Tokens, Types, and Standards: Identification and Utilization in Ethereum

Monika di Angelo, Gernot Salzer
TU Wien, Vienna, Austria
{monika.di.angelo,gernot.salzer}@tuwien.ac.at

***Abstract*—Tokens are often referred to as the killer application of blockchains and cryptocurrencies. Some even believe that basically everything can be tokenized, meaning that it can be represented by a tradable digital token. With decentralized P2P networks that hold and distribute the tokens, one can build various decentralized applications around them. As a digital asset on top of a cryptocurrency, crypto tokens are managed by a smart contract, most commonly on Ethereum. A widespread high-level categorization of tokens distinguishes between payment tokens, security tokens, and utility tokens. The need for clarifying the differences lies in the fact that in most jurisdictions, security tokens are more heavily regulated than other tokens.**

In this work, we contribute to the detection and classification of crypto tokens from bytecode. First, we examine how ideas on regulation are currently incorporated into respective standards. Then, we discuss methods for identifying deployed token contracts from bytecode. Furthermore, we analyze deployed contracts regarding the standards to which they comply, and the type of token they may represent. Moreover, we investigate the actual usage of tokens. Our empirical analysis uses the transaction data from the Ethereum main chain up to block 8 750 000, mined on Oct 16, 2019.

***Index Terms*—bytecode, compliance, regulation, smart contract, standard, token**

I. Introduction

Applications on a P2P network that are not controlled by a single entity are called decentralized applications (dApps). Usually, their front end is a browser or app interface, while their backend can – at least partly – be realized on a blockchain or cryptocurrency. Major categories of dApps are still decentralized finance, followed by gambling and games, and marketplaces. Often, they provide their own token in the dApp.

A crypto token is a digital asset on top of a cryptocurrency or blockchain, often as a programmable asset managed by a smart contract, for use within a project or dApp. Crypto tokens are similar to the coins of a cryptocurrency, except that they do not have their own blockchain or distributed ledger. Rather, they are built on top of an existing one.

When we regard crypto tokens as representing the right to something, we enter the area of tokenization. Tokenization is a way to convert the rights to something into a digital artifact, a so-called token. With crypto tokens, the benefits of tokenization lie mainly in higher liquidity, general programmability, and immutable proof of ownership. More precisely, fractional digital ownership lowers the barriers of entry for investors while it increases the liquidity of the tokenized assets. Furthermore, programmability facilitates automated management of investor rights and compliance regarding the tokenized assets, and thus potentially increases speed. Finally, the immutable traces of transfers provide evidence that the transfers of the digital ownership have not been tampered with. However, there is still a lack of tokenization standards and in many jurisdictions also a lack of legal infrastructure in this regard.

Purpose of tokens. As a medium of exchange, tokens can act as a currency themselves. In this respect, they can also be termed as the local currency of a dApp. In the main, crypto tokens are used beyond mere exchange by leveraging their most salient feature: being programmable. In this regard, they are used to trigger certain functions in the smart contract(s) of the dApp. Moreover, tokens may be linked to off-chain assets. They can serve as means of fundraising, pre-order or investment, as well as building an ecosystem or a community.

Creation of tokens on top of an existing blockchain is accomplished via smart contracts, so-called token contracts. As this is a widespread application type, coding patterns and best practice examples are readily available. Furthermore, there are token contract factories, either on-chain or as a web service.

Acquisition of tokens varies. For example, they can be purchased during an initial coin offering (ICO) or through a crypto exchange, traded on-chain or received freely during an airdrop or as a reward for a service or behavior.

The value of a token depends mainly on supply, demand and the trust that the participating community has in it, which is based on credibility and service.

Our Approach

Many projects implement their tokens on Ethereum. With the publicly available Ethereum transaction data, we analyze token contracts to answer the following questions.

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

For our analysis, we examine both the deployed bytecode and the calls to token contracts that we extract from transaction data. We discuss several methods for the (semi-)automatic identification of token contracts and their type.

The presented evaluation of token contracts may benefit the design of future decentralized asset trading systems. It paves the way for an accurate assessment of deployed tokens with respect to their type, based on their bytecode and usage. We

978-1-7281-6978-1/20/$31.00 ©2020 IEEE
DOI 10.1109/DAPPS49028.2020.00-11

contribute to the discussion about detection and classification of crypto assets. Furthermore, our work adds to a clearer insight into decentralized applications by further characterizing the most heavily used application *tokens*.

Roadmap. In section II, we discuss types of crypto tokens. In section III, we summarize typical functionalities of token contracts and token standards. Our methods for analyzing bytecode are detailed in section IV, while the methods for identifying token contracts are discussed in section V. We identify compliant tokens in section VI and non-compliant ones in section VIII. In section VII, we examine the variability and usage of token bytecode. We address the type distinction in section IX. In section X, we compare our approach to related work. Section XI summarizes our findings and concludes.

II. Types of Tokens

A widespread high-level categorization of tokens [1] distinguishes between payment tokens, security tokens, and utility tokens. The need for clarifying their differences lies in the fact that in most jurisdictions, security tokens are more heavily 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 which is typical of utility tokens. Payment tokens fulfil a payment function with little or no other function.

Legally, the distinction is still a grey area in many jurisdictions. In [2], the authors discuss legal aspects of token sales under US law, including a similar classification of token types, and the so-called 'Howey Test' (a list of criteria determining whether a financial instrument qualifies as a security). They argue that jurisdictions should provide "regulatory certainty and a sensible path to compliance". We use the Swiss FINMA [3] definitions for security and utility tokens as a common ground for US [2], EU [4], 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. section III), as well as into wallets and exchanges (cf. [5]).

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.

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 [4]. Based on reviewed literature and a subsequent empirical study, the authors of [1] distill eight archetypes of tokens.

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

III. Token Standards

Standard token interfaces enable applications such as wallets to recognize tokens and interact with them. In this section, we first clarify terms and summarize the functionalities provided by token contracts. Then we introduce already accepted token standards alongside with proposed security token standards. Finally, we reference available coding patterns.

A. Terms

We assume the reader to be familiar with cryptocurrencies. For Ethereum specifics, we refer to [6]–[8].

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.

Abstract Binary Interface (ABI). Most contracts in the Ethereum universe adhere to the ABI standard [9], which identifies functions by signatures that consist of the first four bytes of the Keccak-256 hash of the function name together with the parameter types. Thus, the bytecode of a contract contains instructions to compare the first four bytes of the call data to the signatures of its functions. The presence of a particular function in a contract can be checked by locating the corresponding 4-bytes hash in its deployed bytecode. Thus, the compliance of a contract with interface standards can be determined via its bytecode.

B. Functionalities of Token Contracts

The basic functionality of a token contract comprises: *bookkeeping* of token holdings, *transferring* the ownership of tokens by according changes in the book of token holdings in the respective token contract, and emitting *events* to record the transfers of ownership in the logs. *Safe transfer* is a mechanism where tokens are pulled (withdrawn) from an address after approval, as opposed to being pushed (transferred) to an address where they may be lost in case the address is not prepared for receiving tokens.

Additional token-related functionalities include the creation and destruction of tokens (called minting and burning) as well as their distribution and trading (e.g. via ICOs and airdrops). Often, token contracts also implement general functionalities including authentication and roles, control (like pause, lock),

information provision (like view functions), and utilities (like safe mathematical operations).

C. 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 [10] 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 [11] 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 [12] 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 [13] 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.

D. 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 [14] 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 [15] 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 some optional parts of ERC-20 mandatory. Moreover, it requires certain modifiers and constructor arguments to be implemented.

ERC-1644 Controller Token Operation Standard [16] "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 [17], 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-chain and off-chain restrictions, and ERC-1643 for document and legend management.

E. Coding Patterns for Token Contracts

Most tokens aim at establishing trust and credibility by disclosing their source code on the leading blockchain explorer for Ethereum, *Etherscan*[1]. As a service, this platform checks that the deployed bytecode is the result of compiling the provided source code with the given compiler settings, and labels it as 'verified source code'.

Furthermore, many token related projects are developed publicly on the platform *GitHub*. Finally, we would like to point out the best practice coding patterns provided by ConsenSys[2] (including tokens), as well as the tested implementations of ERC standards by OpenZeppelin[3].

IV. Bytecode Analysis

We use verified source code from Etherscan if available, but relying solely on such contracts would bias the results: In contrast to 18.9 M successful create operations, there is verified source code for 73 k addresses (0.4 %) only. In this section, we describe our methods of bytecode analysis and list the tools we employ.

Unless stated otherwise, statistics refer to the Ethereum main chain up to block 8 750 000 (mined on Oct 16, 2019). We abbreviate factors of 1 000 and 1 000 000 by the letters k and M, respectively.

A. Code 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 PUSH operations 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

[1] https://etherscan.io/
[2] https://consensys.github.io/smart-contract-best-practices/
[3] https://github.com/OpenZeppelin/openzeppelin-contracts

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 18.9 M contract deployments correspond to just 109 k distinct skeletons. This is still a large number, but more manageable then 239 k distinct bytecodes. By exploiting creation histories and the similarity via skeletons, we are able to relate 7.3 M of these deployments to some source code on Etherscan, an increase from 0.4 to 39 %.

B. Contract Interfaces

Understanding the interface of a contract renders its analysis much easier. To this end, we first extract the interface from the bytecode of a contract, and then try to restore the corresponding function headers.

1) Interface Extraction: We developed a pattern-based tool to extract the interface in the form of 4-byte function signatures (cf. section III) from the bytecode. For validation, we applied it to the bytecodes of the 73 k verified contracts (71 k codes, 39 k skeletons) from Etherscan, using the ABIs given there as ground truth. The signatures extracted by our tool differed from the ground truth in 42 cases. Manual inspection revealed 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 consists 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 9.6 M deployed contracts (220 k codes, 107 k skeletons) generated by the Solidity compiler[4].

Another large group of deployed contracts consists of 5.2 M short contracts (18 k codes, just 271 skeletons) without any entry points. It consists mainly of contracts for storing gas (gasToken), but also of proxies (contracts redirecting calls elsewhere) and of contracts involved in attacks.

A third large group are 4.2 M contracts that self-destruct at the end of the deployment phase. They cannot be regularly called and thus contain no entry points either.

What remains is a heterogeneous group of 1.4 k contracts (595 codes, 432 skeletons). For these, our tool shows an error rate of 8 %, extrapolated from a random sample of 60 codes that we manually checked.

For all groups taken together, our tool extracts the 4-byte signatures of the interface with virtually no errors.

2) Interface Restoration: To understand the purpose of contracts we try to recover the function headers from the signatures. As the signatures are partial hashes of the headers, we use a dictionary of function headers with their 4-byte signatures (collected from various sources), which allows us to obtain a function header for 59 % of the 254 k distinct signatures on the main chain.[5] Since signatures occur with varying frequencies and codes are deployed in different numbers, this ratio increases to 91 % (or 89 %) when picking a code (or a deployed contract) at random.

3) Events: Another part of a contract's interface are events, which notify applications external to the blockchain about state changes. At the level of the EVM, events are implemented as `LOG` instructions with the unabridged Keccak-256 hash of the event header as identifier.

We lack a tool that is able to extract the event signatures as reliably as the function signatures. But we can check whether a given signature occurs in the code section of the bytecode, as the 32-byte sequence is virtually unique. Even though this heuristic may fail if the signature is stored in the data section, it performs well in general. E.g., when locating the event *Transfer* in the bytecodes corresponding to the 73 k verified source codes from Etherscan, we obtain just 272 mismatches. Many of them are due to source code declaring the event but not using it, so the compiler adds it to the ABI, even though it does not occur in the bytecode. Therefore, we regard this method as sufficiently reliable for our purpose.

C. The Data and Its Use

Section V details how we identify token contracts based on both static and dynamic data of deployed contracts that we extract from transaction data and execution traces.

The static data consists of the deployment address, the bytecode, its skeleton, the list of the extracted 4-byte signatures of the implemented functions, and a flag indicating whether the bytecode implements an event related to token transfers. Moreover, we use the function headers and contract names where available.

The dynamic data consists of all calls to and from contracts (including the signature of the invoked function) as well as the signatures of the events actually emitted. The dynamic data is sparse: 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. If some behavior is observed with a particular contract, then we may assume that every contract with the same bytecode will behave identically.

D. Third Party Tools

We employ the Ethereum client *Parity* in archive mode to obtain bytecode and execution traces. *Postgres* serves as our

[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 a `PUSH` instruction followed by `0x50` or `0x3014`.

[5]An infinity of possible function headers is mapped to a finite number of signatures, so there is no guarantee that we have recovered the original header. The probability of collisions is low, however. E.g., of the 322 k signatures in our dictionary only 19 appear with a second function header.

primary database for storing messages between and information on contracts. For analyzing contract interactions as graphs, we use the graph database *Neo4j*. We query *Etherscan* over its API for further information on deployed contracts (like source code or contract name) and utilize *Matplotlib* for plotting.

V. Methods for Identifying Token Contracts

In this section, we discuss methods to identify contracts that provide token functionality. We first summarize related work and then present our approach.

A. Methods from Related Work

1) 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 implements functions for querying the data structure and for transferring tokens between users. Some researchers try to detect such behavior by symbolic execution and taint analysis of the bytecode. Due to the difficulty of the problem, this method is still less effective than simpler approaches [18]. We therefore rely on other methods for our analysis.

2) Interface-oriented Approach: Token contracts are supposed to be accessible by wallets and exchanges, hence they offer standardized interfaces. It is therefore a reasonable approach to identify token contracts over the implemented functions headers. This is accomplished by locating the presence of the corresponding function signatures in the bytecode (cf. section IV).

Most authors concentrate on fully ERC-20 compliant contracts [19], [20], while [18] accept tokens when they implement at least five of the six mandatory ERC-20 functions. The latter work also shows the effectiveness of this method.

B. Our Approach

We apply the interface-oriented approach to identify contracts that implement all functions required by the token standards mentioned in section III and call them fully compliant. For not fully compliant contracts, we discuss additional approaches to identification.

1) Compliant Tokens – Identification via the Interface: The interface-oriented approach is a simple and effective means for detecting most compliant token contracts. For obtaining the interface as a list of implemented functions, we rely on our tool for extracting signatures from the bytecode (section IV). To determine full compliance, we consider the required functions, but we disregard events, return values, and further requirements of the respective standards.

Limitations: This approach may misclassify token contracts in the following situations: (a) signatures are extracted inaccurately from the bytecode, (b) the contract uses an idiosyncratic interface instead of adhering to the ABI, (c) token functions are named in a non-standard way, (d) the contract implements non-token functionality with token-specific function headers, and (e) the token contract is implemented as a proxy that forwards calls to another contract and thus does not contain the forwarded signatures.

We disregard (a), as the error rate of our signature extraction tool is low (see its evaluation in the previous section). Situations (b) and (c) are unlikely since deviating from standard interfaces would render the token contract inaccessible for general use. There are cases of (d): On the one hand there are contracts that implement e.g. a function `balanceOf(address)`, but being wallets rather than tokens. On the other hand, we encountered singular bogus contracts selling tokens via a standard interface without recording ownership, but returning the same balance to all queries. The second case is rare and can be ignored (or even may be considered a token), whereas the first one can be countered by using e.g. a combination of features or by checking the code.

2) Non-Compliant Tokens: Above all, the detection of non-compliant tokens requires to define conditions for a contract to be considered as actually implementing a token. We approach a definition over several indicators.

Partial ERC-20 Compliance. Based on the interface method, partial compliance to a standard with a threshold on implemented functions could serve as definition for a token contract. The functions *transfer* or *transferFrom*, and *balanceOf* are the bare essentials of an interface that enables applications to work with the token contract. We therefore mark contracts as potential tokens when they provide these functions. Also, the presence of the signatures for the optional functions *name, decimals,* or *totalSupply* are a strong indicator for a token.

Limitations: This approach has the same issues as the interface method. Additionally, it heavily depends on the threshold.

Token Contracts as Event Issuers. Token standards usually require compliant contracts to emit an event with every token transfer indicating the affected token as well as the sender and receiver. Further events signal other state changes like modified allowances. These events can be used to identify the emitting token contract, by detecting either their signature in the code or the data written to the log.

We mark addresses as potential tokens if their code contains the event `Transfer(address,address,uint256)` or if they actually emitted the event. All standards in section III require this event, except for ERC-1155 that replaces it by *TransferSingle* or *TransferBatch*.

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.

Identifying Tokens by their Contract Names. To detect tokens in a heuristic fashion, we scanned the 70 k source codes on Etherscan for contracts containing the strings 'token' or 'coin' in their name. Then we associated the corresponding source codes to the actual deployments and marked them as potential tokens.

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 yields false positives.

VI. Compliant Tokens

In this section, we analyze deployed token contracts regarding their compliance with at least one of the standards from section III. Since there are tokens that do not comply with any mentioned ERC standard, we further investigate non-compliant contracts in section VIII. For both we employ the methods from the previous sections on all deployed bytecode.

Figure 1 shows the deployments of ERC-compliant tokens on Ethereum's main chain on a time line, with ERC-20 compliance in the upper plot and the other compliances in the lower plot. In each of the two plots, the upper horizontal axis indicates the date line, while the lower one shows the Ethereum block number. Each bar represents the accumulated number of deployments within 50 k blocks (about one week). As depicted in the upper plot, ERC-20 took off in the middle of the year 2017, while the lower plot shows that the NFT standard started to rise with the beginning of the year 2018.

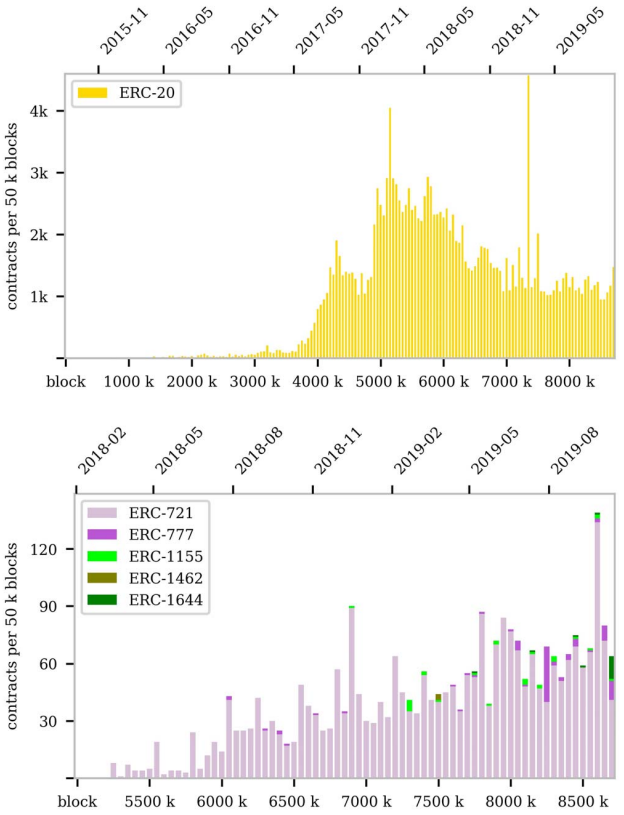

Fig. 1. Creation of ERC-compliant token contracts. The lower horizontal axis indicates the Ethereum blocks, while the upper axis shows the corresponding dates. Each bar represents a bin of 50 000 blocks (corresponding roughly to 1 week). The upper plot shows ERC-20 creations, while the lower plot differentiates the other compliances.

TABLE I
Full Compliance of Deployed Token Contracts

standard	deployments	bytecodes	skeletons
ERC-20	166 191	92 328	32 090
ERC-721	2 740	1 268	999
ERC-777	82	34	29
ERC-1155	28	22	22
ERC-1462	4	4	4
ERC-1450	0	0	0
ERC-1644	19	5	5

After a hype in the first half of the year 2018, ERC-20 deployments seem to stabilize at about 1000 per week.

Table I shows the number of fully compliant token deployments for the standards from section III. With over 166 k deployments, ERC-20 is by far the most commonly used standard. Of all compliant token contracts, over 98% comply with ERC-20. The non-fungible token standard ERC-721 is the second most common one, albeit only deployed in 1.6% of the compliant contracts. The remaining standards are deployed in small numbers only. A few token contracts comply with more than one standard and consequently are counted for each corresponding standard. In total, we identified 169 k fully compliant token contracts.

VII. Code Usage, Variability and Reuse

A. Usage of Token Functions

Table II lists the nine most called functions of compliant token contracts with their respective signature, header and number of calls. Until block 8.75 M, Ethereum produced almost 1 370 M messages. With over 193 M calls, the function *transfer* accounts for 14 % of all Ethereum messages. Summing up all calls to compliant token contracts, we arrive at 326 M or 23.8 %. This clearly suggests that tokens are the killer application for Ethereum.

B. Variability and Reuse

Ethereum is known for its high degree of code reuse (cf. [21]–[23]), with almost 19 M deployed contracts having only about 239 k unique bytecodes and 109 k distinct skeletons. This amounts to a code reuse factor of 79 regarding bytecode and of 174 regarding skeletons.

TABLE II
Calls to Functions in Token Contracts

signature	header	number of calls
A9059CBB	transfer(address,uint256)	193 390 892
70A08231	balanceOf(address)	28 204 061
23B872DD	transferFrom(address,address,uint256)	25 266 224
18160DDD	totalSupply()	10 331 342
DD62ED3E	allowance(address,address)	7 413 635
095EA7B3	approve(address,uint256)	6 446 899
8DA5CB5B	owner()	5 257 436
40C10F19	mint(address,uint256)	3 369 344
313CE567	decimals()	2 998 962

Among the deployed token contracts, we also find code reuse. However, it is far less pronounced than the average for Ethereum. In table I, we observe that 169 k deployed token contracts correspond to 94 k bytecodes and 33 k skeletons. This amounts to a code reuse factor of just 1.8 for bytecodes and 5.1 for skeletons. For tokens, code reuse can be attributed to off-chain services for token contract deployment as well as to readily available blueprints for token contracts in Solidity (cf. section III).

The low reuse factor indicates that token contracts are tailored to different usage scenarios. This is also supported by the fact that the number of functions implemented by any token contract varies between 6 (the bare minimum of ERC20 compliant tokens) and 130. In total, we have 57 k different function signatures occurring in compliant contracts, which means that on average, each skeleton implements two functions unique to it.

VIII. Non-Compliant Tokens

While the identification of tokens via the full compliance to a standard interface is effective, it misses non-compliant tokens. Etherscan lists ERC-20 and ERC-721 tokens even when they are not fully compliant. They report over 200 k ERC-20 tokens for the study period, while only 166.2 k contracts are fully ERC-20 compliant. Even well-known tokens implement standards only partially by leaving out one or more of the mandatory functions. A famous example are the CryptoKitties.

In this section, we discuss approaches to define tokens irrespective of full ERC compliance by looking at potential indicators for tokens. First, we establish a ground truth for token contracts. As indicators, we investigate partial ERC-20 compliance with a possible thresholds on the number of provided ERC-20 functions, as well as contract names and events.

A. Ground Truth for Token Contracts

To compare the indicators, we construct a token ground truth (TGT) consisting of bytecodes implementing a token (positive instances) and bytecodes not doing so (negative instances). Below, the numbers in parentheses indicate the number of instances obtained by the respective method.

In the first round, we consider the names that developers gave to their verified contracts on Etherscan. We mark a bytecode as a positive instance of a token contract if the name of one of its deployments ends, case insensitive, with 'token' (18532) or 'coin' (4535). Likewise, we mark it as a negative instance if the name ends with 'exchange' (307), 'market' (95), 'auction' (137), or 'wallet' (662).

Furthermore, by manual inspection we obtain further negative instances: wallets (2170), gasToken (16080), attack or otherwise nonsensical contracts (1066), and libraries (3215); and positive ones: non-compliant token contracts with minimal interfaces (304).

Finally, we manually classified 250 bytecodes that received the most calls, resulting in 120 positive and 148 negative instances. In total, our TGT consists of 23 k positive and 23 k negative instances.

B. Number of Implemented ERC-20 Functions

ERC-20 is the most widespread standard for tokens with the mandatory functions *transfer, transferFrom, totalSupply, allowance, approve,* and *balanceOf*. We investigated how many of the six mandatory functions (signatures) all deployed contracts actually implement. The functions *transfer* and *balanceOf* are the most essential ones for a dApp to work with the token. At the same time, they are not specific to tokens contracts. Rather, they also appear in applications that interact with a token where these functions may serve as proxies.

TABLE III
Implemented ERC-20 Functions with and without transfer

	deployments		bytecodes		skeletons	
signatures	incl. transfer	excl. transfer	incl. transfer	excl. transfer	incl. transfer	excl. transfer
6 of 6	166 191	—	92 328	—	32 090	—
5 of 6	3 496	82	2 107	64	1 578	54
4 of 6	3 347	2 795	2 637	1343	1 016	1 080
3 of 6	8 802	857	4 518	535	2 586	379
2 of 6	3 473	2 603	2 061	602	887	493
1 of 6	203 708	28 125	539	3 175	476	2 275

Table III lists the number of contracts regarding the number of ERC-20 signatures they provide. We differentiate the numbers according to the presence or absence of the function *transfer*, and also indicate the actual deployments on-chain, the corresponding unique bytecodes, and the respective skeletons.

Table III shows that 166.2 k deployed contracts implement the full ERC-20 interface. Furthermore, there are 203.7 k contracts that provide only the function *transfer* but none of the other mandatory ERC-20 functions. The latter contracts also show a remarkably small set of only 539 corresponding bytecodes, which represents a code reuse factor of 378 regarding bytecode and of 428 regarding skeletons. This hints towards factory-produced non-token contracts (e.g. wallets) implementing a function *transfer*.

In summary, the numbers in table III do not suggest a threshold for a minimum of functions required to implement a token. Furthermore, the number of contracts implementing between two and five of the mandatory functions is non-negligible. Hence, it is worthwhile to look for indicators beyond mere ERC compliance of implemented functions.

C. Contract Names

When associating bytecode with the names of its deployments on Etherscan (cf. section V), we can draw the following picture. We list the number of deployed contracts and respective bytecodes, for which we can associate a name from Etherscan in table IV. In the first line, the high number of over 3 M deployments with associated names mainly results from wallets [24]. A high factor between bytecodes and deployments is atypical of tokens and rather an indicator for wallets.

TABLE IV
ASSOCIATED CONTRACT NAMES

contract with	deployments	bytecodes
name	3 162 177	71 179
'token' in name	116 793	23 386
'coin' in name	8 326	5 914
'token' in name and non-ERC	57 831	3 495

D. Effectiveness of Indicators

Table V compares the effectiveness of the indicators above on our token ground truth (TGT), measured by precision and recall. Let tp (true-positive) denote the number of positive TGT instances classified correctly as a token, fp (false-positive) be the number of negative TGT instances classified wrongly as a token, and fn (false-negative) be the number of positive TGT instances classified wrongly as a non-token. *Precision* is computed as the quotient $tp/(tp+fp)$. A precision value close to one means that the number of negative instances mistaken as positive ones is small; if the indicator classifies a bytecode as a token, then it most likely is one. *Recall* is computed as the quotient $tp/(tp+fn)$. 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.

The highest precision and the second best recall is achieved by checking the contract name for 'token' or 'coin'. This seems plausible: if a developer calls a contract a token, then it probably is one. However, our ground truth is biased towards this indicator, as the majority of positive instances has been selected by this criterion; moreover, the indicator can only be applied if the contract name is available.

TABLE V
EFFECTIVENESS OF INDICATORS FOR IDENTIFYING TOKEN BYTECODES

indicator	precision	recall
The contract name contains either 'token' or 'coin'.	99.7	98.5
Bytecode is fully ERC compliant (section VI).	99.5	92.5
Bytecode implements at least three functions of some ERC standard.	99.4	98.2
Some deployment of bytecode has emitted the event *Transfer*.	99.4	80.9
Bytecode implements at least one of the functions *transfer* or *transferFrom*.	99.3	98.8
Bytecode contains the signature of event *Transfer*.	99.2	96.4
Bytecode contains the event *Transfer* or it has been emitted by any of its deployments.	99.1	96.6

IX. DISTINCTION OF TOKEN TYPES BY PURITY

In this section, we try to gain insights into the purpose of token contracts at large, in particular with regard to their use as security or utility tokens. A precise classification of tokens in large quantities would require automated code analysis to check for semantic properties, which is a difficult problem not yet adequately solved. Instead, we present a heuristic test that starts from the observation that the functions of token interfaces can be categorized into the following three groups.

First, we have the core functions mandated by token standards, as well as related ones for creating, destroying, and distributing tokens. Second, there are neutral functions that may 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 the token.

We call a token contract pure if its interface contains only functions from the first and second group. Pure tokens do not provide functions that indicate a service or product, hence they probably are security rather than utility tokens. Non-pure tokens, on the other hand, are more likely to be utility tokens.

To partition the functions of an interface into the three groups we analyze the function names. Unsurprisingly, the names in the first and second group are quite uniform and stereotypical as the functions perform standardized tasks. Therefore, this heuristic approach seems reasonable.

In the remainder of this section, we first detail our method for function classification and then use it to classify compliant token contracts.

A. Token-Related and Neutral Functions

To classify function headers, we use rules of the form 'If a header matches the regular expression re_1 but not re_2, then it belongs to category C.' We devise the rules empirically in a repetitive manner, by searching yet unclassified headers for patterns, deducing new rules, checking their effect on all headers, refining them, removing already classified headers, and repeating the process with the remaining ones.

As an example, one of our rules reads 'If the header matches `^(get|is|total|balance)` but not `^issue`, then it belongs to the category *getter*'. It labels headers like `getAuthor()` and `IsActive()` as getters, whereas `issueToken(address)` is skipped. Another rule detects functions related to ICOs, taking into account that unicorns and icoins are not ICO-related: 'If the header matches `ico` but not `unicorn|icoin`, then it is of category *ico*'.

Limitations: The effectiveness of this method hinges on how carefully the rules are chosen. Moreover, the method assumes that at least for the first and second group, the names of functions indicate the implemented functionality. Finally, a contract may delegate some function calls to another contract (like a library) such that the signatures extracted from the contract represent only part of the interface.

B. Pure Compliant Token Contracts

We apply our classification scheme to the 94 k compliant token codes (deployed 169 k times). They contain 57 k distinct signatures, of which 43 k can be decoded to function headers. Because of their uniformity, we expect signatures from the first two groups to be mostly among the decoded ones. This assumption can also be justified ex-post by the large number of tokens that turn out to be pure tokens in our sense and thus do not contain any unknown signatures.

We specified 22 rules that divide 35 k functions into 17 categories, with 8 k remaining unclassified. We regard headers

of the categories *token*, *distribution*, *auction*, *minting*, *approval*, *kyc*, *ico*, *transfer*, *crowdsale*, *airdrop*, and *burning* as token-related (first group), whereas the categories *control*, *math*, *getter*, *setter*, *trading*, and *roles* count as neutral (second group). 14 k unknown signatures and 8 k unclassified functions form the other (third) group.

TABLE VI
PURITY OF COMPLIANT TOKEN CONTRACTS

# other functions	bytecodes	deployments	received calls
= 0 (pure)	63 496	123 021	171 343 421
> 0	30 120	45 936	166 586 678
all compliant	93 616	168 957	337 930 099

Table VI groups compliant token contracts depending on whether their interface contains any function of the *other* (third) group. Interestingly, 63.5 k distinct bytecodes (corresponding to 123 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. Consequently, 30.1 k bytecodes implement other functions as well. As this group contains 22 k different signatures, it is not apparent how to decide automatically whether they offer a genuine service or product. Thus, a different approach is required to classify the further functionalities of token contracts.

X. COMPARISON TO RELATED WORK

Token Aspects. Most of the distantly related work focuses on the transfer of assets and network communication on Bitcoin and other cryptocurrency platforms. Regarding Ethereum, [25] examines "whether an attacker can de-anonymize addresses from graph analytics against transactions on the blockchain". The authors of [26] "leverage graph analysis to characterize three major activities, namely money transfer, contract creation, and contract calls" with the aim to address security issues. Applying network science theory, [27] "find that several transaction features, such as transaction volume, transaction relation, and component structure, exhibit a heavy-tailed property and can be approximated by the power law function."

Regarding ERC-20 tokens on Ethereum, the authors of [20] study the token trading network in its entirety with graph analysis and show power-law properties for the degree distribution. Similarly, the authors of [19] 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.

Instead of examining the trading of tokens, our investigation puts a focus on the compliance of tokens to standards and investigates the types of tokens. We look at token interactions in the form of calls to the token contracts. We combine the analysis of interactions with the analysis of bytecode to identify deployed contracts more reliably as tokens.

EVM Bytecode Analysis. To detect code clones, the authors of [22] first deduplicate 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 deduplicated contracts by a customized version of fuzzy hashing and compute pair-wise similarity scores. In another approach to clone detection, the authors of [28], [29] 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.

To detect token systems automatically, the authors of [18] compare the effectiveness of a behavior-based method combining symbolic execution and taint analysis, to a signature-based approach limited to ERC20-compliant tokens. They demonstrated that the latter approach detects 99 % of the tokens in their ground truth data set. For all deployed bytecode, though, it bears a "false positive risk in case of factory contracts or dead code".

Our method of computing code skeletons is comparable to the first step for detecting similarities by [22]. 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. While the usage of signatures is in line with [18], we extend it beyond ERC-20 compliance by including other standards as well and by discussing partial compliance. Furthermore, as we aim at distinguishing pure tokens from those with additional functionality, we also discuss methods for automatic type inference.

XI. CONCLUSIONS

In this work, we contributed to the detection and classification of token contracts from the publicly available transaction data of the Ethereum main chain. More specifically, we examined token types and standards, and we discussed methods for identifying deployed contracts as tokens. Furthermore, we analyzed deployed contracts regarding the standards to which they comply, and the type of token they may represent. Moreover, we investigated the actual usage of tokens including code reuse.

Compliance. ERC-20 is the most general and most prevalent token standard. Of the deployed compliant token contracts, over 98 % are ERC-20 compliant. The non-fungible token standard ERC-721 is the second most common one, albeit only deployed in 1.6 % of the compliant contracts. Compliance with regulations is still work in progress regarding both, the standards and the deployments because regulations are either not yet enacted or diverge in different jurisdictions.

Identification. Token contracts complying with standards can be readily identified by extracting characteristic signatures from the bytecode (interface-oriented method) and by watching out for events mandated by the standard. Recognizing non-compliant tokens is more challenging, as they often implement the bare minimum of a token interface and may be mixed up with non-token contracts. Here a combination of different indicators seems promising. Ultimately, to reach certainty, a

thorough analysis of the bytecode and its semantics is necessary, which cannot be fully automatized yet. Our indicators, however, help to focus on relevant contracts. Moreover, they may be sufficient for a statistical analysis of contracts where the misclassification of single contracts does no harm.

Token Type. Distinguishing token types automatically in a heuristic fashion is feasible in a first step for pure tokens that do not provide other functionalities than token-related and neutral ones, and thus do not indicate a genuine product or service on-chain. Pure tokens amount to 70 % of the deployed compliant token contracts (66 % regarding unique bytecodes).

Code Reuse is far less pronounced for compliant token contracts than the average for Ethereum, with a factor between deployed bytecode and unique bytecode of less than 2 as opposed to a factor of nearly 80 on average. The unusually high code variability refers to the bytecode level. However, on the level of functionality we are presented with the more uniform picture of a high number of pure tokens.

Code Usage. The fully compliant token contracts are responsible for almost a quarter of all messages of Ethereum, which makes them the dominant application.

Limitations of Scope

We did not discuss regulations in depth as they are still in flux. The safety of the managed asset was out of scope as well as an analysis of security issues in token contracts. Moreover, we were not interested in the trading or market value of tokens.

Ideas for Further Work

Functionality that is split between several contracts is challenging to analyze. Heuristics for analyzing and understanding contracts are a promising field, as they may provide effective, yet simple means.

A semantic analysis of bytecode is indispensable for reliably determining what a smart contract actually implements. For a massive automated semantic code analysis, better tool support is desirable.

REFERENCES

[1] L. Oliveira, L. Zavolokina, I. Bauer, and G. Schwabe, "To token or not to token: Tools for understanding blockchain tokens," in *International Conference on Information Systems (ICIS)*. AIS eLibrary, 2018.

[2] J. Rohr and A. Wright, "Blockchain-Based Token Sales, Initial Coin Offerings, and the Democratization of Public Capital Markets," *Hastings LJ*, vol. 70, p. 463, 2019.

[3] FINMA, accessed 2019-10-12. [Online]. Available: https://www.finma.ch/en/documentation/dossier/dossier-fintech/entwicklungen-im-bereich-fintech/

[4] P. Hacker and C. Thomale, "Crypto-securities regulation: Icos, token sales and cryptocurrencies under eu financial law," *European Company and Financial Law Review*, vol. 15, no. 4, pp. 645–696, 2018.

[5] Bittrex, "Controlled wallet," 2017, accessed 2019-10-12. [Online]. Available: https://etherscan.io/address/0xA3C1E324CA1CE40DB73ED6026C4A177F099B5770#code

[6] Ethereum Wiki, "A next-generation smart contract and decentralized application platform," accessed 2019-02-02. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[7] B. Vitalik, "Blockchain and smart contract mechanism design challenges (slides)," 2017, accessed 2018-08-09. [Online]. Available: http://fc17.ifca.ai/wtsc/Vitalik%20Malta.pdf

[8] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, Tech. Rep., 2019, https://ethereum.github.io/yellowpaper/paper.pdf.

[9] "Contract ABI Specification," 2019, accessed 2019-09-09. [Online]. Available: https://solidity.readthedocs.io/en/latest/abi-spec.html

[10] F. Vogelsteller and V. Buterin, "ERC-20 token standard," 2015, accessed 2019-10-12. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[11] W. Entriken, D. Shirley, E. Evans, and N. Sachs, "ERC-721 non-fungible token standard," 2018, accessed 2019-10-12. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[12] J. Dafflon, J. Baylina, and T. Shababi, "ERC-777 token standard," 2015, accessed 2019-10-12. [Online]. Available: https://eips.ethereum.org/EIPS/eip-777

[13] W. Radomski, A. Cooke, P. Castonguay, J. Therien, E. Binet, and R. Sandford, "ERC-1155 multi token standard," 2015, accessed 2019-10-12. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155

[14] M. Kupriianov and J. Svirsky, "Base security token standard draft," 2019, accessed 2019-10-12. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1462

[15] J. Shiple, H. Marks, and D. Zhang, "Ldgrtoken standard draft," 2019, accessed 2019-10-12. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1450

[16] A. Dossa, P. Ruiz, F. Vogelsteller, and S. Gosselin, "Controlled token standard proposal," 2019, accessed 2019-10-12. [Online]. Available: https://github.com/ethereum/EIPs/issues/1644

[17] ——, "Security token standard proposal," 2019, accessed 2019-10-12. [Online]. Available: https://github.com/ethereum/EIPs/issues/1411

[18] M. Fröwis, A. Fuchs, and R. Böhme, "Detecting token systems on ethereum," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019.

[19] F. Victor and B. K. Lüders, "Measuring ethereum-based erc20 token networks," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019.

[20] S. Somin, G. Gordon, and Y. Altshuler, "Network analysis of erc20 tokens trading on ethereum blockchain," in *International Conference on Complex Systems*. Springer, 2018, pp. 439–450.

[21] L. Kiffer, D. Levin, and A. Mislove, "Analyzing ethereum's contract topology," in *Proceedings of the Internet Measurement Conference*, ser. IMC '18. New York, NY, USA: ACM, 2018, pp. 494–499. [Online]. Available: http://doi.acm.org/10.1145/3278532.3278575

[22] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," *arXiv preprint arXiv:1905.00272*, 2019.

[23] M. Di Angelo and G. Salzer, "Mayflies, Breeders, and Busy Bees in Ethereum: Smart Contracts Over Time," in *Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts (BCC'19)*. ACM Press, 2019.

[24] ——, "Wallet Contracts on Ethereum," *arXiv preprint: 2001.06909*, 2020.

[25] W. Chan and A. Olmsted, "Ethereum transaction graph analysis," in *12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2017, pp. 498–500.

[26] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding ethereum via graph analysis," in *IEEE INFOCOM: Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.

[27] D. Guo, J. Dong, and K. Wang, "Graph structure and statistical properties of ethereum transaction relationships," *Information Sciences*, vol. 492, pp. 58–71, 2019.

[28] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "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*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 900–903. [Online]. Available: http://doi.acm.org/10.1145/3236024.3264596

[29] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE Press, 2019, pp. 105–115.