Characterizing Types of Smart Contracts in the Ethereum Landscape

Monika di Angelo[0000−0002−4217−4530] and Gernot Salzer[0000−0002−8950−1551]

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

Abstract. After cryptocurrencies, smart contracts are the second major innovation of the blockchain era. Leveraging the immutability and accountability of blockchains, these event-driven programs form the basis of the new digital economy with tokens, wallets, exchanges, and markets, but facilitating also new models of peer-to-peer organizations. To judge the long-term prospects of particular projects and this new technology in general, it is important to understand how smart contracts are used. While public announcements, by their nature, make promises of what smart contracts might achieve, openly available data of blockchains provides a more balanced view on what is actually going on.
We focus on Ethereum as the major platform for smart contracts and aim at a comprehensive picture of the smart contract landscape regarding common or heavily used types of contracts. To this end, we unravel the publicly available data of the main chain up to block 9 000 000, in order to obtain an understanding of almost 20 million deployed smart contracts and 1.5 billion interactions. As smart contracts act behind the scenes, their activities are only fully accessible by also considering the execution traces triggered by transactions. They serve as the basis for this analysis, in which we group contracts according to common characteristics, observe temporal aspects and characterize them quantitatively and qualitatively. We use static methods by analyzing the bytecode of contracts as well as dynamic methods by aggregating and classifying the communication between contracts.

Keywords: bytecode analysis · empirical study · EVM · execution trace · smart contract · transaction data.

1 Introduction

Smart contracts (SCs) are small, event-triggered programs that run in a trustless environment on a decentralized P2P network. They may be self-sufficient in offering a service or may be part of a decentralized application (dApp). In the latter scenario, they implement trust-related parts of the application logic like the exchange of assets, while the off-chain frontend interacts with users and other applications. Frequently, SCs extend the concept of cryptocurrencies by implementing tokens as a special purpose currency.

Information on the purpose of SCs is often scarce. Colorful web pages advertise business ideas without revealing technical details, whereas technical blogs are anecdotal and selective. A comprehensive, but not readily accessible source is the blockchain data itself, growing continuously. Ethereum as the most prominent platform for SCs has recorded so far half a billion transactions, among them millions of contract creations. Although Ethereum is a well-researched platform, many questions about the usage of SCs remain unanswered.

Our work contributes to a deeper understanding of the *types of contracts* on the Ethereum blockchain, of their *quantities*, and of their *activities*. Starting from the publicly available transaction data of Ethereum's main chain we draw a comprehensive picture that accounts for the major phenomena. Along the way, we also investigate two claims frequently put forward: 'In Ethereum, the majority of deployed contracts remain unused' (claim 1), and 'Tokens are the killer application of Ethereum' (claim2).

Blockchain activities are usually described in terms of transactions clustered into blocks. This view is too coarse as contract activities become only visible when taking into account internal messages like calls, creates, and self-destructs. We base our static and dynamic analysis on the bytecode of contracts and their execution traces, define classes of contracts with common behavior or purpose, and observe their temporal evolution.

Roadmap. The next section summarizes related work. Section 3 clarifies terms, while section 4 defines the types of contracts we intend to investigate. Section 5 describes our methods in detail. We start our analysis with general statistics in section 6 before we examine various contract groups in section 7. Section 8 puts the pieces together to arrive at the general picture. Finally, section 9 summarizes our findings and concludes.

2 Previous Smart Contract Analyses

Contract Types. The authors of [13] find groups of similar contracts employing the the unsupervised clustering techniques *affinity propagation* and *k-medoids*. Using the program *ssdeep*, they compute fuzzy hashes of the bytecodes of a set of verified contracts and determine their similarity by taking the mean of the Levenshtein, Jaccard, and Sorenson distance. After clustering, the authors identify the purpose of a cluster by the associated names (and a high transaction volume). With k-medoids, they are able to identify token presale, DAO withdrawal, some gambling, and empty contracts. The analysis is based on the bytecode deployed until February 2017.

The authors of [1] provide a comprehensive survey of Ponzi schemes on the Ethereum blockchain. They start from bytecode with known source code and extend their analysis to bytecode that is similar regarding normalized Levenshtein distance, collecting 191 Ponzi schemes in total. The analysis is based on the bytecode deployed until July 2017. The authors of [4] also detect Ponzi schemes, with data until October 2017.

The authors of [5] investigate the lifespan and activity patterns of SCs on Ethereum. They identify several groups of contracts, provide quantitative and qualitative characteristics for each identified type, and visualize them over time. The analysis is based on bytecode and messages until December 2018.

Topology. In the empirical analysis [2], the authors investigate platforms for SCs, cluster SC applications on Bitcoin and Ethereum into six categories, and evaluate design patterns for SCs of Solidity code until January 2017.

The authors of [9] measure "the control flow immutability of all smart contracts deployed on Ethereum." They apply "abstract interpretation techniques to all bytecode deployed on the public Ethereum blockchain, and synthesize the information in a complete call graph of static dependencies between all smart contracts." They claim that debris from past attacks biases statistics of Ethereum data. Their analysis is based on bytecode of 225000 SCs until May 2017.

In a graph analysis [3], the authors study Ether transfer, contract creation, and contract calls. They compute metrics like degree distribution, clustering, degree correlation, node importance, assortativity, and strongly/weakly connected components. They conclude that financial applications dominate Ethereum. The analysis is based on the messages until June 2017.

In their empirical study on Ethereum SCs [12], the authors find that SCs are "three times more likely to be created by other contracts than they are by users, and that over 60% of contracts have never been interacted with" and "less than 10% of user-created contracts are unique, and less than 1% of contract-created contracts are so." The analysis is based on the messages until January 2018.

3 Terms

We assume familiarity with blockchain technology. For Ethereum basics, we refer to [8, 14].

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 Ether 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 EVM and are reflected in the execution trace and potential state changes.

For the sake of uniformity, we use the term message as a collective term for any (external) transaction or (internal) message, including SELFDESTRUCT operations that also may transfer Ether.

The Lifecycle of a Contract. For a contract to exist, it needs to be created by an account via *deployment code* (see below). As part of this deployment, the so-called *deployed code* is written to the Ethereum state at the contract's address. The contract exists upon the successful completion of the CREATE operation.

A contract may call other contracts, may create further contracts or may destruct itself by executing a SELFDESTRUCT operation. This results in a state change because the code at the contract's address is cleared. It is worth noting that this change happens only at the end of the whole transaction; until then the contract may still be called.

Deployment Code. A CREATE message passes bytecode to the EVM, the so-called deployment code. Its primary purpose is to initialize the storage and to provide the code of the new contract (the deployed code). However, deployment code may also call other contracts and may even contain CREATE instructions itself, leading to a cascade of contract creations. All calls and creates in the deployment code will seem to originate from the address of the new contract, even though the account contains no code yet. Moreover, the deployment code need not provide reasonable code for the new contract in the end. In particular, it may destruct itself or just stop the execution.

Gas. Users are charged for consumed resources. This is achieved by assigning a certain number of *gas units* to every instruction and each occupied storage cell, which are supposed to reflect the costs to the network of miners. Each transaction specifies the maximum amount of gas to be used as well as the amount of Ether the user is willing to pay per gas unit. The amount of gas limits the runtime of contracts, whereas the gas price influences the likelihood of a transaction to be processed by miners.

4 Definitions

We define the types of contracts that we will investigate in subsequent sections.

Dormant: a contract that has never been called since deployment. More precisely, we check the condition that the code (deployment or deployed) neither self-destructs in the transaction creating it nor receives a call later on.

Active: a contract that has received at least one successful call after the transaction creating it.

Self-destructed: a contract that successfully executed the operation SELFDESTRUCT at some point in time.

Short-Lived: a contract with an extremely short lifespan. More precisely, a short-lived contract self-destructs in the same transaction that has created it.

Prolific: a contract that creates at least 1 000 contracts.

Token: a contract that maintains a mapping from accounts to balances of token ownership. Moreover, it offers functions for transferring tokens between accounts. We approximate this informal notion by defining a token to be a contract that implements the mandatory functions of the token standards ERC-20 or ERC-721, or that is a positive instance of our ground truth [6].

Wallet: a contract that provides functionality for collecting and withdrawing Ether and tokens via its address. We consider a contract a wallet if it corresponds to one of the blueprints that we identified in earlier work as wallet code [7].

GasToken: a contract with the sole purpose of self-destructing when called from a specific address, this way causing a gas refund to the caller because of the freed resources. GasToken contracts can be identified by their behavior (numerous deployments and self-destructions) and subsequent code analysis.

Attack: a contract involved in an attack. Attack contracts stick out by unusual behavior (like executing certain instructions in a loop until running out of gas, or issuing an excessive number of specific messages); subsequent code analysis reveals the intention.

ENS deed: a contract created by another contract, the so-called ENS Registrar, that registers a name for an Ethereum address.

5 Methods for Identifying Contracts

Contract groups characterized by their function (like wallets) are detected mainly statically by various forms of code analysis; transactional data may yield further clues but is not essential. Groups characterized by operational behavior, on the other hand, require a statistical analysis of dynamic data, mostly contract interactions. This section describes the data forming the basis of our analysis as well as the methods we use. As a summary, table 1 relates the contract groups defined above to the methods for identifying them.

5.1 The Data

Our primary data are the messages and log entries provided by the Ethereum client `parity`, which we order chronologically by the triple of block number, transaction id within the block, and message/entry id within the transaction.

A message consists of type, status, context, sender, receiver, input, output, and value. Relevant message types are contract creations, four types of calls, and self-destructions. The status of a message can be 'success' or some error. 'Context' is the address affected by value transfers and self-destructs, whereas 'sender' is the user or contract issuing the message. The two addresses are identical except when the contract sending a message has been invoked by DELEGATECALL or CALLCODE. In this case, the sender address identifies the contract, whereas the context is identical to the context of the caller. For calls, 'receiver' is the address of the user or contract called, 'input' consists of a function identifier and argument values, and 'output' is the return value. For create, 'receiver' is the address of the newly created contract, 'input' is the deployment code, and 'output' is the deployed code of the new contract. 'Value' is the amount of Ether that the message transfers from 'context' to 'receiver'.

Log entries arise whenever the EVM executes a LOG instruction. They contain the context, in which the instruction was executed, and several fields with event-specific information. The most frequent log entries are those resulting from a 'Transfer' event. In this case, the context is the address of a token contract, whereas the fields contain the event identifier as well as the sender, the receiver, and the amount of transferred tokens.

A second source of data is the website `etherscan.io`, which provides the source code for 76.5 k deployments (0.4 % of all deployments), as well as supplementary information for certain addresses. The website speeds up the process of understanding the purpose of contracts, which otherwise has to rely on disassembled/decompiled bytecode.

5.2 Static Analysis

Code skeletons. To detect functional similarities between contracts we compare their *skeletons*, a technique also used in [5,11]. 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. Skeletons allow us to transfer knowledge gained for one contract to others with the same skeleton.

As an example, the 19.7 M contract deployments correspond to just 112 k distinct skeletons. This is still a large number, but more manageable then 247 k distinct bytecodes. By exploiting creation histories and the similarity via skeletons, we are able to relate 7.7 M of these deployments to some source code on Etherscan, an increase from 0.4 to 39.2 %.

Function signatures. The vast majority of deployed contracts adheres to the convention that the first four bytes of call data identify the function to be executed. Therefore, most deployed code contains instructions comparing this function signature to the signatures of the implemented functions. We developed a pattern-based tool that reliably[1] extracts these signatures from the bytecode. Thus we obtain for each deployed contract the list of signatures that will trigger the execution of specific parts of the contract. The signatures are computed as the first four bytes of the Keccak-256 hash of the function name concatenated with the parameter types. Given the signature, it is not possible in general to recover name and types. However, we have compiled a dictionary of 328 k function headers with corresponding signatures that allows us to find a function header for 59 % of the 254 k distinct signatures on the main chain.[2] 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.

Event signatures. On source code level, so-called events are used to signal state changes to the outside world. On machine level, events are implemented as `LOG` instructions with the unabridged Keccak-256 hash of the event header

[1] For the 76.5 k source codes from Etherscan, we observe 50 mismatches between the signatures extracted by our tool and the interface there. In all these cases our tool works actually correctly, whereas the given interface on Etherscan is inaccurate.

[2] An infinity of possible function headers is mapped to a finite number of signatures, so there is no guarantee that we recover the original header. The probability of collisions is low, however. E.g., of the 328 k signatures in our dictionary, only 19 appear with a second function header.

as identifier. We currently lack a tool that can extract the event signatures as reliably as the function signatures. We can check, however, 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: For the event *Transfer* and the 76.5 k source codes from Etherscan, we obtain just 0.2 k mismatches.

Code patterns. Some groups of bytecodes can be specified by regular expressions that generalize observed code patterns. A prime example are the code prefixes that characterize contracts and libraries generated by the Solidity compiler, but also gasTokens, proxies, and some attacks can be identified by such expressions.

Symbolic execution. To a limited extent, we execute bytecode symbolically to detect code that will always fail or always yield the same result, a behavior typical e.g. of early attacks targeting underpriced or poorly implemented instructions.

5.3 Dynamic Analysis

Time stamps. Each successful create message gives rise to a new contract. We record the time stamps of the start and the end of deployment, of the first incoming call, and of any self-destructs. There are several intricacies, since self-destructs from and calls to a contract address before and after deployment have different effects. Moreover, since March 2019 different bytecodes may be deployed successively at the same address, so contracts have to be indexed by their creation timestamp rather than their address.

Message statistics. By counting the messages a contract/code/skeleton sends or receives according to various criteria like type or function signature, we identify excessive or otherwise unusual behavior.

Temporal patterns of messages. Certain types of contacts can be specified by characteristic sequences of messages. This approach even works in cases where the bytecode shows greater variance. E.g., a group of several million short-lived contracts exploiting token contracts can be detected by three calls with particular signatures to the target contract followed by a self-destruct.

Log entry analysis. In contrast to the static extraction of event signatures from the bytecode, log entries witness events that have actually happened. For transfer events, the log information reveals token holders. Log analysis complements static extraction as it uncovers events missed by extraction as soon as they are used, whereas the extraction detects events even if they have not been used yet.

5.4 Combined Approaches

Grouping contracts by their purpose is a complex task and usually requires a combination of methods.

Interface method. Some application types, like tokens, use standardized interfaces. Given unknown bytecode, one can detect the presence of such an interface and then draw conclusions regarding the purpose of the code. In [10], the authors show that testing for the presence of five of six mandatory signatures is an effective method for identifying ERC20 tokens. As many token contracts are not fully compliant, we can lower the threshold for signatures even further. To maintain the level of reliability, we can additionally check, statically and dynamically, for standardized events.

Blueprint fuzzing. Many groups of contracts are heterogeneous. As an example, wallets have a common purpose, but there are hardly any similarities regarding their interfaces. In such a situation, we start from samples with available source code or from frequently deployed bytecode that we identify as group members by manual inspection (blueprints). We then identify idiosyncratic signatures of these blueprints and collect all bytecodes implementing the same signatures. By checking their other signatures and sometimes even the code, we ensure that we do not catch any bytecode outside of the group. If bytecode has been deployed by other contracts, we can detect variants of such factories by the same method; the contracts deployed by these variants are usually also members of the group under considerations. Altogether, this method turned out to be quite effective, though more laborious than the interface method. As an example, starting from 24 Solidity wallets and five wallets available only as bytecode we identified more than four million wallets [7] deployed on the main chain.

Ground truth. For the validation of methods and tools as well as conclusions about contract types, we compiled a ground truth, i.e. a collection of samples known to belong to a particular group [6, 7]. This required the manual classification of bytecodes (or more efficiently, skeletons) that are particularly active or otherwise prominent. A further approach is to rely on adequate naming of

Table 1. Methods for Identifying Groups of Contracts

	function signatures	event signatures	code patterns	symbolic execution	time stamps	message stats	message patterns	log entries	interface	blueprint fuzzing	ground truth
dormant					✓						
active					✓						
self-destructed					✓						
short-lived					✓		✓				
prolific						✓					
token	✓	✓				✓		✓	✓		✓
wallet	✓					✓			✓	✓	✓
ENS deed	✓		✓								
GasToken			✓								✓
attack			✓	✓		✓					

source code on Etherscan. Moreover, samples identified as positive instances of one group can serve as negative instances for another one.

6 Messages and Contracts

Messages. The 9 M blocks contain 590 M transactions, which gave rise to 1 448 M messages. That is, about 40 % of the messages are from users, who in turn send about two thirds of the messages to contracts. Regarding contract-related messages, of the 1 176 M messages to, from, or between contracts, 81.9 % were successful, while 18.1 % failed.

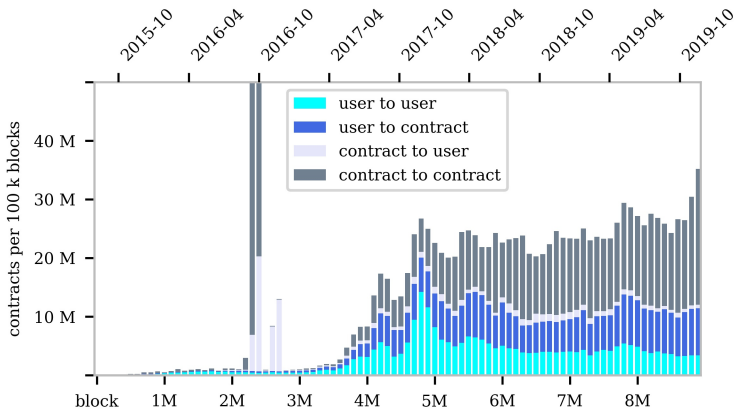

Fig. 1. Stackplot of user-messages (transactions) in blue and contract-sent messages in grey. The two clipped peaks around block 2.4 M depict 137 M and 89 M messages.

Figure 1 shows the distribution of the messages over time in bins of 100 k blocks (corresponding to about two weeks). Messages initiated by users are depicted in blue, while messages emanating from contracts are depicted in grey. The activities on the blockchain steadily increased in the course of the year 2017 and remained on the elevated level since the year 2018, but with more and more activities happening behind the scenes as the share of internal messages slightly keeps increasing. The peak after block 2 M shows the DoS attack in 2016 with the bloating of the address space in the first two elevated light grey bins and the countermeasure in the next two elevated light grey bins, both touching user addresses. At the same time, the unusually high contract interaction indicated by the two huge dark grey bins was also part of this attack.

Contracts. Figure 2 depicts the 19.7 M successful deployments over time, differentiated into user-created contracts in blue and contract-created ones in grey. Interestingly, 111 k different users created about 3 M (15.2 %) contracts, whereas just 21 k distinct contracts deployed 16.7 M (84.8 %) contracts.

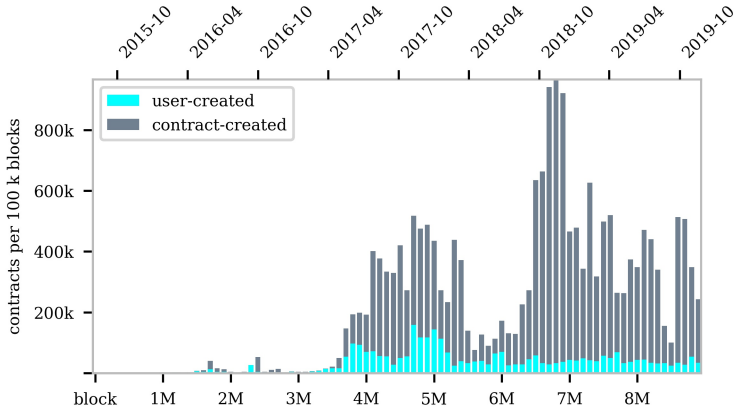

Fig. 2. Stackplot of user-created contracts in blue and contract-created ones in grey.

7 Groups of Contracts

In this section, we explore groups of contracts with specific properties that we defined in section 4. The methods for identifying the groups are indicated and described in section 5. Interesting properties are a high number of deployments with similar functionality, a high number of specific operations like CREATE, SELFDESTRUCT, or calls, as well as special bytecode or call patterns.

Dormant and Active Contracts. It has been observed (e.g. [5,12]) that many deployed contracts have never been called. As of November 2019, 62.4 % (12.3 M) of the successfully created contracts never received a call. On closer inspection, however, the picture is more differentiated (see section 8).

Self-Destructed Contracts. We count 7.3 M self-destructed contracts, which include 4.2 M short-lived contracts, 2.8 M GasTokens, and 0.2 M ENS deeds. The remaining 9 k self-destructed contracts contain a few (778) wallets. We arrive at about 8 k contracts that self-destructed for other reasons. Self-destructed contracts show minimal average activity.

Short-lived Contracts. We count almost 4.2 M short-lived contracts that were created by less than 1 k distinct addresses, mostly contracts. So far, the short-lived contracts predominantly harvested tokens, while some are used to gain advantages in gambling. The main reason for designing such a short-lived contract is to circumvent the intended usage of the contracts with which they interact.

Short-lived contracts appear in two types. Type 1 never deploys a contract and just executes the deployment code. With a total of 4.2 M, almost all short-lived contracts are type 1. Technically, they show no activity (as deployed contract). Type 2 actually deploys a contract that receives calls within the creating transaction, which includes the instruction SELFDESTRUCT that eventually destructs the contract at the end of the transaction. Type 2 is rare (52 k) and was mainly used during the DoS attack in 2016 with high activity.

Prolific Contracts. Interestingly, the set of contracts that deploy other contracts is small, while the number of their deployments is huge. Only 21 k contracts created a total of 16.7 M other contracts, which corresponds to 84.8 % of all Ethereum deployments. Still, the vast majority of these deployments (16.3 M) originate from the very small group of 460 prolific contracts, each of which created at least 1 k other contracts. Thus, the prolific contracts leave about 0.4 M deployments to non-prolific contracts and 3.0 M to users. Apart from over 16.3 M contract creations, the prolific contracts made 65 M calls, so in total, they sent about 4.5 % of all messages.

Tokens. We identified 226 k token contracts that comprise the 175.6 k fully compliant tokens overlapping with the 108.7 k contracts (23.5 k distinct bytecodes) from the ground truth. Tokens are a highly active group since they were involved in 455.5 M calls, which amounts to 31.5% of all messages.

Wallets. On-chain wallets are numerous, amounting to 4.3 M contracts (21.7 % of all contracts). Two-thirds of the on-chain wallets (67.7 %) are not in use yet. It might well be that wallets are kept in stock and come into use later. We define wallets to be *not in use* when they have never been called, neither received any token (which can happen passively), nor hold any Ether (which might be included in the deployment or transferred beforehand). Some never called wallets do hold Ether (385), or tokens (20 k), or both (40). Wallets show a low average activity with 30.8 M messages in total, which amounts to 2.1 % of all messages.

ENS Deeds. The old ENS registrar created 430 k deeds, of which about half (200 k) are already destructed. They exhibit almost no activity.

GasTokens. We identified 5.3 M deployments. About half of them (2.8 M) were already used and thus destructed, while the other half (2.5 M) are still dormant and wait to be used. GasTokens have 16 k distinct bytecodes that can be reduced to 16 skeletons. Naturally, gasTokens are to be called only once.

Attacks. Remnants of attacks also populate the landscape. Some argue that this debris puts a bias on statistics [9]. On the other hand, attacks are a regular usage scenario in systems with values at stake. We identified 49 k attacking contracts that were involved in almost 30 M calls, which amount to 2 % of all messages.

8 Overall Landscape

Dormant, active, and short-lived contracts. For a deployed contract to become active, it has to receive a call. Therefore, contracts fall into three disjoint groups regarding activity: short-lived contracts that are only active during deployment, dormant contracts that get deployed but have not yet been called, and active contracts that have been called at least once.

Figure 3 depicts the 7.4 M active contracts in bright colors and the 12.3 M dormant ones in light colors. Regarding the short-lived contracts, the common type 1 is shown in light pink, while the rare type 2 is shown in bright pink.

Weaken claim 1: a) A few wallets are used also passively when receiving tokens. b) The numerous gasTokens are in use *until* they are called. c) The numerous

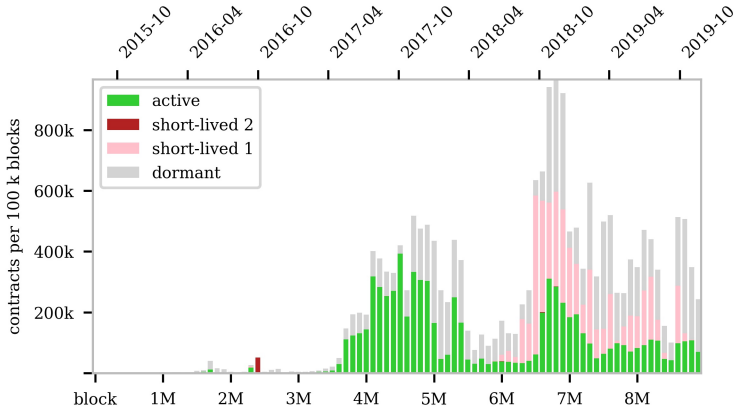

Fig. 3. Deployments of active, short-lived and dormant contracts.

short-lived contracts type 1 are improper contracts as they were never intended to be contracts and actually were active without being called directly. If we disregard the 4.2 M short-lived contracts, the share of never-called contracts drops to 41.1 % (8.1 M).

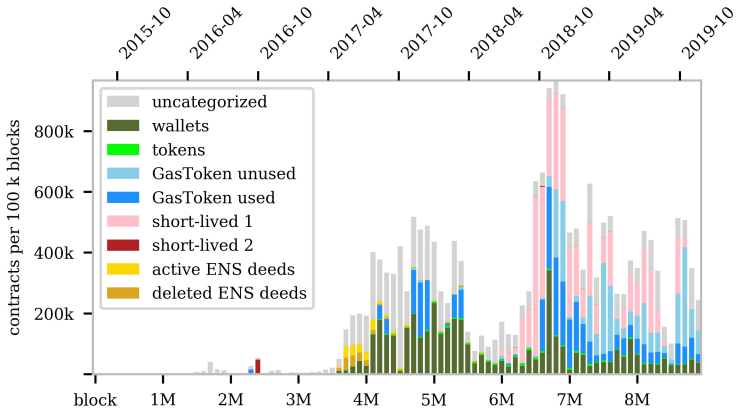

Fig. 4. Deployments of large groups.

Groups with plentiful contracts. Some groups of contracts are deployed in large quantities with a clear usage scenario. At the same time, they are not overly active. Figure 4 shows the deployments of the larger groups wallets, gasTokens, short-lived contracts, and ENS deeds. We also included the highly active, but

smaller group of 0.2 M tokens in light green to facilitate the comparison of group sizes. The most common contract type with 5.3 M deployments is the gasToken in blue (dark blue for the already used ones, and light blue yet unused). The second largest group of 4.3 M wallets is depicted in dark green. The almost equally common group of 4.2 M short-lived contracts is depicted in light and bright pink. Finally, there are the 0.4 M ENS deeds in yellow and brown.

Groups with high activity. Of the total of 1 448 M messages, 21 % are transactions, i.e. they come from users. The remaining 79 % can be attributed to contracts that generate the further messages as an effect of being called. Attributing the messages (calls) to the identified groups delivers a somewhat incomplete picture. First, we can clearly map only about half of the message to the groups. Secondly, some of the groups contain contracts that belong to a (decentralized) application that employs other contracts as well. This is especially true for the tokens that are part of a dApp. Third, we have not yet characterized exchanges, markets, and DAOs, that may make up a substantial part of the messages. Lastly, some quite active dApps may not fall into the mentioned groups.

As depicted in fig. 5, the 225.7 k tokens account for 31.5 % of all messages (calls from users included). This is followed by the 460 prolific contracts (4.5 %). Due to overlaps, we did not include wallets (2.4 %), short-lived (1 %) and attacking contracts (2 %) in the plot.

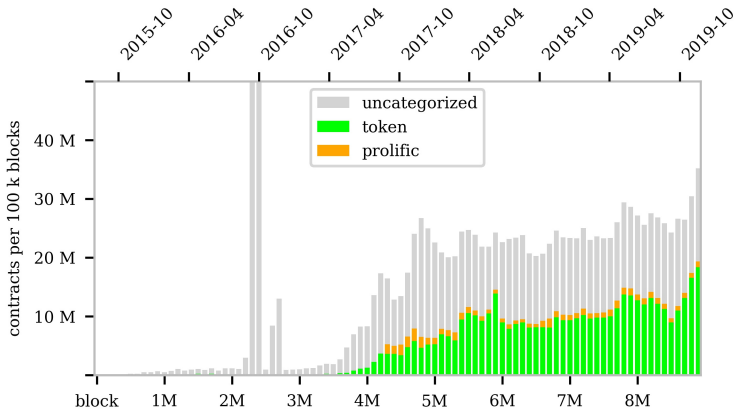

Fig. 5. Group-related message distribution as stackplot.

Regarding exchanges, Etherscan lists 180 addresses (164 user and 16 contract accounts) as well as 111 token accounts. The token activities are already included in our token count. According to Etherscan, the 180 exchange addresses had 76.5 M transactions (!), which corresponds to 12.9 % of the overall 590 M transactions. Of these, 6.6 M transactions (1.1 %) stem from the 16 contracts.

Corroborate claim 2: a) Token interactions are overwhelmingly high. b) Tokens are the base of an ecosystem that includes also wallets for managing tokens and exchanges for trading them. The contracts and messages of the token ecosystem account for such a large part of the activity that tokens justifiably are referred to as the killer application of SCs.

9 Conclusion

We defined groups of smart contracts with interesting properties. Furthermore, we summarized methods for identifying these groups based on the bytecode and call data that we extracted from the execution trace. With this, we characterized many of the smart contracts deployed in Ethereum until November 2019. Based on the identified groups and their interactions, we elaborated an overall picture of the landscape of smart contracts and tested two claims.

Compared to [12], this work draws a much more detailed and recent picture. This work extends the analysis of [5], as it uses similar groups, but adds further groups and methods. For tokens, it builds on [6, 10], for wallets on [7]. In summary, the added value lies in the detail concerning both, temporal aspects and number of groups, as well as in the variety of the employed methods.

Observations. We conclude with some observations resulting from our analysis.

Code variety. As has been widely observed, code reuse on Ethereum is high. Therefore, it is surprising that some authors still evaluate their methods on samples of several hundred thousand *contracts*. Picking one bytecode for each of the 112 k skeletons results in a *complete coverage*, with less effort. By weighting quantitative observations with the multiplicity of skeletons, it is straight-forward to arrive also at conclusions about contracts.

Creations. Deployment is dominated by just a few groups of contracts: gasTokens, wallets, short-lived contracts, and ENS deeds. GasTokens make gas tradable; they are actually in use, even though it is debatable whether they constitute a reasonable use scenario. Wallets are a frequent and natural use case. Short-lived contracts, only active during the creating transaction, are borderline: the contracts they are targeting were probably not intended to be used this way. The ENS deeds were replaced by a more efficient solution without mass deployment. Claim 1 that the majority of contracts remains unused may seem true on a superficial level but becomes less so upon closer inspection.

Calls. Concerning contract activity, there is one dominating group. Tokens form a lively ecosystem resulting in numerous wallets on-chain (and also off-chain) and highly active crypto exchanges (albeit with most of it being non-contract activity). Our work thus confirms that tokens are the killer application before anything else (claim 2). The other groups contribute comparatively little to the call traffic. However, there are still large grey areas in the picture of messages.

Self-destructions. Our work gives a near-complete account of self-destructing contracts. The majority are contracts that fulfill their purpose during deployment

and self-destruct to save resources. After further discounting gasTokens and ENS deeds, only 8 k contracts (of 7.3 M) remain that self-destructed for other reasons.

Future Work. Exchanges should be examined in more detail with respect to activity and regulations. Moreover, markets and DAOs seem worth exploring more closely regarding governance and usage scenarios. Furthermore, a focus on dApps would be interesting. Finally, our methodology would be well complemented by a behavioral analysis of contract activity.

References

1. Bartoletti, M., Carta, S., Cimoli, T., Saia, R.: Dissecting ponzi schemes on ethereum: identification, analysis, and impact. arXiv:1703.03779 (2017)
2. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. In: International Conference on Financial Cryptography and Data Security. pp. 494–509. Springer (2017)
3. Chen, T., Zhu, Y., Li, Z., Chen, J., Li, X., Luo, X., Lin, X., Zhange, X.: Understanding ethereum via graph analysis. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications. pp. 1484–1492. IEEE (2018)
4. Chen, W., Zheng, Z., Cui, J., Ngai, E., Zheng, P., Zhou, Y.: Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In: Proceedings of the 2018 World Wide Web Conference. pp. 1409–1418. WWW '18, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland (2018). DOI: 10.1145/3178876.3186046
5. Di Angelo, M., Salzer, G.: 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)
6. Di Angelo, M., Salzer, G.: Tokens, Types, and Standards: Identification and Utilization in Ethereum. Intl. Conf. Decentralized Applications and Infrastructures (DAPPS) (2020)
7. Di Angelo, M., Salzer, G.: Wallet Contracts on Ethereum. arXiv preprint: 2001.06909 (2020)
8. Ethereum Wiki: A next-generation smart contract and decentralized application platform, https://github.com/ethereum/wiki/wiki/White-Paper, accessed 2019-02-02
9. Fröwis, M., Böhme, R.: In code we trust? In: Data Privacy Management, Cryptocurrencies and Blockchain Technology, pp. 357–372. Springer (2017)
10. Fröwis, M., Fuchs, A., Böhme, R.: Detecting token systems on ethereum. In: International conference on financial cryptography and data security. Springer (2019)
11. He, N., Wu, L., Wang, H., Guo, Y., Jiang, X.: Characterizing code clones in the ethereum smart contract ecosystem. arXiv preprint arXiv:1905.00272 (2019)
12. Kiffer, L., Levin, D., Mislove, A.: Analyzing ethereum's contract topology. In: Proceedings of the Internet Measurement Conference 2018. pp. 494–499. IMC '18, ACM, New York, NY, USA (2018). DOI: 10.1145/3278532.3278575
13. Norvill, R., Awan, I.U., Pontiveros, B.B.F., Cullen, A.J., et al.: Automated labeling of unknown contracts in ethereum. In: ICCCN 26th Int. Conf. on Computer Communications and Networks. IEEE (2017)
14. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Tech. rep., Ethereum Project Yellow Paper (2018), https://ethereum.github.io/yellowpaper/paper.pdf