Characteristics of Wallet Contracts on Ethereum

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

***Abstract*—For the management of cryptocurrencies or cryptographic tokens, many users employ a software wallet that facilitates the interaction with a blockchain in general or with on-chain programs (smart contracts) in particular. While many blockchain wallets execute their core program code off-chain, some wallets implement core functionality on-chain as smart contracts with the intent to increase trust and security by using transparent and verifiable execution.**

In this work, we investigate smart contracts for wallets with regard to the functionality that makes use of cryptographically secured blockchain technology. We focus on wallet contracts deployed on Ethereum, as it is the most prominent platform for tokens and smart contracts with readily available data. We aim at a better understanding of this frequently deployed group of smart contracts by analyzing characteristics of wallet contracts and grouping them into six types. To this end, we present approaches to identify wallet contracts by analyzing source code, bytecode, and execution traces extracted from transaction data. Moreover, we investigate usage scenarios and patterns. From the derived data, we extract blueprints for wallets and compile a ground truth. We provide numbers and temporal perspectives regarding the creation and use of wallets.

***Index Terms*—analysis, EVM bytecode, smart contract, transaction data, wallet**

I. Introduction

Wallets keep valuables, credentials, and items for access rights (like cash, licenses, credit cards, key cards) in one place, for ease of access and use. On the blockchain, cryptocurrencies play a role similar to cash, while cryptographic tokens are a universal tool for handling rights and assets. Blockchain wallets manage the cryptographic keys required for authorization and implement the protocols for interacting with blockchains.

Smart contracts are described as a disruptive financial technology, and crypto tokens are often termed the killer application of smart contracts. Token and wallet contracts form a large ecosystem with regard to the number of smart contracts and transactions as well as market value. They already started to change financial processes and markets.

Wallet contracts hold cryptocurrencies and access to tokens and may offer advanced methods for manipulating the assets. Simply by introducing the role of an 'owner' it becomes possible to transfer all assets of a wallet contract transparently and securely in a single transaction. More refined methods include multi-signature wallets, which grant access only if sufficiently many owners agree.

Ethereum is the major platform for smart contracts and tokens. This paper investigates the usage and purpose of wallet contracts on the main chain of Ethereum qualitatively as well as quantitatively. In particular, we address the following questions.

- How can deployed wallet contracts be identified from transaction data?
- Regarding functionality, which types of wallet contracts are deployed?
- When and in which quantities are wallets created, and how many are actually used?

Methodologically, we start from available source code of wallets and determine characteristic functions. Then we search the deployed bytecode for variants of the wallets with the same profile. Some wallets can also be detected by their creation history or by the way they interact with other contracts. We group the wallets according to their functionality and collect creation and usage statistics from the transaction data. Finally, we relate wallets to other frequently occurring contract types.

This work contributes to a better understanding of the actual usage of smart contracts in general and wallet contracts in particular. This may concern users, investors, companies, developers, and regulators. To facilitate assessment, we seek methods for detecting, classifying and monitoring the rapidly growing ecosystem around crypto assets. Specifically, we extract a comprehensive collection of blueprints for wallet contracts and thereby compile a ground truth. Moreover, the wallet blueprints and the features they implement may serve as a resource for designing further decentralized trading apps. For an extended version of this work see [1].

Roadmap: Section II clarifies terms and presents our methods for bytecode analysis. Section III discusses methods for the identification of potential wallet contracts. Section IV describes characteristic features of wallets and categorizes them into types. Section V analyzes interactions of wallets. Section VI compares our approach to related work. Finally, section VII concludes with a summary of our results.

II. Bytecode Analysis

A. Terms and Data

Ethereum [2]–[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

978-1-7281-7091-6/20/$31.00 ©2020 IEEE

contract. Messages are not recorded on the blockchain, since they are deterministic consequences of the initial transaction. They only exist in the execution environment of the Ethereum Virtual Machine (EVM) and are reflected in the execution trace and potential state changes. We use 'message' as a collective term for any (external) transaction or (internal) message.

Unless stated otherwise, statistics refer to the Ethereum main chain up to block 10 000 000 (mined on May 4, 2020). We abbreviate factors of 1 000 and 1 000 000 by the letters k and M, respectively.

To a large extent, our analysis is based on the EVM bytecode of deployed contracts. If available we use verified source code from `etherscan.io`, but relying solely on such contracts would bias the results: in contrast to 25.4 M successful create operations, there are verified source codes for 86.5 k addresses (0.34 %) only.

B. Code Skeletons

To detect functional similarities between contracts we compare their *skeletons*. These 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 impact on the functional behavior, like the swarm hashes added by the Solidity compiler or hard-coded addresses of companion contracts. Skeletons allow us to transfer knowledge gained about one contract to others with the same skeleton. Note that the 25.4 M contract deployments correspond to only 280 k distinct bytecodes and 131 k distinct skeletons. Thus, we are able to relate 11.5 M of these deployments to some source code on `etherscan.io`, an increase from 0.34 to 45 %.

C. Contract Interfaces

Most contracts in the Ethereum universe adhere to the ABI standard [5], which identifies functions by signatures that consist of the first four bytes of the Keccak-256 hash of the function name and the parameter types. The bytecode of a contract contains instructions to compare the first four bytes of the call data to the signatures of its functions. To understand the implemented interface of a contract, we extract it from the bytecode, and then try to restore the corresponding function headers.

1) Interface Extraction: We developed a pattern-based tool to extract the interface contained in the bytecode. As ground truth for validation, we used the combination of verified source code, corresponding bytecode, and ABI provided by Etherscan. The signatures extracted by our tool differed from the ground truth in 86 cases. Manual inspection revealed that our tool was correct also in these cases, whereas the ABIs did not faithfully reflect the signatures in the bytecode (e.g. due to compiler optimization or library code).

Before applying the tool to all deployed bytecodes, a few considerations are due. Apart from a small number of LLL or Vyper contracts, the validation set consists almost exclusively of bytecode generated by the Solidity compiler, covering virtually all its releases (including early versions). Regarding the large group of 11.3 M deployed contracts (259 k codes, 130 k skeletons) generated by the Solidity compiler, the validation set is thus representative.[1]

Another interesting group of deployed contracts consists of 9.8 M short contracts (24 k codes, only 420 skeletons) without entry points. They are mostly contracts for storing gas (gasToken), but also some proxies (contracts redirecting calls elsewhere) and contracts involved in attacks. A third large group are 4.2 M contracts that self-destruct at the end of the deployment phase (mayflies [35]). They cannot be regularly called and thus do not contain any entry points either.

What remains are 1.4 k contracts (595 codes). For them, our tool shows an error rate of 8 %, estimated from a random sample of 60 codes that we manually checked.

2) Interface Restoration: To understand the purpose of contracts we try to recover the function headers from the signatures. As the signatures are partial hashes of the headers, we use a dictionary of function headers with their 4-byte signatures (collected from various sources), which allows us to obtain a function header for 58 % of the 294 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 90 % (or 89 %) when picking a code (or a deployed contract) at random.

III. Identifying Potential Wallets

We define a proper wallet to be a contract whose sole purpose is to manage assets. In contrast, contracts that serve other purposes as well are termed non-wallet contracts. The latter group includes all applications that require Ether or tokens.

Our approach first identifies potential wallet contracts and then checks if the bytecode actually implements a proper wallet. In this section, we discuss four methods to identify potential wallets, of which we utilize the last three in combination for the first step. In section IV we elaborate on the second step, the check whether they implement a proper wallet.

A. Wallets as Recipients of Ether or Tokens

In a broad sense, any address that has sent or received Ether or tokens at some point in time may be called a wallet, regardless of being a contract or user address. Addresses transferring Ether can be easily identified by looking at the senders and receivers of successful messages with an Ether value greater than zero. Token transfers are harder to detect, as the addresses receiving tokens are not directly involved in the transfer.

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

[2] An infinity of possible function headers is mapped to a finite number of signatures, so there is no guarantee that we have recovered the original header. The probability of collisions is low, however. E.g., of the 354 k signatures in our dictionary only 21 appear with a second function header.

TABLE I
ACCOUNTS HAVING HELD ETHER OR TOKENS

	only Ether	only tokens	both	neither
contracts	1.8 M	5.9 M	0.4 M	17.3 M
users	38.9 M	19.5 M	31.8 M	

We identify token holders as the addresses that appear in calls of the methods `transfer(address,uint256)`, `transferFrom(address,address,uint256)`, `mint(address,uint256)`, `balanceOf(address)` or in events of the type `Transfer(address,address,uint256)`.

Table I lists the number of accounts that ever held Ether or tokens. The number of user accounts that never held Ether or tokens is difficult to assess. According to `etherscan.io`, the number of addresses in the state was 96.3 M. However, as accounts no longer in use are removed from the state space, it is smaller than the sum of the numbers in the table above.

Limitations. The liberal definition of wallets as senders or receivers of assets gives a first idea of the quantities involved. As we will see below, many wallet contracts have not yet been used and thus cannot be detected by the above method. Inherently, this method yields many non-wallets, while it misses the large number of unused proper wallets. Thus, we did not use this method for the further analyses.

B. Identifying Wallets by their Interface

Given the source code of a wallet contract, we can use its bytecode and ABI to identify *similar* contracts on the chain. Employing the methods described in section II, we first locate all deployed contracts with identical bytecode or skeleton.

In order to capture variants of the already found wallets, we then look for contracts that implement the same characteristic functions as a given wallet. This is achieved by allowing for some fuzziness regarding additional signatures. For this, the choice of signatures is crucial. One has to avoid using unspecific signatures for the search or being too liberal with additional signatures. This can be achieved by checking the functionality of contracts, e.g. by reading the bytecode or by looking at the interaction patterns of deployed contracts. As we will see, the number of wallet blueprints is small enough to actually read its code.

Limitations. This approach misses contracts that do not adhere to the ABI specification. Moreover, contracts with similar signatures may implement different functionality, and thus may not be related.

C. Identifying Wallets by their Factory

Wallets appearing in large quantities are usually deployed by a small number of contracts (so-called factories) or external addresses. Factories can be located either by the same methods as wallets, or by specifically looking for addresses that create many other contracts and by verifying that the latter are indeed wallets. Once the factories are identified, a database query is sufficient to select all wallets created by them.

Limitations. When looking for factories, we may encounter the same problems as for the wallet interfaces. Otherwise, this method is robust as the signatures and the interaction patterns of factories are distinctive. This approach misses wallets not deployed by factories.

D. Identifying Wallets by their Name

To detect wallets in a more systematic fashion and to include also less popular ones, we scanned the 86.5 k source codes on `etherscan.io` for contracts containing the string 'wallet' or 'Wallet' in their name.

Limitations. This heuristic approach yields false positives and misses wallets named differently. Again, checking the code and looking at the interaction patterns of the deployed contracts is indispensable.

E. Combination of Identification Methods

We started with a few known wallets that had Solidity sources. As some wallets are deployed in large quantities, we looked for (their) factories. For some of these, we could also find Solidity sources. These sources served as a test set for a first verification of both, the interface method including skeletons and fuzzing and the factory method. Finally, we scanned all verified source codes for wallets contracts, and used the resulting set of contracts as a starting point for the fuzzed interface method. To this, we added the wallets found by the factory method.

We would like to mention, that – in retrospect – most wallets can be identified uniquely by a small set of functions they implement, often just one function.

Limitations. The combination of the three methods (interface, factory, name) might still miss some wallets when they do not adhere to the ABI specification, are not deployed by a factory, and have non-descript contract names.

IV. CLASSIFICATION AND COMPARISON OF WALLETS

In this section, we base our discussion on the functionalities of wallets. First, we detail our definition of a proper wallet. Then, we determine and compare types of proper wallets.

A. Proper Wallet

The main functionality of wallets consists in funding the wallet as well as in submitting, confirming, and executing transactions to transfer Ether and tokens. Some wallets offer additional features. To distinguish proper wallet contracts from non-wallet contracts, we define that *optional wallet functions* (beyond the transfer of assets) must fall into the categories: administration and control, security mechanisms, lifecycle functions, and extensions.

Whether an implemented function belongs to one of these categories was decided upon reading the code. This was possible due to the heavy code reuse factor for wallets. Employing the technique of skeletons in combination with the fuzzed interface method (cf. sections II and III), we had to examine 739 skeletons that could be grouped into 28 blueprints for wallets.

TABLE II
CHARACTERISTICS OF WALLET CONTRACTS

			handles			control						security			life		ext.	
Type	# Found	Name	Ether	ERC20	advanced tokens	owner administration	cosigner (2-of-2)	multi signature	third party control	forwarding of assets	flexible transaction	daily limit	time lock	recovery mechanism	safe mode, pause, halt	destroy	update logic	module administration
Simple	9 250	AutoWallet [6]	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	4 685	BasicWallet [7]	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	8 467	ConsumerWallet [8]	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
	84 656	EtherWallet1 [9]	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	112 987	EtherWallet2 [10]	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	1 288	SimpleWallet3 [11]	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	46 832	SmartWallet [12]	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
	6 430	SpendableWallet [13]	✗	✓	(✓)	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	204	TimelockedWallet [14]	(✓)	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
	76 621	Wallet1 [15]	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiSig	16	Argent [16]	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
	207 807	BitGo [17]	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
	12 714	Gnosis/ConsenSys [18]	✓	✓	✗	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
	96	Ivt [19]	✓	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗
	3 616	Lundkvist [20]	✓	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
	996	NiftyWallet [21]	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗
	50 420	Parity/Eth/Wood [22]	✓	✓	✗	✓	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗	✗
	852	TeambrellaWallet [23]	✓	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	(✓)	✗	✗	✗	✗
	134	Unchained Capital [24]	✓	✗	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Forwarder	1 436 543	BitGo [25]	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
	2 520	IntermediateWallet [26]	✓	✓	(✓)	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
	527	SimpleWallet2 [27]	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Controlled	2 520 353	Bittrex [28]	✓	✓	(✓)	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Update	3 535	Eidoo [29]	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	8 922	LogicProxyWallet [30]	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗
Smart	17 964	Argent [31]	✓	✓	✗	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
	36 592	Dapper [32]	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓
	3 158	Gnosis [33]	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✗	✗	✓	✓

B. Types of Wallets

The identified 28 variants of proper wallets (blueprints) differ in functionality and number of deployments. Based on their features, we assign them to one of six groups.

1) Simple Wallets: provide little extra functionality beyond handling Ether and tokens. A sample can be found at [12].

2) MultiSig Wallets: require that m out of n owners sign a transaction before it is executed. Usually the required number of signatures (m) is smaller than the total number of owners (n), meaning that not all owners have to sign. In most cases, the set of owners and the number of required signatures can be updated.

3) Forwarder Wallets: forward the assets they receive to some main wallet. They may include owner management. BitGo employs large numbers of forwarder wallets in combination with its variant of a multiSig wallet [17].

4) Controlled Wallets: can be compared to traditional bank accounts. They are assigned to customers, who can use them as target of transfers, but the control over the account remains with the bank. Withdrawals are executed by the bank on behalf of the customer. This construction allows to comply with legal regulations that may restrict transactions. Regarding the number of deployments, controlled wallets are the most common type.

5) Update Wallets: provide a mechanism to update their main features at the discretion of the owner. A sample can be found at [29], [30]

6) Smart Wallets: offer enhanced features like authorization mechanism for arbitrary transactions, recovery mechanisms for lost keys, modular extension of features, or advanced token standards.

C. Features of Wallets

Table II shows for each identified wallet blueprint its type, number of deployed instances, name, reference to the source or bytecode, as well as an overview of the implemented features as detailed below.

Ether. To handle Ether a wallet has to be able to receive and transfer it. Some wallets are intended for tokens only and thus refuse Ether. But even then well designed wallets provide a withdraw method, since some Ether transfers (like mining rewards and self-destructs) cannot be blocked.

ERC20 tokens. For ERC20 token transfers, the holding address initiates the transfer by sending to the respective token contract the address of the new holder who is not informed about this change. No provisions have to be made to receive such tokens. However, to s(p)end the tokens a wallet needs to provide a way to call a transfer method of the token contract.

Advanced tokens require the recipient to provide particular functions that are called before the actual transfer of the token.

Owner administration enables the transfer of all assets in a wallet to a new owner in one sweep without revealing private keys. With an off-chain wallet, one has to transfer each asset separately or share the private key with the new owner.

MultiSig wallets face a trade-off between flexibility, transaction costs, and transparency. Each Ethereum transaction carries only one signature. To supply more, the wallet either has to be called several times by different owners (incurring multiple transaction fees), or the signatures have to be computed off-chain by a trusted app and supplied as data of a single transaction. A wallet may offer a few fixed multSig actions that are selected transparently via the entry point, or there may be a single entry point that admits the execution of arbitrary calls. The latter case requires a trusted app that composes the low-level call data off-chain in a manner transparent for the owners who are supposed to approve it.

Cosigner is a form of MultiSig where exactly two signatures are required. Moreover, it can be employed for implementing further functionality via another contract that acts as cosigner. This may include multiSig or off-chain signing.

Third party control means that the actual control over the wallet stays with a central authority.

Forwarding wallets are additional addresses for receiving assets that are transferred to a main wallet.

Flexible transactions means that the wallet is able to execute arbitrary calls after adequate authorization.

Daily limits and time locking restrict the access to the assets based on time. Spending more than a daily limit may e.g. require additional authorization. Time locks are useful if assets should be used only at a later point in time, after a vesting period, like after an ICO, for a smart will, or a trust fund.

Recovery mechanisms provision against lost or compromised credentials.

Life cycle management enable wallets to be put into safe mode, paused, or halted. Early wallets were able to self-destruct, which results in the loss of assets sent thereafter. When put on hold, a wallet can still reject transfer attempts.

Update logic enables to switch to a newer version of the wallet logic. This is implemented by means of proxy wallets, which derive their functionality from library code stored elsewhere, and keeps the wallets small and cheap to deploy.

Module administration offers the inclusion or deselection of modules to customize the wallet to user needs. This reprsents a modular and more fine-grained version of update logic.

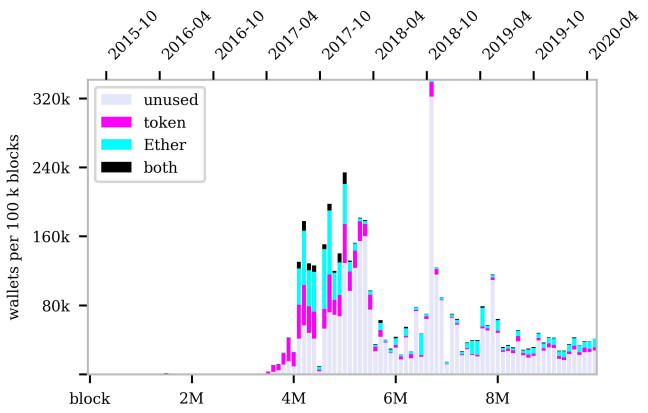

Fig. 1. Creation and usage of wallet contracts over time. The stack plot depicts for each bar the number of deployed wallets in a bin of 100 k blocks (corresponding to 2 weeks) differentiated into the usage groups by color.

V. Interaction Analysis

In this section, we examine the usage of wallets, differentiate token holdings, and discuss the role of wallets within the smart contracts landscape.

The 10 M blocks of the Ethereum main chain contain 697.4 M transactions, which gave rise to 1 761 M messages and 25.4 M successfully created contracts. About 4.66 M of them (18.4 %) are wallets. The wallets received 28.2 M and sent 54.0 M calls, with 2.8 M inter-wallet calls. In total, 85.0 M messages (4.8 %) involve wallets.

A. Creation and Usage of Wallets over Time

Figure 1 depicts the number of wallets created per 100 k blocks (about two weeks) as a stack plot, differentiating the wallets according to their later usage: wallets used for tokens as well as Ether, wallets used for only one of them, and unused wallets. (See section III-A for a discussion of how to identify token and Ether holders.)

Of the 4.66 M wallets, 68 % have not been used so far (3.15 M, grey). The other wallets are either used for tokens (613 k, magenta) or for Ether (772 k, cyan), but only a few wallets are used for both (119 k, black).[3]

When comparing the two most common wallet types, controlled and forwarder wallets, we notice a marked difference regarding their usage (Fig. 2). Controlled wallets (2.5 M, upper part) are used for Ether as well as tokens, while forwarder wallets (1.4 M, lower part) are used predominantly for Ether. Both show a large portion of unused wallets (grey), namely 60.5 % for the controlled and 81.7 % for the forwarder wallets.

The other types of wallets were deployed in smaller numbers (Fig. 3). For the first two years, only multiSig wallets (yellow) were created. Simple wallets (green) started to appear in the second half of 2017 after block 4.4 M. Update wallets (black) are as recent as block 5.9 M (mid 2018), while smart

[3]Initially, we considered also the property of having received a call as a further sign of activity. However, it strongly correlates with the other activities, so we omitted it for the sake of a clearer presentation.

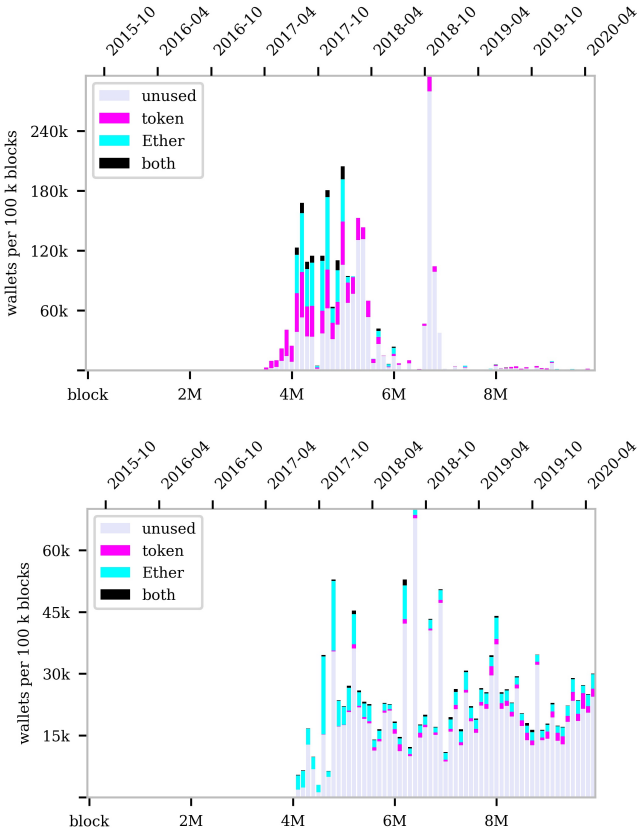

Fig. 2. Creation and usage of the two most frequent types of wallets. The upper plot depicts the most common controlled wallets, while the lower plot shows the forwarder wallets. Each bar represents the respective number of deployed wallets in a bin of 100 k blocks (corresponding to 2 weeks) differentiated into the usage groups by color

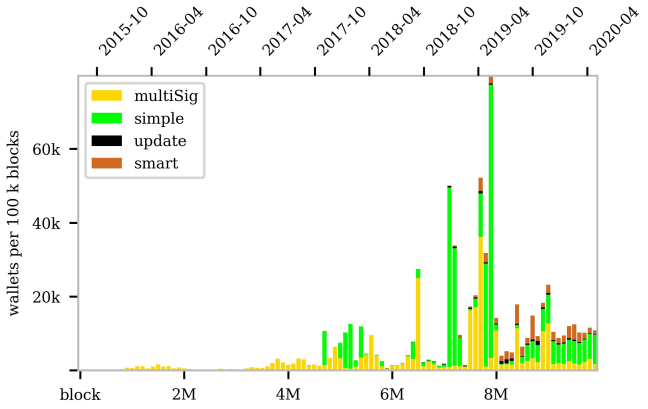

Fig. 3. Creation of the less frequent types of wallet contracts: simple, multiSig, update, and smart wallets. Each bar represents the respective number of deployed wallets in a bin of 100 k blocks (corresponding to 2 weeks) differentiated into the wallet types by color.

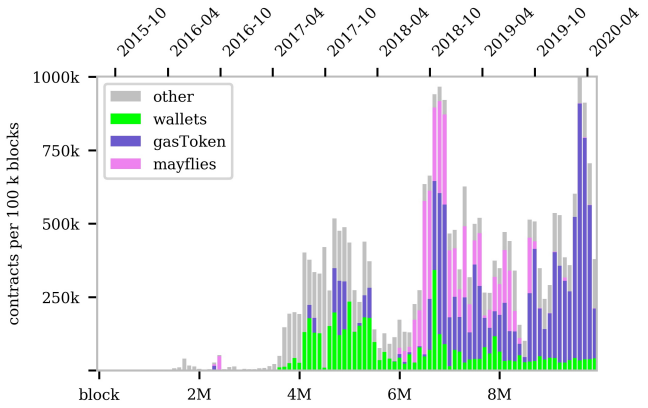

Fig. 4. Creation of all contracts on Ethereum. Each bar represents the respective number of deployed contracts in a bin of 100 k blocks (corresponding to 2 weeks) differentiated into wallets, gasTokens, mayflies and other contracts.

wallets (brown) start towards the end of 2018 after block 6.5 M. All of them are still being created.

B. Token Holdings

Most wallets are designed for token management. Still, only 0.73 M wallets (15.7 %) have ever received a token, while 3.93 M (84.2 %) did not. Even though the percentage of wallets without a single token varies with the type, it is always more than 60 %. If wallets do hold tokens, the number of different tokens is small for the majority of them. Less than 3300 wallets each held more than 10 different tokens. Only single wallets held a substantial number of different tokens over time, the maximum being 705.

C. Landscape

Figure 4 depicts the wallets (in green) within the landscape [34] of Ethereum contracts. Given that mayflies [35] and gasTokens are single-use contracts – self-destructing at the end of the deployment or at the first call – wallets are probably the largest group of 'durable' contracts identified by a common purpose.

VI. Comparison to Related Work

1) Wallets: In their analysis of ERC20 token trading, the authors of [36] take any *address holding tokens* to be a wallet. They demonstrate that the token trading network shows power-law properties and that it is decentralized, diverse, and mature. *Off-chain wallets* are compared extensively in [37]. The authors of [38] focus on 2-factor authentication for *contract wallets*, but do not discuss wallet contracts in detail. The broad analysis of smart contracts by [39] does not focus on wallets, but concludes that most contracts that collect substantial amounts of Ether are wallet contracts.

In this paper, we focus on wallet contracts that implement only characteristic functionality. A major challenge is to identify such proper wallet contracts.

2) Ethereum Graph Analysis: Most work focuses on the transfer of assets and network communication on Bitcoin and other cryptocurrency platforms. Regarding Ethereum, [40] examines "whether an attacker can de-anonymize addresses from

graph analytics against transactions on the blockchain". The authors of [41] "leverage graph analysis to characterize three major activities, namely money transfer, contract creation, and contract calls" with the aim to address security issues. Applying network science theory, [42] "find that several transaction features, such as transaction volume, transaction relation, and component structure, exhibit a heavy-tailed property and can be approximated by the power law function."

Regarding ERC20 tokens on Ethereum, the authors of [36] study the tokens trading network in its entirety with graph analysis and show power-law properties for the degree distribution. Similarly, the authors of [43] measure token networks, which they define as the network of addresses that have owned a specific type of token at any point in time, connected by the transfers of the respective token.

Instead of examining the trading of assets, our investigation focuses on contracts that manage the access to the traded assets, namely wallet contracts. We employ call graphs as they are a suitable abstraction for identifying interaction patterns.

3) EVM Bytecode Analysis: To detect code clones, the authors of [44] first deduplicate contracts by "removing function unrelated code (e.g., creation code and Swarm code), and tokenizing the code to keep opcodes only". Then they generate fingerprints of the deduplicated contracts by customized fuzzy hashing and compute pair-wise similarity scores. In another approach to clone detection, the authors of [45], [46] characterize each smart contract by a set of critical high-level semantic properties. Then they detect clones by computing the statistical similarity between the respective property sets.

To detect token systems automatically, the authors of [47] compare the effectiveness of a behavior-based method combining symbolic execution and taint analysis, to a signature-based approach limited to ERC20-compliant tokens. They demonstrated that the latter approach detects 99 % of the tokens in their ground truth data set. For all deployed bytecode, though, it bears a "false positive risk in case of factory contracts or dead code".

Our method of computing code skeletons is comparable to the first step for detecting similarities by [44]. Instead of their second step of fuzzy hashing though, we rely on the set of function signatures extracted from the bytecode and manual analysis, as our purpose is to identify wallets reliably. Relying on the interface is in line with the results in [47].

VII. Conclusions

We examined smart contracts that provide a wallet functionality on the Ethereum main chain up to block 10 000 000, mined on May 4, 2020. For a semi-automatic identification of wallet contracts, we discussed methods based on deployed bytecode and interactions. By analyzing source code, bytecode, and execution traces, we derived features and types of wallets in use, and compared their characteristics. Moreover, we provided a quantitative and temporal perspective on the creation and use of identified types of wallets, and discussed their role in the smart contracts landscape.

Identification of wallets. The identification of wallets as recipients of tokens or Ether can be done automatically, but includes many contracts beyond proper wallets. Our method of identifying wallets by name, interface, and ancestry yields blueprints for wallets, which then are used to locate contracts with similar implementations or same deployers. This approach is only semi-automatic, but more reliable.

Blueprints for wallets. Since we manually verify the Solidity source code, our work yields a ground truth of wallets that can be used for evaluating automated tools.

Wallet Features. Features of wallets in use beyond the transfer of assets can be grouped into administration and control, security mechanisms, lifecycle functions, and extensions. By distilling a comprehensive list of features for pure wallets, we are able to separate wallet contracts from non-wallets. Moreover, we could depict actual use cases via the extracted features.

Wallet Types. Wallets can be categorized into the six types simple, multiSig, controlled, forwarder, update, and smart wallet according to the features they provide. MultiSig wallets were the first to appear shortly after the launch of Ethereum, while controlled and forwarder wallets followed in 2017. Update wallets and smart wallets with a modular design are a recent phenomenon starting at the end of 2018. We observe an evolution of features in the wallet types. Still, the multiSig wallet seems popular, either as it is or incorporated into smart wallets.

Usage of Wallets. Wallet contracts are numerous, amounting to 18.4 % of all deployed contracts. If we discount the 4.2 M improper contracts that self-destruct during deployment [35], the share of wallets even rises to 22 %. Next to gasToken contracts, wallets are the second largest application group regarding contract deployments.

Then again, 68 % of the wallet contracts are not in use yet. They may have been produced on stock for later use. Interestingly, wallet contracts are used either for tokens or for Ether, but rarely for both. Even though most wallets are designed for token management, only 16 % so far received at least one token. Of the few wallets holding tokens, 83 % hold just one type, while 99.5 % hold at most 10 different types of tokens.

Solidity and Code Reuse. On the surface, only 587 wallet addresses (0.02 % of all wallets) have a verified Solidty source code on `etherscan.io`. However, taking into account that most wallets are created by factories whose code may be found, this number rises to 63 %. By exploiting the similarity of code skeletons, we can relate even 83 % of the wallets to publicly available source code.

The 4.66 M wallets correspond to just 1357 distinct deployed bytecodes (631 distinct code skeletons). This homogeneity results from the small number of on- or off-chain factories that generate most of the wallets.

Future Work

When aiming at a deeper understanding of the landscape of smart contracts and dApps, there are still some pieces of the puzzle missing. Our contribution to understanding wallet contracts may serve as a basis for further research in this direction, because wallets are a major application type. Moreover, as wallets link many dApps, removing them from the overall picture may let other applications stand out clearer.

To determine reliably what smart contracts actually implement, it is still indispensable to analyze bytecode. Adequate tool support for a massive automated semantic code analysis would be helpful to obtain a comprehensive picture of the smart contract ecosystem.

REFERENCES

[1] M. Di Angelo and G. Salzer, "Wallet Contracts on Ethereum," *arXiv preprint: 2001.06909*, 2020.

[2] Ethereum Wiki, "A Next-Generation Smart Contract and Decentralized Application Platform," accessed 2020-01-14. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[3] V. Buterin, "Blockchain and Smart Contract Mechanism Design Challenges (slides)," 2017, accessed 2018-08-09. [Online]. Available: http://fc17.ifca.ai/wtsc/Vitalik%20Malta.pdf

[4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, Tech. Rep., 2020, https://ethereum.github.io/yellowpaper/paper.pdf.

[5] "Contract ABI Specification," 2019, https://solidity.readthedocs.io/en/latest/abi-spec.html.

[6] Wallet_1, "AutoWallet," 2019, https://etherscan.io/address/0x1991af53e07b548a062a66e8ff3fac5cc9e63b22#code.

[7] Wallet_2, "BasicWallet," 2019, https://etherscan.io/address/0xa4db5156d3c581da8ac95632facee7905bc32885#code.

[8] Wallet_3, "ConsumerWallet," 2019, https://github.com/tokencard/contracts/.

[9] Wallet_4, "EtherWallet1," 2019, https://etherscan.io/address/0xbe1390c5bbc48f731511bcba0dc7052Ceea0a48a#code.

[10] Wallet_5, "EtherWallet2," 2019, https://etherscan.io/address/0x1a9deebf2d30687688750107c02bad2a97b2a767#code.

[11] Wallet_7, "SimpleWallet3," 2019, https://etherscan.io/address/0xc84764ea01cb3a713566c076544d5c6c01d79a1c#code.

[12] Wallet_6, "SmartWallet," 2019, https://etherscan.io/address/0xefc7de761ae038b3bb3080ecfb98cea51fd442ea#code.

[13] Wallet_8, "SpendableWallet," 2019, https://etherscan.io/address/0x35a1700AC75f6e9E096D9A5c90e3221B658096e0#code.

[14] Wallet_9, "TimelockedWallet," 2019, https://etherscan.io/address/0x5119b5e3a7bff084732a7ec41efed8aa0c4cd6d4#code.

[15] Wallet_10, "Wallet1," 2019, https://etherscan.io/address/0x2dd27d3597c137c94c99d868167bbd1e06919bfa#code.

[16] Wallet_11, "MultiSig Argent," 2019, https://github.com/argentlabs/argent-contracts/blob/develop/contracts/MultiSigWallet.sol.

[17] Wallet_12, "MultiSig BitGo," 2019, https://github.com/BitGo/eth-multisig-v2/blob/master/contracts/WalletSimple.sol.

[18] Wallet_13, "MultiSig Gnosis," 2019, https://github.com/gnosis/MultiSigWallet/blob/master/contracts/MultiSigWalletWithDailyLimit.sol.

[19] Wallet_14, "MultiSig IVT," 2019, https://etherscan.io/address/0x36d3f1c3ea261ace474829006b6280e176618805#code.

[20] Wallet_15, "MultiSig Lundkvist," 2019, https://github.com/christianlundkvist/simple-multisig.

[21] Wallet_16, "MultiSig NiftyWallet," 2019.

[22] Wallet_17, "MultiSig Parity," 2017, https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol.

[23] Wallet_18, "MultiSig Teambrella," 2019, https://etherscan.io/address/0x44852FAEFcb42E392f2c55c6df53A50A732Df298#code.

[24] Wallet_19, "MultiSig Trezor," 2018, https://github.com/unchained-capital/ethereum-multisig/blob/master/contracts/MultiSig2of3.sol.

[25] Wallet_20, "Forwarder BitGo," 2019, https://github.com/BitGo/eth-multisig-v2.

[26] Wallet_21, "Forwarder IntermediateWallet," 2019.

[27] Wallet_22, "Forwarder SimpleWallet2," 2019, https://gist.github.com/bitgord/e7e39a90552ef10c57940d0f4f2e9a00.

[28] Wallet_23, "Controlled Bittrex," 2017, https://etherscan.io/address/0xA3C1E324CA1CE40DB73ED6026C4A177F099B5770#code.

[29] Wallet_24, "Update Eidoo," 2018, https://etherscan.io/address/0x0409b14cac8065e4972839f9dafabb20cef72662#code.

[30] Wallet_25, "Update LogicProxyWallet," 2019, https://etherscan.io/address/0x5d4a945271fb3e16481bf6ce0bad5f6b2e9d13db#code.

[31] Wallet_26, "Smart Argent," 2019, https://github.com/argentlabs/argent-contracts.

[32] Wallet_27, "Smart Dapper," 2019, https://github.com/dapperlabs/dapper-contracts.

[33] Wallet_28, "Smart Gnosis," 2019, https://github.com/gnosis/safe-contracts.

[34] M. Di Angelo and G. Salzer, "Characterizing Types of Smart Contracts in the Ethereum Landscape," in *4th Workshop on Trusted Smart Contracts, Financial Cryptography*. Springer, 2020.

[35] ——, "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.

[36] S. Somin, G. Gordon, and Y. Altshuler, "Network Analysis of ERC20 Tokens Trading on Ethereum Blockchain," in *International Conference on Complex Systems*. Springer, 2018, pp. 439–450.

[37] T. Haigh, F. Breitinger, and I. Baggili, "If I Had a Million Cryptos: Cryptowallet Application Analysis and a Trojan Proof-of-Concept," in *nternational Conference on Digital Forensics and Cyber Crime*. Springer, 2018, pp. 45–65.

[38] I. Homoliak, D. Breitenbacher, A. Binder, and P. Szalachowski, "An Air-Gapped 2-Factor Authentication for Smart-Contract Wallets," *arXiv preprint arXiv:1812.03598*, 2018.

[39] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, "A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics," *IEEE Access*, vol. 7, pp. 78 194–78 213, 2019.

[40] W. Chan and A. Olmsted, "Ethereum Transaction Graph Analysis," in *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, 2017, pp. 498–500.

[41] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding Ethereum via Graph Analysis," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 1484–1492.

[42] D. Guo, J. Dong, and K. Wang, "Graph Structure and Statistical Properties of Ethereum Transaction Relationships," *Information Sciences*, vol. 492, pp. 58–71, 2019.

[43] F. Victor and B. K. Lüders, "Measuring Ethereum-based ERC20 Token Networks," in *International Conference on Financial Cryptography and Data Security*, 2019.

[44] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing Code Clones in the Ethereum Smart Contract Ecosystem," *arXiv preprint arXiv:1905.00272*, 2019.

[45] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "EClone: Detect Semantic Clones in Ethereum via Symbolic Transaction Sketch," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 900–903. [Online]. Available: http://doi.acm.org/10.1145/3236024.3264596

[46] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling Clone Detection for Ethereum via Smart Contract Birthmarks," in *2019 IEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE Press, 2019, pp. 105–115.

[47] M. Fröwis, A. Fuchs, and R. Böhme, "Detecting Token Systems on Ethereum," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019.