2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)

DataEther: Data Exploration Framework For Ethereum

Ting Chen*†, Zihao Li*†, Yufei Zhang*, Xiapu Luo†, Ang Chen‡, Kun Yang*, Bin Hu*, Tong Zhu§,
Shifang Deng*, Teng Hu*, Jiachi Chen†, Xiaosong Zhang*
*Center for Cybersecurity, University of Electronic Science and Technology of China, China
†Department of Computing, The Hong Kong Polytechnic University, China
‡Department of Computer Science, Rice University, USA
§School of Computer Science and Engineering, University of Electronic Science and Technology of China, China

***Abstract*—Ethereum is the largest blockchain platform supporting smart contracts with the second biggest market capitalization. Ethereum data can yield many useful insights because of the large volume of transactions, accounts and blocks as well as the popular applications developed as smart contracts. Studying Ethereum data can also reveal many new attacks to the platform and its smart contracts. Unfortunately, it is non-trivial to systematically explore Ethereum because it involves massive heterogeneous data, which are produced and stored in different ways. Although a few recent studies report some interesting observations about Ethereum, they are limited by their data acquisition methods which cannot provide comprehensive and precise data. In this paper, to fill the gap, we propose DataEther, a systematic and high-fidelity data exploration framework for Ethereum by exploiting its internal mechanisms. Besides supporting the analyses in existing studies, DataEther further empowers users to explore unknown phenomena and obtain in-depth understandings. We first describe how we tackle the challenging issues in developing DataEther, and then use four data-centric applications to demonstrate its usage and report many new observations.**

I. Introduction

Being the second largest blockchain in market capitalization [1], Ethereum distinguishes itself as the most successful blockchain supporting smart contracts. A *smart contract* is a piece of autonomous program running on the blockchain according to the program logic defined beforehand [2]. Exploring this complex system can yield many useful insights because of the huge volume of transactions, accounts and blocks [3], popular applications developed as smart contracts (e.g., token ICOs [4], phenomenal games [5]), as well as attacks to the platform and its smart contracts [6].

It is non-trivial to systematically explore Ethereum because it involves massive heterogeneous data, such as blocks, transactions, smart contracts, execution traces, and accounts. Although a few recent studies report some interesting observations by exploring data from Ethereum [3], [7]–[13], they are limited by their data acquisition methods, which can be divided into four categories, including **C1**: downloading & parsing block files [8], [14]; **C2**: invoking Web3 APIs provided by Ethereum [9], [10]; **C3**: crawling blockchain explorer websites [11], [12]; **C4**: instrumenting Ethereum node [3], [13]. In particular, they suffer from the following limitations: *incomplete data*, *confusing information*, and *inefficiency*.

First, all four categories of methods produce incomplete data. Concretely, **C1** and **C2** methods cannot obtain internal transactions (to be explained in §II) launched by smart contracts, thus missing the interactions among smart contracts (e.g., how malicious contracts exploit the vulnerabilities in legitimate contracts). The blockchain explorer websites used by **C3** methods usually list partial data and discourage automated crawlers to guarantee the availability to all visitors. For example, Etherscan [15], the most popular Ethereum block explorer, just shows the last 0.5 million transactions. Although some blockchain explorer websites provide APIs for fetching data, they also just support partial data (e.g., Etherscan's APIs return only the most recent 10,000 transactions [16]). Note that the APIs provided by Ethereum are different from the APIs provided by blockchain explorer websites. The former collect data from an Ethereum node, whereas the latter acquire data from the websites of blockchain explorers. Existing **C4** methods only collect limited data for specific applications. For instance, our previous work only collects the sender, receiver, and the amount of money transferred in each transaction to investigate money transfer, contract creation and contract invocation [3]. Grossman et al. just acquire internal transactions and storage operations to detect the reentrancy vulnerability of smart contracts [13].

Second, **C3** methods may lead to confusing information. For example, the 'Contract Creation Code' box in Etherscan shows the deployment bytecode of a smart contract. Since the 'Switch To Opcodes View' button is next to this box, users may expect to see the disassembly of the deployment bytecode after clicking the button. However, Etherscan shows the disassembly of the runtime bytecode. The difference between the deployment bytecode and the runtime bytecode is detailed in §II-B. Moreover, Etherscan sometimes just lists partial internal transactions and displays the statement "... Produced N Contract Internal Transactions", where N is smaller than the correct number. Consequently, users may mistakenly think that there are just N internal transactions. For instance, we find 243 internal transactions incurred by a transaction whose hash is "0x8023679696f1db7fe00adfdab4fcd8600a7cd6e4c5d8a8054df0d97cd06503de", but Ethereum just lists 31 out of them and displays the statement "... Produced 31 Contract Internal Transactions" [17].

2575-8411/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDCS.2019.00137

Third, **C2** and **C3** methods are inefficient. The latter is restricted by the network bandwidth and the rate limit imposed by the web server. For example, Etherscan restricts the frequency of invoking its APIs to 5 queries/sec [16]. For the former, some Web3 APIs return data slowly due to their design. For example, `debug_traceTransaction()` takes a long time to return the execution trace of each queried transaction. We learn the root cause after inspecting the source code of Ethereum: before executing the queried transaction, this API has to initialize the runtime environment, construct the correct state before the execution of the block containing the queried transaction, and then replay the preceding transactions before the queried transaction in the same block. Moreover, Web3 APIs use Remote Procedure Calls (RPC) to communicate with an Ethereum node, which introduces further delay. Hence, the long time required for collecting the execution traces for all transactions (more than 370 million) through this API would be unacceptable.

To address the above limitations in existing studies, we propose DataEther, a systematic and high-fidelity data exploration framework for Ethereum by exploiting its internal mechanisms and carefully instrumenting an Ethereum full node. The rationale of our approach lies in the fact that each node keeps the *same* copy of the blockchain and the execution of smart contracts can be replayed by leveraging the transactions invoking those smart contracts [2]. Different from existing methods, DataEther acquires all blocks (§III-B), transactions (§III-D), execution traces (§III-C), and smart contracts (§III-E) from Ethereum, and collects ERC20 token activities (§III-F). Table I lists the information that can be acquired by DataEther and existing methods.

We overcome several technical challenges in developing DataEther. First, Ethereum has massive heterogeneous data distributed in different sources, such as execution traces in Ethereum Virtual Machine and transactions in blocks, without a detailed document about the internals of Ethereum. We overcome this issue by first scrutinizing Ethereum implementation and then carefully instrumenting an Ethereum node to acquire the data. Second, not all data can be directly obtained from Ethereum. We address this challenge by first figuring out the relationships among data and then deciding how to obtain them. For example, the balance of an account is determined by several factors, including the genesis block, transactions that increase/decrease the balance, block rewards, uncle block rewards, and gas rewards. Hence, we first restore such information to recover all the historical values of account balance. Third, we need to collect the information of tokens (token is a kind of smart contract) but less than 1% smart contracts are open-source [18]. We recognize tokens and their behaviors by conducting bytecode analysis and trace analysis.

Based on the plenty of diverse data collected by DataEther, we develop four new applications to demonstrate its usage, including profiling important entities in Ethereum such as account balance and tokens (§IV), characterizing reentrancy attacks through transaction-based analysis (§V), revealing non-deployable contracts through contract-based analysis (§VI), and detecting underpriced DoS attacks through trace-based analysis (§VII). We obtain many new insights from these applications. For example, In §IV, we learn that Nanopool, a mining pool, gets stable profit and distributes it to its miners. Moreover, most token holders invest just one kind of tokens while most tokens receive little attention. In §V, we discover 22 accounts that exploited the reentrancy vulnerability of the DAO contract. In particular, 13 out of them have not been reported before, and 7 out of 22 accounts exploited an *unreported attack surface*. In §VI, we discover a special kind of smart contracts, named *non-deployable contracts*, and learn their usages, including sending Ether to multiple accounts by one transaction, transferring Ether 'forcibly', and using non-deployable contracts to steal Ether in a new way. In §VII, we uncover 27,313 *unreported* attacking transactions that exploited 3 underpriced operations to launch DoS attacks.

TABLE I
INFORMATION THAT CAN BE ACQUIRED BY EXISTING METHODS AND DATAETHER. ✓ AND × DENOTE THAT THE INFORMATION CAN/CANNOT BE COLLECTED, RESPECTIVELY. △ MEANS THAT THE INFORMATION CAN ONLY BE PARTIALLY OBTAINED. WE SELECT ETHERSCAN AS A REPRESENTATIVE OF **C3**, OUR PREVIOUS WORK [3] AND GROSSMAN ET AL.'S WORK [13] AS REPRESENTATIVES OF **C4**.

	C1	C2	C3 (Etherscan)	C4 [3]	C4 [13]	DataEther
external transaction	√	√	△	√	×	√
internal transaction	×	×	△	√	√	√
block	√	√	△	×	×	√
uncle block	√	√	△	×	×	√
trace	×	√	△	×	△	√
contract bytecode	△	√	△	×	×	√
account balance	×	√	△	×	×	√
token transfer	×	×	△	×	×	√
token name/symbol	×	×	△	×	×	√

We also note that although DataEther has been implemented for Ethereum, it can be easily extended to support other blockchains (e.g., NEO [19], TRON [20], Qtum [21]) which have the similar design of smart contracts with Ethereum. In summary, we make the following contributions.

(1) We propose DataEther, a systematic and high-fidelity data exploration framework for Ethereum by exploiting its internal mechanisms and instrumenting an Ethereum full node.

(2) We implement DataEther after addressing several technical challenges. It obtains all historical data and enables new functionalities. It is also more efficient than existing methods.

(3) We develop four data-centric applications based on the data collected by DataEther, and obtain many new observations. DataEther and the collected data will be released after the paper is published.

The remainder of this paper is organized as follows. §II introduces the necessary background. After detailing DataEther in §III, we describe four applications based on DataEther in §IV, §V, §VI, and §VII, respectively. We review related studies in §VIII and conclude the paper in §IX.

II. BACKGROUND

A. Account and Transaction

Ethereum has two kinds of accounts: (1) Externally Owned Account (EOA). Each EOA has a pair of <public key, private key> [2] and its address is derived from the public key; (2)

Smart contract account. Such account is created by an EOA or another smart contract, and has executable bytecode of smart contract [2]. An account is referred to by its address. A transaction is a message sent from an EOA to another account, which carries information like the amount of Ether sent, the method invoked and the corresponding parameters [2]. An *internal transaction* is a message sent from a smart contract account to another account [2]. Note that internal transactions are *not* stored in the blockchain and hence cannot be obtained by parsing blocks. To avoid ambiguity, we use *external transaction* to denote the transaction sent from an EOA.

B. Smart Contract and Ethereum Virtual Machine

Smart contracts are usually developed in high-level languages(e.g., Solidity [22]) and then compiled into bytecode to be executed by the Ethereum Virtual Machine (EVM). EVM is a stack-based virtual machine with 130+ operations [2]. Besides stacks, EVM also has *Memory*, which provides transient storage, and *Storage*, which provides permanent storage [2].

The bytecode of a smart contract can be deployed to Ethereum by an external or internal transaction. In the former case, the *input data* property of the external transaction contains the deployment bytecode of the contract [2]. In the latter case, the deployment bytecode is stored in the memory [2]. The *deployment bytecode* consists of the *initialization bytecode* for initializing the smart contract and its parameters as well as the *runtime bytecode*. The runtime bytecode will be stored in the blockchain, but the initialization bytecode will be discarded after contract deployment [2].

A contract can be invoked by an external transaction, whose *input data* property provides the parameters and specifies the invoked method and *to* property contains the contract address, or an internal transaction [2]. In the latter case, the address of the contract to be invoked is in the stack and the invoked method and parameters are in the memory.

Ethereum allows a contract to self-destruct, which is useful in several situations (e.g., stopping a buggy contract). When a contract self-destructs, its runtime bytecode will be removed from the blockchain and the remaining Ether in the contract will be sent to a specified account [2].

C. Ether, Token and Gas

Ether is the cryptocurrency of Ethereum [2]. A user can get Ether by mining blocks, purchasing from markets, or from other users. *Token* is a kind of alternative cryptocurrency coins, which is implemented in smart contracts and operated on Ethereum [23]. Ethereum supports launching tokens for ICO (Initial Coin Offering) by deploying the token contacts, which usually follow certain standards (e.g., ERC20 [24]). A token standard defines standard methods and standard events whose semantics are well-documented. Besides standard methods and standard events, a token standard allows developers to implement non-standard methods and non-standard events [24].

To thwart resource abuse and incentivize nodes to contribute their computing resources, Ethereum introduces the gas mechanism that requires transaction senders to pay execution fees [2]. Ethereum defines the *gas cost* of EVM operations [2] and lets the *gas price* be changeable by transaction senders. The execution fee of a transaction is the multiplication of the gas price with the total gas cost of all executed operations [2].

D. Block, Mining, and Synchronization

A *block* consists of a block header containing meta information and a block body comprising zero or more transactions [2]. Note that the block stores neither execution traces nor token activities. The *genesis* block (i.e., the first block) sets the initial state of the blockchain, such as the balances of some accounts, etc. All blocks are linked together. The *parent block* of a block A is the block mined exactly before A [2], and every block (except the first one) has the block hash of its parent.

The accounts who mine blocks (i.e., miners) will be rewarded. The mining reward consists of three parts: block reward, uncle block reward and gas reward. The *uncle block* of a block A has the same parent block with the parent block of A [2] but it does not include transactions. Since uncle blocks accelerates block confirmation and improves the security of blockchain [25], mining uncle blocks is also rewarded. A miner will receive gas reward if its mined block includes transactions. Since proof-of-work mining needs lots of computing resources [26], miners tend to form a large mining pool to increase the success probability of mining blocks. The mining pool aggregates the computation results from its miners and distributes mining reward to them. Note that mining pools have diverse reward distribution schemes.

When an Ethereum node joins the Ethereum network, it first synchronizes with its peers for downloading blockchain or the state of blockchain before validating or mining blocks. Ethereum supports three synchronization modes, namely full, fast and light [27]. Only a *full* node implements the full mode that downloads the entire blockchain, stores it in block files, and executes all historical transactions [27].

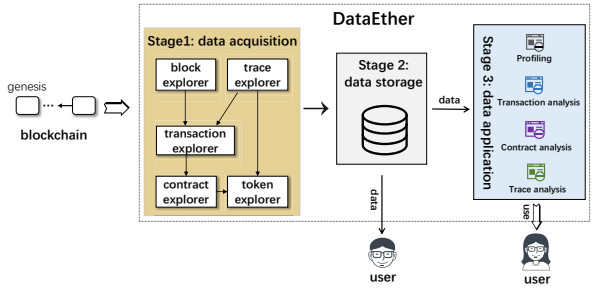

Fig. 1. High-level overview of DataEther

III. DataEther

A. Overview

As shown in Fig. 1, DataEther consists of three stages. In the data acquisition stage, DataEther obtains the raw data from the blockchain during synchronization and processes it through five modules to collect information from blocks (§III-B), execution traces (§III-C), transactions (§III-D), smart contracts (§III-E), and tokens (§III-F), respectively. The transaction explorer needs the data collected by the block explorer

and trace explorer, because external transactions are stored in blocks and internal transactions are extracted from the traces. The contract explorer relies on the transaction explorer, because a contract is deployed and invoked by a transaction. The token explorer depends on the contract explorer and trace explorer, because we identify tokens by contract analysis and recognize token behaviors by trace analysis. It is worth noting that acquiring such information is *not* straightforward because the massive heterogeneous data are distributed in diverse sources and not all data can be directly obtained from Ethereum. We address these issues by first figuring out the relationships among various types of data and then exploiting the internal mechanisms of Geth, an Ethereum full node, through instrumentation. The details are reported in the following subsections.

In the data storage stage, all extracted data are stored in ElasticSearch [28] for the ease of data management. Users can get access to the collected data in ElasticSearch. In the application stage, users can conduct various analyses on the collected data. We demonstrate it through four data-centric applications, including profiling account balance and tokens (§IV), characterizing reentrancy attacks through transaction-based analysis (§V), revealing non-deployable contracts through contract-based analysis (§VI), and detecting underpriced DoS attacks through trace-based analysis (§VII).

B. Block Explorer

Block explorer collects the information of genesis block and normal/uncle blocks as well as mining rewards. During synchronization, an Ethereum full node constructs the entire blockchain locally and validates every block, especially the fields in each block (e.g., gas consumption, the hash of execution results of transactions, timestamp, etc.) using the function `ValidateBlock()`. DataEther records the block data after block validation by extracting all properties of the validated block. For example, the timestamp in each block is necessary for trend analysis (§IV), and the coinbase (i.e., account) that mines this block is used for investigating mining behaviors. If a block contains transactions, the Transaction Explorer (§III-D) extracts and processes them. DataEther gathers the data of uncle blocks by instrumenting `ValidateBlock()` since it also validates uncle blocks. Since the block header stores the property *parent hash*, DataEther correlates a block with its uncles by checking their parent hashes.

The data in the genesis block sets the initial state of Ethereum. We *cannot* use the above approach to acquire the data in the genesis block because there are no transactions in the genesis block and there is no miner for it. Fortunately, we find that Ethereum creates the genesis block by invoking the function `ToBlock()`, which sets the initial states of many accounts. These accounts are organized in a map which associates an account address with the data structure `GenesisAccount`. Since `GenesisAccount` contains the initial state information of an account, which is unmarshalled from a configuration file, we collect the information in the genesis block by instrumenting the `ToBlock()` function.

Miners will be rewarded after they mine blocks/uncle blocks, which consist of block rewards, uncle block rewards and gas rewards. The reward is of interest for profiling miners' behaviors but it is not recorded in the block. By inspecting Geth, we find that the function `accumulateRewards()` sets block rewards and uncle block rewards and the function `TransitionDb()` sets the gas rewards, and hence DataEther instruments these two functions to record rewards.

C. Trace Explorer

Since executing smart contracts is a salient feature of Ethereum, we propose a new approach to acquire the detailed information of *every* executed EVM operations. Such information is important to many applications, such as recognizing token transfers (§III-F) and detecting underpriced DoS attacks (§VII). More precisely, since Geth provides interpretation handlers to interpret EVM bytecode, we carefully instrument all 130+ interpretation handlers by inserting recording code to log the detailed information of each operation, including its program counter, opcode and operand, the stack/memory/storage items that are read/written, and the gas consumption of the operation.

```
1  func opJumpi(pc *uint64, evm *EVM, contract *Contract, ···){
2    pos, cond := stack.pop(), stack.pop();
3    Log(*pc, cond, pos)
4    if cond.Sign() != 0 {...
5      *pc = pos.Uint64()
6    } else {
7      *pc++}
8    evm.interpreter.intPool.put(pos, cond)
9    return nil, nil }
```

Fig. 2. Instrumenting the handler of JUMPI

We use the operation JUMPI as an example to explain how to instrument the operations. Fig. 2 presents the code snippet of its interpretation handler with our code (Line 3). JUMPI is used for the conditional jump, which consumes top two stack items. The top stack item is the jump target and the second item is the outcome of the latest comparison (Line 2). After the execution of JUMPI, the program counter (PC) moves to the jump target (Line 5) if the comparison result is not 0; otherwise, it moves to the next operation (Line 7). DataEther logs PC and the two stack items read by JUMPI (Line 3).

After gathering the execution trace, we need to correlate it with the transaction because the same trace can be triggered by different transactions. It is non-trivial to achieve this goal because the interpretation handlers do not know which transaction is executed. After analyzing Geth, we find that it invokes `ApplyTransaction()` to execute a transaction. Specifically, this function first extracts a message from the transaction being executed, and then initializes an EVM context and creates an EVM object. After that, it executes the message by calling `ApplyMessage()`, in which EVM operations are executed in order. Therefore, DataEther instruments these functions to correlate a transaction with its execution trace.

D. Transaction Explorer

Transaction explorer collects internal and external transactions when DataEther processes blocks. With the information of transactions, we can extract smart contracts from them

(§III-E). It is easy to collect external transactions, but it is not straightforward to capture internal transactions, because they are not stored in the blocks. To address this issue, we first identify all EVM operations that can trigger internal transactions [2] because internal transactions are invoked by smart contracts, and then investigate the execution traces that contain any of such operations. Eventually, we identify 6 EVM operations (i.e., CREATE, CALL, CALLCODE, DELEGATECALL, STATICCALL, SELFDESTRUCT).

CREATE is used to create a smart contract. CALL invokes a smart contract whose address is specified by the second stack item. The id of the invoked method and the parameters passed to the method are stored in the memory. A contract can also send Ether to another account using CALL and the amount is specified in the third stack item. CALLCODE is equivalent to CALL except that the callee runs in the caller's context. For example, when the callee writes to the storage, it writes to the caller's storage. DELEGATECALL is equivalent to CALLCODE except that it persists the current values of properties *sender* (i.e., the transaction sender) and *value* (i.e., the amount of Ether). DELEGATECALL is considered to be a bug-free version of CALLCODE [29]. STATICCALL functions equivalently to CALL, except that it disallows any modifications to the state during the call (and its subcalls, if present) [30]. SELFDESTRUCT is used to self-destruct a smart contract, and the account specified by the top stack item will receive the remaining Ether of the self-destructed contract [2].

By analyzing the execution traces containing any of these 6 operations, DataEther collects all internal transactions triggered by these operations. DataEther acquires various information about an internal transaction, including the corresponding EVM operation (e.g., CREATE, CALL), transaction sender (i.e., the contract being executed), transaction receiver (i.e., the contract being created/called, or the account receiving Ether from a self-destructed contract), the amount of Ether being transferred, and the gas allowing the callee to spend. Since an external transaction can produce multiple internal transactions, DataEther associates an external transaction with its internal transactions because such correlation helps users better understand the consequence of executing an external transaction. Fig. 3 illustrates how DataEther identifies such correlation, where the EOA account (i.e., EOA1) sends an external transaction t1 to execute a smart contract SC1. The execution of SC1 produces an internal transaction it1, which calls the smart contract SC2. Then, SC2 invokes SC3 through another internal transaction it2. By conducting backtracking on the call chain, DataEther knows that both it1 and it2 are triggered by t1 whose sender is EOA1.

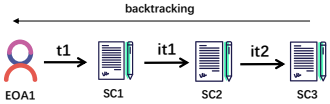

Fig. 3. Correlating an external transaction with its internal transactions.

E. Contract Explorer

Contract explorer acquires the information of smart contracts, which is used to determine token information (§III-F) and characterize non-deployable contracts (§VI). When a contract is being deployed, its deployment bytecode includes three parts, i.e., the initialization bytecode, runtime bytecode and the region for storing initialization parameters. Since an external transaction and an internal transaction deploy a contract in different ways, DataEther handles the two cases separately.

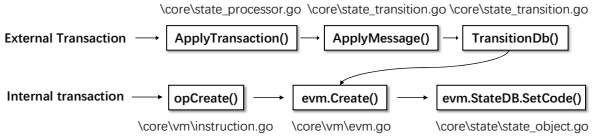

Fig. 4. The process of creating a contract

Contract deployed by an external transaction. If an external transaction deploys a contract, we can obtain the contract's deployment bytecode in the transaction's *input data* property. We propose a two-step approach to identify the initialization bytecode, runtime bytecode, and parameter region from the deployment bytecode. First, we locate and extract the runtime bytecode by inspecting five major functions in Geth as shown in Fig. 4, which depicts the process for an external or an internal transaction to deploy a smart contract. Specifically, `ApplyTransaction()` extracts a message (a data structure in Geth to be explained below) from the external transaction and then invokes `ApplyMessage()`. A message contains the necessary information for running the transaction, e.g., the address of the receiver, the data carried by the transaction, the Ether that should be transferred by the transaction. `ApplyMessage()` creates a new state transition and calls `TransitionDb()`, in which the function `evm.Create()` is invoked. Finally, `evm.Create()` executes the initialization bytecode and calls `evm.StateDB.SetCode()` to store the runtime bytecode in the storage. Therefore, we instrument `evm.Create()` to obtain the runtime bytecode. Second, DataEther extracts the initialization bytecode and parameter region by dividing the deployment bytecode into three parts according to the runtime code. For example, Fig. 5 shows the deployment bytecode of a real contract, Issuer [31], which has 1,069 bytes. DataEther first obtains its runtime bytecode, whose length is 823 bytes. Then, DataEther locates the beginning (offset 150) of the runtime bytecode in the deployment bytecode. After that, we get the initialization bytecode from offset 0 to offset 149 and learn that the parameter region starts from offset 973.

Initialization bytecode (150 bytes)

```
6060604052346100057604051606806103cd833981016040908152151602
0830151919092015155b5b60008054600160……909255600380548584169083 16
1790556002805492841692909116919091790555b5050505b6103378061009
6600039
```

Runtime bytecode (823 bytes)

```
6060604052361561006757631ffffffff60e060020a6000350416630b0f77438
11461006c578063867904b41461008b5780……16815600a165627a7a72305820
c2b40c8157f7dc3973e813f1d6522ffae1772f4bd4813c0ec756e2d97a17408
80029
```

Parameter region (96 bytes)

```
0000000000000000000000006efd5665ab4b345a7ebe63c679b651f375dddb7e
00000000000000000000000e75cd82ada6200356a5a879b31d87c3c5e6f70d0
00000000000000000000000b2e59493763d0d0be2634b2d1afe066914b0fcc2
```

Fig. 5. Deployment bytecode partitions

Contract deployed by an internal transaction. As shown in Fig. 4, the operation CREATE (its interpretation handler is `opCreate()`) is called to create a contract. After that,

`opCreate()` invokes `evm.Create()` and the remaining process is the same as the process for an external transaction to deploy a contract. Based on this observation, we insert recording code after the call to `evm.StateDB.SetCode()` to log the address of the deployed contract, which is the first parameter of `evm.StateDB.SetCode()`, and the runtime bytecode of the created contract, which is the second parameter of `evm.StateDB.SetCode()`. Since the deployment bytecode of a contract created by an internal transaction is stored in the memory, which is the fourth parameter of `opCreate()`, we acquire the deployment bytecode by instrumenting `opCreate()`. After that, we locate the initialization bytecode, runtime bytecode and parameter region in the deployment bytecode through the same process described above.

Contract invocation. Besides handling contract creation, contract explorer also collects the information of contract invocation, including the method id of the invoked method and parameters passed to the method. When a transaction calls a method of a smart contract, it should provide the corresponding method id [2]. When an external transaction calls a smart contract, DataEther gets the call data from its *input data* property, which is the first four bytes of *input data*. The parameters are the remaining bytes of *input data*. If an internal transaction invokes a contract, the interpretation handler `opCall()` or `opCallCode()` or `opDelegateCall()` or `opStaticCall()` will be called depending on which EVM operation triggers the internal transaction. Since the call data (including the method id and parameters) is stored in the memory, DataEther obtains it by reading the memory when the interpretation handler is being executed [22].

F. Token Explorer

Although many tokens have been created, distributed, and controlled by smart contracts running on Ethereum, little is known about token behaviors. DataEther takes the first step to examine such smart contracts by gathering token information, such as its name, symbol, and token transfer activities. Token contracts usually follow certain standards so that they can be discerned, transferred and traded by third-party applications (e.g., wallets, exchange markets) [32], but they can also define non-standard methods.

ERC20 tokens. ERC20 [24] is the most popular standard, which defines 6 standard methods (i.e., `totalSupply()`, `balanceOf()`, `transfer()`, `transferFrom()`, `approve()` and `allowance()`) and 2 standard events (i.e., Transfer and Approval). Based on this fact, DataEther determines whether a contract is an ERC20 token by checking whether it implements those standard methods and events. More precisely, since the method id of a method is derived from its prototype and the event is represented by the hash from its prototype, we first calculate the method ids and event hashes of those standard methods and events, respectively, and then check whether they exist in the runtime bytecode of a contract.

Tokens transfer. ERC20 defines two standard methods for token transfer (i.e., `transfer()` and `transferFrom()`), and therefore we can recognize them in the transaction according to their method ids. However, a token contract can also provide other methods for token transfer. For example, we find that Zilliqa [33] offers the function `burn()` and VeChain [34] provides the function `offerBonus()` to transfer tokens. Note that such non-standard methods cannot be recognized by third-party applications, because these applications do not know the prototypes of non-standard methods. To address this issue, we propose to monitor the Transfer event to quantify token transfer because ERC20 standard requires the emission of Transfer whenever token transfers [24]. Technically, DataEther captures events by analyzing execution traces because the EVM operations LOG0, LOG1, LOG2, LOG3 and LOG4 are used for recording events on the blockchain [2].

Token name and symbol. We find two ways to set token names and symbols: (1) hardcode in the initialization bytecode, e.g., Tronix [35] sets its name using the statement "string public name = "Tronix"" in its constructor; (2) providing them as the parameters to the initialization bytecode. For the first scenario, DataEther looks for PUSH32 operations whose operands are human-readable in the initialization bytecode of token contracts, because token contracts use PUSH32 to push data on the stack and the operand of PUSH32 stores the name and symbol. For the second scenario, DataEther searches the initialization parameters from the transaction to create contracts for human-readable characters.

G. Efficiency of DataEther

We compare the efficiency of DataEther and that of **C2** methods (i.e., invoking Web3 APIs provided by Ethereum), because 6 out of 9 kinds of data can be fully collected by **C2**, which is higher than the other existing methods as shown in Table I. More precisely, we record the time required by DataEther and the **C2** methods to collect the execution traces of 10,000 transactions that invoke smart contracts. All experiments are conducted on a server equipped with an Intel Xeon E5-2609 v3, 32GB main memory and 10TB HDD, and we repeat each experiment 30 times.

Since DataEther instruments an Ethereum full node and acquires data during synchronization, we measure the time consumption of the first two stages (i.e., data acquisition, data storage) of DataEther by launching it to acquire data (including blocks, traces, transactions, smart contracts, and tokens) until it records 10,000 execution traces. In our experiment, the average time consumption of DataEther is 100.4s.

We then measure the time consumption of collecting the same 10,000 execution traces by invoking the web3 API, `debug.traceTransaction()`. Note that before invoking the API we have to finish the synchronization of block files that cover the transactions triggering these 10,000 execution traces [36]. When measuring the time consumption of such API, we *exclude* the time used for synchronization and just count the time from sending the query to receiving the result. The experimental results show that such approach consumes 498.5s, which is 18.6x larger than that of DataEther. Fig. 6 shows the time consumption of DataEther and the API-based approach. A box at x-axis *m* shows the statistics of

time consumptions for collecting m traces in 30 repeated experiments. We can see that DataEther is much faster than `debug.traceTransaction()`, because this API needs to construct the exact state for executing the queried transaction [36] and uses RPC to communicate with the caller.

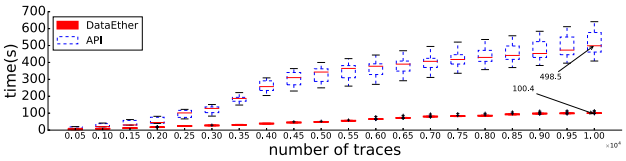

Fig. 6. The time consumption of DataEther and the API-based approach.

IV. Profiling Ethereum Entities

Analyzing the various data collected by DataEther, we can profile Ethereum entities from many aspects and examine the evolution of their features as DataEther correlates the data with timestamps. Due to page limit, we only report the study of account balance and tokens because to the best of our knowledge, they have not been examined by other works.

A. Account Balance

DataEther computes the historical balances of any account by considering all ways that can change balances. Eq.(1) shows how to compute the balance of an account a at block n:

$$B(a,n) = B(a,1) + \sum_{i=2}^{n} (A(a,i) + R(a,i) - S(a,i)), \quad (1)$$

where $B(a,1)$ is the initial balance of a set by the genesis block. $A(a,i)$ is the mining reward to a at block i, including block reward, uncle block reward and gas reward. $A(a,i) == 0$ if a neither mine the block nor the uncle block with block number i. $R(a,i)$ (or $S(a,i)$) denotes the amount of Ether received (or sent) by a according to the transactions in block i. DataEther collects $B(a,1)$ and $A(a,i)$ by its block explorer (§III-B), $R(a,i)$ and $S(a,i)$ by its transaction explorer (§III-D). The historical account balance allows us to analyze the changes in account balance over time. Such information can be correlated with other information or events to uncover new insights. As an example, we leverage the historical balances of Nanopool, a mining pool, to understand how it distributes the earned Ether to its miners.

Historical balances of Nanopool. Fig. 7 shows the balance of an account from Jan. 13, 2017 02:08:41 AM (T1) to Jun. 06, 2017 09:35:56 AM (T2). This account is the coinbase of the mining pool, Nanopool [37]. The x-axis gives the time points (in seconds) since T1. Assuming the account balance was m at T1, a point (x, y) in the black dash-dot line means that the balance became m+y at the time T1+x. This figure also shows the received mining reward and the amount of Ether sent by the account. A point (x, y) in the cyan solid line means that the accumulative mining reward to the account reaches y Ether from T1 to T1+x. A point (x, y) in the pink dashed line indicates that the accumulative amount of Ether sent from the account reaches y Ether from T1 to T1+x.

We observe that the balance did not change drastically. The reason may be that the Ether earned by the mining pool is distributed to the miners who contribute their computing

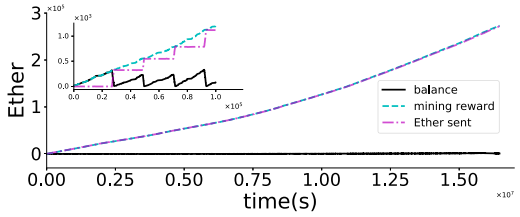

Fig. 7. The change of balance, mining reward and Ether sent over time

resources. Moreover, Fig. 7 shows that the mining reward and the amount of Ether sent increased at the same pace, because the Ether sent to participant miners comes from the mining reward. By zooming in on the first 10^5 seconds in the subfigure of Fig. 7, we observe that the mining reward increased steadily (mining 1 block about every 1.7 min on average), indicating that the mining pool with massive computing power can mine blocks steadily. This observation supports the viewpoint that *miners can get stable income by joining a mining pool* [38]. Interestingly, the amount of Ether sent increased periodically (every 5.5h on average) and the balance decreased correspondingly. This observation reveals the reward distribution strategy of Nanopool, which *pays the miners several times a day* [39] instead of distributing the block reward once a block is mined.

Remark: DataEther can acquire all historical balances of any accounts, which can be correlated with other information (e.g., Ether transfer) to reveal new insights. In particular, we observe that the mining pool Nanopool has stable mining reward and distributes the reward to its miners periodically.

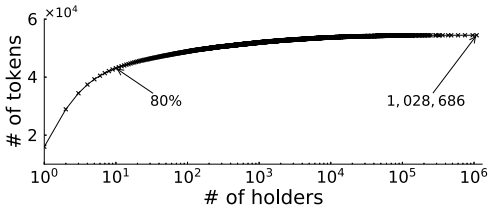

Fig. 8. Token holders of each token

B. Token

We profile token holders and token transfer behaviors to understand the investors and their behaviors. Fig. 8 shows the number of holders for all ERC20 tokens that implement all six standard methods and two standard events [24]. A point (x, y) means that there are y tokens, each of which is held by no more than x holders. DataEther considers an account as a token holder if it has ever participated in transferring (sending or receiving) certain token. We find that the token with the most holders (1,028,686) is the TRON token, which is ranked 9 of all tokens in terms of market capitalization [40]. Moreover, we find that about 80% (43,097/54,448) of tokens have no more than 10 holders. In other words, most tokens receive little attention.

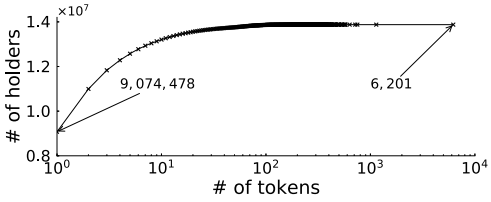

Fig. 9. The number of tokens held by token holders

A point (x, y) in Fig. 9 denotes that there are y holders, each of which holds no more than x different tokens. One observation is that most token holders (9,074,478) just invest one kind of tokens. One account holds 6,201 tokens. Manual investigation reveals that it is a smart contract named EtherDelta_2 belonging to an exchange market named EtherDelta [41]. EtherDelta_2 holds many kinds of tokens because it allows users to buy/sell tokens by invoking the contract.

Remark: DataEther can identify tokens and create the mapping between tokens and their holders. Moreover, it can recognize token transfer behaviors by interpreting the Transfer event from execution traces. We observe from the data collected by DataEther that most tokens receive little attention and most token holders just invest one kind of tokens.

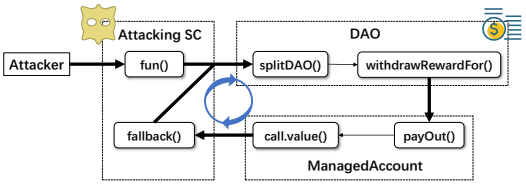

Fig. 10. A cycle of size 3 in the DAO attack

V. Transaction-Based Analysis

Analyzing the transactions can benefit various applications (e.g., attack forensics, anomaly detection, de-anonymization, etc.). We present an application of anomaly detection that concerns the *cycle* in contract invocation triggered by an external transaction. Fig. 10 shows an example where a thick edge denotes an inter-contract call and a thin edge stands for an intra-contract call. The contract *Attacking SC* calls the contract DAO which in turn calls the contract ManagedAccount. Then, ManagedAccount calls *Attacking SC*, and the smart contract invocation forms a cycle. We count the number of contracts in the cycle as the *size of a cycle*. For instance, the size of the cycle in Fig. 10 is three. We regard a cycle as a *suspicious* one if an external transaction triggers the cycle to be executed for more than H times ($H = 10$ in our experiments). Our anomaly detector identifies a suspicious cycle by first identifying a cycle and then counting the number of executions of that cycle. DataEther eases the development of our anomaly detector because it records the information of all internal and external transactions and correlates an external transaction with all its internal transactions (detailed in §III-D).

Our detector finds many suspicious cycles related to the DAO contract. This contract has a reentrancy vulnerability, and the attacker has exploited this vulnerability to withdraw a large amount of money by invoking it repeatedly [42]. To better understand this vulnerability and the corresponding attacks, we first examine the reported attacking transactions and the code of the DAO contract [43], and then investigate the collected transactions to answer *how many accounts exploited the DAO contract*. As shown in Fig. 10, in the reported attack transactions, `splitDAO()` is an attack surface in the DAO contract, which was called by the malicious contract (i.e., *Attacking SC*). Then, `splitDAO()` invokes `withdrawRewardFor()` in the same contract. After that, `withdrawRewardFor()` calls `payOut()` in the ManagedAccount contract, which invokes `call.value()` to transfer Ether to the attacker. `call.value()` will trigger the execution of the *fallback* method [2] in the attacking account, which invokes `splitDAO()` again.

TABLE II
Unreported attacking accounts and the number of attacking transactions involved in the DAO attack

Surface	Attacking account	#trans
splitDAO()	0xda6b0caeff67ebea3d0ea305b0ee4ce9bb7ab6c1	16
	0x0fe611f0ccf3b0da12e575a89072201c1832f413	2
	0x2399c2c86fbf159d81b7cb90014038f31080b4fc	752
	0xcc6415ad7e2403f1652233c364d6297729d0411a	99
	0x8172970a694509d9ba6379e1f2bad61c2437daf4	3
	0xee68fa275e27860779de0b02a18fa154ab601a78	13
getMyReward()	0xb27d4478815fcaf79054e6b561903db11501fa9b	25
	0x6007cf65baf6a2c2981650341300dccc441ba486	1
	0x971b8456de474e836049c0671c29d97fd25708f6	7
	0x900a979cfcc4a9e5f0dcac1f7cc629873e2528ec	65
	0x46562484f41893c8a4d3f7cb1af15c1b6eb2f91b	314
	0xd6266bfe94742c361683969e802c895ababdb4f9	19
	0xd92833b0990b599704f2b35e9bca2810bc09b1b6	1

We capture thousands of external transactions sent to the DAO contract, each of which triggers a cycle of size 3 and forces the cycle to be executed for more than 10 times. These transactions are sent by 22 attacking accounts, *13 of which were unreported* [42]. Besides, we reveal that 7 out of 13 unreported attacking accounts take advantages of an *unreported attack surface*, `getMyReward()` in the DAO contract. In particular, those 7 attacking accounts invoke `getMyReward()`, which also calls `withdrawRewardFor()`. That is, the interaction pattern of contracts involved in the attack that exploits the unreported attack surface is the same as the one shown in Fig. 10 except that `splitDAO()` is replaced with `getMyReward()`. Table II lists all unreported attacking accounts, their attack surfaces and the numbers of attacking transactions.

Remark: The transaction data obtained by DataEther can enable many applications. As an example, our anomaly detector discovers 22 accounts that attack the DAO contract, which include 13 unreported accounts. Moreover, our detector identifies an unreported attack surface of the DAO contract.

VI. Contract-Based Analysis

Inspecting smart contracts is an important step in many applications, such as vulnerability discovery [44], Ponzi schemes detection [12], and gas-inefficient code detection [45], [46]. This section presents a contract-based application: discovering and understanding abnormal smart contracts. In particular, by examining the initialization bytecode of smart contracts, we discover a special kind of smart contracts whose initialization bytecode ends with the EVM operation SELFDESTRUCT. That is, the contract will self-destruct immediately after initialization. We name such contracts as non-deployable contracts. Such contracts have a special feature: nobody (including the creator) can execute the contract again since no runtime bytecode is deployed on the blockchain.

We first design an automated approach to identify non-deployable contracts from the initialization bytecode of smart contracts collected by DataEther. To the best of our knowledge, no existing studies analyzed initialization bytecode. Given the initialization bytecode of a smart contract, our approach first constructs its control flow graph (CFG) based on

OYENTE [44] (a static analysis tool for EVM bytecode) and then traverses the CFG to check whether every path contains the SELFDESTRUCT operation. If so, a non-deployable contract is found. Our approach discovers 26,730 non-deployable contracts in our dataset. Fig. 11 shows the number of non-deployable contracts created within each hour from the first to the last discovered non-deployable contracts. We can see that 81% (21,612/26,730) non-deployable contracts were created from June 11 to August 1, 2018.

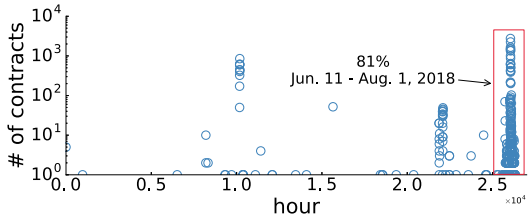

Fig. 11. Number of non-deployable contracts

To understand the potential usages of non-deployable smart contracts, we scrutinize all non-deployable contracts manually and identify three patterns. One is to execute the EVM operation CALL repeatedly for transferring Ether to multiple accounts. 584 (2%) contracts use this pattern. Different from Bitcoin whose transaction can have multiple outputs [47], an Ethereum transaction can send Ether to only one account without using contracts. Note that sending Ether to multiple accounts in one transaction costs less gas than using multiple transactions, each of which sends Ether to one receiver, because a transaction consumes at least 21,000 units of gas [2]. Hence, one potential usage of non-deployable contracts is to conduct multiple transfers using one transaction.

The second pattern is to transfer Ether via self-destruction. 4,534 (17%) non-deployable contracts adopt this pattern. A contract cannot receive Ether if none of its methods is marked as 'payable', except mining rewards and the Ether from self-destructed contracts [22]. Moreover, a contract cannot refuse the Ether from mining rewards and self-destruction [22]. We call such Ether transfer as *forcible transfer*. The forcible transfer can be used to send Ether to a buggy contract that misses the 'payable' keyword. Besides, forcible transfer can be exploited by attackers to subvert the contracts that check their balances before conducting sensitive tasks [48]. That is, an attacker can affect the comparison result through forcible transfer. Hence, another potential usage of non-deployable contracts is to forcibly transfer Ether to certain accounts.

The third pattern is to steal Ether. All 21,612 non-deployable contracts created from June 11, 2018 to August 1, 2018 use this pattern to steal Ether from the smart contract of a phenomenal game, Fomo3D [49]. This game collects Ether from all participants and sends the rewards to the last participant before the timer goes to zero. A recent news disclosed that the rewards were claimed by an attacker that blocked the blockchain to prevent other users from joining the game (i.e., becoming the last participant) [50]. Differently, we find an *unreported* attack mode after investigating the non-deployable contracts. More precisely, an account who sends Ether to Fomo3D has a probability to win other kinds of rewards (not the rewards claimed when the timer goes to zero [50]). Fomo3D records the accounts that have been rewarded, and it does not reward any account twice. Hence, the attacker creates many non-deployable contracts, and instructs each contract to send Ether to Fomo3D for claiming such rewards. Consequently, the rewards obtained by all non-deployable contracts are sent to the attacker. By using non-deployable contracts, attackers are more likely to hide themselves because no bytecode will be deployed on the blockchain. Besides, attackers can get the refund of execution fee due to the execution of SELFDESTRUCT [2]. After checking all 21,612 contracts and their transactions, we find that 280.9 Ether (worthy of about 0.14 million USD at that time) are stolen. For example, the non-deployable contract (address: 0xf3063e7cEcb382b7812EBaEff638bec1cf50Ed08) sends 0.1 Ether to Fomo3D (transaction hash: 0x01bd3cbfbe5f69646cfa0aff609637759433555eeedee160890c69d22312776f) and gets about 0.24 Ether back. Therefore, the attacker steals about 0.14 Ether by leveraging a single non-deployable contract.

Remark: Scrutinizing the bytecode of smart contracts can discover many interesting behaviors. As an example, we design an automated approach to detect non-deployable contracts from the data collected by DataEther and obtain an in-depth understanding of their potential usages, including (1) transferring Ether to multiple accounts by one external transaction; (2) forcible Ether transfer; and (3) stealing Ether through an unreported attack pattern that exploits non-deployable contracts.

VII. Trace-Based Analysis

Investigating the traces of a smart contract can help us learn its execution states. As an example, this section reports how we detect underpriced DoS attacks from the execution traces collected by DataEther. The gas cost of an EVM operation is expected to be proportional to the resources consumption of executing that operation in order to thwart resources abusing [2]. However, the gas costs of some EVM operations are lower than they should be. We call such operations *underpriced* operations. An attacker can launch underpriced DoS attack by executing underpriced operations repeatedly. Such attacks can waste the computing resources of Ethereum nodes and hence deny the service of blockchain. Our previous work proposes an adaptive gas mechanism to defend against underpriced DoS attack [10]. Differently, in this paper, we propose a detector to discover *the EVM operations that have been exploited by underpriced DoS attacks*.

We propose a three-step approach to detect underpriced DoS attack by looking for the EVM operations with abnormally high execution frequencies. First, we compute the frequency of each EVM operation in every execution trace. Second, we construct a frequency sequence for every analyzed operation. This sequence lists the execution frequencies of an operation in all execution traces in chronological order. Third, we apply the Tukey's range test [51] to the frequency sequences for locating outliers. Tukey's range test is a well-known method to find outliers, which are values that are significantly different from

TABLE III
RESULTS OF UNDERPRICED DOS ATTACK DETECTION

Underpriced OP	#trans	#OP per trans
EXTCODESIZE	11,246	18,370
SLOAD	1	26,459
BALANCE	8,190	960
DELEGATECALL	19,122	8,319

other values [51]. An outlier in this application refers to an EVM operation that has been executed much more frequently in one trace than in other traces. We do not set a fixed threshold to the execution frequencies of EVM operations, because some operations will be executed much more times than other operations in normal situations.

Table III shows the detection results. Column 1 lists the four underpriced EVM operations that have been exploited by *real* DoS attacks. Columns 2, 3 present the numbers of external transactions for attacking purpose and the average numbers of the underpriced operations executed per attacking transaction. It has been reported that EXECODESIZE was used to launch DoS attacks [52]. However, the attacks that exploit SLOAD, BALANCE, or DELEGATECALL have *not* been reported before, and these attacks sent 27,313 (i.e., $1 + 8,190 + 19,122$) attacking transactions. We further explore why these operations were exploited to launch DoS attacks. Since SLOAD loads a value from the storage and BALANCE gets an account balance stored in the storage [2], their operations need disk I/O. DELEGATECALL produces an internal transaction which invokes another smart contract [2]. Since smart contracts are stored in the storage, the execution of DELEGATECALL consumes considerable CPU and disk resources. As these operations were underpriced, attackers repeatedly execute them to cause extensive disk I/O and high CPU usage in every Ethereum full node. Although we cannot launch DoS attacks against the Ethereum blockchain by exploiting these three operations, we believe they are *not* false positives because the gas costs of SLOAD, BALANCE, and DELEGATECALL have been increased from 50, 20 and 40 to 200, 400 and 700 in the updated Ethereum, respectively [53].

Remark: The execution traces enable analyzing smart contracts and detecting attacks. Based on the execution traces collected by DataEther, we discover 27,313 unreported attacking transactions that exploited 3 underpriced operations.

VIII. RELATED WORK

There are four typical ways to acquire data from Ethereum.

C1: downloading & parsing block files. Kiffer et al. studied the hard fork of Ethereum due to the divergences in the solutions for the DAO attack by parsing block files [8]. MAIAN discovered three kinds of vulnerabilities in smart contracts that are collected by the downloading & parsing approach [14].

C2: invoking Web3 APIs. Our previous work, GASPER [45] detected several gas-costly code patterns from smart contracts that were collected by invoking `web3.eth.getCode()`. Our previous work collected execution traces by calling `debug.traceTransaction()` for calculating the execution frequencies of EVM operations [10]. Bartoletti et al. designed a general-purpose framework to support data analytics on Bitcoin and Ethereum, where data is collected by invoking the APIs provided by Bitcoin and Ethereum, respectively [9]. EtherQL [54] collected blockchain data through the APIs exported by EthereumJ (a Java implementation of Ethereum) [55], and maintained the data in MongoDB [56].

C3: crawling blockchain explorer websites. ZEUS verified the safety of 22.4 thousand smart contracts via abstract interpretation and symbolic model checking [11]. Two studies detected and dissected the contracts that implement Ponzi schemes [12], [57]. They all collect contract by crawling blockchain explorers, including Etherscan, Etherchain and EtherCamp. Some studies combined the downloading & parsing approach and the web crawling approach. Huang et al. estimated the potential profitability of mining and speculating tokens using real-world blockchain and trade data [58]. More precisely, they obtained the data from Bitcoin and Litecoin by parsing their block files, and acquired the data from the other blockchains by web crawling. Bartoletti et al. conducted an empirical analysis of smart contracts with two data sets. One is crawled from Ethereum explorer websites and the other is extracted from the block files of Bitcoin [59].

C4: instrumenting Ethereum node. Our previous work obtained the senders, receivers, and the amount of Ether transferred from transactions, and then conducted graph analysis [3]. In comparison, DataEther acquires much more information of various types. Even for an internal transaction, DataEther collects new information, such as its gas limitation, the EVM operation (e.g., CALLCODE, DELEGATECALL) triggering it, the actual gas consumption, and its input data. Moreover, our previous work did not handle STATICCALL because it is a new operation introduced first in Geth V 1.7.0 [3]. The consequence is that the internal transactions triggered by STATICCALL will be missed. Grossman et al. obtained the information of internal transactions and storage operations to effectively detect callback free objects in smart contracts [13]. Our previous tool, GasReducer [46] collected data by both APIs and instrumentation for contract optimization. In particular, it obtained contracts by calling `web3.eth.getCode()`, and recorded every executed operation and its gas consumption from an instrumented node.

IX. CONCLUSION

To address the limitations of existing methods for exploring Ethereum, we propose and develop DataEther, a systematic and high-fidelity data exploration framework which exploits Ethereum's internal mechanisms and carefully instruments an Ethereum full node. DataEther acquires all historical data and collects ERC20 token activities, thus enabling many new applications. We demonstrate its usefulness through four applications, including profiling account balance and tokens, characterizing reentrancy attacks, revealing non-deployable contracts, and detecting underpriced DoS attacks, which have yielded many new insights. DataEther and the collected data will be released after the paper is published.

REFERENCES

[1] coinmarketcap. (2018) Top 100 cryptocurrencies by market capitalization. [Online]. Available: https://coinmarketcap.com/

[2] G. Wood. (2018) Ethereum: a secure decentralised generalised transaction ledger – byzantium version. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[3] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," in *IEEE International Conference on Computer Communications*, 2018.

[4] eidoo. (2018) Erc20 tokens list. [Online]. Available: https://eidoo.io/erc20-tokens-list

[5] Applancer. (2017) The ethereum killer app cryptokitties sales reach $12 million. [Online]. Available: https://www.reddit.com/r/CryptoKitties/comments/7j1o3t/the_ethereum_killer_app_cryptokitties_sales_reach/

[6] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *International Conference on Principles of Security and Trust*, 2017.

[7] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan. (2017) Blocksci: Design and applications of a blockchain analysis platform. [Online]. Available: https://arxiv.org/abs/1709.02489

[8] L. Kiffer, D. Levin, and A. Mislove, "Stick a fork in it: Analyzing the ethereum network partition," in *ACM Workshop on Hot Topics in Networks*, 2017.

[9] M. Bartoletti, S. Lande, L. Pompianu, and A. Bracciali, "A general framework for blockchain analytics," in *Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, 2017.

[10] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks," in *International Conference on Information Security Practice and Experience*, 2017.

[11] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *The Network and Distributed System Security Symposium*, 2018.

[12] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia. (2017) Dissecting ponzi schemes on ethereum: identification, analysis, and impact. [Online]. Available: https://arxiv.org/abs/1703.03779

[13] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Detection of effectively callback free objects with applications to smart contracts," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2018.

[14] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. (2017) Finding the greedy, prodigal, and suicidal contracts at scale. [Online]. Available: https://arxiv.org/abs/1802.06038

[15] M. Tan. (2018) Etherscan – the ethereum block explorer. [Online]. Available: https://etherscan.io/

[16] Etherscan. (2018) Ethereum developer apis. [Online]. Available: https://etherscan.io/apis

[17] ——. (2018) Ethereum shows 31 internal transactions of 0x8023679696f1db7fe00adfdab4fcd8600a7cd6e4c5d8a8054df0d97cd06503de. [Online]. Available: https://etherscan.io/tx/0x8023679696f1db7fe00adfdab4fcd8600a7cd6e4c5d8a8054df0d97cd06503de#internal

[18] M. Fröwis and R. Böhme, "In code we trust? measuring the control flow immutability of all smart contracts deployed on ethereum," in *International Workshops on Data Privacy Management, Cryptocurrencies and Blockchain Technology*, 2017.

[19] NEO. (2019) Neo - an open network for smart economy. [Online]. Available: https://neo.org/

[20] TRON. (2019) Tron - decentralize the web. [Online]. Available: https://tron.network/index?lng=en

[21] Qtum. (2019) Qtum - defining the blockchain economy. [Online]. Available: https://qtum.org/en

[22] (2018) Solidity. [Online]. Available: https://solidity.readthedocs.io

[23] Ethereum. (2018) Ethereum homestead documentation. [Online]. Available: http://www.ethdocs.org/en/latest/

[24] Wikipedia. (2018) Erc20 token standard. [Online]. Available: https://theethereum.wiki/w/index.php/ERC20_Token_Standard

[25] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*, 2015.

[26] Wiki. (2018) Proof of work. [Online]. Available: https://en.bitcoin.it/wiki/Proof_of_work

[27] StackExchange. (2018) What is geth's "light" sync, and why is it so fast? [Online]. Available: https://ethereum.stackexchange.com/questions/11297/what-is-geths-light-sync-and-why-is-it-so-fast

[28] Elastic. (2018) Elasticsearch. [Online]. Available: https://www.elastic.co/

[29] StackExchange. (2016) Difference between call, callcode and delegatecall. [Online]. Available: https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall

[30] V. Buterin and C. Reitwiessner. (2017) Eip 214: New opcode staticcall. [Online]. Available: https://eips.ethereum.org/EIPS/eip-214

[31] Etherscan. (2018) The contract issuer, 0x28982952e9dc211ec9cd7e23df67e907c5a88740 is created by the transaction 0xc2d8af2bd53108e8899d961b4354d93c27beb2f021846da08cae4d018f9da046. [Online]. Available: https://etherscan.io/tx/0xc2d8af2bd53108e8899d961b4354d93c27beb2f021846da08cae4d018f9da046

[32] ChronoBank.io. (2018) Ethereum token standards. [Online]. Available: https://blog.chronobank.io/ethereum-token-standards-19fbcc54fe27

[33] Etherscan. (2018) The address of zilliqa is 0x05f4a42e251f2d52b8ed15E9FEdAacFcEF1FAD27. [Online]. Available: https://etherscan.io/address/0x05f4a42e251f2d52b8ed15e9fedaacfcef1fad27

[34] ——. (2018) The address of vechain is 0xD850942eF8811f2A866692A623011bDE52a462C1. [Online]. Available: https://etherscan.io/address/0xD850942eF8811f2A866692A623011bDE52a462C1

[35] ——. (2018) The address of tronix is 0xf230b790E05390FC8295F4d3F60332c93BEd42e2. [Online]. Available: https://etherscan.io/address/0xf230b790E05390FC8295F4d3F60332c93BEd42e2

[36] P. Szilágyi. (2017) Tracing: Introduction. [Online]. Available: https://github.com/ethereum/go-ethereum/wiki/Tracing:-Introduction

[37] Nanopool.org. (2018) Nanopool. [Online]. Available: https://nanopool.org/

[38] bitcoinwiki. (2017) Pool vs. solo mining. [Online]. Available: https://en.bitcoin.it/wiki/Pool_vs._solo_mining

[39] Nanopool.org. (2018) General questions. [Online]. Available: https://eth.nanopool.org/faq

[40] CoinMarketCap. (2018) Tron (trx) price, charts, market cap, and other metrics. [Online]. Available: https://coinmarketcap.com/currencies/tron/

[41] EtherDelta. (2018) Etherdelta. [Online]. Available: https://etherdelta.com

[42] StackExchange. (2016) How many the dao recursive call vulnerability attacks have occurred to date? [Online]. Available: https://ethereum.stackexchange.com/questions/6320/how-many-the-dao-recursive-call-vulnerability-attacks-have-occurred-to-date

[43] Etherscan. (2018) The source code the dao is available. [Online]. Available: https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

[44] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[45] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[46] T. Chen, Z. Li, H. Zhou, J. Chen, X. Li, X. Luo, and X. Zhang, "Towards saving money in using smart contracts," in *International Conference on Software Engineering*, 2018.

[47] Xavhorn. (2014) How to create a raw transaction using multiple outputs in bitcoin qt. [Online]. Available: https://www.reddit.com/r/Bitcoin/comments/2ee3kr/how_to_create_a_raw_transaction_using_multiple/

[48] SmartCheck. (2018) Checking for strict balance equality. [Online]. Available: https://tool.smartdec.net/knowledge/SOLIDITY_BALANCE_EQUALITY

[49] Fomo3D. (2018) Fomo3d. [Online]. Available: https://exitscam.me/shakedown

[50] K. Sedgwick. (2018) Someone wins $3 million jackpot in ethereum ponzi fomo3d. [Online]. Available: https://news.bitcoin.com/someone-wins-3-million-jackpot-in-ethereum-ponzi-fomo3d/

[51] Wikipedia. (2017) Tukey's range test. [Online]. Available: https://en.wikipedia.org/wiki/Tukey%27s_range_test

[52] J. Wilcke. (2016) The ethereum network is currently undergoing a dos attack. [Online]. Available: https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/

[53] J. Ray. (2018) Design rationale. [Online]. Available: https://github.com/ethereum/wiki/wiki/Design-Rationale

[54] Y. Li, K. Zheng, Y. Yan, Q. Liu, and X. Zhou, "Etherql: A query layer for blockchain system," in *International Conference on Database Systems for Advanced Applications*, 2017.

[55] ethereum.org. (2018) Java implementation of the ethereum yellowpaper. [Online]. Available: https://github.com/ethereum/ethereumj
[56] mongoDB. (2018) Mongodb atlas database as a service. [Online]. Available: https://www.mongodb.com/
[57] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting ponzi schemes on ethereum: Towards healthier blockchain technology," in *The International World Wide Web Conference*, 2018.
[58] D. Y. Huang, K. Levchenko, and A. C. Snoeren, "Short paper: Estimating profitability of alternative cryptocurrencies," in *International Conference on Financial Cryptography and Data Security*, 2017.
[59] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *Workshop on Trusted Smart Contracts*, 2017.