Mayflies, Breeders, and Busy Bees in Ethereum: Smart Contracts Over Time

Monika di Angelo
Technische Universität Wien, Austria
Eurecom, France
monika.di.angelo@tuwien.ac.at

Gernot Salzer
Technische Universität Wien, Austria
Eurecom, France
gernot.salzer@tuwien.ac.at

ABSTRACT

Smart contracts on a blockchain are programs running in a distributed, transparent, and trustless environment, being one of the major assets of this new technology. They give rise to innovative applications and business models, with their potential and lasting impact still open. In this situation, it is interesting to understand what smart contracts are actually doing. While public announcements, by their nature, make promises of what smart contracts might achieve, the openly available data of blockchains provides a more balanced view on what is actually going on.

In this paper, we analyze the activities of smart contracts on the Ethereum blockchain, the most prominent platform for smart contracts with all blockchain data visible. However, contracts operate behind the scenes. Their activities are only accessible by looking beyond the mere blockchain data that records external transactions. We also use *all internal messages* caused by contracts interacting with other addresses. In particular, we investigate the activities of smart contracts in their quantitative and temporal aspects. Based on lifespan and activity patterns, we identify particular groups like mayflies, loners, breeders, busy bees, sleepers, self-destructed and bonkers contracts and visualize their temporal characteristics. To gain insights into the purpose of these smart contracts we perform a basic analysis of code and message content including deployment code. We consider data up to Ethereum block 6 900 000 (end of 2018).

CCS CONCEPTS

• **Computing methodologies** → *Distributed computing methodologies*; • **Information systems** → *Computing platforms*; • **Computer systems organization** → Peer-to-peer architectures;

KEYWORDS

contract activity, Ethereum, smart contracts, temporal aspects

ACM Reference Format:
Monika di Angelo and Gernot Salzer. 2019. Mayflies, Breeders, and Busy Bees in Ethereum: Smart Contracts Over Time. In *Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts (BCC '19), July 8, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3327959.3329537

BCC '19, July 8, 2019, Auckland, New Zealand

ACM ISBN 978-1-4503-6785-1/19/07…$15.00
https://doi.org/10.1145/3327959.3329537

1 INTRODUCTION

Smart contracts are still being hyped with promises on their potential benefits. They may handle valuable assets like crypto-currencies or tokens in a transparent, decentralized manner. As the core of decentralized applications (dApps), they may encapsulate essential parts of the business logic.

Suitable information on how smart contracts are actually used and which goals they do achieve, is scarce, though. Colorful web pages advertize 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 smart contracts has recorded so far billions of transactions, among them millions of contract creations. Over the past two years, scientific publications have started to analyze this wealth of data from different angles. On the one end of the spectrum, we find surveys that evaluate the data statistically, on the other end new methods are developed for reverse engineering particular smart contracts. Given the complexity of the problem, it is not surprising that many questions, especially regarding smart contracts, have remained open.

The aim of this paper is to obtain a deeper understanding of the *types of contracts* on the Ethereum blockchain and of their *activities*. Our analysis intends to contribute to the bigger picture of how Ethereum smart contracts are actually used. We define groups of contracts with common properties and observe their temporal evolution. We quantify the messages they are involved in throughout their lifespan, from creation to selfdestruction. For a qualitative assessment, we look at the bytecode of contracts, which is available as deployed code (content of a contract) and as deployment code (code executed during contract deployment).

Compared to related work, the novel aspects of our approach are threefold: In addition to the external transactions, we include the internal messages caused by contracts, which only exist within the execution environment of the EVM. Moreover, instead of overall average numbers, we investigate how the numbers of numbers of contracts and messages evolve over time. Finally, we coarsely analyze the semantics of contract code in order to distill usage patterns.

Roadmap. The next section defines basic terms and notions that are essential for understanding the paper. Section 3 lays out the data, methods, and tools for our analysis. Section 4 looks at the data set and its structure as a whole. Section 5 continues with a temporal view on specific groups of contracts, while section 6 concentrates on the interactions between active contracts. Section 7 concludes with a discussion of related work and a summary of our results.

2 TERMS AND DEFINITONS

We assume familiarity with blockchain technology. A great introductory book on cryptocurrencies is [11]. Regarding Ethereum, we refer to the Ethereum basics [5, 15] and Buterin's blockchain and smart contract mechanism design challenges [14].

2.1 Accounts, Transactions, and Messages

Ethereum[4] 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.

For the sake of uniformity, we use the term message as a collective term for any (external) transaction or (internal) message, including selfdestructs that transfer value between addresses.

2.2 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. Henceforth, the contract can be called (become active). A contract call may include call data and/or value, but it can also have empty data and zero value. When called, the contract's code is executed by the EVM. Contracts can be organized so that with a call only specific parts of their code are executed, like in the case of a function call.

A contract may call or create other 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 upon completion of the transaction; until then the contract may still be called.

2.3 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 execution.

2.4 Interaction and Activity

The EVM provides the following operations that are relevant for our analysis: CREATE (`0xF0`) and SELFDESTRUCT (`0xFF`) to deploy and destruct a contract, as well as CALL (`0xF1`), CALLCODE (`0xF2`), DELEGATECALL (`0xF4`), and STATICCALL (`0xFA`) for interactions between contracts. Unless stated otherwise, we consider only successful operations.

Interaction means to send or receive one of the four types of call messages.

Contract activity. The creation of a contract is considered as an activity of the code containing the CREATE instruction, not of the contract being created. Activities of the deployment code, on the other hand, are attributed to the new contract, as they are executed, according to EVM semantics, already in the context of the contract being created. Finally, we count as contract activity whatever results from the create and call instructions of the deployed code.

The differences between the various types of calls are quite technical and largely irrelevant for our purpose. As we are interested in the activities of the code, not of the context in which the code is executed, we attribute calls to the address containing the code.

Gas. Users are charged for all 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.

3 DATA, METHODS, AND TOOLS

Throughout the paper we abbreviate large numbers by denoting multiplicative factors of 1 000 and 1 000 000 with the letters k and M, respectively.

3.1 The Data

The activities on the Ethereum chain are usually described in terms of transactions clustered into blocks. This view is too coarse for our purpose since transactions may be composed of several internal messages that create, call, or destruct contracts. We therefore use messages as our central data structure, which are characterized by the following fields.

Message type: relevant types for this paper are CREATE, the four call types CALL, CALLCODE, DELEGATECALL, and STATICCALL as well as SELFDESTRUCT.

Sender (from): address of the external account or of the code containing the instruction causing the message. For messages issued by deployment code during a create, *from* is the future address of the contract being deployed.

Receiver (to): address of external account or contract receiving the message. For contracts, this is the location of the code being called or where a newly created contract is being deployed. For SELFDESTRUCT messages, *to* is the receiver of the balance of the destructed account.

Context: address of account whose balance is reduced by a transfer of Ether and who is destructed by SELFDESTRUCT. It is identical to *from* except for messages resulting from code invoked by CALLCODE and DELEGATECALL, since these instructions execute foreign code without context change.

Block id (bid): number of the block where the message occurs.

Transaction id (tid): index of transaction within the block that causes the message.

Message id (mid): index of message within the transaction.

The triple (*bid*, *tid*, *mid*) identifies a message uniquely and serves as an abstract timestamp that can be related to real time via the Unix timestamp of the block. The lexicographic ordering of the triples corresponds to the temporal ordering of messages.

Parent id (pid): identifies the parent message, or is NULL for the first message of a transaction. Message $p = (bid, tid, pid)$ is the parent of message $m = (bid, tid, mid)$ if p calls code that contains the instruction responsible for message m.

Input data: input data for the called contract, or the deployment code for CREATE messages.

Output data: return value of the called contract, or the deployed code for CREATE messages.

Status: status code indicating whether the message succeeded or why it failed. Apart from success and EVM-related errors that revert the transaction, the effect of a message can be deliberately reverted by the instruction REVERT. Moreover, a message may complete successfully but may be reverted later due to a failure elsewhere in the transaction. A failed message consumes computational resources even when its effects are reverted, and thus is relevant for a comprehensive analysis.

Further fields and message types not relevant for this paper record the amounts of Ether, gas, fees, endowments, etc.

Source of the data. The message data described above was extracted from the traces provided by the Ethereum client `parity`, version 2.1.4-beta in archive mode, with a patch to fix a bug regarding CALLCODE traces. On recent hardware the archive node takes 10 days to synchronize with the main chain and occupies 2 TB. Loaded into an SQL database the messages occupy 180 GB (not counting the storage for indexes).

3.2 Code Analysis

We perform elementary bytecode analysis to identify code that is not useful, semantically equivalent, or token compliant.

3.2.1 Bonkers Code. The code of some contracts (deployed code or deployment code) is not useful. We define code to be 'not useful' (bonkers) if its execution unconditionally leads to a fail, a revert, or returns a fixed value (or nothing) without causing a state-change. To detect bonkers code we execute the code symbolically up to the first jump, call, or halting instruction, but perform unconditional JUMPS if the jump destination is pushed on the stack immediately before the jump. Moreover, we keep track of state changing operations like SSTORE and LOG. This way we catch the following situations.

- Endless loop (resulting in an out-of-gas error).
- Stack under- and overflows, illegal instructions, bad jump destinations.
- REVERT instruction that returns a fixed value.
- RETURN instruction that returns a fixed value, without prior state change.
- STOP instruction without prior state change.

These tests are reliable in the sense that code declared as bonkers is indeed not useful in the above sense. The tests may miss some useless code, though. A comprehensive test has to include gas consumption – without it we are dealing with an undecidable problem – as well as code deployed elsewhere on the chain.

3.2.2 Code Skeletons. To detect functional similarities between bytecodes we consider their *skeletons*. A code skeleton is obtained by replacing 20-byte-addresses occurring literally as the argument of a PUSH operation by placeholders, and by removing swarm hashes (metadata not affecting the functionality) and constructor arguments. This transformation captures some of the variability introduced by the Solidity compiler, as well as contracts that interact with differing hard-coded addresses but are equal otherwise.

3.2.3 Contract interfaces. Most contracts, in particular those obtained from Solidity code, follow the convention that the first four bytes of the input data specify the called function. This so-called *function signature* is computed as the first four bytes of the hashed function header consisting of function name and argument types. By identifying function signatures at appropriate locations in the bytecode, it is possible to reconstruct partially the interface.

Obviously, it is not possible to regain the header from the function signature, which is a hash fragment. We use a directory of about 275 k signature-header pairs that we extracted from available Solidity sources. This way, one often obtains translations of function signatures to possible headers that help in understanding the meaning of messages and contracts.

3.2.4 Manual code inspection and transaction analysis. We use the tool `Vandal` [2, 12] to disassemble EVM bytecode and to display the control flow graph. For decoding and visualizing the messages within a transaction and to automatize some tasks, we use homegrown Python scripts.

3.3 Visualization

We use `matplotlib`[1] and `numpy`[2] to visualize the temporal evolution of activities and the distribution of contracts.

Temporal views (like the one in Fig. 1) show the number of contracts over consecutive periods of 10 k blocks (roughly 1.7 days), from genesis to block 6.9 M. The lower horizontal axis of diagrams indicates the time in blocks, the upper one in years and months. As further frame of reference, pink overlays indicate periods of attack. These were, from left to right, the DAO and the DoS attack in 2016, and a spam attack as well as the parity theft and the parity freeze in 2017. Occasionally, we clip the vertical axis when less relevant data (most notably the DoS attack of 2016) exceeds the more important parts of the plot.

Apart from stackplots, we use a grey line to plot the 690 values. In spite of 10 k aggregation, successive values may still differ significantly. To make trends more visible, we add a black line that shows the moving average over 25 values.

Where reasonable, we plot the proportion of particular contracts (or messages) to all contracts (or messages) in a second figure below the main plot.

In cases where numbers differ by orders of magnitudes, we use a logarithmic scale.

[1] https://matplotlib.org
[2] https://www.numpy.org

4 GENERAL STATISTICS

In this section, we analyze the data set as a whole, while the following two sections focus on particular groups of contracts.

4.1 Messages

We analyze data of the Ethereum main chain up to (but not including) block number 6 900 000, which was mined on Dec 16, 2018. The 6.9 M blocks contain 360 M transactions, which gave rise to 900 M messages.

Figure 1 shows the distribution of messages over time. The blue areas, light and dark, represent external messages initiated by users, while the grey and black areas correspond to internal messages emanating from contracts. The activities on the blockchain steadily rose during 2017 and have remained on a high level since the beginning of 2018, but with more and more activities happening behind the scenes as the share of internal messages increases.

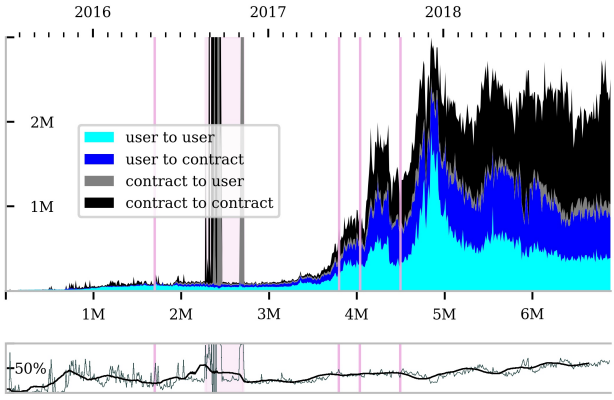

Figure 1: Distribution of messages over time. The upper part shows a stackplot of user-sent external messages in blue and contract-sent internal messages in grey and black. The lower part indicates the percentage of internal messages with regard to all messages, per 10k blockrange.

Of the 713 M contract-related messages (messages to, from, or between contracts) 72 % were successful and 28 % failed. Of the latter, 80% failed due to an error at EVM-level (like out of gas or stack overflow), 5 % were deliberately reverted by the contracts, and 15 % were initially successful but later reverted because of a failure elsewhere.

4.2 Contracts

Of the 72 M addresses appearing as sender or receiver of a message, about 50 M are currently stored by the network nodes. The other addresses either never came into existence because the transaction was reverted, or they were cleared at some point from the state space. A bit more than 11 M addresses belong to contracts. Figure 2 depicts the creation of contracts over time. We see that also this particular activity is dominated by contracts: external accounts (users) created about 2.1 M contracts, while the remaining 9 M contracts were created by just 17 100 contracts. Of the latter, almost all were created by contracts already deployed; only 38 k contracts were created by deployment code during the creation of some other contract.

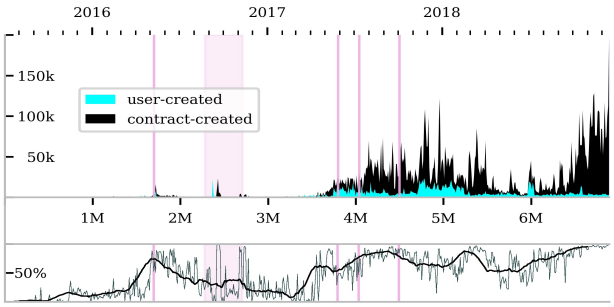

Figure 2: Distribution of contract creations. The upper part shows a stackplot of user-created contracts in blue and contract-created contracts in black. The lower part indicates the percentage of contract-created contracts with regard to all contract creations, per 10 k block range.

4.3 Contract code

The 11 134 232 successfully created contracts originate from 472 495 different deployment codes, which deployed 177 759 distinct bytecodes. The variability shrinks even more when we look at the skeletons (cf. section 3.2) of contract code. We arrive at just 94 311 distinct code skeletons for 11.1 M deployed contracts, a reduction by a factor of 100.

Table 1 shows the numbers of codes and skeletons as used by external accounts (users) and contracts, respectively. With 4263 skeletons for 9 M contracts the variability is particularly low for contracts created by contracts. This can be explained by the facts that only 17 k contracts are creators, that the code of the new contracts is part of their code, and that some contracts are particularly prolific with up to 1.5 M deployments per contract.

Variability is higher for contracts created by external accounts, but still low. This, together with the fact that some of the accounts deploy up to 400 k contracts, suggests that many of these contracts are not deployed by hand but by programs outside of the chain.

Table 1: Numbers on Contract Creation. Of the accounts creating contracts, 17 % are contracts, who create 81 % of the new contracts. The variability in the deployed contracts is low: on average, external accounts (users) reuse skeletons for 12 new contracts, while contracts reuse them 2 109 times.

	external accounts	contracts
# of creators among	85 828	17 100
# of contracts created by	2 145 615	8 988 617
# of unique deployment codes used by	347 571	125 115
# of unique deployed codes used by	171 881	6 095
# of unique deployed skeletons used by	90 436	4 263
max # of contracts by single creator among	391 518	1 539 319

Etherscan.io collects Solidity source code for the contracts on the chain and verifies that it actually compiles to the deployment code used for contract creation. At the end of 2018, source code for 53 290 contract addresses was available, corresponding to 52 684 unique deployment codes. Of these, only 1 469 contracts (1 377 unique codes) were created by contracts. Compared to the entirety of 472 495 distinct deployment codes, this means that the source code for about 89% of the deployed contracts is not directly accessible. However, one should bear in mind that the source code of contract-created contracts is usually literally included in the source code of the master contract deployed by the external account and thus might be recovered by tracing it back to an ancestor with known source code.

5 TEMPORAL PERSPECTIVES

In this section, we explore the lifespan and activities of interesting contract types based on numbers, plots visualizing the numbers, and interpretations. First, we look at the reportedly large group of never called contracts. Next, destructed contracts are differentiated according to the number of selfdestructs (single vs. multiple). As a third group, we look at peculiarities of mayfly contracts. For sleepers, we examine sleep durations and wake-up times. Bonkers contracts are viewed with regard to their occurrence and code. As a last group, we analyze breeders and their creations.

5.1 Loners

It has been observed in the past that the number of contracts never called is large. E.g., for data up to January 2018, [9] quantifies the share with 60 % and states that *'the large fraction of them that have remained dormant is nevertheless surprising'*. A first count seems to confirm this observation. Up to the end of 2018, 63 % of the addresses that where the target of a successful create operation have never received a call.

A closer look at contract deployment shows a more differentiated picture. The aim of contract creation, by users or contracts, may not be the deployment of a new contract but rather the execution of the deployment code itself. The latter is inaccessible from the outside; in particular, it cannot receive calls. When deployment finishes, the purpose of the create command is fulfilled and no code is deployed. This pattern has recently become quite popular and will be investigated in section 5.3. If we disregard such non-deployments, the share of never-called contracts drops to 46 %. This number is still high but becomes less surprising when further examining the purpose of the contracts.

We define a *loner* to be a contract where code gets deployed that is never called. More precisely, we check the condition that the code (deployment or deployed) neither selfdestructs in the transaction creating it nor receives a call later on. There were 5 156 658 such loners within the study period, which account for 46 % of all creations. The other 6.0 M contracts are either short-lived (section 5.3) or active (section 6).

Fig. 3 depicts the temporal distribution of loner creations. In the second half of 2017, we see several peaks, with the highest one beyond 100 k loner creations per 10 k block range. They can be attributed to the *GasToken* contract, which until the end of 2018 has deployed more than a million of identical copies of a code

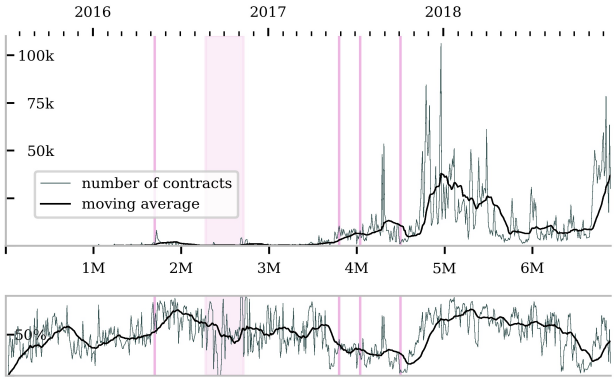

Figure 3: Occurrence of loners. The upper plot shows the number of loners created within a block range, the lower plot their percentage with regard to all creations.

with just 22 bytes. Each of the tiny contracts corresponds to a gas token which can be purchased as gas reserve, and thus render gas tradable. These contracts essentially store gas that they release by self-destruction as soon as they are called. So far only 0.2 % have been called, leaving the rest as loners.

Next in the list of most frequent loners, we find wallet contracts that sum up to 2.2 M. It should be noted, though, that there are roughly 1 M contracts with the same bytecode that are already in use. It might well be that wallets are deployed on stock and only gradually come into use. Moreover, a wallet address need not be called to receive or hold tokens, so some loners might be effectively in use in a wider sense.

Altogether we expect that a considerable number of these contracts will receive a call in the future, effectively leaving even less than 46 % loners.

5.2 Destructed Contracts

We call a contract *destructed* if it executed a selfdestruct operation at some point in time. We count 2 540 995 of such contracts within the study period. Figure 4 shows the temporal distribution of selfdestructions. A contract can selfdestruct successfully multiple times as long as this happens within a single transaction. A few contracts utilized this possibility (e.g. for an attack) and performed hundreds of selfdestructs.

There were 24 438 879 successfully executed selfdestructs within the analyzed messages, about 10 times more than destructed contracts. Only 48 506 contracts selfdestructed multiple times (some of them up to 10 920 times). Almost all of them were part of the DoS attack in October 2016 with a total of almost 22 M successful selfdestructs.

The rising number of selfdestructs starting around block 6 M can be largely attributed to mayflies (see the next section).

5.3 Mayflies

We call contracts with an extremely short lifespan *mayflies*. More precisely, a mayfly is a contract that selfdestructs in the same transaction as it has been created. During the study period we count

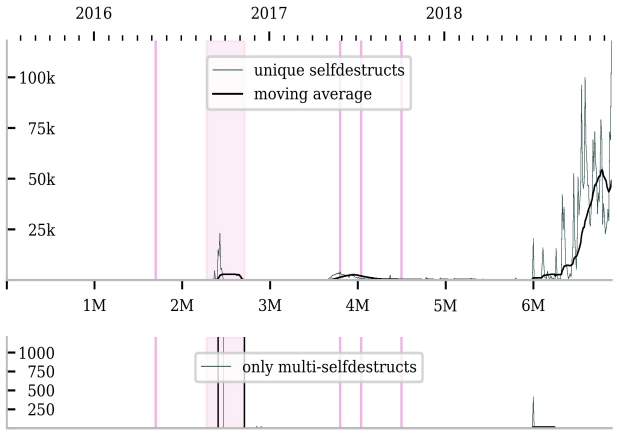

Figure 4: Contract destruction. The upper plot shows the number of contracts that selfdestructed within a block range. The lower plot depicts the accumulated selfdestructs of contracts that selfdestructed multiple times. The vertical axis of the lower plot was clipped, since numbers greater than 1000 only appeared during the DoS period of 2016.

1 856 655 mayflies that were created by just 8 992 distinct addresses. Most addresses are contracts; the 405 users among them account for 7 460 mayflies only. 975 mayflies created 17 681 other mayflies.

Figure 5 depicts the distribution of mayflies over time. It shows a small number of mayflies during the DoS attack in 2016, while mayflies in large quantities start to appear in the middle of 2018 and even dominate contract creation in late 2018.

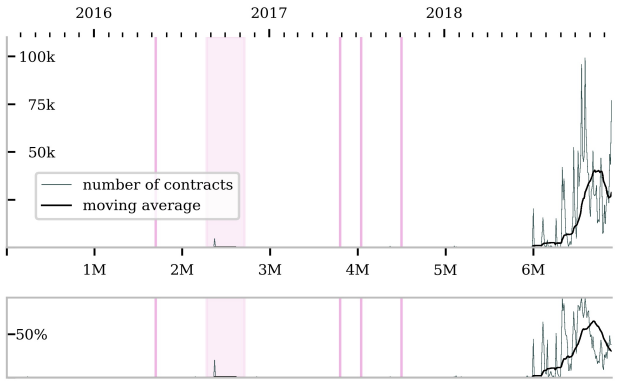

Figure 5: Occurrence of mayflies. The upper plot depicts the number of mayflies created within a block range, the lower plot shows their percentage with regard to all creations.

5.3.1 Construction of a Mayfly. To understand the construction of a mayfly, the EVM deployment mechanism is crucial. Deployment code is executed as if originating from the new contract (here the mayfly). Any deployed code will exist at the new contract's address only upon successful completion of the deployment code.

In the majority of cases, a mayfly was *just* the deployment code that ends with a selfdestruct (equally executed as if it came from the mayfly, thereby destructing it). For this, the mayfly does not even need a deployed code.

In rarer cases, mayflies do have a deployed code, which is even called. This requires subsequent messages within the *same* transaction: first the create and then one or more calls to the mayfly, especially to a function containing the selfdestruct. Note that a contract is only destructed upon successful completion of the transaction containing the selfdestruct (thereby allowing multiple selfdestructs of the same contract).

5.3.2 Usage Scenario: Token Harvesting. This scenario exemplifies the predominant pattern we found for mayflies. Typically, several dozens of contracts are created in a row that claim free tokens, transfer them to a fixed address, and destruct themselves while still in the deployment phase. These multi-creations are either themselves implemented as deployment code or are part of a deployed contract that can be called multiple times.

To see why this has become a popular coding pattern we take a look at token contracts and airdrops. For funding a business idea in the blockchain era, you could deploy a token contract with the promise that token holders will profit once your enterprise skyrockets. To create an initial community for your idea and advertise it, you distribute parts of your tokens for free. A common implementation of this so-called airdrop is to provide a function that for a certain period of time transfers free tokens to any address that calls it. To counter-act hoarding of tokens, the function blacklists addresses once they have received their share.

```
MESSAGES OF TRANSACTION 6409724/70
43132826
└─45884811 call('')=''
   ├─68649799 CREATE(174439375)=''
   │  ├─42687408 call('')=''
   │  ├─42687408 call(balanceOf(68649799))='1285.6'
   │  ├─42687408 call(transfer(43132826,'1285.6'))='1'
   │  └─46429544 SELFDESTRUCT
   ├─68649800 CREATE(174439375)=''
   │  ├─42687408 call('')=''

ACCOUNTS
  42687408: 0xe30a76ec9168639f09061e602924ae601d341066
  43132826: 0xcb44e6ac67fca11e46a0e4d0758455a06846fb5b
  45884811: 0x955074147610509817a092db3a7e353478cd1bf1
  46429544: 0x808284ad89f21c512f9a5d43ad24aaf79780849a
  68649799: 0x837f4c693c134014811524b0f96fd1d1ee41910b

DATA
 174439375: 0x6080604052348015610010576000080fd5b50 ...
```

Figure 6: Trace of a mayfly harvesting tokens. During deployment, the mayfly calls the fallback function, `call('')`, of token contract 42687408 to request free tokens, queries the precise amount, transfers 1285.6 tokens to the user, and selfdestructs. The trace continues with 39 further mayflies.

This is where token harvesting mayflies come into play. Figure 6 shows the start of such a harvesting sequence in block 6 409 724. The external user 43132826 calls an existing contract at address 45884811, which creates new contracts in a loop. The first one is deployed at address 68649799. From this new address, the deployment code calls the fallback function of the token contract at

address 42687408 (*NewIntelTechMedia*) and receives free tokens. The amount often diminishes with each airdrop, so the code queries the precise amount with `balanceOf`. Finally, 1285.6 tokens are transferred to the account of the external user, and the deployment code selfdestructs. (Note that we abbreviate addresses and data by 4-byte-integers.)

Using deployment code is essential for token harvesting. First, each mayfly operates under a new address and thus is eligible for free tokens from the point of view of the token contract. Second, mayflies pass the 'is human' test that some contracts employ. By checking the absence of code at the caller's address, they try to ensure that they interact with an external address. Since there is no deployed code yet while the deployment code is executed, mayflies are taken to be users.

This practice has even become a business model. Mayfly breeders have been deployed than can be called by anyone. The breeder will generate as many mayflies as the gas supplied with the call permits. Each mayfly harvests tokens for the initial caller, except for the last one, which transfers the tokens to an address of the breeder provider.

Depending on the token contract, functions like `getTokens`, `Mine`, or `register` need to be called in place of the fallback function, or the airdrop functionality is integrated into `balanceOf` or `transfer`. The harvesters also differ regarding where the selfdestruct occurs. Until the end of 2018, 1.7 M harvesters collected 136 different types of token for 6 674 beneficiaries.

5.3.3 Purpose of Mayflies. So far, mayflies were primarily used for token harvesting, followed by advantage gains in gambling and games (like correctly 'guessing' pseudo-random numbers). In each case, the reason for using deployment code is to bypass the interaction pattern intended by the target contracts. This could be called exploitation.

5.4 Sleepers

We refer to the number of blocks between the creation of a contract and it receiving the first call as its *sleep time*. We define a *sleeper* to be a contract with a long sleep time, where a reasonable value for 'long' will be determined below by analyzing the data.

Fig. 7 depicts the sleep time for all contracts that ever received a call, i.e., that are not loners. Each value represents the number of contracts that slept for this many blocks before their first call, aggregated over 10 k blocks. We use a logarithmic scale since most contracts sleep for a short time only, being used briefly after creation. The timeline is capped at 3 500 k blocks since longer sleep times are rare. The maximum sleep time is 5.8 M blocks.

As we use a logarithmic scale, a sharp change of slope in the upper regions indicates a noteworthy point. The first two occur at 48 k blocks (8 days) and 130 k blocks (25 days), respectively, and are marked by green lines. Accordingly, we distinguish short-, medium-, and long-term sleepers depending on whether they sleep up to 8, 8 to 25, or more than 25 days after creation.

Another way of looking at sleep times is to consider the wake-up times. Fig. 8 shows all wake-up times color-coded according to sleep duration. Short sleep, depicted in grey, is the norm for most contracts. Medium and long sleep, depicted in red and black, is rare. The first noticeable increase in contracts that woke up concurrently

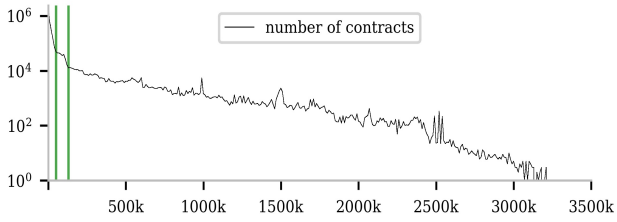

Figure 7: Sleep time of contracts. The horizontal axis indicates the sleep time in 10 k blocks. The green vertical lines mark the first two significant changes in slope, which occur at 8 and 25 days of sleep, respectively.

is during the DoS attack in 2016 (pink overlay). The attack area shows an increased number of long-term sleepers at the onset and at the end, as well as an increased number of short-term sleepers in the first half.

Generally, the number of sleepers is equal to the number of active contracts by definition (both need to have at least one call after creation). The number of sleepers increases towards the later blocks since contract deployment markedly increased after block 3.5 M (cf. fig. 2). Accordingly, the increase in the grey area after block 3.5 M reflects the general increase in contract deployment and usage.

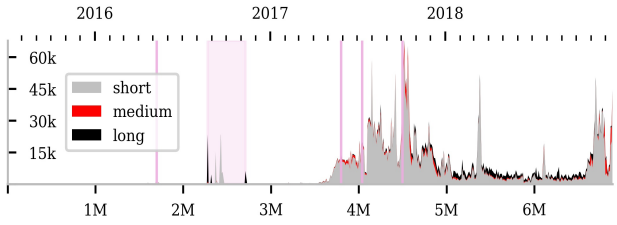

Figure 8: Wake-up times of sleeping contracts. The grey area represents short-term sleepers, i.e. contracts that slept for less than 8 day (48 k blocks). The red area is stacked on top and represents contracts with a sleep time between 8 and 25 days (130 k blocks). The black line is again stacked on top and represents long-term sleepers (first call after 25 days).

5.5 Bonkers Contracts

We define a contract as *bonkers* when it has no useful deployed code (see section 3.2.1 for the definition of 'useful'). Some of these codes can be executed successfully, while others always fail or revert. Without the empty non-contracts deployed by mayflies (see above), we count 44 883 bonkers contracts. Figure 9 shows their creation over time. Right after the DAO attack in 2016 (first pink vertical line, before block 2 M), we see the largest number of bonkers contracts: 10 673 empty contracts were created by unusual deployment code, possibly in preparation of the DoS attack to follow. By their nature, bonkers contracts are not expected to receive many calls, and indeed, more than two thirds were loners. Surprisingly,

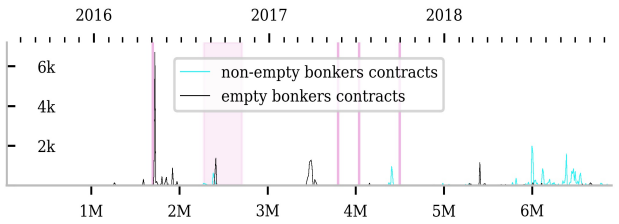

Figure 9: Occurrence of bonkers contracts. The black line depicts the number of empty bonkers contracts created within a block range, the cyan line the number of non-empty bonkers contracts.

14 k bonkers contracts (including the empty contracts from above) did receive at least one call. This phenomenon can be traced to the DoS attack and counterattack that involved calls to various empty contracts. It can be observed in Fig. 8, which shows the wakening of long-term sleepers (black) at the onset of the DoS attack.

The bytecode of bonkers. The majority of bonkers contracts are the remnants of deployments that went awry. In these cases, the deployed code is either empty (53 %) or stops, throws, reverts, or fails within a few instructions (45 %). Most of the latter contracts show the signature of the Solidity compiler, i.e., they start with `0x60606040` or `0x60806040`.

The remaining 764 contracts are part of the DoS attack and subsequent cleanup in 2016. We find 635 contracts with excessive amounts of instructions that were underpriced or inefficiently implemented at that time, like `EXTCODESIZE`, `EXP`, `SLOAD`, or `BALANCE`. The remaining 129 pseudo-contracts do not contain executable code but are collections of 512 or 1024 addresses that were loaded and called as part of the counterattack.

5.6 Breeders

Table 1 showed that 17 100 contracts created a total of 9.0 M contracts. But the group of contracts responsible for the majority of these creations is in fact much smaller. We define a *breeder* to be a contract that creates at least 1 000 contracts. There are just 276 breeders, which account for 8.76 M creations. This leaves slightly over 230 k contract creations to non-breeders contracts and around 2.1 M to users.

Breeders are responsible for 1.7 M mayflies (94% of the mayflies). Other breeders created a total of 2.7 M wallets. Among the top breeders we also find GasToken2[3] with 1 M contracts and ENS-Registrar[3] with 0.4 M contracts.

Apart from nearly 9 M contract creations, breeders made 27 M calls, so in total they sent about 4 % of all messages.

6 ACTIVE CONTRACTS

We call a contract *active* if it receives at least one call during its lifetime. We count 4.2 M active contracts within the study period, of which 0.7 M have selfdestructed at some point. The maximal numbers of calls sent/received by a single contract were 19.4 M for

[3]The translation of address to name is from Etherscan.

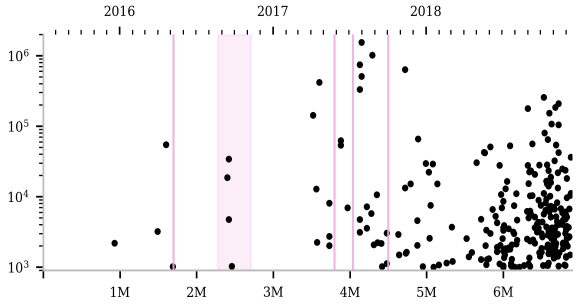

Figure 10: Creation of breeders. The vertical axis uses a logarithmic scale for the number of contracts that a breeder created so far.

incoming and 18.5 M for outgoing messages. The most interactive contract sent and received a total of 21.5 M messages.

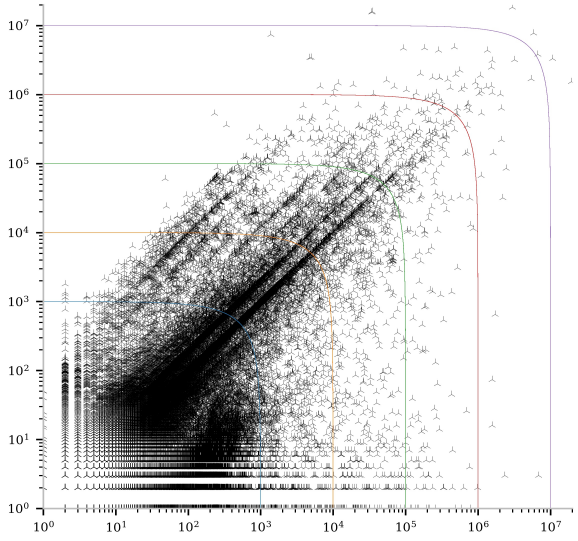

Figure 11: Interactions of active contracts. The horizontal axis indicates the number of incoming calls, the vertical axis the number of outgoing calls, both in a logarithmic scale. The colored lines connect points with the same sum of incoming and outgoing calls.

Fig. 11 shows the number of interactions for all active contracts, where each mark represents one of the 4.2 M contracts, with the counts of incoming and outgoing calls serving as coordinates. Contracts near the diagonal have roughly the same number of incoming and outgoing calls, whereas contracts close to an axis show an asymmetric behaviour by either mostly calling or mostly getting called. The extreme cases are contracts that never send – they lie on the horizontal axis – and contracts that received a single call – they form the first vertical stripe to the right of the vertical axis. The stripes parallel to the diagonal are series of contracts that have a fixed ratio of incoming to outgoing messages. Without logarithmic

scaling the stripes would form straight lines leaving the origin at different angles. The topmost of the more clearly visible stripes corresponds to contracts that on average send 100 calls for every incoming one.

6.1 Busy Bees

We call a contract with more than 1 000 interactions (sent or received messages) a *busy bee*. Only 27 k contracts, or 0.6 % of all active contracts, are busy bees. They deserve their name as they account for 505 M sent and received messages, slightly over 60 % of all messages. In figure 11, busy bees are located from the first connecting line (blue) outwards.

If we look at busy bees with at least 10 000 interactions, they still account for 471 M interactions while being only 4 945 contracts. Even busier bees with more than 100 000 interactions account for 392 M interactions, while being only 898 contracts. With at least 1 M interactions we are down to 96 contracts with an interaction volume of 255 M. Finally, 7 contracts have over 10 M interactions each and about 98 M interactions in total.

6.2 Casual Workers

Most contracts had just a few interactions. We call contracts with less than 1000 interactions *casual workers*. 3.6 M contracts had at most 10 interactions, 509 k contracts had between 11 and 100 interactions, and 103 k contracts had between 101 and 1000. So, more than 99% of the active contracts are casual workers.

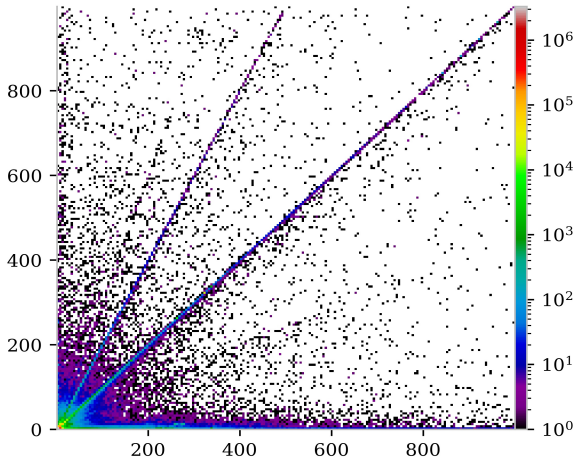

Figure 12: Heatmap of contracts with at most 1000 incoming and outgoing calls each. The colormap uses a logarithmic scale.

Fig. 12 depicts the casual workers in a heatmap with 200 bins. The incoming calls are represented on the horizontal axis and the outgoing calls on the vertical axis. The number of contracts are color-coded with a logarithmic scale. A box in the heatmap represents a range of 5 calls each. The tiny greyish area with a dark red dot next to the origin represents the roughly 3.5 M casual workers with no more than 10 interactions. The straight lines emanating from the origin correspond to contracts with the same ratio of outgoing to incoming calls. The most clearly visible lines, from the diagonal upwards, represent ratios of 1:1, 2:1, and 3:1, respectively.

7 DISCUSSION

7.1 Related Work

Of the papers analyzing Ethereum smart contracts, a fair part deals with security issues, which are beyond the scope of our paper. Similarly, papers concerning the control flow in smart contracts [6, 7] or Solidity aspects [3, 8, 13] are only distantly related.

Papers addressing the behaviour of smart contracts are closer to our work. In [10], the authors represent code by vectors obtained by counting critical instruction patterns during symbolic execution, and use these vectors to detect semantic clones of programs, like bytecode obtained from the same source with different compiler versions or settings. Their analysis is based on 2 117 Solidity smart contracts from November 2017. In [16], the authors describe methods for analyzing and reverse engineering smart contracts given as bytecode. They collect statistics of all deployed contracts until January 2018 to show how their complexity increases, and use their decompilation techniques to decode four popular contracts.

Some papers analyze blockchain activities by considering addresses and transactions as a network. In [1], the authors propose graph-based quantitative indicators and use them to relate internal activities of the Ethereum mainchain until summer 2017 to external events like the foundation of the Ethereum Alliance. They are not interested in detailed contract activities, though.

Most closely related to our work is the analysis in [9], where the authors examine the creation and interaction of contracts based on the transactions and internal messages until January 2018. Their findings include that contracts are three times more often created by other contracts than by users; that over 60 % of contracts have never been interacted with; less than 10 % of user-created contracts are unique, and less than 1 % of contract-created contracts are so; that the fraction of activity involving contracts remains constant at about one third; and that code is often re-used and highly similar.

7.2 Summary of Our Results

Our analysis is based on the external and internal messages of the Ethereum mainchain until the end of 2018. On a methodological level, we employed both, a numerical analysis and an investigation of code and interaction patterns. By alternating the utilzation of structural and behavioral similarities, it was possible to classify large classes of contracts.

Mayflies (contracts that selfdestruct in the same transaction as they are created) appear in large quantities as a recent phenomenon starting around block 6 M. Their main purpose is to circumvent intended interaction patterns and to take advantage of unaware design of contracts. The main application area is token harvesting, followed by exploits of games with bad random number generators.

Deployment code is used extensively beyond deploying contracts. Its mechanics have stayed the same since genesis in 2015, but seem to be not sufficiently known among contract developers.

Selfdestruction of contracts has become popular again. While we saw 20 M multi-selfdestructs in the DoS attack in October 2016, we

are now confronted with 1.8 M selfdestructing deployment codes. Selfdestruction *per se* is beneficial by freeing resources, while its massive use has been an indication of attacks and exploits so far.

Bonkers Contracts (non-sensical contracts) occasionally occur as the result of deployment mishaps or during attacks.

Code reuse became even more massive than already pointed out in earlier work. When comparing the 11.1 M contracts deployed until the end of 2018 to the corresponding 94 k different code skeletons, we observe a reduction by a factor of 100. This phenomenon is also a consequence of the high degree of automation in deployment, including contract creations from external addresses.

The message volume increased drastically in the middle of 2017 and has remained high since. The share of messages involving contracts rose to over 70 % in 2018.

Contract Creations peaked in the second half of 2017, dropped in mid 2018 and showed again a marked increase towards the end of 2018 that can be attributed to mayflies.

The distinction between user- vs. contract-created contracts becomes less relevant as external addresses use programs to deploy contracts in large numbers. The two methods differ in deployment costs and transparency, though.

A small number of breeders (contracts creating a large number of contracts) are responsible for the vast majority of contract creations.

Loners (never called contracts) are still widespread and comprise almost half of the deployed contracts. Their existence, however, becomes less surprising when looking at their purpose. Many of them will probably be called in the future (like GasToken and wallets), turning them into sleepers.

Sleepers (contracts with an extended delay between creation and first call) are common, though most contracts, if called at all, receive their first call rather sooner than later. Given the diverse application areas of contracts, it seems difficult to state general rules.

Busy Bees (contracts sending and receiving over 1 k calls) are rare but account for over 60 % of all messages. In contrast, the overwhelming majority of contracts (more than 99 %) only has 10 interactions or less.

7.3 Final remarks

When trying to understand the use of smart contracts, the transactions recorded on the blockchain are only of limited use, since smart contracts act mostly behind the scenes. We therefore recorded all creations, calls, and selfdestructs during execution and treated each contract as a separate entity. This choice of granularity is debatable since some contracts may in fact belong to the same application, so what we observe may sometimes be internal traffic. A reasonable clustering of contracts into applications is desirable but seems out of reach for the moment.

The numerical analysis of messages yields first insights, but also produces oddities (like the large number of loners) that can only be resolved by digging deeper. The analysis of code and interaction patterns seems indispensable for reliable conclusions.

The huge amount of contracts and messages is overwhelming at first, but clustering of contracts with similar behavior and code allowed us to understand classes of contracts with millions of instances and messages. We can confirm the well-known fact that tokens (and their ICOs) are still popular in the Ethereum ecosystem, followed by games and gambling, exchanges, and wallets. We also encountered numerous exploits, especially in late 2018. Moreover, we identified breeders and mayflies as use cases that currently dominate contract creation.

Blockchains in general and Ethereum in particular are still evolving, and some design choices lead to probably unintended use cases. Admitting arbitrary code during contract deployment provides maximal flexibility, while it opens an attack window at a point that should be transparent and routine. The difficulty of writing safe contracts may contribute to the heavy reuse of code. In any case, the proliferation of highly similar contracts that have to be stored permanently puts an unnecessary burden on the state space. A solution might be to deploy parametrized contracts by copying them verbatim to the chain (instead of generating them by code), to make the creation of new contracts costlier, and to incentivize the creation of audited libraries via royalties paid by their users.

REFERENCES

[1] Andra Anoaica and Hugo Levard. 2018. Quantitative Description of Internal Activity on the Ethereum Public Blockchain. In *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*. IEEE, 1–5.

[2] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *arXiv:1809.03981* (2018).

[3] T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*. 442–446. https://doi.org/10.1109/SANER.2017.7884650

[4] Ethereum Community. 2018. Ethereum Homestead. https://media.readthedocs.org/pdf/ethereum-homestead/latest/ethereum-homestead.pdf.

[5] Ethereum Wiki. [n. d.]. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper Accessed 2019-02-02.

[6] Michael Fröwis and Rainer Böhme. 2017. In Code We Trust? In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 357–372.

[7] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158136

[8] Péter Hegedus. 2018. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. In *IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 35–39.

[9] Lucianna Kiffer, Dave Levin, and Alan Mislove. 2018. Analyzing Ethereum's Contract Topology. In *Proceedings of the Internet Measurement Conference 2018 (IMC '18)*. ACM, New York, NY, USA, 494–499. https://doi.org/10.1145/3278532.3278575

[10] Han Liu, Zhiqiang Yang, Chao Liu, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. 2018. EClone: Detect Semantic Clones in Ethereum via Symbolic Transaction Sketch. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 900–903. https://doi.org/10.1145/3236024.3264596

[11] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press.

[12] Smart Contract Research at USYD. 2018. Vandal. https://github.com/usyd-blockchain/vandal.

[13] Roberto Tonelli, Giuseppe Destefanis, Michele Marchesi, and Marco Ortu. 2018. Smart contracts software metrics: a first study. *arXiv:1802.01517* (2018).

[14] Buterin Vitalik. 2017. Blockchain and Smart Contract Mechanism Design Challenges (slides). http://fc17.ifca.ai/wtsc/Vitalik%20Malta.pdf Accessed 2018-08-09.

[15] Gavin Wood. 2018. *Ethereum: A secure decentralised generalised transaction ledger*. Technical Report. Ethereum Project Yellow Paper. 1–32 pages. https://ethereum.github.io/yellowpaper/paper.pdf.

[16] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: Reverse engineering ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, Vol. 1.