Wallet Contracts on Ethereum

Monika di Angelo     Gernot Salzer
TU Wien, Vienna, Austria
{monika.di.angelo,gernot.salzer}@tuwien.ac.at
19 January 2020

Abstract

In the area of blockchains, a wallet is anything that manages the access to cryptocurrencies and tokens. Off-chain wallets appear in different forms, from paper wallets to hardware wallets to dedicated wallet apps, while on-chain wallets are realized as smart contracts. Wallet contracts are supposed to increase trust and security by being transparent and by offering features like daily limits, approvals, multiple signatures, and recovery mechanisms.

Ethereum is the most prominent platform for both, tokens and smart contracts, and thus also for on-chain wallets. Our work aims at a better understanding of Ethereum on-chain wallets, which represent one of the most frequent types of smart contracts. By analyzing source code, bytecode, and execution traces, we derive usage scenarios and patterns. We discuss several methods for identifying wallet contracts in a semi-automatic manner by looking at the deployed bytecodes and their interaction patterns. We extract blueprints for wallets and thereby compile a ground truth. Furthermore, we differentiate characteristics of wallets in use, and group them into six types. We provide numbers and temporal perspectives regarding the creation and use of wallets. We analyze the data of the Ethereum main chain up to block 8 450 000, mined on August 30, 2019.

Keywords: analysis, EVM bytecode, smart contract, transaction data, wallet
# Contents

1 Introduction  4

2 Terms, Bytecode Analysis, and Tools
   2.1 Terms and Data  5
   2.2 Code Skeletons  5
   2.3 Contract Interfaces
       2.3.1 Interface Extraction  6
       2.3.2 Interface Restoration  6
   2.4 Third Party Tools  6

3 Identifying Potential Wallets
   3.1 Wallets as Recipients of Ether or Tokens  7
   3.2 Identifying Wallets by their Interface  8
   3.3 Identifying Wallets by their Factory  8
   3.4 Identifying Wallets by their Name  8
   3.5 Combination of Identification Methods  8

4 Classification and Comparison of Wallets
   4.1 Proper Wallet  9
   4.2 Types of Wallets
       4.2.1 Simple Wallets  9
       4.2.2 MultiSig Wallets  9
       4.2.3 Forwarder Wallets  11
       4.2.4 Controlled Wallets  11
       4.2.5 Update Wallets  11
       4.2.6 Smart Wallets  11
   4.3 Features of Wallets  11

5 Interaction Analysis
   5.1 Creation and Usage of Wallets over Time  12
   5.2 Token Holdings  15

6 Wallets in the Landscape of Contracts
   6.1 Graph View  15

7 Comparison to Related Work  16

8 Conclusions
   8.1 Future Work  18

References  20
### A Wallet Profiles

| Section                  | Page |
|--------------------------|------|
| A.1 Simple Wallets       | 23   |
| A.2 MultiSig Wallets     | 27   |
| A.3 Forwarder Wallets    | 33   |
| A.4 Controlled Wallets   | 34   |
| A.5 Update Wallets       | 37   |
| A.6 Smart Wallets        | 38   |
1 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. Software wallets manage the cryptographic keys required for authorization and implement the protocols for interacting with blockchains in general and smart contracts (on-chain programs) in particular.

On-chain wallets are smart contracts that hold cryptocurrencies and access to tokens and that may offer advanced methods for manipulating the assets. Simply by introducing the role of an ‘owner’ it becomes possible to transfer all assets of an on-chain wallet transparently and securely in a single transaction. More refined methods include multi-signature wallets, which grant access only if sufficiently many owners sign.

Regarding the number of transactions and public availability of data, Ethereum is the major platform for smart contracts, and thus also for tokens and on-chain wallets. This paper investigates the usage and purpose of on-chain wallets on the main chain of Ethereum qualitatively as well as quantitatively. In particular, we address the following questions.

1. How can deployed wallets be identified from transaction data?
2. Regarding functionality, which types of wallets are deployed?
3. When and in which quantities are wallets created, and how many are actually used?
4. What is the role of wallets in the overall Ethereum smart contract landscape?

Methodologically, we start from the 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 blockchain data. Finally, we relate wallets to other frequently occurring contract types.

Regarding their number, on-chain wallets form a substantial part of the smart contracts on the chain. This work thus contributes to a better understanding of what smart contracts on Ethereum are actually used for. Moreover, the collection of wallet features and blueprints may serve as a resource when designing further decentralized trading apps. Our methods for detecting wallets and analyzing their activities may help in accessing the liveliness of on-chain projects. E.g., a temporal view on the use of wallets is more informative than just the number of wallets initially deployed.

Roadmap

Section 2 clarifies terms and presents our methods for bytecode analysis. Section 3 discusses methods for the identification of potential on-chain wallets. Section 4 describes characteristic features of wallets and categorizes them into types. Section 5 analyzes interactions of wallets. Section 7 compares our approach to related work. Finally, section 8 concludes with a summary of our results.
2 Terms, Bytecode Analysis, and Tools

We assume the reader to be familiar with blockchain essentials. For Ethereum specifics, we refer to [1, 2, 3].

2.1 Terms and Data

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

Unless stated otherwise, statistics refer to the Ethereum main chain up to block 8450000 (mined on Aug 30, 2019). We abbreviate factors of 1000 and 1000000 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 18 M successful create operations, there are verified source codes for 70 k addresses (0.4 %) only.

2.2 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 18 M contract deployments so far give rise to 252 k distinct bytecodes and just 119 k distinct skeletons. This is still a large number, but manageable by exploiting creation histories and the similarity of skeletons. We are able to relate 6.7 M of these deployments to some source code on etherscan.io, an increase from 0.4 to 37 %.

2.3 Contract Interfaces

Most contracts in the Ethereum universe adhere to the ABI standard [4], which identifies functions by signatures that consist of the first four bytes of the Keccak-256 hash of the function names and the parameter types. Thus, the bytecode of a contract contains instructions to compare the first four bytes of the call data to the signatures of its functions.
To understand the implemented interface of a contract, we extract it from the bytecode, and then try to restore the corresponding function headers.

2.3.1 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 42 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 very few 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 9.6 M deployed contracts (220 k codes, 107 k skeletons) generated by the Solidity compiler, it is thus representative. Another interesting group of deployed contracts consists of 5.2 M short contracts (18 k codes, only 271 skeletons) without entry points. They are mainly contracts for storing gas (gasToken), but also proxies (contracts redirecting calls elsewhere) and contracts involved in attacks. As a last group of deployed contracts, we are left with remaining 595 codes. For them, our tool shows an error rate of 8%, estimated from a random sample of 60 codes that we manually checked.

2.3.2 Interface Restoration

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

2.4 Third Party Tools

We employ the Ethereum client parity in archive mode to obtain the execution traces. PostgreSQL serves as our primary database that stores the messages extracted from the traces as well as information on the contracts. For analyzing contract interactions as

---

1 Deployed code generated by solc can be identified by the first few instructions. It starts with one of the sequences 0x6060604052, 0x6080604052, 0x608060408152, 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 322 k signatures in our dictionary only 19 appear with a second function header.
graphs, we use the graph database Neo4j. Furthermore, we utilize etherscan.io for information on deployed contracts and matplotlib for plotting.

3 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 beyond managing assets are termed non-wallet contracts. The latter group contains 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 4 we elaborate on the second step, the check whether they implement a proper wallet.

3.1 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: on-chain wallet in case of a contract address and off-chain wallet otherwise. 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.

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 1 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 74.3 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 on-chain wallets 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 high amount of unused proper wallets. Thus, we did not use this method for the further analyses.
3.2 Identifying Wallets by their Interface

Given the source code of a wallet contract, we can use its bytecode and partially restored ABI to identify similar contracts on the chain. Employing the methods described in section 2, 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.

3.3 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.

3.4 Identifying Wallets by their Name

To detect wallets in a more systematic fashion and to include also less popular ones, we scanned the 70k source codes on etherscan.io for contracts containing the string ‘wallet’ or ‘Wallet’ in their name.

Limitations. This approach is a heuristic that yields false positives and misses wallets named differently. Again, reading the bytecode or looking at the interaction patterns of the deployed contracts is indispensable.

3.5 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.

4 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.

4.1 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 2 and 3), we had to examine 631 skeletons that could be grouped into 24
blueprints for wallets.

4.2 Types of Wallets

The identified 24 variants of proper wallets (blueprints) differ in functionality and number
of deployments. Based on their features, we assign them to one of six groups.

4.2.1 Simple Wallets

provide little extra functionality beyond handling Ether and tokens. A sample can be
found at [8].

4.2.2 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.
| Type       | # Found | Name                           | handles | control | security | life       | ext.       |
|------------|---------|--------------------------------|---------|---------|----------|------------|------------|
|            |         |                                | ether   | ERC20   | owner    | forwarding | flexible   | daily      | recovery   | safe       | destroy    | update      | admin       | modal       |
|            |         |                                | advanced| admin   | admin    | of assets  | transaction| limit      | mechanism  | mode, pause| halt       | logic       | administration| control     |
| Simple     | 9250    | AutoWallet [5]                 | ✓✓✓✓    | ✓✓✓✓    | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 4635    | BasicWallet [6]                | ✓✓✓✓    | ✓✓✓✓    | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 3634    | ConsumerWallet [7]             | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 46340   | SmartWallet [8]                | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 1436    | SpendableWallet [9]            | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 202     | TimelockedWallet [10]          | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
| MultiSig   | 11      | Argent [11]                    | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 135457  | BitGo [12]                     | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 10150   | Gaasis/ConSensys [13]          | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 96      | Ivit [14]                      | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 3391    | Lundkvist [15]                 | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 995     | NiftyWallet [16]               | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 44235   | Parity/Eth/Wood [17]           | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 822     | TeambrellaWallet [18]          | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 131     | Unchained Capital [19]         | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
| Forwarder  | 1087257 | BitGo [20]                     | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 2520    | IntermediateWallet [21]        | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 527     | SimpleWallet2 [22]             | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
| Controlled | 2488845 | Bittrex [23]                   | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
| Update     | 2862    | Eidoo [24]                     | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 3926    | LogicProxyWallet [25]          | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
| Smart      | 4098    | Argent [26]                    | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 15749   | Dapper [27]                    | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
|            | 1065    | Gaasis [28]                    | ✓✓✓✓✓   | ✓✓✓✓✓   | ✓✓✓✓✓✓   | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       | ✓✓✓✓       |
4.2.3 **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 [12].

4.2.4 **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.

4.2.5 **Update Wallets**

provide a mechanism to update their main features at the discretion of the owner. A sample can be found at [24, 25]

4.2.6 **Smart Wallets**

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

4.3 **Features of Wallets**

Table 2 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 (p)send 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 represents a modular and more fine-grained version of update logic.

5 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 8.45 M blocks of the Ethereum main chain contain 530 M transactions, which gave rise to almost 1 293 M messages and 17.9 M successfully created contracts. About 3.9 M of them (21%) are wallets. The wallets received 19.2 M and sent 40.5 M calls, with 2 M inter-wallet calls. In total, 57.7 M messages (4.5%) involve wallets.
5.1 Creation and Usage of Wallets over Time

Figure 1 depicts the number of wallets created per 100k 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 3.1 for a discussion of how to identify token and Ether holders.)

Of the 3.9 M wallets, 68 % have not been used so far (2.6 M, grey). The other wallets are either used for tokens (529 k, magenta) or for Ether (626 k, cyan), but only a few wallets are used for both (91 k, black).

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 (1.1 M, lower part) are used predominantly for Ether. Both show a large portion of unused wallets (grey), namely 62 % for the controlled and 83 % 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 wallets (brown) start towards the end of 2018 after block 6.5 M. All of them are still being created.

\footnote{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.}
Figure 2: Creation and usage of controlled wallets (above) and forwarder wallets (below).
5.2 Token Holdings

Most wallets are designed for token management. Still, only 0.62 M wallets (16%) have so far received at least one token, while 3.25 M (84%) 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 substantial amounts of different tokens over time, the maximum being 705.

6 Wallets in the Landscape of Contracts

Smart contracts may perform their tasks stand-alone or in cooperation with companion contracts, with the number of contracts belonging to one application going up to a million in extreme cases (like for gasTokens or wallets). The landscape is as diverse as the purpose of smart contracts. We find exchanges, markets, wallets, tokens, games, attackers, and all kinds of dApps implementing part of their logic on-chain. Delineating the border between all of these groups, but especially between applications turns out to be difficult. We assume that applications are connected to each other mainly over general services like wallets.
6.1 Graph View

In order to analyze the landscape of smart contracts, their interactions in form of calling each other provide useful information. For this, we build the call graph from the execution trace (as provided by the parity client in archive mode), where the contracts serve as nodes that are connected by an edge whenever the trace lists a call from one contract to another. After removing singletons (i.e. contracts that are called just by users or not all) and reducing multiple edges between two contracts to just one edge, we are left with 10.3 M contracts as nodes and 13.0 M calls as edges. This graph consists of 12.9 k connected components, with the largest one containing virtually all nodes (9.7 M, 94%), while most components have less than 100 nodes. The high interconnection indicates that applications do not separate naturally.

Proper Wallets. To test the assumption that wallets act as a major connecting element, we additionally remove all contracts from the graph that we identified as wallets. This further reduces the graph by 1.2 M nodes and 2.1 M edges. However, the largest component still consists of 8.5 M nodes (93%), while the number of components slightly increases to 15.1 k. We conclude that wallets probably contribute to the cohesion of the graph but are not solely responsible for it.

Token holders. If we take any token holder to be a wallet and repeat the analysis, the graph falls apart. The remaining 5.1 M nodes yield 4.4 M graph components, the largest one containing 158 k nodes (3% of all nodes). Thus, for a liberal definition of wallet, the assumption of wallets serving as cohesive holds true. However, removing all token holders is probably too coarse, as we remove exchanges, markets, applications that employ their own token, and token contracts in general.

7 Comparison to Related Work

Wallets

In their analysis of ERC20 token trading, the authors of [29] 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 [30]. The authors of [31] focus on 2-factor authentication for contract wallets, but do not discuss wallet contracts in detail. The broad analysis of smart contracts by [32] 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 on-chain wallets.

Ethereum Graph Analysis

Most work focuses on the transfer of assets and network communication on Bitcoin and other cryptocurrency platforms. Regarding Ethereum, [33] examines “whether an attacker can de-anonymize addresses from graph analytics against transactions on the blockchain”. The authors of [34] “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, \[35\] “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 \[29\] study the tokens trading network in its entirety with graph analysis and show power-law properties for the degree distribution. Similarly, the authors of \[36\] measure token networks, which they define as the network of addresses that have owned a specific type of token at any point in time, connected by the transfers of the respective token.

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.

**EVM Bytecode Analysis**

To detect code clones, the authors of \[37\] 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 \[38, 39\] 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 \[40\] 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 \[37\]. 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 \[40\].

**Smart Contract Landscape**

In their empirical study \[41\], the authors investigate Ethereum smart contracts by looking at contract creation, interaction, and code reuse. The authors of \[42\] complement their quantitative analysis of smart contracts by temporal views. Moreover, they identify particular groups of smart contracts based on their activity patterns.

Our contribution also aims at understanding the landscape of smart contract, but concentrates on the specific group of wallet contracts. We argue that wallets are one of the backbones of the landscape keeping the call graph connected.
8 Conclusions

We examined smart contracts that provide a wallet functionality on the Ethereum main chain up to block 8,450,000, mined on August 30, 2019. 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. On-chain wallets are numerous, amounting to 3.9 M contracts (21% of all contracts). If we discount mayflies (contracts that self-destruct during deployment \cite{42}), the share of wallets even rises to 28%. Next to gasToken contracts, wallets are the largest application group regarding contract deployments.

Then again, most on-chain wallets (68%) are not in use yet. They may have been produced on stock for later use. Interestingly, on-chain wallets are used either for tokens or for Ether, but rarely for both. Even though most wallets are designed for token management, only 0.62 M wallets (16%) have so far received at least one token. Of the few wallets holding tokens, 83% hold just one type of tokens, while 99.5% hold at most 10 different types.

Solidity and Code Reuse. On the surface, only 587 wallet addresses (0.02% of all wallets) have a verified Solidity source code on Etherscan. 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 3.9 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.

**Landscape.** Our assumption that wallets act as a cohesive in the graph of executed calls between contracts holds only for a very broad definition of wallets. To dissect the landscape of smart contracts effectively into applications, we may have to identify further contract groups that handle assets.

### 8.1 Future Work

When aiming at a deeper understanding of the role of dApps and smart contracts, there are still some pieces of the puzzle missing. Our contribution to understanding on-chain wallets may serve as a basis for further research in this direction, as 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. Examples of such applications are markets and exchanges, which also act as a cohesive in the call graph and which still need thorough investigation. Additionally, we can use the number of calls as weights on the edges and apply neighborhood algorithms. First experiments with the latter approach show that it may be effective if we manage to remove some of the major connecting applications.

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

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

[3] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” Ethereum Project Yellow Paper, Tech. Rep., 2018, https://ethereum.github.io/yellowpaper/paper.pdf.

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

[5] Wallet_1, “Autowallet,” 2019, https://etherscan.io/address/0x1991af53e07b548a062a66e8f3fac5cc9e63b22#code.

[6] Wallet_2, “Basicwallet,” 2019, https://etherscan.io/address/0xa4db5156d3c581da8ac95632f1acee79055c32885#code.

[7] Wallet_3, “Consumerwallet,” 2019, https://github.com/tokencard/contracts/

[8] Wallet_4, “Smartwallet,” 2019, https://etherscan.io/address/0xefc7de761ae038b3bb3080ecb98cea51fd442ea#code.

[9] Wallet_5, “Spendablewallet,” 2019, https://etherscan.io/address/0x35a1700AC75f6e9E096D9A5c90e3221B658096e0#code.

[10] Wallet_6, “Timelockedwallet,” 2019, https://etherscan.io/address/0x5119b5e3a7bf084732a7ec41efed8a0c4cd6d4#code.

[11] Wallet_7, “Multisig argent,” 2019, https://github.com/argentlabs/argent-contracts/blob/develop/contracts/MultiSigWallet.sol

[12] Wallet_8, “Multisig bitgo,” 2019, https://github.com/BitGo/eth-multisig-v2/blob/master/contracts/WalletSimple.sol

[13] Wallet_9, “Multisig gnosis,” 2019, https://github.com/gnosis/MultiSigWallet/blob/master/contracts/MultiSigWalletWithDailyLimit.sol

[14] Wallet_10, “Multisig ivt,” 2019, https://etherscan.io/address/0x36d3f1c3ea261ace474829006b6280e176618805#code

[15] Wallet_11, “Multisig lundkvist,” 2019, https://github.com/christianlundkvist/simple-multisig

[16] Wallet_12, “Multisig niftywallet,” 2019.
[17] Wallet_13, “Multisig parity,” 2017, https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol.

[18] Wallet_14, “Multisig teambrella,” 2019, https://etherscan.io/address/0x44852FAEFcb42E392f2c55c6d53A50A732Df298#code

[19] Wallet_15, “Multisig trezor,” 2018, https://github.com/unchained-capital/ethereum-multisig/blob/master/contracts/MultiSig2of3.sol.

[20] Wallet_16, “Forwarder bitgo,” 2019, https://github.com/BitGo/eth-multisig-v2

[21] Wallet_17, “Forwarder intermediatewallet,” 2019.

[22] Wallet_18, “Forwarder simplewallet2,” 2019, https://gist.github.com/bitgord/e7e39a90552ef10c57940d0f4f2e9a00.

[23] Wallet_19, “Controlled bittrex,” 2017, https://etherscan.io/address/0xA3C1E324CA1CE40DB73ED6026C4A177F099B5770#code.

[24] Wallet_20, “Update eido,” 2018, https://etherscan.io/address/0x0409b14cac8065e4972839f9dafab820cceil72662#code.

[25] Wallet_21, “Update logicproxywallet,” 2019, https://etherscan.io/address/0x5d4a945271fb3e164816f6ce0bad5f6b2e9d13dbc#code.

[26] Wallet_22, “Smart argent,” 2019, https://github.com/argentlabs/argent-contracts

[27] Wallet_23, “Smart dapper,” 2019, https://github.com/dapperlabs/dapper-contracts

[28] Wallet_24, “Smart gnosis,” 2019, https://github.com/gnosis/safe-contracts

[29] 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.

[30] T. Haigh, F. Breitinger, and I. Baggili, “If i had a million cryptos: Cryptowallet application analysis and a trojan proof-of-concept,” in International Conference on Digital Forensics and Cyber Crime. Springer, 2018, pp. 45–65.

[31] 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.

[32] 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.
[33] 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.

[34] 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.

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

[36] F. Victor and B. K. Lüders, “Measuring ethereum-based erc20 token networks,” in International Conference on Financial Cryptography and Data Security, 2019.

[37] 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.

[38] 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

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

[40] M. Fröwis, A. Fuchs, and R. Böhme, “Detecting token systems on ethereum,” in International conference on financial cryptography and data security. Springer, 2019.

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

[42] M. Di Angelo and G. Salzer, “Mayflies, Breeders, and Busy Bees in Ethereum: Smart Contracts Over Time,” in Third ACM Workshop on Blockchains, Cryptocurrencies and Contracts (BCC’19). ACM Press, 2019.
A Wallet Profiles

In this appendix, we detail each identified wallet blueprint with regards to

- the author(s) if known
- the location of the Solidity source code that it is based on
- a brief description of features
- the function headers used for identifying its instances
- the deploying factory contract if used
- subtleties of the detection procedure if other than finding signatures

When specifying the location of source code, we use ES:0x... as an abbreviation for the link https://etherscan.io/address/0x...#code.

A.1 Simple Wallets

AutoWallet

Source: ES:0x1991af53e07b548a062a66e8ff3fac5cc9e63b22

Description: Simple wallet that handles Ether, ERC20 tokens and non-fungible tokens (ERC721). Received Ether is forwarded automatically to the owner of the wallet, but the wallet provides also a sweep function to access Ether that was deposited e.g. as a mining reward or by a self-destruct.

Identification: The wallet can be uniquely identified by the following function:

transferNonFungibleToken(address,address,uint256)

Addresses: So far there is only one bytecode for this contract in use, deployed e.g. at address

0x1991af53e07b548a062a66e8ff3fac5cc9e63b22. Of the 9250 instances on the chain, all but two where deployed by the externally owned account

0x13d0c7ada3f98eec232ed7e57fefc4c300f25095.

BasicWallet

Source: ES:0xa4db5156d3c581da8ac95632facee7905bc32885

Description: Simple wallet with owner management for Ether, ERC20, and ERC223 tokens.

Identification: The wallet can be uniquely identified by the four functions it implements:

changeOwner(address)
transfer(address,uint256)
transferToken(address,address,uint256)
tokenFallback(address,uint256,bytes)
Addresses: This wallet was deployed 4635 times with two versions of the bytecode (due to different versions of the Solidity compiler) by the externally owned account 0xff3249da62ca5286997f31f458959de9ae2f4dad. 

| wallets | version | deployed e.g. at |
|---------|---------|-----------------|
| 3180    | newer   | 0xa4db5156d3c581da8ac95632facee7905bc32885 |
| 1455    | older   | 0x850c3beae3766e3efcf76ade7cbd6e3e0aec517e |

**ConsumerWallet**

**Source:** https://github.com/tokencard/contracts/

**Description:** Simple wallet for Ether and ERC20 tokens with various security features, like white-listing of receivers, daily limits (using an oracle for converting tokens to Ether), two factor authentication, and gas management.

**Identification:** We identify the wallets by checking whether one or two of the following functions are among their signatures:

- `topUpAvailable()` 
- `bulkTransfer(address,address[])`

This allows us to identify seven variants of deployed bytecode, which we check manually to make sure that they are indeed the same type of wallet. Most wallets are deployed by on-chain factories that can be found by looking for the characteristic signature

`deployWallet(address)`

**Addresses:** Of the seven bytecodes of this wallet, the three most frequent ones have been deployed by four factories, while the less frequent ones have been deployed by one particular externally owned account.

| wallets | code deployed e.g. at | creator | creator address | user? |
|---------|-----------------------|---------|-----------------|-------|
| 1380    | 0x20ab867160e73788e0db311f445e67bc596e0ec0  | A       | 0x95bebe7bfc6acc186c13d055d0aacc2de5f81502 | no    |
| 1267    | 0xa8e7213d64e29f6e6e81cb56d6cd48bdc5f722dc4 | B       | 0x85bb8a852c29d8f100cb97ecdf4589056d1be2dd | no    |
| 531     | 0xd883f8a6080ea6c473efc05c8ff3238255ad0e02 | C,D     | 0x5c76fb5fb117d190beac217bc3568e70f2b6b71d | no    |
| 255     | 0xe6510c19c7768ca0937e8f4daf0b16859af9c271 | E       | 0xe6510c19c7768ca0937e8f4daf0b16859af9c271 | E     |
| 53      | 0xe6510c19c7768ca0937e8f4daf0b16859af9c271 | E       | 0xe6510c19c7768ca0937e8f4daf0b16859af9c271 | E     |
| 4       | 0x1e7d250a2ac2646125be3823290fbb5d61d57c13 | E       | 0xe0731c1a30ebed0c6e9162eb87fc85e831caf382 | yes   |

24
SmartWallet

Source: [ES:0xefc7de761ae038b3bb3080ecfb98cea51fd442ea]

Description: Simple wallet for ERC20 tokens. The older version allows the user to configure a backup account where to transfer the tokens, the newer version implements a time lock (cooling period) for withdrawals.

Identification: The wallet can be identified by the following function:

```solidity
transferToUserWithdrawalAccount(address,uint256,address,uint256)
```

The older version additionally contains the function

```solidity
transferToBackupAccount(address,uint256)
```

whereas the newer version can be identified e.g. by the additional function

```solidity
requestWithdraw()
```

The newer version of the wallet keeps the cooling period in a separate contract called `WithdrawalConfigurations`, which can be found by looking for the function

```solidity
withdrawalCoolingPeriod()
```

Addresses: All wallets of this type have been deployed by externally owned accounts. We find five distinct code skeletons.

| wallets version deployed e.g. at creator | creator address user? |
|---|---|
| 37629 older | 0x5e63e5f352fa0167b241a9824165b5cf38ee2973 | A, B, C, D, E |
| 8700 newer | 0x69667ce8641ed03abec2627866543b1126240044 | A, B, D, E |
| 7 older | 0x0d341e13f9f9bd6e02fa97e249a0a27522b0efb1 | F |
| 2 older | 0x0c3e4f2961f6b8d62be9353ac2376e6438a9cf20 | F |
| 2 older | 0x86acc9df6292662bb6b5dd0e46409be37ffe36a | F |

Spendable Wallet

Source: [ES:0x35a1700AC75f6e9E096D9A5c90e3221B658096e0]

Description: Simple wallet to manage a single ERC20 token as specified on deployment. Owner administration. Emergency function to withdraw Ether and tokens other than the one the wallet is intended for. A small number of wallets contain the `tokenFallback` function required by ERC223.

Identification: The wallets of this type can be uniquely identified by looking for bytecode implementing the functions
spend(address,uint256)
claimTokens(address)

Depending on the version, the wallets implement three or four further, less distinctive functions.

All instances of this wallet have been deployed by contracts. These factories can be uniquely identified by looking for bytecode implementing the function

newPaymentAddress(address,address)

Addresses: In total, six versions of bytecode can be found on-chain. The last two in the list below implement the tokenFallback function.

| wallets | code deployed e.g. at | factory |
|---------|------------------------|---------|
| 611     | 0x3ba1700ac75f6e9e096d9a5c90e3221b658096e0 | 0x16e73d7262163c49db410350b94f9f48898821 |
| 452     | 0x28f5378494c693d1ea80aaffaaf53b547879df5 | 0x2048360f36f3cbb858f8e2dbae248bb8ad2e94c23 |
| 366     | 0x2254f466da2a03f900b45647a40c0fca0f73 | 0x365443b7b08861f5f8073de723644f5927ce |
| 3       | 0x6bb5a4f4dc6eeb5de65b983c29f63913 | 0x25f93d0a70c3a9054a9ab5a4a39abf7238f0f3 |
| 2       | 0x0a9d9b3a9e8da2855f5b0c6203963f17e7 | 0x6cac9f1478902be42551813577be5be610ed9688d |
| 2       | 0x409fd06a822c62e7c0d790c673c7ee50f922 | 0x04d4afe9f324b16af5b8e05b6e2b87930c8536 |

Timelocked Wallet

Source: ES:0x5119b5e3a7b7f084732a7ec41fed8aa0c4cd6d4

Description: Simple wallet to handle ERC20 tokens and in most cases also Ether. During deployment, the owner and a lock period is fixed. The wallet may receive assets at any time, but only after the lock period the owner may withdraw the assets.

Identification: This type of wallet can be singled out by looking for bytecode that implements the functions

info()
unlockDate()

Most wallets are deployed by factories that can be detected by looking for bytecode implementing the function

newTimeLockedWallet(address,uint256)

Addresses: We list the top creators that created at least 10 wallets. All except the fifth one are factories.

| wallets | code deployed e.g. at | creator |
|---------|------------------------|---------|
| 54      | 0x1b9fd1b8a967f37f71e74d929baafe86c896f77 | 0x21746d3ce46b4abf471ed47e8d16119b7c87 |
| 36      | 0x009d24660fb8ed4e546c29ee4c94a7db85f1c13 | 0x2664c36e772759e1670862697b5f11ade292ec9 |
| 30      | 0x699caeb2e308b1ca70699567f79f901b9f2a5d4e1eb5 | 0x72093e5b8c6dd7fa1a2ed34777393313 |
| 24      | 0xc9b016bd0a3b9c4cafe6c56f13f5280d20012b453bb | 0x9a150b0070d4fc54f0dc34ec117a07e849756 |
| 13      | 0xa0b43a45f489377f6d44h9b2e1ed5858f06f6ddaf | 0x293e599c77a81619c5fe2fed6bfa600833d6d5 |
| 11      | 0xdd6516b191ca6b7979bfedc0c446e09cb5cbf2374 | 0xb449851de9285467820aa222acbe807285e094 |
A.2 MultiSig Wallets

Argent MultiSig Wallet

Author: Julien Niset

Source: https://github.com/argentlabs/argent-contracts/blob/develop/contracts/MultiSigWallet.sol

Description: m-out-of-n multisig wallet for Ether and ERC20 tokens. The transaction as well as all signatures are computed off-chain. A single call to the wallet is required to execute the multisig transaction, which may be an arbitrary call. The list of owners and the number of required signatures can be modified. The wallet has been deployed only in small numbers, but is worth mentioning because of its clear design.

Identification: The instances of this wallet can be identified by looking for bytecode that implements the function

execute(address,uint256,bytes,bytes)

Addresses: So far, four distinct bytecodes have been deployed at eleven addresses:

- 0x1975c3586e46abd928b5fb0c61136ad789a18cc4
- 0x19a5312e13f458e8afe59c2c213b41243285f2fd
- 0x1b512c29fa62960afb06c292adbe35513f75e2a1
- 0x3f2d0f18530e6e283e9a8bf0ea4cd15c9e19f4f9
- 0x4fee588f5f23474cdafe78e17d2feaa2e3234ed78
- 0xabc603e1c27a961714873e0649b01c56248d68
- 0xc022d15ddcf47d33a3472f92f694ef5cf6549590
- 0xc672c57301483ff384136f59fed46bb72ab1160f
- 0xc6a5d95a62a394bb2eda85c0387a951a09eb2d20
- 0xd60cff99be043d2d2f1c770d781d9b509c569f3
- 0xf8ab2f7464f91bdec950632147c9e0980b6d5c2e

Bitgo MultiSig Wallet

Source: https://github.com/BitGo/eth-multisig-v2

Description: 2-out-of-3 multisig wallet for Ether and ERC20 tokens without owner management. The transaction and one signature are computed off-chain, while the second signature is the one from the message sender. A single call to the wallet is required to execute the multisig transaction, which may be an arbitrary call. Optionally, a second entry point is specialized on token transfers. Some variants of the wallet can create forwarder wallets. There are also restricted forms that lack the possibility for signing general calls.

Identification: We identify the wallets by checking whether one or two of the following functions are among their signatures:

sendMultiSig(address,uint256,bytes,uint256,bytes)
sendMultiSigToken(address,uint256,address,uint256,bytes)
Addresses: Wallets of this type have been deployed roughly 135 k times with 105 distinct bytecodes. For bytecodes that occur at least 10 times, the following table lists the number of deployments, the address of one such deployment, and the number of creators. All wallets were created by an externally owned account, with exception of the wallets in the third row below.

| wallets | code deployed e.g. at | creators |
|---------|-----------------------|----------|
| 134981  | 0xe0f42a3f573d83452e1c3c9c8d14f4499a415cd4 | 353 |
| 163     | 0xf738a52a5835ca3535199645046354271c0eb6   | 2       |
| 72      | 0x5d3bb8aa930bbcb552e4ad4c7bc40be1a4429b   | 1       |
| 43      | 0x3c8640e3a6d57ce9157c1932d8897f5f16408151 | 1       |
| 36      | 0xe1d31b32f45273146f32c84d923b3eadd2396213 | 1       |
| 10      | 0xeb99bf5f5d6e284c1c3196871442e328be6631ad | 3       |

Gnosis MultiSig Wallet

Author: Stefan George

Source: [https://github.com/Gnosis/MultiSigWallet](https://github.com/Gnosis/MultiSigWallet)

Description: m-out-of-n multisig wallet. Each signature is provided with a separate call. Owners sign arbitrary call data that has been composed off-chain as well as the amount of Ether to send along. May receive Ether and ERC20 tokens, no provisions for advanced token handling. An extended version adds support for daily limits. Most wallets are deployed by factories.

Identification: Wallets of this type can be identified by looking for bytecode that implements at least the functions

- changeRequirement(uint256)
- confirmTransaction(uint256)
- executeTransaction(uint256)
- submitTransaction(address,uint256,bytes)

This approach captures several variants, including those with daily limits.

Addresses: Overview of top wallet creators (contracts or users):
### Ivt MultiSig Wallet

**Source:** [ES:0x36d3f1c3ea261ace474829006b6280e176618805](https://github.com/christianlundkvist/simple-multisig)

**Description:** MultiSig wallet with flexible transactions, safe mode, and w/o owner administration.

**Identification:** The wallet is characterized by the following four functions:

```solidity
submitTransaction(address,string,string,uint8[],bytes32[],bytes32[])
submitTransactionToken(address,address,string,string,uint8[],bytes32[],bytes32[])
confirmTransaction(address)
activateSafeMode()
```

**Addresses:** tbd

### Lundkvist MultiSig Wallet

**Author:** Christian Lundkvist

**Source:** [https://github.com/christianlundkvist/simple-multisig](https://github.com/christianlundkvist/simple-multisig)

**Description:** The wallet aims at being the “smallest possible multisig wallet” w/o owner management. It handles Ether and ERC20 tokens, and supports flexible transactions.

**Identification:** The wallet is characterized by the following functions:

```solidity
nonce()
threshold()
ownersArr(uint256)
execute(uint8[],bytes32[],bytes32[],address,uint256,bytes,address,uint256)
```

and in an older version:
execute(uint8[], bytes32[], bytes32[], address, uint256, bytes)

Addresses: tbd

**Nifty MultiSig Wallet**

*Author:* Duncan Cock Foster

*Source:* ES: 0x412Fc2E898a4A89d40bA8c13c7287F75A295D027

*Description:* This multiSig wallet is copied from Gnosis, but added non-fungible token handling.

*Identification:* The wallet is characterized by the following three functions:

- returnUserAccountAddress()
- returnWalletTxCount()
- callTx(bytes, address, uint256, bytes)

MasterMultiSig is one of the two helper contracts containing the functions:

- transactions(uint256)
- confirmations(uint256, address)
- isOwner(address)
- owners(uint256)
- required()
- transactionCount()
- MAX_OWNER_COUNT()
- staticCallContractAddress() -- sufficient to identify contract
- user_control_accounts(uint256)
- returnUserControlAddress(uint256)
- returnIsValidSendingKey(address)
- returnStaticContractAddress()
- changeUserControlAddress(uint256, address)
- addSendingKey(address)
- removeSendingKey(address)
- changeStaticLocation(address)
- recover(bytes32, bytes)
- returnTxMessageToSign(bytes, address, uint256, uint256)
- addOwner(address)
- removeOwner(address)
- replaceOwner(address, address)
- changeRequirement(uint256)
- submitTransaction(address, uint256, bytes)
- confirmTransaction(uint256)
- revokeConfirmation(uint256)
- executeTransaction(uint256)
- isConfirmed(uint256)
- getConfirmationCount(uint256)
getTransactionCount(bool, bool)
getOwners()
getConfirmations(uint256)
getTransactionIds(uint256, uint256 bool, bool)

NiftyStaticCalls is the other helper contract containing the functions:

retAdd()
isValidSignature(bytes32, bytes)
isValidSignature(bytes, bytes)
onERC721Received(address, address, uint256, bytes)
ERC1155_RECEIVED_SIG()
ERC1155_BATCH_RECEIVED_SIG()
ERC1155_RECEIVED_INVALID()
lastaData()
lastOperator()
lastId()
lastValue()
onERC1155Received(address, address, uint256, uint256, bytes)
onERC1155BatchReceived(address, address, uint256[], uint256[], bytes)

Addresses: tbd

Parity/Eth/Wood

Author: Gavin Wood
Source: https://github.com/paritytech/parity/blob/4d08e7b0aec46443bf26547b17d10cb302672835/js/src/contracts/snippets/enhanced-wallet.sol

Description: MultiSig wallet with owner management, flexible transactions, and daily limit.

Identification: The wallet is characterized by the following function:

hasConfirmed(bytes32, address)
m_required()
m_numOwners()

Additional functions are:

version() -- Ethereum wallet
revoke(bytes32) -- Ethereum wallet/Parity library
changeOwner(address, address) -- Ethereum wallet/Parity library
addOwner(address) -- Ethereum wallet/Parity library
removeOwner(address) -- Ethereum wallet/Parity library
changeRequirement(uint256) -- Ethereum wallet/Parity library
isOwner(address) -- Ethereum wallet/Parity library/Parity wallet
hasConfirmed(bytes32,address) -- Ethereum wallet/Parity library/Parity wallet
setDailyLimit(uint256) -- Ethereum wallet/Parity library
resetSpentToday() -- Ethereum wallet/Parity library
kill(address) -- Ethereum wallet/Parity library
execute(address,uint256,bytes) -- Ethereum wallet/Parity library
confirm(bytes32) -- Ethereum wallet/Parity library
m_required() -- Ethereum wallet/Parity library/Parity wallet
m_numOwners() -- Ethereum wallet/Parity library/Parity wallet
m_dailyLimit() -- Ethereum wallet/Parity library/Parity wallet
m_spentToday() -- Ethereum wallet/Parity library/Parity wallet
m_lastDay() -- Ethereum wallet/Parity library/Parity wallet
getOwner(uint256) -- Parity library/Parity wallet
initMultiowed(address[],uint256) -- Parity library
initDaylimit(uint256) -- Parity library
initWallet(address[],uint256,uint256) -- Parity library
Deposit(address,uint256) -- extra signature

Addresses: tbd

Teambrella MultiSig Wallet

Source: [ES:0x44852FAEFcb42E392f2c55c6df53A50A732Df298]

Description: MultiSig wallet with flexible transactions and owner management. In one variant, there is a recovery mechanism.

Identification: The wallet is characterized by the following function:

m_teamId()

The wallet contains the further functions:

m_opNum()

m_owner()

m_cosigners(uint256)

m_cosignersApprovedDisband(uint256)

assignOwner(address[],uint256,address)

changeAllCosigners(uint256,address[],uint256[3],bytes,bytes,bytes)

changeAllCosigners2(uint256,address[],bytes,bytes,bytes)

transfer(uint256,address[],uint256[],bytes,bytes)

transfer2(uint256,address[],uint256[],bytes,bytes,bytes)

approveDisband()

and in a variant also:

rescue(address)

Addresses: tbd

32
Unchained Capital MultiSig Wallet

Source: [https://github.com/unchained-capital/ethereum-multisig](https://github.com/unchained-capital/ethereum-multisig)

Description: Basic MultiSig wallet with flexible transactions.

Identification: The wallet is characterized by the following five functions:

- spendNonce()
- unchainedMultisigVersionMajor()
- unchainedMultisigVersionMinor()
- generateMessageToSign(address,uint256)
- spend(address,uint256,uint8,bytes32,bytes32,uint8,bytes32,bytes32)

Addresses: tbd

A.3 Forwarder Wallets

BitGo Forwarder Wallet

Source: [https://github.com/BitGo/eth-multisig-v2](https://github.com/BitGo/eth-multisig-v2)

Description: The wallet can forward its assets to the parent who created it.

Identification: The wallet is characterized by the following three functions:

- parentAddress()
- flushTokens(address)
- flush()

Optinal entry points are:

Forwarder()
initialize()
logger()
initialize(address)
feeAddress()
0x7cade34a
updateOwner(address)
changeParent(address)
0x0e18b681
newAdmin()
changeAdmin(address)
changeTarget(address)
admin()
0x813f0f59
0xf4069e83

Addresses: tbd
IntermediateWallet

**Source:** ES:0x5a01a4A46108794055Fe90DC5dbb8a3823a1c2e6

**Description:** Simple wallet for Ether and ERC20 tokens with forwarding of assets to hardcoded address.

**Identification:** The wallet is characterized by the following functions:

- owner()
- wallet()
- transferOwnership(address)
- setWallet(address)
- retrieveTokens(address,address)

A variant also contains the function:

- tokenFallback(address,uint256)

**Addresses:** tbd

SimpleWallet2

**Source:** ES:0xcqq2f21e3d752834a38a0f8a68c4cefaef84df1eb4541

**Description:** Simple wallet that forwards all its Ether to its owner.

**Identification:** The wallet is characterized by the following four functions:

- owner()
- transferOwnership(address)
- weiBalance()
- claim(address)

**Addresses:** tbd

### A.4 Controlled Wallets

**Bittrex**

**Author:** Bittrex

**Source:** ES:0xa3C1E324CA1ce40db73eD6026c4A177F099B5770

**Description:** This controlled wallet involves three types of contracts: controller, sweeper, and user wallets. The controller creates wallets and maintains a list of sweepers that are used by the wallets to perform transfers. A separate sweeper can be set for each kind of token or Ether. In practice the so-called default sweeper is used most of the time; it is deployed at the same time as the controller. With a total of 14 addSweeper call, only eight controllers have set a new sweeper so far: three times for each of the OMG, SONM, Tether token, twice for the Binance token and for Ether, and once for the Mithril token.

Figure 4 shows the interaction of controller, (default) sweeper, and user wallets. A
Figure 4: Bittrex controlled wallet: interactions between controller, sweeper, and wallets.

user or contract representing e.g. an exchange, \( E \), deploys two companion contracts in a single transaction: a controller \( C \) and a default sweeper \( D \) (1). In further transactions, \( E \) calls \( C \) with \texttt{makeWallet} (2). This prompts \( C \) to deploy wallets \( W_i \) (3). In total almost 2.5M wallets have been deployed this way. The wallet addresses can be given to customers, who announce them to receive Ether and tokens. The control over the wallets, however, stays with \( E \), similar to accounts of traditional banks.

Now suppose Ether or tokens owned by \( W_1 \) are to be transferred to the destination address \( d \). First, \( E \) calls \( C \) with \texttt{changeDestination} to store \( d \) in the controller (4). Destination addresses are changed rarely (so far 65 times for all controllers, in total). Then \( E \) initiates the transfer by calling the wallet with \texttt{sweep}, passing \( W_1 \) the token address (0 for Ether) and the amount (5). The wallet asks the controller for the sweeper in charge of the token (6). Next, the wallet performs a delegate call to the sweeper, \( D \), handing over token address and the amount (7). The delegate call has the effect that all actions by the sweeper will seem to originate from the wallet even though the code is actually contained in the sweeper. The sweeper calls the controller several times to learn the destination address and to check permissions (8). Then it performs the token transfer or sends the Ether (9). Finally, on successful completion, the sweeper prompts the controller to issue the event \texttt{LogSweep} (10).

\textit{Identification:} The controllers and default sweepers are detected by a combination of signature and message analysis. A contract, \( C \), is a controller if it satisfies the following criteria.

- During the deployment of \( C \) another contract, \( D \), is created.

- The only signatures of \( D \) are \texttt{sweep(address,uint256)} and optionally \texttt{controller()}.

- The only contracts created by \( C \) are wallets, \( W \).

- The only signatures of \( W \) are \texttt{sweep(address,uint256)} and optionally \texttt{tokenFallback(address,uint256,bytes)}.

Table 3 lists the addresses of all controllers that have deployed at least ten wallets with their default sweepers.
| wallets | controller | default sweeper |
|---------|------------|----------------|
| 1578470 | 0xa3c1e324ca| 0xb2233fcee4c |
| 63344   | 0xdf9101c1| 0x88e44a3e173 |
| 124580  | 0xedce88316| 0x88de41a3487 |
| 50909   | 0x9edf853a | 0x3105d1027f4 |
| 292015  | 0x2754b2822| 0x32403d04d6 |
| 10693   | 0x8bd7c1e9| 0xf9dd79ef77 |
| 7593    | 0xe4746809| 0xc27304dcd4 |
| 3093    | 0xe91ebf1b| 0xc74681c066 |
| 3350    | 0x26735a2c| 0x3cb2271be7 |
| 325     | 0x9edf853a| 0x88de41a348 |
| 32327   | 0x7563    | 0x27304dcd46 |
| 1058    | 0x36791fa| 0x3c8ba598d8 |
| 977     | 0x8e474680| 0x88e44a3e17 |
| 887     | 0xfcdcc1a| 0x44746809e6 |
| 600     | 0x4202b629| 0xc27304dcd4 |
| 422     | 0x6933127b| 0x32403d04d6 |
| 305     | 0x357f9a1| 0xc27304dcd4 |
| 300     | 0xb4cbc0c| 0xc27304dcd4 |
| 228     | 0xa7ab11e| 0xc27304dcd4 |
| 212     | 0xadbe6e| 0xc27304dcd4 |
| 209     | 0x00e8c2| 0xc27304dcd4 |
| 201     | 0x6658d2c| 0xc27304dcd4 |
| 150     | 0x2c164b1| 0xc27304dcd4 |
| 141     | 0x69e6030| 0xc27304dcd4 |
| 135     | 0x9d44d08| 0xc27304dcd4 |
| 119     | 0xeb818c6| 0xc27304dcd4 |
| 117     | 0x6f6873| 0xc27304dcd4 |
| 107     | 0xb82464| 0xc27304dcd4 |
| 71      | 0xc227f| 0xc27304dcd4 |
| 66      | 0xf0833b4| 0xc27304dcd4 |
| 62      | 0x7c9338b| 0xc27304dcd4 |
| 60      | 0x2081c| 0xc27304dcd4 |
| 60      | 0x721a6f3| 0xc27304dcd4 |
| 50      | 0x945ef5| 0xc27304dcd4 |
| 40      | 0xe030f| 0xc27304dcd4 |
| 35      | 0x6cb075| 0xc27304dcd4 |
| 35      | 0xe4460| 0xc27304dcd4 |
| 31      | 0x127925| 0xc27304dcd4 |
| 25      | 0x26f844| 0xc27304dcd4 |
| 24      | 0x1e7311| 0xc27304dcd4 |
| 19      | 0x3c861c| 0xc27304dcd4 |
| 16      | 0xb0db51| 0xc27304dcd4 |
| 15      | 0x539a| 0xc27304dcd4 |
| 14      | 0xa1df42| 0xc27304dcd4 |
| 11      | 0xd7d4| 0xc27304dcd4 |
| 7      | 0xe030f| 0xc27304dcd4 |
| 5      | 0x4202b6| 0xc27304dcd4 |
| 3      | 0xe030f| 0xc27304dcd4 |
A.5 Update Wallets

Eidoo Update Wallet

Source: [ES:0x0409b14cac8065e4972839f9dafabb20cef72662]

Description: The wallet can be updated, manage Ether and ERC20 tokens, and place orders.

Identification: The wallet is characterized by the following functions:

- owner()
- exchange()
- tokenBalances_(address)
- logic()
- birthBlock()
- depositEther()
- depositERC20Token(address,uint256)
- updateBalance(address,uint256,bool)
- updateExchange(address)
- updateLogic(uint256)
- verifyOrder(address,uint256,uint256,address)
- withdraw(address,uint256)
- balanceOf(address)

The helper contract WalletLogic contains the functions:

- owner()
- exchange()
- tokenBalances_(address)
- deposit(address,uint256)
- updateBalance(address,uint256,bool)
- verifyOrder(address,uint256,uint256,address)
- withdraw(address,uint256)
- safeTransfer(address,address,uint256)
- safeTransferFrom(address,address,address,uint256)
- safeApprove(address,address,uint256)

Addresses: tbd

LogicProxyWallet

Source: [ES:0x5d4a945271fb3e16481bf6ce0bad5f6b2e9d13db]

Description: This wallet can execute flexible transactions, as well as change the owner and the version of the implementation.

Identification: The wallet is characterized by the following functions:
execute(address, bytes, uint256, uint256)
owner()
setOwner(address)
isAuth(address)
registry()

The helper contract registry contains the functions:
proxies(address)
build()
build(address)
record(address, address)
logicProxiesStatic(address)
logicProxies(address)
logic(address)
logicStatic(address)
enableStaticLogic(address)
enableLogic(address)
disableLogic(address)
getAddress(string)
setAddress(string, address)

Addresses: tbd

A.6 Smart Wallets

Argent Smart Wallet

Author: Julien Niset
Source: https://github.com/argentlabs/argent-contracts/tree/develop/contracts/wallet/
Description: Modular wallet where the base wallet manages the modules that are entitled to call the wallet, and authorized modules may execute any transaction.
Identification: The wallet is characterized by the following function:
init(address, address[])

and contains the further functions:
implementation()
owner()
authorised(address)
enabled(bytes4)
modules()
authoriseModule(address, bool)
enableStaticCall(address, bytes4)
setOwner(address)
invoke(address, uint256, bytes)
The deploying factory is identified by the function:

```solidity
createWallet(address, address[], string)
```

and contains the further functions:

```solidity
moduleRegistry()
walletImplementation()
ensManager()
ensResolver()
changeModuleRegistry(address)
changeWalletImplementation(address)
changeENSManager(address)
changeENSResolver(address)
init(address)
getENSReverseRegistrar()
addManager(address)
getENSRegistry()
revokeManager(address)
ADDR_REVERSE_NODE()
owner()
resolveEns(bytes32)
changeOwner(address)
managers(address)
```

Addresses: tbd

**Dapper Smart Wallet**

*Source:* [https://github.com/dapperlabs/dapper-contracts](https://github.com/dapperlabs/dapper-contracts)

*Description:* A wallet suited for Ether, ERC20 and advanced tokens with cosigner functionality, flexible transactions, and a recovery mechanism. The multiSig functionality employs signed messages based on ERC191 for confirmation support.

*Identification:* The wallet is characterized by the following function:

```solidity
invoke0(bytes)
```

and contains the further functions:

```solidity
EIP191_VERSION_DATA()
EIP191_PREFIX()
VERSION()
COMPOSITE_PLACEHOLDER()
AUTH_VERSION_INCREMENTOR()
authVersion()
authorizations(uint256)
nonces(address)
```
delegates(bytes4)
initialized()
init(address,uint256,address)
setDelegate(bytes4,address)
setAuthorized(address,uint256)
emergencyRecovery(address,uint256)
recoveryAddress(address)
setRecoveryAddress(address)
recoverGas(uint256,address[])
isValidSignature(bytes32,bytes)
supportsInterface(bytes4)
invoke1CosignerSends(uint8,bytes32,bytes32,uint256,address,bytes)
invoke1SignerSends(uint8,bytes32,bytes32,bytes)
invoke2(uint8[2],bytes32[2],bytes32[2],uint256,address,bytes)
onERC721Received(address,uint256,bytes)
onERC721Received(address,address,uint256,bytes)
tokenFallback(address,uint256,bytes)
ERC223_ID()

Its factory is identified by:

deployFullWallet(address,address,uint256)

and contains the further functions:

reclaimEther()
cloneWalletAddress() -- also a unique identifier
deployCloneWallet(address,address,uint256)
deployCloneWallet2(address,address,uint256,bytes32)
deployFullWallet2(address,address,uint256,bytes32)

Addresses: tbd

Gnosis

Author: Stefan George, Richard Meissner, Ricardo Guilherme Schmidt
Source: https://github.com/gnosis/safe-contracts/blob/development/contracts/GnosisSafe.sol
Description: A modular multiSig wallet with flexible transactions for Ether, ERC20 and advanced tokens with daily limit and recovery mechanism.
Identification: The wallet is characterized by the following function:

approvedHashes(address,bytes32)

and contains the further functions:
NAME()
VERSION()
DOMAIN_SEPARATOR_TYPEHASH()
SAFE_TX_TYPEHASH()
SAFE_MSG_TYPEHASH()
nonce()
domainSeparator()
signedMessages(bytes32)

Addresses: tbd