network that are not controlled by a single entity. Decentralization can be achieved by implementing critical components on a blockchain. Governance of and access to DApps are often controlled by application-specific tokens, but tokens can also act as the local currency of a DApp.

In addition to these use cases (exchange of value, part of an application), tokens may be linked to off-chain assets. Moreover, they can serve as means of fundraising, pre-order or investment, as well as means for building an ecosystem or a community.

Tokens gained in importance, the more value was attached to them. At the same time, they sparked the interest of regulatory bodies. With the proliferation of tokens, one may ask what people intend to achieve by using tokens and how they attempt to achieve it.

As Ethereum is the major platform for tokens, we search for a clarification on the actual usage of tokens in its public data. More specifically, we investigate the following regulatory and technological aspects of tokens:

- Which types of tokens can be distinguished?
- Which standards for token contracts are in use?
- How can token contracts be identified in transaction data?
- How can the type of a token be automatically inferred?

✉ Monika Di Angelo
monika.di.angelo@tuwien.ac.at

Gernot Salzer
gernot.salzer@tuwien.ac.at

[1] TU Wien, Vienna, Austria

Published online: 03 September 2021

Our approach. We address these questions by analyzing the transaction traces of Ethereum with regard to the deployed bytecodes (static data) as well as the calls to token contracts (dynamic data). Concerning methods, we discuss the automated identification and classification of token contracts. The methods rely on reconstructing the interface of contracts from bytecode as well as on observing the actual behavior of contracts.

Contribution. Most work focuses on tokens with a high market cap and on the flow of Ether and tokens. In contrast, we view tokens as a particular group of smart contracts that includes all contracts, from unused to top tokens. Like other authors, we discuss tokens complying with the most prevalent ERC-20 standard [37], but we include other token standards as well. Furthermore, we give an account of the utilization of the standards and depict their usage over time.

Based on our exploration of contract interfaces and activities, we derive indicators for detecting token contracts that do not comply with any of the standards considered. Moreover, we evaluate these indicators systematically on a carefully selected ground truth of tokens and non-tokens. Finally, we propose a heuristic approach to assess the type of token contracts—security versus non-security—and evaluate it qualitatively against decisions of the US Securities and Exchange Commission (SEC).

Overall, this paper advances the field of blockchain analytics, in particular regarding tokens on Ethereum.

Roadmap. Section 2 introduces blockchain tokens and typical functionalities of token contracts. In Sect. 3, we discuss types of tokens with an emphasis on regulatory aspects. In Sect. 4, we summarize relevant token standards. In Sect. 5, we compare our approach to related work. Section 6 introduces terms and data. We present methods for the identification of compliant tokens in Sect. 7 and discuss their prevalence in Sect. 8. In Sect. 9, we characterize token contracts beyond standard compliance and discuss indicators for identifying non-compliant tokens. We compare the indicators in Sect. 10. To assess the type of tokens, we introduce the concept of purity in Sect. 11 and give examples in Sect. 12. Section 13 concludes with a summary of our findings.

2 Token basics

Token contracts maintain a ledger that records the ownership of tokens. Most contracts implement *fungible* tokens, which are mutually indistinguishable. In this case, it suffices to store the amount of tokens for each holder. *Non-fungible* tokens, on the other hand, are uniquely identified by individual bit patterns, like numbers, and the contract has to associate each individual token with its owner. The ledger is safeguarded by the cryptographic mechanics of the underlying blockchain.

The core functionality of token contracts consists of methods that allow holders to transfer some of their tokens to a specified address. Moreover, the contracts often enable administrators to create or destroy tokens (known as minting and burning).

2.1 Benefits

Three main characteristics make tokens on a blockchain particularly attractive.

- Programmability: Token contracts facilitate an automated management of aspects like the enforcement of regulations.
- Tamper evidence: The immutable traces of transfers on the blockchain provide evidence whether the digital ownership has been tampered with.
- Liquidity: With tokens, ownership can readily be divided into fractions, which increases the liquidity of otherwise indivisible assets.

2.2 Acquisition and value

Tokens can be purchased (e.g., during an initial coin offering (ICO) or through a crypto exchange), traded on-chain or received freely (e.g., during an airdrop or as a reward for a service or behavior).

The value of a token depends on supply and demand as well as on the trust of the participating community, which is based on the credibility and service.

2.3 Design of token contracts

As tokens are a widespread application, coding patterns and best practice examples are readily available, like in the collections provided by ConsenSys[1] and OpenZeppelin.[2] Many token contracts are generated by factories (on-chain or as a web service) according to a given specification.

Most tokens aim at establishing trust and credibility by disclosing their source code on Etherscan.io. As a service, this platform checks that the deployed bytecode is the result of compiling the source code with the given compiler settings and labels it as 'verified source code.'

3 Types of tokens

A common high-level categorization of tokens distinguishes between payment, security, and utility tokens [30]. The need

[1] https://consensys.github.io/smart-contract-best-practices/.

[2] https://github.com/OpenZeppelin/openzeppelin-contracts.

for clarifying the differences lies in the fact that in most jurisdictions, security tokens are more strictly regulated than other tokens. The main distinguishing feature is the investment purpose of security tokens as opposed to the added value for the functioning of a product that is typical of utility tokens. Payment tokens offer little to no other functionality beyond the transfer of values. Legally, the distinction is still a gray area in many jurisdictions.

3.1 Howey test

In [32], Rohr et al. base their discussion of legal aspects of token sales under US law on a similar classification of tokens and emphasize the importance of the so-called Howey test. They argue that jurisdictions should provide 'regulatory certainty and a sensible path to compliance.'

The Howey test essentially identifies three criteria as characteristic of securities. A financial instrument is considered a security, if it requires (i) the investment of money, (ii) in a common enterprise, (iii) with the expectation of profits mainly from the efforts of others [35]. For crypto-tokens, criterion (i) is met if the token is sold on-chain in exchange for a cryptocurrency or other crypto-assets. Whether a token is related to a 'common enterprise' mainly depends on the legal assessment of off-chain factors. For criterion (iii), an analysis of the underlying token contract may contribute to the overall assessment of the token. In Sect. 11, we will introduce our concept of 'purity' as an indicator that the token contract itself does not provide any means that would allow a token holder to make efforts on-chain.

3.2 Definitions

In this work, we rely on the distinction of token types as stated by the Swiss FINMA [19] as a common ground for US [32], EU [22], and other jurisdictions.

Security Tokens are 'assets, such as a debt or equity claim on the issuer. In terms of their economic function, therefore, these tokens are analogous to equities, bonds or derivatives.' Typically, it is a share in the issuing company (equity token).

Regarding legal compliance, there is an ongoing discussion on how it could be integrated into a token standard (cf. Sect. 4), as well as into wallets and exchanges (cf. [2]).

Utility Tokens are usually backed by a project, an application, or a DApp with a definable benefit (like access) and intend to 'provide access digitally to an application or service by means of a blockchain-based infrastructure. The issue of utility tokens does not require supervisory approval if the digital access to an application or service is fully functional at the time the tokens are issued.' The purpose of a utility token may include voting rights, some sort of reward, or staking governance.

3.3 Categorization

As these purposes and categories may overlap for a specific token, a finer-grained classification scheme may be more adequate. Many tokens are hybrids concerning this coarse categorization [22]. Based on a literature review and a subsequent empirical study, Oliveira et al. [30] distill eight archetypes of tokens.

It would be desirable to automatically identify the type of a token that a contract implements. In this work, we discuss first steps toward this goal.

4 Interface standards for tokens

Standardized interfaces for token contracts enable applications such as wallets to recognize tokens and to interact with them. In this section, we first introduce accepted token standards and then proposed security token standards.

4.1 Accepted token standards

The community continuously discusses and establishes standard interfaces for tokens in the programming language Solidity, which is prevalent on Ethereum. The following standards have been accepted so far.

ERC-20 Token Standard [37] is the most widely used and most general token standard that 'provides basic functionality to transfer tokens, as well as allows tokens to be approved so they can be spent by another on-chain third party.' It lists six mandatory and three optional functions as well as two events to be implemented by a conforming API.

ERC-721 Non-Fungible Token Standard [17] concerns tokens where each token is distinct (aka non-fungible) and thus enables the tracking of distinguishable assets. Each asset must have its ownership individually and atomically tracked. This standard requires compliant tokens to implement 10 mandatory functions and three events.

ERC-777 Token Standard [8] defines advanced features to interact with tokens while remaining backwards compatible with ERC-20. It defines operators to send tokens on behalf of another address and hooks for sending and receiving in order to offer token holders more control over their tokens. This standard requires compliant tokens to implement 13 mandatory functions and five events.

ERC-1155 Multi Token Standard [31] allows for the management of any combination of fungible and non-fungible tokens in a single contract, including transferring multiple token types at once. This standard requires compliant tokens to implement six mandatory functions and four events.

4.2 Proposed security token standards

Apart from the accepted standards, several others are proposed and discussed, but not yet finalized. From the legal perspective, the following security token standards seem interesting. While the first one is rather general, the other two are project-specific and company-backed.

ERC-1462 Base Security Token [25] is a minimal extension to ERC-20 that 'provides compliance with securities regulations and legal enforceability' and aims at general use-cases, while additional functionality and limitations related to projects or markets can be enforced separately. Furthermore, it includes 'KYC (Know Your Customer) and AML (Anti Money Laundering) regulations and the ability to lock tokens for an account, and restrict them from transfer due to a legal dispute.' Moreover, it provides means to attach documents to tokens. This standard requires compliant tokens to implement four further mandatory checking functions (on top of ERC-20) and two optional documentation functions.

ERC-1450 LDGRToken [33] is a 'security token for issuing and trading SEC-compliant securities' that extends ERC-20. This standard 'facilitates the recording of ownership and transfer of securities sold in compliance with the Securities Act Regulations CF, D and A.' Apart from its own mandatory functions, it makes optional parts of ERC-20 mandatory. Moreover, it requires certain modifiers and constructor arguments to be implemented.

ERC-1644 Controller Token Operation Standard [15] 'allows a token to transparently declare whether or not a controller can unilaterally transfer tokens between addresses.' This is motivated by the fact that 'in some jurisdictions the issuer (or an entity delegated to by the issuer) may need to retain the ability to force transfer tokens.' This standard requires compliant tokens to implement three mandatory functions and two events.

ERC-1644 is part of ERC-1400 [16], a library of standards for security tokens, which requires the contained standards to be backwards compatible with ERC-20 and via extensions also with ERC-777. Additionally, the library contains ERC-1410 for differentiated ownership and transparent restrictions, ERC-1594 for on- and off-chain restrictions, and ERC-1643 for document and legend management.

5 Comparison to related work

Most of the distantly related work focuses on the financial aspects (specifically the transfer of assets), network aspects (like address clustering), or cryptocurrency platforms other than Ethereum.

5.1 Ethereum token networks and transactions

The work mentioned here is related to our approach to the extent that it deals with Ethereum tokens and transaction data.

Ethereum transactions. Chan et al. [4] analyze the transactions as a graph in order to de-anonymize addresses. With the aim to address security issues, Chen et al. [5] analyze the transaction graph in regard to money transfer, contract creation, and contract call. Applying network science theory onto the transaction graph, Guo et al. [21] conclude that 'transaction volume, transaction relation, and component structure, exhibit a heavy-tailed property and can be approximated by the power law function.' Likewise, Chen et al. [7] employ a graph approach to analyze the token ecosystem by constructing a graph each for the creators, holders, and transfers of tokens.

ERC-20 token networks. Somin et al. [34] study the token trading network in its entirety by analyzing it as a graph and show power-law properties for the degree distribution. Similarly, Victor et al. [36] measure token networks, which they define as the network of addresses that have owned a specific type of token at any point in time, connected by the transfers of the respective token.

Our Approach. Rather than de-anonymization, security issues, or trading aspects, our investigation puts a focus on the identification of token contracts that comply to an interface standard, fully or partially. Furthermore, we aim at automatically inferring the type of an implemented token. To this end, we consider transactions not from a network or graph perspective, but on the level of contract deployment (for the bytecode of the contract) and event logs as well as call frequency of functions and contracts. Moreover, we employ the analysis of calls as an add-on to the analysis of bytecode in order to identify aspects of deployed contracts more reliably than we can achieve by relying merely on bytecode.

5.2 EVM bytecode analysis

The work mentioned here is related closely to our approach since we employ bytecode analysis for identifying both standard compliant and non-compliant token contracts.

Code Clones. To detect code clones, He et al. [23] first de-duplicate contracts by 'removing function unrelated code (e.g., creation code and Swarm code), and tokenizing the code to keep opcodes only.' Then, they generate fingerprints of the de-duplicated contracts by a customized version of fuzzy hashing and compute pairwise similarity scores. In another approach to clone detection, Liu et al.[27,28] characterize each smart contract by a set of critical high-level semantic properties. Then, they detect clones by computing the statistical similarity between the respective property sets. On source code level, Kondo et al. [24] applied a tree-based

clone detector to 33,000 verified contracts from Etherscan up to the year 2018.

ERC-20 Compliance. Fröwis et al. [20] as well as Norvill et al. [29] demonstrate the feasibility to identify ERC-20 compliance over the interface of a contract. To detect token systems automatically, Fröwis et al. [20] compare the effectiveness of a behavior-based method combining symbolic execution and taint analysis, to a signature-based approach limited to ERC-20 compliant tokens. They demonstrated that the latter approach detects 99% of the tokens in their ground-truth data set. Extracting function signatures and restoring the interface is also reported in our previous work [10,13].

Partial Compliance. Moreover, Fröwis et al. [20] consider partially ERC-20 compliant tokens when they implement at least 5 of the 6 mandatory functions. While the usage of signatures of the interface is in line with [20,29], our previous work extends it beyond ERC-20 compliance by including other standards as well and by discussing partial compliance [13].

Type Distinction. Next to employing a graph approach to analyze the token ecosystem, Chen et al. [7] try to classify token contracts by reading the descriptive texts in their source code, albeit less than 1% of the tokens provide such a text. In our previous work [14], we infer the token type over a semantic classification of the token interface.

Our Approach. The method of computing code skeletons is comparable to the first step for detecting similarities by [23]. Instead of fuzzy hashing as a second step though, we rely on the set of function signatures extracted from the bytecode and manual analysis, as our purpose is to identify token contracts reliably. This is in line with previous work on ERC-20 standard compliance [10,13,20,29].

Regarding non-compliant tokens, we devise further methods for their identification that extend our previous work [13].

Additionally, we aim at an automatic distinction of token types. In contrast to [7] where Chen et al. use descriptive texts from source code, we work at bytecode level and approach it over the concept of pure token contracts that we define via the set of implemented functions in the bytecode and apply this concept to exemplary security tokens.

6 Terms and data

In this section, we introduce relevant terms and describe the data used for the analysis. Throughout the paper, we abbreviate the factors 1000, 1,000,000 and 1,000,000,000 by the letters k, M, and G, respectively.

6.1 Terms

We assume the reader to be familiar with blockchain technologies and cryptocurrencies in general. Regarding the specifics of Ethereum, we refer to [3,18,38].

6.1.1 Accounts, transactions, and messages

Ethereum distinguishes between externally owned accounts, often called *users*, and contract accounts or simply *contracts*. Accounts are uniquely identified by addresses of 20 bytes. Users can issue *transactions* (signed data packages) that transfer value to users and contracts, or that call or create contracts. These transactions are recorded on the blockchain. Contracts need to be triggered to become active, either by a transaction from a user or by a call (a *message*) from another contract. Messages are not recorded on the blockchain since they are deterministic consequences of the initial transaction. They only exist in the execution environment of the Ethereum Virtual Machine (EVM) and are reflected in the execution trace and potential state changes. We use 'message' as a collective term for any (external) transaction or (internal) message.

6.1.2 Abstract binary interface (ABI)

Most contracts in the Ethereum universe adhere to the ABI standard [1], which identifies functions by a particular hash of the header. More precisely, such a function signature consists of the first four bytes of the Keccak-256 hash of the function name concatenated with the parameter types. The bytecode of a contract contains instructions that compare the first four bytes of the call data to the signatures of its functions. The latter can be usually found literally in the deployed bytecode and indicate that the contract implements functions with these headers.

Another component of the interface are *events*. Emitting an event during the execution of a contract results in a log entry that can be observed by off-chain programs. Events are implemented via the instruction `LOG` whose first argument is the hash of the event header. The presence of the hash in the bytecode indicates the ability to issue the corresponding event.

6.2 Database

Our analysis is based on the transaction data of the Ethereum main chain up to block 10.5 M, which was mined on July 21, 2020. We retrieve the blocks, transactions, and execution traces via the RPC interface of the Ethereum client OpenEthereum v3.0.1. To speed up the analysis of contracts, we use the verified source code of contracts at Etherscan. If not available, we resort to disassembling or decompiling the bytecode.

For efficient querying, we store the data in a Postgres database. Each of the 2 G messages (creations, calls, and self-destructions) is uniformly represented by a record composed of an abstract timestamp, the message type, the success status, the addresses of context, sender and recipient, the input and output data, and the transferred amount of Ether.

6.2.1 Contracts

For each of the 28.1 M successful creation messages, our table of contracts contains an entry with the timestamps of start and end of deployment, of the contract's first use after deployment, and of an optional self-destruction. Moreover, we store the deployment and the deployed bytecode, the deployment address, and the address of the creator.

6.2.2 Bytecodes

Frequently, contracts share the same bytecode. For each of the 300 k distinct codes, our table of codes contains the function and event signatures. Moreover, we maintain dictionaries with 400 k function and 60 k event headers that allow us to reconstruct the headers for the majority of signatures. See the next section for details.

6.2.3 Logs

For each of the 710 M instructions `LOG` that have been executed so far, our table of log entries records a timestamp, the context address and several fields with log data. The first field holds the hash of the event header. We are particularly interested in the standardized event accompanying token transfers, accounting for 60% of the entries.

6.2.4 Messages

The dynamic data, i.e., the calls to and from contracts as well as the emitted events, are sparse and noisy. For most contracts, only a small fraction of the offered functions has ever been called, and many events have never been emitted. Moreover, observing a call to a contract with a particular signature does not mean that the corresponding function is indeed implemented; often a so-called fallback function catches unknown signatures without raising an error. Only if a function is frequently called, it is safe to assume that it is part of the interface. To get a more complete picture, we accumulate the dynamic data for all contracts with the same bytecode.

6.2.5 Proxies

Furthermore, proxies are a phenomenon to be considered. They forward incoming calls to a central contract via a particular type of call. This way the proxy contract may implement an interface without containing the corresponding signatures in its bytecode. We identify proxies statically via their bytecode as well as dynamically by detecting the forwarding of calls.

7 Methods for ERC-compliant token contracts

In this section, we concentrate on contracts that comply with the token standards in Sect. 4, referring to them as 'fully compliant', and summarize methods to identify them. In Sect. 9, we consider methods for token contracts that are partially or not compliant.

Behavior-oriented approach. The central task of a token contract is bookkeeping. Each token contract maintains a data structure that maps user ids like addresses to quantities of fungible tokens or lists of non-fungible ones. Moreover, it usually implements functions for querying the data structure and for transferring tokens between users.

Chen et al. [6] observe the EVM execution trace to capture changes in the bookkeeping of a token. Then, they try to match the found changes with emitted events in order to detect inconsistencies.

Fröwis et al. try to detect bookkeeping by symbolic execution and taint analysis of the bytecode in order to identify token contracts. Due to the difficulty of the problem, this method is still less effective than the interface approach [20]. We therefore resort to interface methods in our analysis.

Interface-oriented approach. Token contracts need to be accessible by wallets and exchanges; hence, they offer standardized interfaces. We therefore expect fully compliant token contracts to be identifiable by the functions and events they implement. It is unlikely that a contract offers six or more functions with the profiles prescribed by a standard without implementing token semantics. We found a single bogus contract whose interface pretends to be a token contract but that does not record token holdings.

Figure 1 gives an overview of the procedure for interface reconstruction. In the first step, we split the raw bytecode into sections. Then, we locate all function entry points as well as selected events in the first code section; their signatures form

the interface. For many signatures, we are able to restore the original headers, which helps to understand the purpose of the contract. In the following, we describe the algorithms in more detail.

7.1 Skeletons

To detect functional similarities between contracts, we compare their *skeletons*. They are obtained from the bytecodes of contracts by replacing meta-data, constructor arguments, and the arguments of the operations `PUSH` uniformly by zeros and by stripping trailing zeros. The rationale is to remove variability that has little to no impact on the functional behavior (like the swarm hashes added by the Solidity compiler or hard-coded addresses of companion contracts). Skeletons allow us to transfer knowledge gained about one contract to others with the same skeleton.

As an example, the 28.1 M contract deployments correspond to just 140 k distinct skeletons. This is still a large number, but more manageable then 298 k distinct bytecodes. By exploiting creation histories and the similarity via skeletons, we are able to relate 13.7 M contract addresses to one of the 92 k source codes on Etherscan, an increase from 0.3 to 49%.

7.2 Sectioning EVM bytecode

As preparation for code analysis techniques like code skeletons, signature extraction, and control flow graphs, we decompose the bytecode of contracts into code, data, and meta-data sections, as otherwise parts of the bytecode may be misinterpreted. Apart from the proper contract code at the beginning, the bytecode may contain the code of further contracts to be deployed as well as literals. Moreover, the Solidity compiler adds meta-data with information on the source code and the compiler version. Meta-data may be followed by constructor arguments. Some bytecodes consist of more than 40 sections with as many as 14 meta-data parts.

The decomposition takes place in three stages. First, meta-data can be unambiguously detected as CBOR encoded mappings that contain one of the keys `bzzr0`, `bzzr1` or `ipfs`. Second, the byte strings before, between and after meta-data, are split at instruction sequences that are characteristic for the start of a new contract; they are marked as code. Finally, the parts after meta-data that do not start with a characteristic sequence are labeled as constructor arguments.

Evaluation. To validate our method, we count the number of good and bad jumps. For each instruction `JUMP(I)` preceded by a `PUSH` of the target address, we determine whether the target instruction is `JUMPDEST` (good jump) or not (bad jump). Bad jumps raise an exception that reverts the entire transaction, so they are used only infrequently in regular code. If, on the other hand, the sectioning algorithm determines the start of a code section incorrectly, then virtually all jumps will be bad jumps. We find that our decomposition heuristics works correctly for 99.9% of the bytecodes. The first code section, relevant for extracting function signatures (see below), is faulty for only 0.03% of the bytecodes. Among others, the faulty cases are 'contracts' that are actually data repositories for other contracts and are not meant to be executed.

7.3 Extracting function signatures

When calling a contract that adheres to the standard for abstract binary interfaces (ABI), the first four bytes of the call data identify the function to be executed. The contract compares these bytes to the signatures of the implemented functions and branches to the respective code. To aid code analysis, tools like Mythril[3] identify heuristically byte sequences involved in comparisons, look them up in a database and, if successful, annotate the code with the function header found. Since a function header exists, chances are high that the byte sequence indeed is a signature.

Our goal is different, as we want to reconstruct the interfaces reliably, regardless of whether signatures correspond to known function headers or not. We need to avoid that arbitrary data or signatures of other code sections are mistaken as part of the interface. Therefore, we identify the first code section of the contract and then apply algorithm 1. It uses eight pairs of regular expressions, where the first expression in each pair locates the code that reads the call data, and the second one is applied repeatedly to extract the signatures from the comparisons. Tables 1 and 2 show one such pair.

Algorithm 1 Extracting signatures from bytecode.

```
reData_1 := ⟨RE for pushing first four bytes of calldata on stack⟩
reSig_1 := ⟨RE for comparing four bytes to signature, returning latter⟩
… 7 more pairs (reData_i, reSig_i) …

function ExtractSignatures(code)
    code = RemoveDataSection(code)
    sigs = ∅
    for (reData, reSig) in [(reData_1, reSig_1), ...] do
        c = code
        if reData matches c then
            c = RemoveMatchedPart(c)
            while reSig matches c do
                sigs = sigs ∪ {⟨signature returned by reSig⟩}
                c = RemoveMatchedPart(c)
        if sigs ≠ ∅ then
            break
    return sigs
```

Evaluation. We evaluated the algorithm on the bytecodes of 81,000 verified contracts from Etherscan, using the ABIs

[3] http://github.com/ConsenSys/mythril.

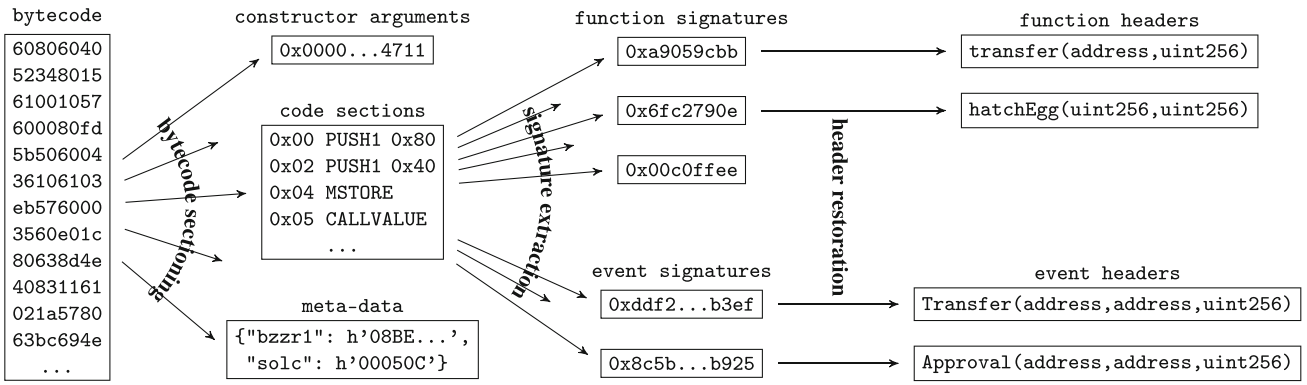

Fig. 1 Interface reconstruction: after decomposing the bytecode into code sections, data and meta-data, we extract the function and event signatures from the first code segment. Using dictionaries with known headers, we are able to restore most function and event headers

Table 1 One of the regular expressions $reData_i$. It specifies 44 equivalent code fragments that push the first four bytes of the calldata on the stack. PUSH 2^{224} is shorthand for a reg.exp. describing five ways of putting the constant 2^{224} on the stack. Question marks with the same index denote elements that are simultaneously present or missing

(PUSH 0xffffffff)?$_1$
(PUSH 2^{224}, PUSH 0x00, CALLDATALOAD, DIV
| PUSH 0x00, CALLDATALOAD, PUSH 2^{224}, SWAP1, DIV
| PUSH 0x00, CALLDATALOAD, PUSH 0xe0, SHR
), (PUSH 0xffffffff, AND)?$_2$, (AND)?$_1$

Table 2 One of the regular expression $reSig_i$. It specifies two equivalent code fragments that compare *signature* to the top of the stack and, on equality, jump to *offset*

(PUSH *signature*, DUP2 | DUP1, PUSH *signature*), EQ, PUSH *offset*, JUMPI

listed by Etherscan as ground truth. The signatures extracted by our tool differed from the ground truth in 71 cases. We verified manually that our tool was correct also in these cases, whereas the ABIs on Etherscan did not faithfully reflect the signatures in the bytecode (e.g., due to compiler optimization or library code). The validation set consisted almost exclusively of bytecode generated by the Solidity compiler (covering most of its releases), with just a few samples of LLL and Vyper code. We therefore regard the validation as representative of the 12.5 M deployed contracts generated by the Solidity compiler.[4]

Another group of contracts consisted of 11.4 M short contracts, mainly gasTokens, but also proxies (contracts redirecting calls elsewhere) and contracts involved in attacks. They do not have entry points, and our algorithm also does not detect any. The same holds for a third large group of 4.2 M contracts that self-destruct at the end of the deployment phase.

After subtracting these groups from the total of 28.1 M, we are left with 3.1 k contracts (732 skeletons). For these, our tool shows an error rate of 8%, extrapolated from a random sample of 60 skeletons that we manually checked. This amounts to an error rate below 10^{-5} in relation to all deployments.

7.4 Extracting event signatures

On source code level, so-called events are used to signal state changes to the world outside the blockchain. On machine level, an event is implemented as the instruction LOG with the unabridged Keccak-256 hash of the event header as identifier. We currently lack a tool that extracts the event signatures as efficiently and reliably as the function signatures. The instructions LOG and its arguments are harder to detect, as they are distributed throughout the code. We can check, however, whether known event signatures occur in the code

[4] Deployed code generated by `solc` can be identified by the first few instructions. It starts with one of the sequences 0x6060604052, 0x6080604052, 0x60806040818152, 0x60806040819052, or 0x60806040908152. In the case of a library, this sequence is prefixed by an instruction PUSH followed by 0x50 or 0x3014.

section of the bytecode, as the 32-byte sequences are virtually unique. This heuristic fails in cases where the event is actually implemented in another contract that is called via `DELEGATECALL`, where the signature is kept in the data section, or where the signature is missing in our collection of 58 k event signatures. In spite of that, event signature extraction performs reasonably well: Evaluating the method with the most frequent *Transfer* event and the 81 k source codes from Etherscan yields less than 0.2 k mismatches.

7.5 Header restoration

To understand the purpose of a contract, we try to recover the original function and event headers from the signatures. This reverse translation of the (partial) hashes is accomplished with a database of headers (plain text) with corresponding signatures (hash). We use our own collection[5] of signatures that extends the verified contracts of the main chain by signatures from test nets as well as from 600 projects we found on GitHub.

For event signatures, we always succeed, as the method in the last section detects the signatures of only those 58 k event headers that we have collected from various repositories. There are no ambiguities, since the signature is a 256-bit cryptographic hash of the header.

Our method for extracting function signatures, on the other hand, will detect any signature. Up to block 10.5 M, a total of 312 k different signatures was in use. Over time we have collected 402 k function headers with their signatures. By using this dictionary in reverse, we are able to restore 60% of the extracted signatures. Taking the deployment frequency into account—some signatures are used more frequently than others—the ratio rises to 90%. In contrast to the event signatures, we may encounter collisions for the 32-bit signatures of function headers. These are rare, however: Of the 402 k signatures, only 27 occur with a second header in the dictionary.

For example, when extracting the function signatures by applying Algorithm 1 on the bytecode of the address `0x776f55fa27644705156a46e8c1b2dc28ca122832` created in block 268036, we obtain the signatures `0x41c0e1b5`, `0x6b590248` and `0xecfc0073`. Our dictionary translates the first two signatures to the headers `kill()` and `getDigit()`, whereas the header for `0xecfc0073` remains unknown.

[5] https://ethereum.logic.at/dictionary.

Table 3 Full compliance of deployed token contracts

Standard	Deployments	Bytecodes	Skeletons
ERC-all	221,232	114,654	40,144
ERC-20	214,528	112,606	38,439
ERC-721	6588	1983	1642
ERC-777	312	141	113
ERC-1155	125	73	71
ERC-1462	3	3	3
ERC-1450	0	0	0
ERC-1644	36	8	8

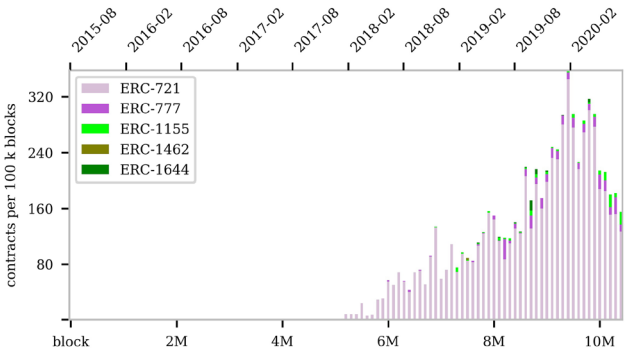

Fig. 2 Creation of ERC-compliant token contracts other than ERC-20. The lower horizontal axis indicates the Ethereum blocks, while the upper axis shows the corresponding dates. Each bar represents a bin of 100,000 blocks (corresponding roughly to 2 weeks)

8 ERC-tokens over time

In this section, we show the results of extracting interfaces from all deployed bytecode on Ethereum for identifying ERC-compliant token contracts.

A contract is called fully ERC-compliant when providing at least one of the standard interfaces mentioned in Sect. 4. Table 3 lists the number of compliant token contracts, including the numbers of unique bytecodes and skeletons. With over 214 k deployments (97%), ERC-20 is by far the most commonly used standard. The remaining 3% are almost exclusively contracts adhering to the ERC-721 standard for non-fungible tokens. The other standards are deployed in small numbers. A few token contracts comply with more than one standard and are counted more than once. In total, we count 221 k fully ERC-compliant token contracts.

Figure 4 in Sect. 11 shows the deployment of ERC-20 compliant tokens on a time line, while the contracts complying with other standards are depicted in Fig. 2. The mass deployment of ERC-20 tokens started in the middle of 2017, peaked in the first half of 2018 and later stabilized at about 1000 deployments per week. The deployment of ERC-721 compliant contracts started in 2018, with the numbers rising steadily to 150 deployments per week at the beginning

of 2020. Since then, the numbers have fallen to 60 deployments per week. The other standards start to appear in small numbers at the beginning of 2019.

9 Identification of non-compliant tokens

In this section, we focus on methods for identifying non-compliant token contracts. In order to be able to evaluate these methods, we first need to clarify the notion of tokens and token contracts. Based on our definition of tokens, we compile a list of contracts that we manually classify as tokens or non-tokens, serving as a ground truth for the evaluation. Then, we discuss four indicators regarding their potential effectiveness in detecting token contracts. The indicators rely on the bytecode techniques described in Sect. 7, on message statistics as well as on some additional techniques described with the indicators below.

9.1 When is a contract a token contract?

Related work. Oliveira et al. [30] introduce several token archetypes that go beyond the common distinction into security, utility, and payment token. The semantic characterization of the numerous types demonstrates that understanding the purpose of a token involves more factors than just the code. Moreover, the level of code analysis required for most distinctions is not readily automatable.

Chen et al. present the tool TokenScope [6] that monitors transfer events, transfer calls as well as changes to the token balance in storage. Whenever any two of them differ, the contract is flagged as behaving inconsistently with regard to the ERC-20 standards. The concept of token contract is closely tied to the standard.

Lambert et al. [26] put a focus on security token offerings (STOs) as opposed to initial coin offering (ICOs) and clarify how security tokens differ from both utility and payment tokens. According to them, 'a security token is a digital representation of an investment product, recorded on a distributed ledger, subject to regulation under security laws.'

Darisi et al. [9] propose mechanisms for the exchange of tokens within and between blockchains. They characterize tokens by the basic parameters name, symbol, initial supply, decimals, and fungibility.

Our Definition of token contracts. Our aim is to develop criteria that enable us to determine whether a contract can be considered a token or not. The criteria should neither be too abstract as we need to apply them to code, nor should they refer to particular standards.

The main functionality of token contracts comprises the maintenance of a ledger that records token holdings and the ability to change token ownership by modifying the ledger. The change of ownership may take different forms, including simple transfers initiated by the owner, safe transfers where the recipient has to claim the approved tokens, the distribution of tokens via airdrops, and the trading of tokens for other crypto-assets. Additionally, token contracts may implement features like administrative roles, token locking, contract halting, and getters/setters for state variables.

As a minimum requirement, a token contract has to satisfy the following criteria:

- Bookkeeping: The token contract maintains a ledger that maps the id of token holders (e.g., their addresses) to the tokens they own.
- Token flow: The token contract provides functionality to transfer tokens between holders, to trade tokens for crypto-assets, and/or to consume tokens.

Whether and in which way the values in a ledger represent tokens depends on the code semantics. In our manual assessment of contracts, we encountered only a small number of borderline cases.

Table 4 Contracts for the evaluation of the indicators (ground truth)

Type	Deployments	Bytecodes	Skeletons
Non-compliant			
Etherscan	14,877	644	270
Manual	943	148	63
Non-token			
Wallet	5,956,205	1885	752
Manual	435,915	1727	19

9.2 A ground truth for token contracts

To evaluate the indicators below, we compile a collection of contracts, or rather bytecodes, for which we determine whether they are non-compliant (not fully ERC-compliant) token contracts or non-tokens. Table 4 provides an overview of this collection.

Etherscan offers a list of several thousand contracts labeled as tokens, of which we selected the non-compliant ones. In prior work [12], we identified numerous wallet contracts. These are interesting, as they interact with token contracts and sometimes include functions similar to token contracts. For the manually classified samples, we selected the bytecodes of frequently deployed or called contracts as well as a random assortment of less active contracts that share some function with the ERC-20 standard.

Limitations. It remains unclear how Etherscan identifies token contracts. We asked the maintainers about the criteria, but have not yet received an answer. Among the manually classified contracts, there are some tokens that fit our defini-

tion but with hardly any similarities to standard tokens. For example, in the lottery game 'Fomo3DSoon', a player buys 'keys' that later can be exchanged for the reward. We therefore expect that none of our simple indicators will be able to recognize such tokens.

9.3 Indicator I_1: single ERC-20 signatures

As interfaces are collections of signatures, we evaluate the predictive power of single ERC-20 signatures. Table 5 lists the frequencies of the most common ones. The first nine signatures are precisely the mandatory and optional functions of the ERC-20 standard. The next two signatures are used in all sorts of contracts (including tokens) to manage ownership, while the last three are again related to tokens.

Limitations. This approach has the same issues as the interface method in Sect. 7.

9.4 Indicator I_2: multiple ERC-20 signatures

Non-compliant token contracts often implement at least some of the mandatory functions. Hence, we investigate subsets of the ERC-20 interface and attempt to find a threshold.

Clearly, transfer functions play a central role, as is documented by both Tables 5 and 6. Table 6 lists the most frequently called signatures, with the function `transfer` being by far the most common. A variant of the transfer function is listed as the third most common.

Even though the function `transfer` is essential, it is not specific to tokens contracts. Therefore, we investigate its interplay with the other ERC-20 functions. Table 7 lists the number of contracts that provide a subset of the six mandatory ERC-20 signatures. We differentiate the numbers according to the presence or absence of the function `transfer` and indicate the actual deployments on-chain, the corresponding unique bytecodes, and the respective skeletons.

Table 7 shows that 224.5 k deployed contracts implement the mandatory ERC-20 interface. Moreover, there are 211.2 k contracts that provide only the function `transfer` but none of the other mandatory ERC-20 functions. These contracts are remarkably uniform as the small set of just 660 bytecodes and 579 skeletons shows, corresponding to a code reuse factor of several hundreds. This hints toward factory-produced non-token contracts (e.g., wallets) implementing a function `transfer`.

The numbers in Table 7 do not suggest a threshold for the number of functions in order to detect tokens. At the same time, the number of contracts implementing two to five mandatory functions is nonnegligible. For this indicator, we therefore compare different thresholds for the number of ERC-20 signatures, with and without the transfer function.

Limitations. This approach has the same issues as the interface method in Sect. 7. Additionally, it hinges on the threshold.

9.5 Indicator I_3: contract name

For some deployed contracts, the source code has been uploaded to Etherscan (cf. Sect. 6). In these cases, the name of the contract assigned by the developers may reveal its purpose. This indicator considers all bytecodes for which any corresponding source code has a contract name that ends with 'token' or 'coin' (case insensitive).

Table 8 lists the number of deployed contracts and respective bytecodes that we can associate with a name from Etherscan. In the first line, the high number of over 3 M deployments with associated names mainly results from wallets, since a high factor between bytecodes and deployments is atypical for tokens, but typical for wallets [12].

The last line in Table 8 shows that there is a substantial number of deployments (and bytecodes) that are not ERC-compliant but where the contract name ends with 'token' or 'coin.' Therefore, this indicator seems worth to be investigated.

Limitations. Even though token contracts are more likely to have their source code on Etherscan (as a means of building trust), the source and thus the name of many contracts is not available. Moreover, this approach misses tokens named differently or may yield false positives.

9.6 Indicator I_4: transfer events

Token standards usually require compliant contracts to emit an event when transferring tokens. It indicates the affected token contract as well as the sender and receiver. Thus, events may help to identify token contracts.

We use two approaches to associate tokens with events. First, we search the bytecode for the signature of relevant events (see Sect. 7.4 for details). Secondly, we search the log entries for events that actually happened. Both methods complement each other, as events overlooked by static extraction (e.g., because of proxying) show up as log entries at their first use, whereas extraction detects events even if they have not been emitted so far.

Indicator I_4 considers bytecodes that contain the signature of the event *Transfer(address,address,uint256)* or if one of the deployments of the bytecode actually emitted this event. This particular event accounts for 61% of the 710 M log entries and signals that the number of tokens given as third argument has been transferred from the first to the second address. All standards in Sect. 4 require this event, except for ERC-1155 that replaces it by *TransferSingle* and *TransferBatch*.

Table 5 Top signatures ranked by the number of bytecodes they appear in. The six functions mandated by ERC-20 are kept in bold, the three optional ones in italic

Signature	Header	Bytecodes
70A08231	**balanceOf(address)**	131,254
06FDDE03	*name()*	128,363
18160DDD	**totalSupply()**	126,082
95D89B41	*symbol()*	125,933
A9059CBB	**transfer(address,uint256)**	125,862
313CE567	*decimals()*	121,825
23B872DD	**transferFrom(address,address,uint256)**	117,577
095EA7B3	**approve(address,uint256)**	117,150
DD62ED3E	**allowance(address,address)**	116,972
8DA5CB5B	owner()	116,480
F2FDE38B	transferOwnership(address)	87,924
CAE9CA51	approveAndCall(address,uint256,bytes)	40,046
42966C68	burn(uint256)	37,525
79CC6790	burnFrom(address,uint256)	23,602

Table 6 Top signatures ranked by their frequency in calls

Signature	Header	Calls
A9059CBB	transfer(address,uint256)	314,764,683
70A08231	balanceOf(address)	68,465,404
23B872DD	transferFrom(address,address,uint256)	45,083,909
18160DDD	totalSupply()	21,550,045

Table 7 Implemented ERC-20 functions with and without transfer

Sigmatures	Deployments		Bytecodes		Skeletons	
	Incl. transfer	Excl. transfer	Incl. transfer	Excl. transfer	Incl. transfer	Excl. transfer
6 of 6	214,528	–	112,606	–	38,439	–
5 of 6	3975	88	2274	70	1684	59
4 of 6	3730	6538	2881	1977	873	1647
3 of 6	9753	1288	5217	692	2950	517
2 of 6	3601	3034	2163	709	944	591
1 of 6	211,239	31,766	660	3598	579	2612

Limitations. This approach misses contracts if they do not implement the event, or if the signature cannot be detected in the code and the event is never emitted because the contract remains unused. In rare cases, non-token contracts use this event for other purposes.

According to the token standards, both addresses are indexed, whereas the token amount is added as further data. Some token contracts choose other indexing schemes, leading to ambiguities in the interpretation of log entries. Moreover, a few contracts do not use regular Ethereum addresses but idiosyncratic addresses. Both situations do not occur with fully compliant tokens.

Table 8 Contracts with associated names

Contract with	Deployments	Bytecodes	Skeletons
Name exists	3,531,310	90,090	50,444
Name ends with ‘token’	113,315	22,195	11,249
Name ends with ‘coin’	7680	5233	2673
Token/coin and non-ERC	48,467	1627	1132

Table 9 Indicator I_1: single ERC-20 signatures

Header	Precision	Recall
allowance(address,address)	100%	26%
approve(address,uint256)	100%	23%
balanceOf(address)	66%	71%
decimals()	59%	50%
name()	22%	70%
symbol()	65%	70%
totalSupply()	**100**%	**68**%
transfer(address,uint256)	64%	64%
transferFrom(address,address,uint256)	98%	24%

Table 10 Indicator I_2: multiple ERC-20 signatures, in three varieties: unrestricted selection, always including and always excluding the simple transfer function. '6 of 6' does not apply to non-compliant contracts. None of the samples in the ground truth satisfies '5 of 6 signatures excl. transfer'

Threshold	Any signature		Incl. transfer		Excl. transfer	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
5 of 6	100%	20%	100%	20%	–	–
4 of 6	100%	45%	100%	41%	100%	24%
3 of 6	**100**%	**67**%	100%	61%	100%	47%
2 of 6	67%	71%	64%	63%	**99**%	**69**%
1 of 6	66%	72%	64%	64%	67%	71%

Table 11 Indicator I_3: Contract name

Names ends with	Precision	Recall
'Token'	99%	9%
'Coin'	100%	2%
'Token' or 'coin'	**99**%	**11**%

10 Comparison of indicators for non-compliant tokens

In this section, we first compare the effectiveness of the indicators for non-compliant token contracts on our token ground truth (TGT), measured by precision and recall. Then, we discuss combinations of indicators.

Let *tp* (true-positive) denote the number of positive TGT instances classified correctly as a token, let *fp* (false-positive) be the number of negative TGT instances classified wrongly as a token, and let *fn* (false-negative) be the number of positive TGT instances classified wrongly as a non-token. *Precision* is computed as the quotient $tp/(tp + fp)$. It is the ratio of token contracts correctly identified to all contracts identified as token. A precision value close to one means that the number of non-tokens mistaken as tokens is small; if the indicator classifies a bytecode as a token, then it most likely is one. *Recall* is computed as the quotient $tp/(tp + fn)$. It is the ratio of token contracts correctly identified to all token contracts. A recall value close to one means that the number of positive instances not recognized as tokens is small; if the indicator is applied to a token contract, then it is most likely classified as such.

10.1 Indicator I_1: single ERC-20 signatures

In Table 9, we list precision and recall for the indicator that tests for the presence of a specific ERC-20 signature in the interface of a bytecode.

Only four ERC-20 signatures with values of about 100% are sufficiently precise to serve as indicator. Of these, only the function `totalSupply` is able to detect the majority (68%) of non-compliant tokens in the ground truth. Thus, the indicator 'implements `totalSupply()`' is distinctive, even though the function is not present in about a third of the tokens contracts of or our sample.

The low precision of the three optional functions `name`, `symbol`, and `decimals` stems from the fact that they also appear in wallets and thus have a low specificity.

10.2 Indicator I_2: multiple ERC-20 signatures

In Table 10, we list precision and recall for indicators that test whether the number of signatures that an interface shares with the ERC-20 standard is above a given threshold. Due to the significance of the transfer function, we consider also the variant where the transfer function has to be among the shared ones as well as the one where the transfer function is omitted when counting the overlap.

Two indicators stand out, the one testing for the presence of at least three out of six ERC-20 functions and the one with a threshold of two out of five functions (with `transfer` excluded). Both have a precision close to 100% and a recall of almost 70%. The other indicators are either far worse in precision or in recall.

The reason for the slightly better recall in the absence of the function `transfer` may lie in its low specificity.

10.3 Indicator I_3: contract name

In Table 10, we list precision and recall for the indicator that tests for specific endings in the contract names in the source code.

All variants show a high precision: A contract called coin or token is indeed a token. Recall is poor, as we do not have associated source code for most bytecodes. Moreover, even

Table 12 Indicator I_4: Transfer events

Transfer event	Precision	Recall
Is implemented	99%	66%
Has been emitted	99%	68%
Either of the two	**99%**	**80%**

a token contract with available source code may have a non-descriptive name.

Because of its high precision, however, this indicator may still be helpful when combined with others.

10.4 Indicator I_4: transfer events

In Table 12, we list precision and recall for the indicator that considers the event *Transfer(address,address,uint256)* in the bytecode or among the log entries.

The high precision shows that this event is indeed typical of tokens. The two ways of detecting the event apparently complement each other, as their combination shows a significantly better recall.

10.5 Combination of indicators

After having analyzed the four indicators individually, we look for combinations that reduce the number of false negatives and positives even better. Table 13 ranks the best individual indicators from above as well as the best combinations that we found.

As demonstrated above, the best single indicator is not related to function signatures, but to events. It can be improved by a few percent when using it in conjunction with one of the other top indicators, like a test for the function `totalSupply`. Adding even more indicators may increase the recall at the price of lowering precision.

One may wonder about the remaining 16% token contracts that go undetected. A few of them are the manually selected samples that conceptually are tokens but that bear no resemblance with the ERC standards. To detect such contracts, we need more sophisticated methods that analyze the code. The majority of undetected 'tokens' are contracts labeled as such by Etherscan. Closer inspection of random samples reveals that these contracts are in fact not tokens. As future work, we will have to clean the data to get a better picture.

10.6 Non-compliant tokens

Based on the evaluation of indicators above, we regard a contract as a non-compliant token if it complies with none of the ERC standards, but has a transfer event in the bytecode or the log entries or shows at least two of five ERC-20 signatures (ignoring `transfer(address,uint256)`) in its interface.

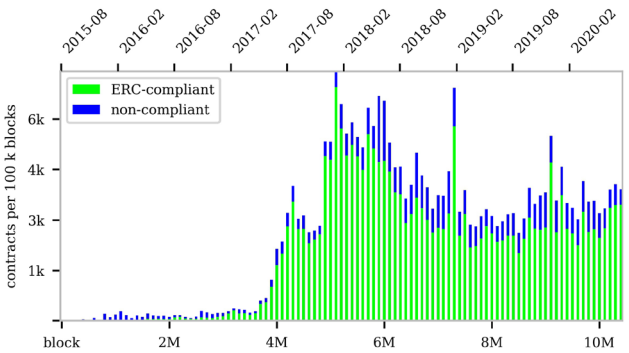

Fig. 3 Deployment of all token contracts, differentiated into ERC-compliant ones in green and non-compliant ones in blue. The lower horizontal axis indicates the Ethereum blocks, while the upper axis shows the corresponding dates. Each bar represents a bin of 100,000 blocks corresponding roughly to 2 weeks (color figure online)

Figure 3 depicts the creation of 221 k (81%) compliant and 51 k (19%) non-compliant tokens over time. Both groups show the same level of activity: 648 M (82%) of the messages are related to compliant tokens, and 141 k (18%) to non-compliant ones. Together, tokens are responsible for 40% of the total message volume on Ethereum.

The high number of non-compliant tokens may come as a surprise. While it took a while in the beginning for ERC-20 to be finalized and adopted, we still see many newly deployed non-compliant tokens. It should be noted that for a token to be usable, not all features of a standard are needed (e.g., `approve, transferFrom, allowance`).

11 Purity of token contracts

In this section, we focus on the distinction between security and utility tokens. As laid out in section 3, a utility token should provide some service or product for the token holder. Our aim is to detect the absence of such a service or product in the code of a token contract as an indicator for a potential security token.

We approach the task by assessing whether a token contract implements functionality beyond token and user management. Our heuristic method uses a set of pattern-based rules to partition the signatures of an interface into the three groups 'token-related', 'neutral' and 'other' (see below for details).

Definition of Purity. We call a token contract *pure* if its interface consists exclusively of functions that our algorithm classifies as 'token-related' or 'neutral.' For a pure token contract, our method finds no evidence that it offers a genuine service or product on-chain; it thus may be a security token. Non-pure tokens, on the other hand, are more likely to be

Table 13 Combination of indicators for non-compliant tokens. Legend: 'coin/token' stands for 'name ends with coin or token'; '$\geq$ 3 sigs' for 'at least three ERC-20 signatures'; 'totalSupply' and 'approve' for the occurrence of the respective function in the interface; '$\geq$ 2 sigs excl transfer' for 'at least two ERC-20 signatures, but not `transfer(address,unit 256)`'; and 'ETransfer' for a transfer event in the bytecode or the log entries

	Indicator	Precision	Recall
I_3	coin/token	98.9%	11.3%
I_2	$\geq$ 3 sigs	100.0%	67.0%
I_1	totalSupply	100.0%	68.0%
$I_1 + I_1$	totalSupply or approve	100.0%	68.9%
I_2	$\geq$ 2 sigs excl transfer	99.3%	69.0%
I_4	ETransfer	99.1%	80.0%
$I_3 + I_4$	ETransfer or coin/token	98.9%	81.9%
$I_2 + I_4$	ETransfer or $\geq$ 3 sigs	99.1%	83.9%
$I_1 + I_4$	ETransfer or totalSupply	99.1%	84.1%
$I_2 + I_4$	ETransfer or $\geq$ 2 sigs excl transfer	99.1%	84.2%

utility tokens, implementing a service by means of the 'other' functions.

Limitations. This approach considers contracts in isolation, disregarding companion contracts and off-chain components. A pure token might in fact be part of a decentralized application that, as a whole, offers a service. A comprehensive assessment requires a manual analysis of the context in which the token contract operates.

Algorithm 2 Classifying a function header wrt. its purpose.

```
REIN_1 := ⟨regular exp. for inclusion of header⟩
REEX_1 := ⟨regular exp. for exclusion of header⟩
LABEL_1 := ⟨label identifying class⟩
… 21 more triples (REIN_i, REEX_i, LABEL_i) …
function CLASSIFYHEADER(header)
    classification = ∅
    for (rI, rE, l) in [(REIN_1, REEX_1, LABEL_1), ...] do
        if rI matches header and rE does not match header then
            classification = classification ∪ {l}
    return classification
```

11.1 Grouping function headers

A precise classification of function headers in large quantities would require an automated code analysis that checks for semantic properties, which is a difficult problem and not yet adequately solved. Instead, we present a heuristic test that is based on the observation that the functions of token interfaces can be categorized into the following three groups.

The token-related group comprises the core functions mandated by the standards, as well as related functions to create, destroy, and distribute tokens. The second group contains the neutral functions that can appear in any type of contract, like getters and setters for public variables or role management. The third group consists of the remaining functions, which may rely on tokens but are not necessary for operating them.

Algorithm 2 classifies functions according to their name. For a given header, it repeatedly applies rules like those in Table 14. Each rule consists of an inclusion pattern, an exclusion pattern, and a label. If the inclusion but not the exclusion pattern matches the function header, then the header is tagged with the label.

For our proof of concept, we specified 22 rules that divide headers into 17 categories. The categories *token*, *distribution*, *auction*, *minting*, *approval*, *kyc*, *ico*, *transfer*, *crowdsale*, *airdrop*, and *burning* determine the group of 'token-related' headers, whereas the categories *control*, *math*, *getter*, *setter*, *trading*, and *roles* constitute the 'neutral' group. Function headers without a tag as well as signatures without restored header form the 'other' group.

In general, we cannot expect to understand the purpose of a contract by just looking at the names of its functions. However, the names in the first and second group are quite uniform and stereotypical as the functions perform standardized tasks. Therefore, this heuristic seems worthwhile in the context of token contracts.

Limitations. The effectiveness of this method hinges on the careful choice of the rules. Moreover, for the first and second group, the method assumes that names of functions indicate the implemented functionality. Finally, 6% of ERC20-compliant tokens delegate some function calls to another contract (like a library) such that the signatures extracted from the contract represent only parts of the interface. To simplify our analysis, we neglect such contracts.

11.2 Share of pure token contracts

When applying the semantic classification of the function signatures as described above, we arrive at a share of pure contracts a listed in Table 15 and depicted in Fig. 4.

Regarding the function signatures in the 215 k deployed ERC-20 tokens, we find 59 k distinct signatures, of which we can decode 45 k to function headers. Our algorithm classifies 37 k headers as 'token-related' or 'neutral', while 8 k remain

Table 14 Three of 22 classification rules

Rule	REIN_i	REEX_i	LABEL_i
1	`(get|is|total|balance)'`	`'issue'`	`'getter'`
	If header starts with 'get', 'is', 'total', or 'balance', but not with 'issue', then label it as 'getter.'		
2	`'ico'`	`'unicorn|icoin'`	`'ico'`
	If header contains 'ico' but neither 'unicorn' nor 'icoin', then label it as 'ico'-related.		
3	`'icoinfo'`		`'ico'`
	If header contains 'icoinfo', then label it as 'ico'-related.		

Table 15 Purity of ERC-20 compliant token contracts

# other functions	Bytecodes	Deployments	Received calls
= 0 (pure)	84,639	159,239	268,909,820
> 0	27,967	55,289	243,520,411
all ERC-20	112,606	214,528	512,430,231

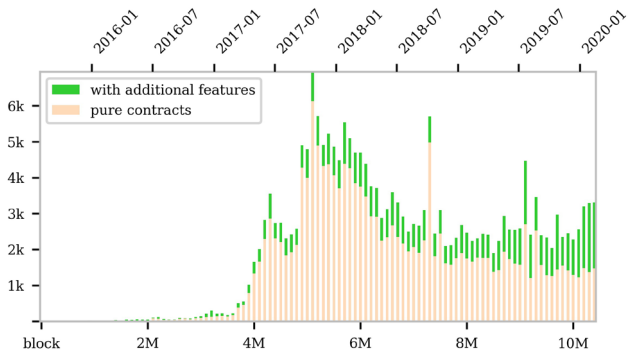

Fig. 4 Deployment of ERC-20 token contracts, divided into pure (peach) and non-pure (green) tokens. The lower horizontal axis indicates the Ethereum blocks, while the upper axis shows the corresponding dates. Each bar represents a bin of 100,000 blocks corresponding roughly to 2 weeks (color figure online)

untagged. The latter form the 'other' group, together with the 14 k signatures that we cannot decode.

Based on this classification of signatures, we distinguish ERC-20 token contracts with respect to their purity and list in Table 15 the respective numbers of bytecodes, deployments and received calls. Interestingly, 85 k distinct bytecodes (corresponding to 159 k deployments) implement only token-related and neutral functions. According to our definition, they are pure tokens that do not implement a recognizable service or product and therefore could be security tokens.

The remaining 28 k bytecodes (55 k deployments) implement also functions from the 'other' group. For the 22 k signatures in this group, it is not apparent how to decide automatically whether the corresponding code offers a genuine service or product.

For a temporal perspective, Fig. 4 shows the deployment of pure and non-pure tokens over time. Most of the time, a vast majority of deployed tokens is pure. Only since the end of 2019, non-pure tokens begin to dominate.

12 Purity for sample tokens

In this section, we look at several examples of tokens to evaluate the purity approach qualitatively. We searched for complaints, litigation, and press releases from the US Securities and Exchange Commission (SEC)[6] that concern Ethereum tokens. The summary of the settlements underlines the importance of clarifying the type of a token (for which we presented first steps).

12.1 Ethereum tokens and the SEC

The SEC considered the tokens in Table 16 as securities violating the Securities Act, with the exception of TKJT and Q2, for which it issued a no-action letter.

Table 16 shows the number of token-related, neutral and other functions for these tokens. From Etherscan, we included the number of token holders, token transfers, and the fully diluted market cap on October 17, 2020. For many security tokens, the number of 'other' functions is zero, as we would expect.

For tokens with 'other' functions, Table 17 lists the restored headers. For CTR, all function names refer to cards, which fits the token's purpose as a financial service. For BOON, ICOS, PRG, and XD, the headers indicate token-related or administrative (neutral) functionality. A refined set of rules might classify these cases correctly. The remaining three cases are inconclusive, either because the function header cannot readily be interpreted or because the signatures cannot be restored.

Regarding the non-security tokens TKJT and Q2, we expect the number of other functions to be greater than zero. This is only the case with Q2, which is linked to playing video games. TKJT (linked to air charter services) does not provide a service or product on the chain. Even though we have

[6] https://www.sec.gov/search/search.htm.

Table 16 Ethereum tokens assessed by SEC

Token	Related	Neutral	Other	Holders	Transfers	Market cap (USD)
AGL	11	6	0	1	0	0
AIR	16	14	0	3792	20,748	2203 k
B2G	10	2	0	1	1	0
BLV	12	0	0	140	309	0
BOON	15	6	1	1630	3802	7 k
BQ	9	4	0	20,823	205,214	35 k
CAT	19	5	0	220,353	292,583	430 k
CTR	13	22	**6**	15,401	97,196	1648 k
EOS	11	8	**2**	330,689	3,570,224	0
FLIK	15	6	0	676	5285	173 k
FMT	6	2	1	91	123	0
GLA	12	5	0	9550	23,721	552 k
HLTH	16	6	0	0	0	0
ICOS	11	1	1	739	10,278	249 k
KIN	14	3	0	49,140	504,130	46,900 k
MUN	16	2	0	0	0	0
OPP	10	1	0	1	1	0
PLX	12	1	0	22,672	95,648	5010 k
PRG	10	3	1	7573	44,216	470 k
Q2	17	11	**5**	299	511	0
SHOP	17	19	**19**	2638	3174	0
TKJT	16	0	0	1	2	0
TON	12	8	0	5	6	0
UKG	11	5	0	9870	46,421	1,267 k
VERI	10	1	0	22,774	266,425	249,536 k
XD	22	20	1	1248	13,889	102 k
ZWC	10	1	0	4	6	0

no information on the originally planned off-chain services around TKJT, the lack of 'other' functions shows that the smart contract does not provide any support in this respect.

12.2 Settlements and orders

The information in this section is a terse summary taken from the collected SEC documents and offers a glimpse into the downside of the token world.

Argylecoin (AGL) continued a Ponzi scheme run by Natural Diamonds in 2017 and was charged with fraud in 2019.

End of 2018, AirToken (AIR) settled with the SEC for the ICO in 2017 by returning funds and paying USD 250 k penalty.

In August 2020, the SEC ordered Boon Tech (BOON) to pay USD 150 k penalty and disgorgement of USD 5 M plus prejudgment interest of USD 600 k for the ICO in 2017/2018.

Bitqy token (BQ) refers to a market place. In 2019, the SEC settled with Bitqyck after an alleged fraud.

Table 17 Security tokens and the functions classified as 'other'

token	other	header
BOON	1	allocations(address)
CTR	**6**	cards_gold(uint256)
		cards_start(uint256)
		cards_titanium(uint256)
		cards_blue(uint256)
		cards_black(uint256)
		cards_black_check(address)
EOS	**2**	push(address,uint128)
		pull(address,uint128)
FMT	1	no translation
ICOS	1	approve(address,uint256,uint256)
PRG	1	approve(address,uint256,uint256)
SHOP	**19**	removeShop(address)
		addShop(address)
		no translation for 17 signatures
XD	1	upgrade(uint256)

In May 2020, Bitclave (CAT) was ordered disgorgement of USD 25.5 M, prejudgment interest of USD 3.4 M, and a civil penalty of USD 400 k.

Centra token (CTR) refers to financial services. All three co-founders were indicted for fraud.

In 2018, the company behind EOS launched its own mainnet and was ordered to pay USD 24 M penalty for their ICOs from mid-2016 to mid-2017.

Gladius token (GLA) is linked to a DDoS protection service. The company reported itself and refunded the proceeds from the ICO in 2017 to avoid a fine from the SEC.

In 2019, SimplyVital (HLTH) settled with the SEC by returning all proceeds from the pre-sale to the investors and by not generating any tokens (as can be seen in Table 16).

In 2019, the SEC ordered ICOBox (ICOS) to pay disgorgement and prejudgment interest totaling over USD 16 M.

Kin token (KIN) is linked to a social media platform. In mid-2019, the token migrated to its own mainnet, while the SEC charged KiK Interactive with conducting a USD 100 M unregistered ICO in 2017.

In 2017, the SEC files charge against PlexCorps (PLX) to halt an alleged initial coin offering (ICO) fraud that raised up to USD 15 M.

In 2020, the SEC ordered Paragon (PRG) to pay penalty in the amount of USD 250 k.

In a settlement in 2020, UnitedData (SHOP) was ordered to pay USD 450 k for its fraudulent ICO.

In June 2020, the SEC released that Telegram (TON) had to return USD 1200 M to investors and pay USD 18.5 M penalty to settle SEC charges.

In 2020, Unikrn (UKG) settled with the SEC by being ordered to pay USD 6.1 M penalty.

In 2019, Veritaseum (VERI) was charged in fraudulent ICO and was ordered to pay nearly USD 9.5 M.

The company SoluTech (XD) is insolvent and has ceased business operations; it settled with the SEC paying USD 25 k penalty.

LongFin (ZWC) offered the SEC a settlement for a fraud and violation charge.

For the other security tokens, the SEC filed a complaint.

12.3 Services and games

As a further small ground truth, we analyzed several highly active token contracts, for which we present the results in Table 18. The upper four examples provide diverse services that are not reflected in the respective token contracts (as the number of other functions is 0 to 1). In contrast, the lower five examples are games that implement at least part of the application in the corresponding token contract (6 to 47 other functions). Games are a typical application category with a genuine service or product.

It should be noted that some applications implement the features in several interacting contracts that include a (plain) token contract, while the logic of the application is implemented in separate contracts. These tokens would show 0 other functions as well.

In summary, our approach is able to detect if a token contract implements nothing but token and account management. If the token is part of an application that claims to provide a product or service, it has to be found elsewhere, either in another contract or off-chain.

13 Conclusions

The overarching theme of this work is the identification and classification of token contracts, based on the publicly available transaction data of the Ethereum main chain. Unlike other work, we do not stop at ERC-20 compliance, but consider also other ERC standards as well as non-compliant tokens. We focus on the contract code rather than its activity, meaning that our analysis encompasses unused and top tokens alike. As 'token' is a fuzzy, semantic concept, our methods approximate it by characteristics that are readily accessible, like the signatures of the interface or log data. We even explore the extent to which this approach can be used to assess the type of a token.

Compliant Tokens. The basic standard for fungible tokens, ERC-20, is implemented by 97% of the tokens that comply with any of the ERC standards. The standard ERC-721 for non-fungible tokens accounts for the remaining 3%, except for a few hundred contracts (0.2%) that implement one of the other standards. Contracts complying with standards for security tokens are virtually non-existent, which may be due to unclear legal regulations. Interestingly, new token contracts keep being deployed at a rate of 200 per day, which leaves us wondering at their purpose.

Non-compliant Tokens. We developed feasible indicators to identify a great number of non-compliant tokens that add another 25% to the number of token contracts. They show a similar level of activity as the compliant ones, as judged by the number of messages. The total number of token contracts on Ethereum up to block 10.5 M thus amounts to 272 k.

Token Type. The distinction between security, utility, and payment tokens is essential for legal consequences. As our compilation of rulings concerning security tokens shows, the number of SEC complaints is rising, with settlements involving fines and refunds of millions of US dollars. To support the assessment of token types at large, we propose to divide the functions of contract interfaces into those implementing token-related, neutral and other functionality. We define a token to be pure if it offers exclusively token-related and neutral functions. Pure tokens amount to 70% of the compliant tokens. According to our hypothesis, pure tokens are

Table 18 Assessment of sample tokens (services and games)

Related	Neutral	Other	Name	Purpose
12	6	0	DxToken	platform for computing
16	6	1	Dragon	payment for entertainment
14	4	0	MANAToken	marketplace (Decentraland)
18	7	0	SNT	wallet app (Status)
9	7	13	Centurions	Crypto Rome
13	18	6	Etheremon	Ether monsters
11	4	7	CryptoSaga	RPG
10	10	47	HyperDragons	strategy battle game
18	7	46	Gods Unchained	eSports

more likely to be security tokens, whereas non-pure tokens tend to be utility tokens. As qualitative evidence, we found the hypothesis to be in line with the SEC filings.

Future Work. To improve the assessment of tokens, we need a better understanding of individual contracts and of the context in which they are embedded. On the one hand, we need tools for analyzing the bytecode of contracts semantically, e.g., to detect data structures and code related to token functionality. On the other hand, we need methods that detect groups of on- and off-chain components that form an application and have to be considered as interrelated entities.

Declarations

Funding Open access funding provided by TU Wien (TUW). The authors declare that they have no extra funding but their employment at the university.

Conflict of interest The authors declare that they have no conflict of interest. In particular, they are neither invested in Ethereum nor have ties to companies of its ecosystem.

Availability of data and materials We use publicly available transaction data from the main chain of Ethereum (see Sect. 6). Information on groups of smart contracts that we analyzed can be found on our website https://ethereum.logic.at.

Code availability Our scripts are available at https://github.com/gsalzer/ethutils.

References

1. Contract ABI Specification. https://solidity.readthedocs.io/en/latest/abi-spec.html (2019). Accessed 9 June 2021
2. Bittrex: Controlled wallet. https://etherscan.io/address/0xA3C1E324CA1CE40DB73ED6026C4A177F099B5770#code (2017). Accessed 12 Oct 2019
3. Buterin, V.: Blockchain and smart contract mechanism design challenges (slides) (2017). http://fc17.ifca.ai/wtsc/Vitalik%20Malta.pdf. Accessed 9 Aug 2018
4. Chan, W., Olmsted, A.: Ethereum transaction graph analysis. In: 12th International Conference for Internet Technology and Secured Transactions (ICITST), pp. 498–500. IEEE (2017). https://doi.org/10.23919/ICITST.2017.8356459
5. Chen, T., Li, Z., Zhu, Y., Chen, J., Luo, X., Lui, J.C.S., Lin, X., Zhang, X.: Understanding Ethereum via graph analysis. ACM Trans. Internet Technol. TOIT **20**(2), 1–32 (2020). https://doi.org/10.1145/3381036
6. Chen, T., Luo, X., Zhang, Y., Wang, T., Li, Z., Cao, R., Xiao, X., Zhang, X.: TokenScope: automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 1503–1520 (2019). https://doi.org/10.1145/3319535.3345664
7. Chen, W., Zhang, T., Chen, Z., Zheng, Z., Lu, Y.: Traveling the token world: a graph analysis of Ethereum ERC20 token ecosystem. In: Proceedings of The Web Conference 2020, pp. 1411–1421. ACM, New York, NY, USA (2020). https://doi.org/10.1145/3366423.3380215
8. Dafflon, J., Baylina, J., Shababi, T.: ERC-777 token standard (2015). https://eips.ethereum.org/EIPS/eip-777. Accessed 11 May 2021
9. Darisi, M., Savla, J., Shirole, M., Bhirud, S.: STEM: secure token exchange mechanisms. In: Advances in Cyber Security, vol. CCIS 1132. Springer (2020). https://doi.org/10.1007/978-981-15-2693-0_15
10. Di Angelo, M., Salzer, G.: Mayflies, breeders, and busy bees in Ethereum: smart contracts over time. In: Proceedings of the Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts, BCC '19, pp. 1–10. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3327959.3329537
11. Di Angelo, M., Salzer, G.: A survey of tools for analyzing Ethereum smart contracts. In: 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON), pp. 69–78. IEEE (2019). https://doi.org/10.1109/DAPPCON.2019.00018
12. Di Angelo, M., Salzer, G.: Characteristics of wallet contracts on Ethereum. In: 2nd Conference on Blockchain Research and Applications for Innovative Networks and Services (BRAINS'20). IEEE (2020). https://doi.org/10.1109/BRAINS49436.2020.9223287

13. Di Angelo, M., Salzer, G.: Tokens, types, and standards: identification and utilization in Ethereum. In: International Conference on Decentralized Applications and Infrastructures (DAPPS). IEEE (2020). https://doi.org/10.1109/DAPPS49028.2020.00-11
14. Di Angelo, M., Salzer, G.: Towards the identification of security tokens on Ethereum. In: 2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–5 (2021). https://doi.org/10.1109/NTMS49979.2021.9432663
15. Dossa, A., Ruiz, P., Vogelsteller, F., Gosselin, S.: Controlled token standard proposal (2019). https://github.com/ethereum/EIPs/issues/1644. Accessed 11 May 2021
16. Dossa, A., Ruiz, P., Vogelsteller, F., Gosselin, S.: Security token standard proposal (2019). https://github.com/ethereum/EIPs/issues/1411. Accessed 11 May 2021
17. Entriken, W., Shirley, D., Evans, E., Sachs, N.: ERC-721 non-fungible token standard (2018). https://eips.ethereum.org/EIPS/eip-721. Accessed 11 May 2021
18. Ethereum Wiki: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper. Accessed 2 Feb 2019
19. FINMA. https://www.finma.ch/en/documentation/dossier/dossier-fintech/entwicklungen-im-bereich-fintech/. Accessed 12 Oct 2019
20. Fröwis, M., Fuchs, A., Böhme, R.: Detecting token systems on Ethereum. In: International Conference on Financial Cryptography and Data Security. Springer (2019). https://doi.org/10.1007/978-3-030-32101-7_7
21. Guo, D., Dong, J., Wang, K.: Graph structure and statistical properties of Ethereum transaction relationships. Inf. Sci. **492**, 58–71 (2019). https://doi.org/10.1016/j.ins.2019.04.013
22. Hacker, P., Thomale, C.: Crypto-securities regulation: ICOs, token sales and cryptocurrencies under EU financial law. Eur. Co. Financ. Law Rev. **15**(4), 645–696 (2018). https://doi.org/10.1515/ecfr-2018-0021
23. He, N., Wu, L., Wang, H., Guo, Y., Jiang, X.: Characterizing code clones in the Ethereum smart contract ecosystem. In: International Conference on Financial Cryptography and Data Security, pp. 654–675. Springer (2020). https://doi.org/10.1007/978-3-030-51280-4_35
24. Kondo, M., Oliva, G.A., Jiang, Z.M., Hassan, A.E., Mizuno, O.: Code cloning in smart contracts: a case study on verified contracts from the Ethereum blockchain platform. Empir. Softw. Eng. (2020). https://doi.org/10.1007/s10664-020-09852-5
25. Kupriianov, M., Svirsky, J.: Base security token standard draft (2019). https://eips.ethereum.org/EIPS/eip-1462. Accessed 11 May 2021
26. Lambert, T., Liebau, D., Roosenboom, P.: Security token offerings. SSRN Electr. J. (2020). https://doi.org/10.2139/ssrn.3634626
27. Liu, H., Yang, Z., Jiang, Y., Zhao, W., Sun, J.: Enabling clone detection for Ethereum via smart contract birthmarks. In: IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp. 105–115. IEEE (2019). https://doi.org/10.1109/ICPC.2019.00024
28. Liu, H., Yang, Z., Liu, C., Jiang, Y., Zhao, W., Sun, J.: EClone: detect semantic clones in Ethereum via symbolic transaction sketch. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pp. 900–903. ACM (2018). https://doi.org/10.1145/3236024.3264596
29. Norvill, R., Fiz, B., State, R., Cullen, A.: Standardising smart contracts: automatically inferring ERC standards. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), pp. 192–195. IEEE (2019). https://doi.org/10.1109/BLOC.2019.8751350
30. Oliveira, L., Zavolokina, L., Bauer, I., Schwabe, G.: To token or not to token: tools for understanding blockchain tokens. In: International Conference on Information Systems (ICIS). AIS eLibrary (2018). https://doi.org/10.5167/uzh-157908
31. Radomski, W., Cooke, A., Castonguay, P., Therien, J., Binet, E., Sandford, R.: ERC-1155 multi token standard (2015). https://eips.ethereum.org/EIPS/eip-1155. Accessed 11 May 2021
32. Rohr, J., Wright, A.: Blockchain-based token sales, initial coin offerings, and the democratization of public capital markets. Hastings LJ **70**, 463 (2019)
33. Shiple, J., Marks, H., Zhang, D.: Ldgrtoken standard draft (2019). https://eips.ethereum.org/EIPS/eip-1450. Accessed 11 May 2021
34. Somin, S., Gordon, G., Altshuler, Y.: Network analysis of ERC20 tokens trading on ethereum blockchain. In: International Conference on Complex Systems, pp. 439–450. Springer (2018). https://doi.org/10.1007/978-3-319-96661-8_45
35. U.S. Supreme Court: SEC v. W. J. Howey Co., 328 U.S. 293 (1946)
36. Victor, F., Lüders, B.K.: Measuring Ethereum-based ERC20 token networks. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security (FC), LNCS, vol. LNSC 11598, pp. 113–129. Springer, New York (2019). https://doi.org/10.1007/978-3-030-32101-7_8
37. Vogelsteller, F., Buterin, V.: ERC-20 token standard (2015). https://eips.ethereum.org/EIPS/eip-20. Accessed 11 May 2021
38. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Technical report, Ethereum Project Yellow Paper (2019). https://ethereum.github.io/yellowpaper/paper.pdf