Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

Studying gas exceptions in blockchain-based cloud applications

Chao Liu[1*], Jianbo Gao[1], Yue Li[1], Huihui Wang[2] and Zhong Chen[1]

Abstract

Blockchain-based cloud application (BCP) is an emerging cloud application architecture. By moving trust-critical functions onto blockchain, BCP offers unprecedented function transparency and data integrity. Ethereum is by far the most popular blockchain platform chosen for BCP. In Ethereum, special programs named smart contracts are often used to implement key components for BCP. By design, users can send transactions to smart contracts, which will automatically lead to code execution and state modification. However, unlike regular programs, smart contracts are restricted in execution by gas limit, i.e., a form of runtime resource. If a transaction uses up all available gas, an out of gas exception (OG) will trigger, reverting state until right before that transaction.

In this work, we study the out of gas exceptions (or gas exceptions in short) on Ethereum empirically for the very first time. In particular, we collect exception transactions using an instrumented Ethereum client. By investigation, we found gas exceptions stand out in terms of both occurrences and losses. Moreover, we focused on individual contracts and transactions, aiming at discovering and identifying common causing factors triggering these exceptions. At last, we also investigate existing tools in preventing gas exceptions. Our results suggest further research and study in this direction.

Keywords: Cloud application, Blockchain, Ethereum, Runtime exception, Out of gas, Empirical study

Introduction

In the last few years, the cloud computing community has seen a number of emerging techniques and application categories. For example, edge computing is a new proposed cloud computing paradigm aiming at meeting the increasing computation demand for low latency and fast response, with abroad applications in IoT (internet-of-things) [1], smart cities [2], autonomous cars [3], and etc.

Besides, another trend of cloud computing has also gained vast attention both within and beyond the community, i.e., the ongoing movement towards building a fully decentralized infrastructure in the name of blockchain. In fact, the interest for blockchain technology has seen a steep growth in the last five years, with a number of investigations both in academia [4–10] as well as in industry [11–14].

One particular use case of the movement towards decentralization is the invention of decentralized application (dApp in short), a combination of both traditional cloud application and new blockchain-enabled application (i.e., the smart contract). We give these systems a more straightforward name, i.e., blockchain-based cloud application (BCP for short). Since these names both refer to the similar concept, throughout this work, we use the name decentralized application and blockchain-based cloud application interchangeably. In addition, we will stick to the most popular blockchain platform for BCP in this work, i.e., the Ethereum blockchain.

In general, blockchain is a new paradigm for cloud computing with a unique emphasises on trust-free and decentralization. More specifically, a blockchain can be seen as a replicated append-only log shared by all network peers using the "chain of blocks" (thus given the

*Correspondence: liuchao_cs@pku.edu.cn
[1]School of Electronics Engineering and Computer Science, Peking University, No.5 Yiheyuan Road, 100871, Beijing, China
Full list of author information is available at the end of the article

name blockchain) data structure, where every block consists of an ordered list of user transactions. Through an ingenious combination of distributed consensus, cryptographic primitives, peer-to-peer gossip protocol, plus economic incentives, the blockchain has proven to be a transparent, tamper-free, yet decentralized way of information sharing. For example, the Bitcoin blockchain is implemented as a public available shared ledger on network peers (*aka* nodes, clients, miners), effectively facilitating a digital payment system without trusted third-party (TTP) binding. Some other similar blockchain-based payment systems are proposed, like BitcoinCash [15], Litecoin [16], Zcash [17], and Libra [18].

In the view of cloud computing, a blockchain is not only a persistent place for data storage, but also provides the ability of computation. The crux of such capability lies in the concept of programmable transaction scripts, *aka*, smart contracts. By design, first generation blockchain like Bitcoin implements a simple transaction script (i.e. smart contracts) to achieve flexible processing logic [19]. While this enables non-trivial transaction settlement logic (some examples as escrow service, micropayment channel, and private transaction), the omission of Turing-complete capability as well as UTXO (unspent transaction output) account model has limited its application in areas outside of payment settlement [20].

The Ethereum [14] blockchain was thus proposed to address the issue, and later grew as a canonical design for public decentralized cloud computing platforms. Unlike Bitcoin, Ethereum adopts a more straightforward approach to holding transacting entities, i.e., modelling each transactor an independent account. More specifically, there are two types of accounts in Ethereum, i.e., externally owned account (or EOA) [1] and smart contract, both accounts are identified and can be retrieved by their unique identites called *addresses* (a 160-bit length integer identifier). In Ethereum, every account resides directly on the blockchain and has its own state persisted by so-called state database [14]. An account's state consists of four fields: 1) `nonce` used to prevent replay attack; 2) `balance` standing for account's holding of Ether (or `ETH`), Ethereum's native cryptocurrency; 3) `storageRoot` representing account-owned storage data (structured as a Merkle tree); and 4) `codeHash` referring to self-governance code. Here, the last two fields (i.e., `storageRoot` and `codeHash`) are key for smart contracts, which set them apart from ordinary accounts.

Smart contracts, in essence, are special programs running on the blockchain. When receiving transactions from other accounts, contracts are automatically loaded and then executed according to their predefined logic (as specified by `codeHash`). In Fig.1, we show a simple contract named `EtherBank` in Solidity. Solidity is the statically typed object-oriented high-level programming language dedicated to smart contract programming, and the most popular and widely used such languages in Ethereum. Solidity supports a rich group of features, such as native big integer (`uint256/int256`) type, dynamic array, user-defined `struct`, multiple inheritance, and important blockchain primitives (e.g., `msg.sender`, `block.number`).

As can be seen in Fig.1, the `EtherBank` contract defines a *storage variable* named `balances` (line 4), which is persisted on blockchain within consecutive transactions. This variable represents a balance record of relevant accounts, modelled as a mapping from account address to its corresponding Ether savings. Besides, there are two publicly available *functions* in `EtherBank`, i.e., `deposit()` (line 6) and `withdraw(uint256 amount)` (line 11). These functions define the processing logic for corresponding requests, and will thus get executed if called by other accounts, respectively. For instance, the `withdraw(uint256 amount)` function states, if called, it first ensures there are no potential integer overflows (line 12), then updates account's balance (line 13), and at last begins to transfer requested Ether accordingly (line 14). To facilitate this kind of contract execution, Ethereum implements a Turing-complete stack-based virtual machine called Ethereum Virtual Machine, or EVM in abbreviation (see Section 2). And every smart contract (just like `EtherBank`) has to be compiled into a corresponding EVM bytecode (i.e., a sequence of instructions) before ever deployed and executed in Ethereum.

In Ethereum, smart contracts are always compiled into bytecode, and then deployed and executed in EVM (Ethereum Virtual Machine) along with transaction processing mechanism. In particular, every instruction thus executed will be charged a fee to compensate for resources spent, as well as to prevent potential DoS (denial of service) attacks. This fee is always pre-paid by transaction sender (i.e., `tx.sender`[2]) on a transaction basis, and are further be factorized into two related parameters in concept of gas, i.e., `tx.gasLimit` and `tx.gasPrice` . Here, `tx.gasLimit` specifies maximal amount of gas available to the transaction, whereas `tx.gasPrice` converts gas units into `ETH` value (the exact fee paid by transaction sender). In principal, every transaction sender is required to specify both `tx.gasLimit` and `tx.gasPrice` , and will have to pay the amount of Ether

[1]In Ethereum, EOAs are accounts held by external users (or software on behalf of external users) with the corresponding public/private key pairs, whereas smart contracts are self-governed accounts with their own governance logic embedded in code (as referenced by `codeHash`). In principle, an EOA can send transactions directly to other accounts (both EOAs and smart contracts).

[2]By design [14], the raw data structure of Ethereum transaction does not have a `sender` field. Instead, the transaction sender is derived from a valid signature accompany with this transaction, i.e., `tx.v`, `tx.r`, and `tx.s`.

```
pragma solidity >=0.4.22 <0.7.0;
contract EtherBank {
    mapping (address => uint256) public balances;
    function deposit() external payable {
        require(balances[msg.sender] + msg.value >= balances[msg.
            sender]);
        balances[msg.sender] += msg.value;
    }
    function withdraw(uint256 amount) external {
        require(amount <= balances[msg.sender]);
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```

Fig. 1 `EtherBank`: an example smart contract written in Solidity

(i.e., `tx.gasLimit` · `tx.gasPrice`) before execution. If, after transaction execution, there are unused gas left, the remaining part will be refunded back to transaction sender in `ETH` in the same rate as `tx.gasPrice`.

During execution, there are cases where a transaction ends up using all available gas limit, e.g., it runs into an infinite loop. When that happens, EVM will force that transaction to an immediate stop, reverting all intermediate states modified until right before the transaction. In this case, we say the transaction has encountered an *out of gas (OG) exception* [14], or in short, gas exception.

Out of gas exceptions are problematic or even vulnerable in at least three aspects, i.e., 1) *money loss*; 2) *resource waste*; 3) *potential vulnerabilities*. First of all, gas exception causes money loss for the transaction sender. As of August 30th, 2019, typical market value for this kind of loss spans from several cents towards several tens of cents US dollars per transaction. Besides, these exceptions also mean a kind of resource waste for the entire system as a whole. Instead of choosing and processing transactions that are doomed to fail, miners could have spent scarce resources on other normal transactions, which are more “meaningful” for the network. Last but not least, previous literature [21] has already revealed a direct link between out of gas exception and severe contract vulnerability, which putting billions of US dollars under threat according to the study.

We claim the importance of studying out of gas exception in the scope of blockchain-based cloud application. While there are previous works concerning the gas mechanism of Ethereum [21–25, 32, 44], none of them are either complete or explicitly towards out of gas exceptions in a general form. Besides, there also lacks a comprehensive and empirical treatment on these exceptions, or any other types. In this work, we present a *first systematic and empirical analysis* on out of gas exceptions. In particular, we aim to answer the following research questions (RQs):

- ***RQ1*** How do out of gas exceptions exist in Ethereum? To what extent does it affect external users, network peers, as well as the blockchain as a whole?
- ***RQ2*** What are the main factors or reasons for out of gas exceptions? Are there lessons developers, researchers, and users can learn from?
- ***RQ3*** How effectively do existing tools or methods can help in preventing out of gas exceptions? What are the limitations?

In summary, the main contributions of our work are:

- We give a comprehensive taxonomy of EVM runtime exceptions, and find that the two most commonly seen exception types are out of gas and explicit revert, which combinedly account for around 95% of all exception instances, *w.r.t.* both external transactions as well as internal message calls.
- To the best of our knowledge, we are the first to conduct large scale empirical analysis on out of gas exceptions in blockchain-based cloud applications on Ethereum. Our study shows that this kind of exceptions is very prevalence in the world of smart contracts, and has already caused significant amount of losses.
- We have investigated the reasons behind out of gas exceptions. More specifically, we identify four

possible factors, i.e., misunderstanding of transaction mechanism, conservative gas limit, compiler derived bug, and unbounded mass operation.

- We have studied existing tools and methods in use of preventing out of gas exceptions. The result suggests room for further research and investigations.

Background

Blockchain-based cloud application

The unparalleled characteristics of blockchain, i.e., transparency, no third-party trust dependency, and performance assurance, have motivated widespread interest and exploration of new application opportunities. Among them are a new kind of application called decentralized application (dApp), or more specifically blockchain-based cloud application (BCP). Based on the decentralized blockchain platform, these applications promise to give birth to a new spectrum of emerging use cases, such as open source crypto-collectible games (e.g., CryptoKitties) and a bunch of decentralized finance (DeFi) applications [26].

In Fig. 2, we show a typical blockchain-based cloud application architecture. Like traditional cloud application, a BCP also contains components as `Frontend`, `Middleware`, and `Backend`. Besides, there may be a separate storage service, or `Database`, providing necessary persistent storage functionalities. What makes blockchain-based cloud applications unique is the additional capability to communicate with `Blockchain`. Usually, this capability of interacting with blockchain is provided by the `Blockchain Endpoint`, which might be a dedicated blockchain client (e.g., `Geth` and `Parity` for Ethereum blockchain), or through cloudalized blockchain endpoint service (e.g., Infura for Ethereum).

The integration of blockchain to cloud application (*aka* blockchain-based cloud application) promises to give some positive influence on both the provision and analysis of traditionally cloud applications. In general, the introduction of blockchain can provide a new universal platform for processing, preserving, and testifying of traditionally hidden application logic, in a fully transparent, decentralized, and trusted way. For example, users of current third-party payment system, such as Alipay, WeChat, and Paypal, may not have access to the critical transfer functionality dealing with their money, all they can do is to fully trust these big companies for performing faithfully, errorless, and timely. By placing critical business logic onto the blockchain, however, users are now able to touch the underline infrastructure and processing logic behind

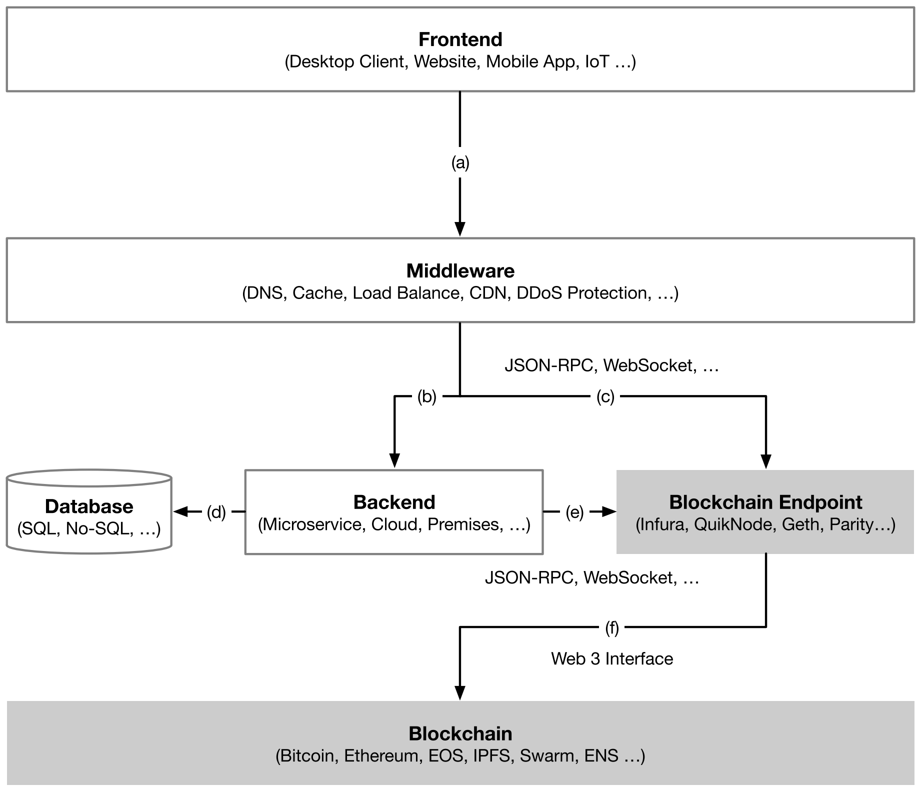

Frontend
(Desktop Client, Website, Mobile App, IoT ...)
(a)
Middleware
(DNS, Cache, Load Balance, CDN, DDoS Protection, ...)
JSON-RPC, WebSocket, ...
(b)
(c)
Database
(SQL, No-SQL, ...)
(d)
Backend
(Microservice, Cloud, Premises, ...)
(e)
Blockchain Endpoint
(Infura, QuikNode, Geth, Parity...)
JSON-RPC, WebSocket, ...
(f)
Web 3 Interface
Blockchain
(Bitcoin, Ethereum, EOS, IPFS, Swarm, ENS ...)

Fig. 2 A typical architecture of a blockchain-based cloud application

their everyday activities, such as components operating on their own money and private data.

Typically, there are two variants of blockchain-based cloud applications, i.e., 1) pure blockchain-based cloud applications (PBCP); and 2) hybrid blockchain-based cloud applications (HBCP).

For pure blockchain-based cloud applications, we require that every component of the application should be built on top of some decentralized platform or service (see process (a), (c), and (f) in Fig. 2), so that both users and developers see the same code base and are confident of the performance logic their application behaves (since both code and data of the application are fully decentralized and transparent). For example, a developer may decide to build an application by using Ethereum as processing engine, IPFS as storage service, Ethereum Name Service (ENS) as name service for both domain names and blockchain accounts, and so on. While appealing for its simplicity and transparency, pure blockchain-based cloud applications are hard to build and maintain using current techniques for problems as scalability, privacy, inter-blockchain communication, and etc.

Instead, for hybrid blockchain-based cloud applications, developers can choose to implement trust-critical functionalities on the blockchain, whereas leave other components as in traditional cloud applications (see process (a) to (f) in Fig. 2), e.g., using microservices. In this way, developers can trade transparency and integrity guarantees with performance and privacy at the minimal level of functions, in the hope to overcome drawbacks of current blockchain platforms while still adding enough trust and transparency to their applications. In fact, most blockchain-based cloud applications seen now adopt the hybrid architecture, and we think the trend to a pure blockchain-based cloud application will take a long time before decentralized platforms and services become mature and easily accessible.

Ethereum, smart contract, and EVM

Ethereum, in its essence, is a decentralized transaction-driven deterministic state machine. In particular, it resides in the *public blockchain* category, where every block as well as transaction is publicly available, and users are free to send transactions to drive a state change. In this respect, it can be seen as an instantiation of Lamport's *state machine replication* [27] approach, where at the core of this state machine transition analogy is the so-called *Ethereum Virtual Machine*, or EVM.

In Ethereum, all miners (i.e., clients, or network nodes) join in the same peer-to-peer network, combinedly maintaining a single view of the so-called world state, where the world state can be seen as an enumeration of accounts, which are further divided into EOAs (externally owned users) and smart contracts. By design, an EOA can send transactions to other accounts. These transactions may specify amount of `ETH` to transfer as well as optional input data. If the transaction target (denoted by `tx.to`) is another EOA, nothing special will happen (other than `ETH` transfer). However, if `tx.to` points to an existing smart contract, Ethereum will load contract's code as well as transaction input data, and send them to EVM (Ethereum Virtual Machine) for further execution. As long as no exception occurs during execution, the result will be persisted and synchronized across the whole network. Besides interacting with an existing smart contract, users can also deploy new contracts by leaving `tx.to` to empty, and filling in the transaction input (i.e., `tx.input`) with appropriately encoded init code [28].

The EVM is a simple stack-based machine with access to a runtime *stack* and a random access *memory*. It also equips with a non-volatile *storage*, which is persisted by state database on the blockchain. To facilitate use of `Keccak256` hash function, EVM adopts a large word size of 256 bits. While the stack and storage are accessible in slots of words, the memory is byte-addressable, so as to be read and written at any preferable byte position. In default, all newly accessed memory and storage locations in EVM are initialized to zero value.

Recall from "Introduction" section, every contract must be compiled into bytecode before executed in transaction. In this regard, EVM offers an instruction set of size 141 until St. Petersburg hardfork[3] in Februray 28th, 2019. These instructions can be categorized into seven groups: (1) *arithmetic and logic operation*, e.g., `ADD`, `EXP`; (2) *cryptographic primitive*, e.g., `SHA3`; (3) *blockchain and environmental information*, e.g., `BALANCE`, `EXTCODESIZE`; (4) *storage manipulation*, e.g., `MLOAD`, `SSTORE`; (5) *control flow*, e.g., `JUMP`, `JUMPI`; (6) *logging*, e.g., `LOG0`, `LOG4`; and (7) *system operation*, e.g., `CALL`, `SELFDESTRUCT`.

In Table 1, we show a selected list of EVM instructions, covering all seven categories, along with their gas cost (as of St. Petersburg hardfork) [14]. Note, the value in Table 1 does not include a *memory expansion cost*, which accounts for the additional memory footprint used by the instruction (see "The gas mechanism of Ethereum"section).

During transaction execution, contracts can interact with each other by calling respective public functions. Since EVM is designed as a single-threaded machine, this kind of internal message call will immediately trigger a new execution frame, and change context to it for further execution. After the call returns (whether normally or exceptionally), execution will resume to where it left before and continue thereafter. In the bytecode level, this internal call is realized by a set of `CALL` instructions, i.e., `CALL`, `CALLCODE`, `DELEGATECALL`, and `STATICCALL`.

[3]In blockchain terminology, a *hardfork* is a backward incompatible protocol update.

Table 1 A selected list of EVM instructions, along with their gas cost (not including memory expansion cost), as of St. Petersburg hardfork, February 28th, 2019

Instruction	Gas Cost
1) *arithmetic and logic operation*	
ADD/SUB	3
MUL/DIV/MOD	5
EXP	$\begin{cases} 10 & \textit{if exponent is } 0; \\ 10 + 50 \times (1 + \lfloor \log_{256}(\texttt{EXP}_{exponent}) \rfloor) & \textit{otherwise.} \end{cases}$
AND/NOT/OR/XOR	3
LT/GT/EQ/ISZERO	3
2) *cryptographic primitive*	
SHA3	$30 + 6 \times \lceil \texttt{SHA3}_{datasize} \div 32 \rceil$
3) *blockchain and environmental information*	
ADDRESS/CALLER/TIMESTAMP/PC	2
BLOCKHASH	20
BALANCE	400
EXTCODESIZE	700
4) *storage manipulation*	
MLOAD/MSTORE	3
SLOAD	200
SSTORE	$\begin{cases} 20000 & \textit{if sets an empty slot to} \\ & \textit{a non} - \textit{empty value}; \\ 5000 & \textit{otherwise.} \end{cases}$
PUSH{*}/DUP{*}/SWAP{*}[†]	3
CALLDATACOPY/CODECOPY/RETURNDATACOPY	$3 + 3 \times \lceil \{\star\}\texttt{COPY}_{datasize} \div 32 \rceil$ [‡]
5) *control flow*	
STOP/RETURN/REVERT	0
JUMPDEST	1
JUMP	8
JUMPI	10
6) *logging*	
LOG{*n*}[◊]	$375 + 8 \times \texttt{LOG}_{datasize} + \text{n} \times 375$
7) *system operation*	
CREATE	32000
CALL/CALLCODE/DELEGATECALL	> 700[♦]
SELFDESTRUCT	$\begin{cases} 30000 & \textit{if creates a new account;} \\ 5000 & \textit{otherwise.} \end{cases}$

[†] In EVM, there are a spectrum of 32 different PUSH, DUP, and SWAP instructions, each starting from 1 to 32, e.g., PUSH1, DUP8, SWAP32. All of these instructions consume the same amount of gas, i.e., 3 units.
[‡] Here $\{\star\}$COPY stands for CALLDATACOPY, CODECOPY, or RETURNDATACOPY.
[◊] There are five LOG instructions in EVM, from LOG0 to LOG4. The variable "n" (as in LOG{n}) represents the number of *topics* for this log.
[♦] The gas cost for CALL-like instructions are very complicated, see [14].

They both expect parameters like ETH value to transfer, message call data, return data position, as well as gas limit for the internal call. Sometime, these contract-generated message calls are also known as internal transactions, as opposite to external transactions fired directly by EOAs. As far as EVM concerns, internal and external transactions are of little difference, since both are processed and executed in exactly the same way. However, for analysis purpose, the internal transactions are much more difficult to capture than external ones since they may only reside during runtime execution.

Like regular programs, contracts in execution may trigger unexpected behaviours, or runtime exceptions, e.g., divide a number by zero, lack necessary instruction parameters, and not enough gas available. In the bytecode level, EVM provides very little support towards handling exceptions. Besides, right until the latest version of Solidity (i.e., `v0.5.11` released in August 13th, 2019), it is still impossible for smart contracts to conduct common `try/catch` operations *w.r.t.* runtime exceptions. Thus, the only safe and possible way for exception handling is to fully revert current call, as well as all its sub-calls. In default, runtime exceptions will automatically "bubble up" or be re-thrown, causing the whole external transaction to revert. A few exceptions are message calls triggered by low-level functions like `call`, `delegatecall`, and `staticcall` of the target contract.

The gas mechanism of Ethereum

To circumvent around the inevitable halting problem stemming from Turing-completeness, as well as to provide economic incentive to external users and blockchain miners, Ethereum defines a systematic expenditure metering mechanism around the concept of gas. In general, gas measures the amount of processing resources that are allowed for or has already been consumed by a specific transaction (*aka* gas cost). The latter can be seen as a form of *transaction gas cost* (see Definition 1). In Ethereum, every transaction must specify a finite number of gas limit, i.e., `tx.gasLimit`, which restricts the maximal amount of gas that can be used by the transaction. The transaction gas limit, together with a *gas price*, i.e., `tx.gasPrice` , combinedly decide how much ETH a transaction sender has to pre-pay (plus additional ETH transferred directly to the receiver) before his or her transaction accepted as valid for further processing[4]. Besides, every valid block also has to set its own gas limit, i.e., `block.gasLimit`, which corresponds to the maximal accumulated gas cost that are allowed for all the transactions in that block.

[4]There are other criteria for a valid transaction. One that is related to gas requires transaction gas limit be greater than or at least equal to *intrinsic gas* [14].

In Ethereum, the exact amount of transaction gas cost can be divided into three parts: 1) *intrinsic gas cost*; 2) *execution gas cost*; and 3) *deploy gas cost*.

Definition 1 (Transaction Gas Cost)**.** *The gas cost for a specific transaction (denoted by* `tx`*) consists of three parts: 1) intrinsic gas cost; 2) execution gas cost; and 3) deploy gas cost.*

$$C(\texttt{tx}) = C_{intrinsic}(\texttt{tx}) + C_{execution}(\texttt{tx}) + C_{deploy}(\texttt{tx}) \tag{1}$$

Note, the Eq. 1 does not include a potential gas refund, since the latter happens after finishing execution and has nothing to do with an out of gas or otherwise exceptional transaction.

Note, whereas Ethereum provides a so-called gas refund mechanism by returning back some part of used gas during transaction execution [14], this operation actually takes place *after* the execution, and thus is not accounted in the above definition. What's more, since this refund happens after finishing execution, any single instance of out of gas exception will *still* cause state revert regardless of the potential refunded gas.

Definition 2 (Intrinsic Gas Cost)**.** *The intrinsic gas cost applies only to external transactions. For a specific external transaction (denoted by* `tx`*), its intrinsic gas cost can be divided into: 1) input data cost; 2) contract creation cost; and 3) basic cost.*

$$C_{intrinsic}(\texttt{tx}) = C_{input}(\texttt{tx}) + C_{create}(\texttt{tx}) + C_{basic}(\texttt{tx}) \tag{2}$$

$$C_{input}(\texttt{tx}) = \sum_{byte \in \texttt{tx.input}} \begin{cases} 4 & if byteiszero; \\ 68 & otherwise. \end{cases} \tag{3}$$

$$C_{creation}(\texttt{tx}) = \begin{cases} 32000 & if\, \texttt{tx.to}\ isempty, i.e., \\ & creatingnewcontract; \\ 0 & otherwise. \end{cases} \tag{4}$$

$$C_{basic}(\texttt{tx}) = 21000 \tag{5}$$

Note, the intrinsic gas cost is only valid for external transactions, whereas internal transactions (*aka* message calls) pays nothing for it. In other words, Ethereum charges intrinsic gas cost only on an external transaction basis.

Definition 3 (Execution Gas Cost)**.** *The execution gas cost of a specific transaction (denoted by* `tx`*) is the sum of individual gas cost for every executed instruction (denoted by* `INS`*).*

$$C_{deploy}(\texttt{tx}) = \sum_{\texttt{INS}} C(\texttt{INS}) \tag{6}$$

Note, for each specific instruction, the exact execution cost is defined by [14] and varies on a case-by-case basis.

A selected list of instructions and their corresponding execution gas can be seen in Table 1.

Definition 4 (Deploy Gas Cost)**.** *The deploy gas cost applies only to contract creation transactions (i.e.,* `tx.to` is empty*). For a specific contract creation transaction (denoted by* `tx`*), it charges for every byte of data that are returned (i.e., the newly created contract's code, denoted by* ***o****) by the execution.*

$$C_{deploy}(\texttt{tx}) = \begin{cases} 200 \times |\boldsymbol{o}| & if\, \texttt{tx.to}\ is empty; \\ 0 & otherwise. \end{cases} \quad (7)$$

Note, the deploy gas cost is only applicable for contract creation transactions (both explicit and implicit), and a deploy gas cost exhaustion will always lead to whole transaction out of gas, even if previous execution halts with remaining gas.

While intrinsic cost and deploy cost are straightforward to calculate [14], the execution gas cost is rather complicated. In fact, EVM charges execution cost in a just-in-time manner, before each instruction execution, until whether it goes to a normal halt or encounters any kind of runtime exception. Particularly, if available gas is not enough to pay for an additional instruction, EVM will trigger out of gas exception, halt execution immediately, and revert intermediate state.

One important rationale and design target for Ethereum gas mechanism is to ensure every transaction as well as instruction uses a "comparable" amount of gas *w.r.t.* resources it spent during execution. Failing to achieve this goal has proven to be dangerous by previous DoS attacks [24, 29–32]. To this end, the execution cost of each instruction can be dividend into three parts *w.r.t.* three different critical resources, i.e., computation, runtime memory, and storage. A complete gas schedule for all EVM instructions can be seen in [14].

For example, while every instruction pays for a computation cost, only two, i.e., `SLOAD/SSTORE`, will cause a storage cost. What's more, both `SLOAD` and `SSTORE` consumes a significantly larger amount of gas than other instructions (since storage access is much slower than computation), and that the cost of `SSTORE` is even higher than `SLOAD` so as to account for the "harder" task of writing than merely reading. Even the same `SSTORE` instruction itself may consume different amount of gas (20000 or 5000), depending on different context. As for memory execution cost, EVM follows the just-in-time fashion, i.e., every instruction only pays for the additional active memory footprint resulted from its execution. This is also known as *memory expansion cost* (see Definition 5) .

Definition 5 (Memory Expansion Cost)**.** *The memory expansion cost (i.e., C_{memory}) for a given instruction (denoted by* `INS`*) corresponds to the difference between active runtime memory cost $C_{active}(\boldsymbol{\mu}_i)$ before and after its execution, where $\boldsymbol{\mu}_i$ is the current active runtime memory size in words (i.e., 32 bytes or 256 bits). Here, we use* ***m*** *to represent the current EVM runtime memory.*

$$C_{memory}(\texttt{INS}) = C_{active}(\boldsymbol{\mu}_i^{\text{after}}) - C_{active}(\boldsymbol{\mu}_i^{\text{before}}) \quad (8)$$

$$C_{active}(x) = 3 \times x + \lfloor x^2/512 \rfloor \quad (9)$$

$$\boldsymbol{\mu}_i = \lceil \boldsymbol{m}_{highestaddress}/32 \rceil \quad (10)$$

Note, since currently EVM do not support reducing active memory size (as of St. Petersburg hardfork), the memory expansion cost shown in Eq. 8 will never go below zero. What's more, the exact value of this memory cost varies depending on the specific memory layout (both before and after execution) as well as instruction parameters. In other words, even the same instruction with the same parameters may cause completely different memory expansion cost, thus the total execution gas cost, under different circumstances.

Methodology

Our study consists of three phases (Fig. 3): 1) data collection; 2) empirical analysis; and 3) tool evaluation. First of all, we collect data by deploying two full-synced Ethereum clients (i.e., `Geth` and `Parity` with different settings), and scraping from blockchain explorer like Etherscan. The collected data are stored into a dedicated offline database for further analysis. Secondly, we use automatic script and manual inspection to investigate the overall status of out of gas exceptions, with a focus on their causing factors or behind reasons (***RQ1*** and ***RQ2***). At last, we investigate the effectiveness of existing tools in helping prevent out of gas exceptions (***RQ3***) using historical transactions as reference.

In particular, we deployed two Ethereum full nodes on the Mainnet, i.e., one `Geth` client and one `Parity` client. Both nodes are set to sync to the latest block height, i.e., 8,547,396 as of September 14th, 2019. We instrumented the `Geth` by adding code to identify and extract transactions triggering at least one instance of any runtime exceptions (including out of gas). The `Geth` node is running in `full` syncmode with state pruning `on` for about one month, on a machine with 2 Intel(R) Xeon(R) E5-2680 v4 CPUs (28 cores, 56 threads), 378 GB RAM, and 2 TB SSD. After successfully synced to block #8, 547, 396 (i.e., as of September 14th, 2019) in about 1 month, the `datadir` directory takes up about 416 GB of disk space. Besides, we also maintain a `Parity` node in `archive` pruning mode with tracing `on`. While this may consume more than 7× SSD space depending on specific setting [5], Archive nodes are special as they also provide the unmatched ability to replay past transactions, retrieve execution traces, as well as send simulated transactions at any point of time

[5] As of September 14th, 2019, a typical fully-synced `Parity` archive node takes around 3 TB of disk space to operate, which may even double when turning `-fat-db` on.

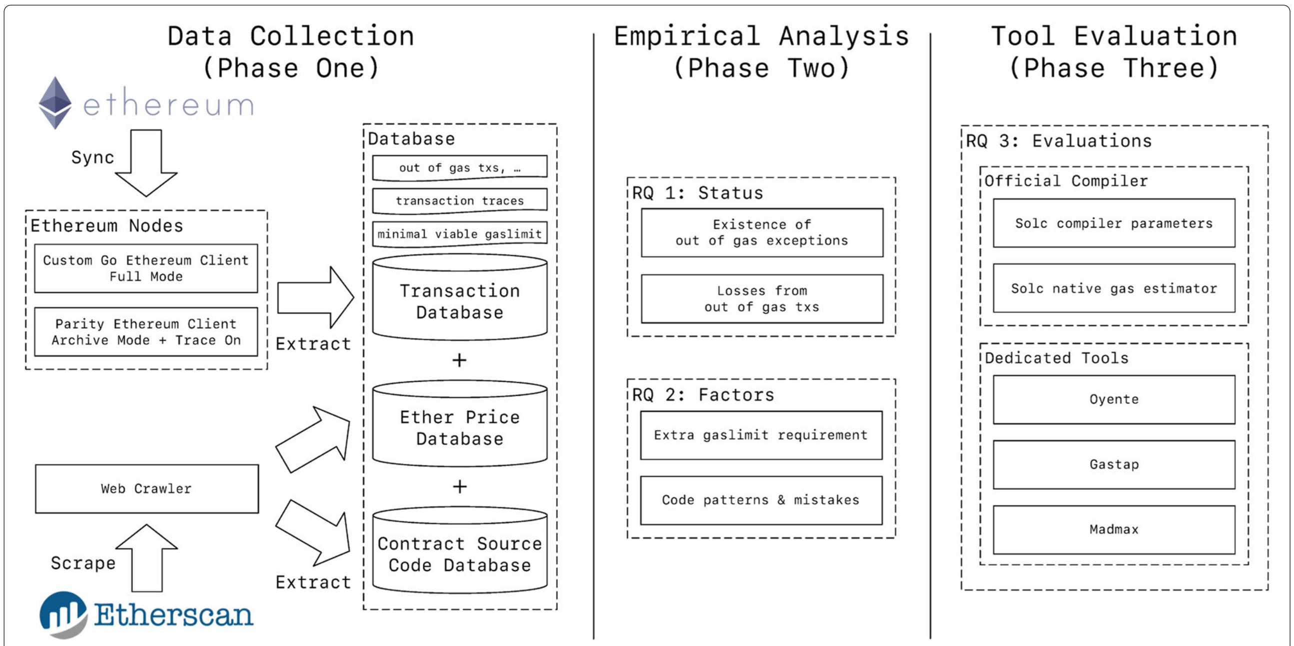


Fig. 3 An overview of our methodology

in history, which normal full nodes (with state pruning on) cannot offer. The Parity node we use in this work is based on QuikNode's dedicated Ethereum node service, which exposes standard Web3 JSON-RPC APIs through both HTTP and WebSocket protocols. It takes about 2 days for this node to fully synchronize.

For analysis and evaluation purpose, we also adopt several Web crawlers to extract both historic ETH price as well as known contract source code from Etherscan. And in the tool evaluation phase, we investigate the effectiveness of using native Solidity compiler in helping prevent these exceptions.

Results

RQ1: status quo

In this section, we investigate the current situation of out of gas exceptions. First of all, we are exceptionally interested in comparing gas exception with other runtime exception types in a macro view. Then, we look at the collective consequences of gas exceptions using historical transaction data collected in Section 2. After that, we alter our attention to the micro cases by focusing on individual smart contracts as well as transactions, in hope of finding clues of the mechanism and causes leading to out of gas exceptions.

Exception taxonomy

We first look at the runtime exceptions of EVM. By referring to both the design paper [14] as well as canonical client implementation [33], we identify a number of 16 different exception types in EVM, and further group them into six major categories. In Table 2, we show this exception taxonomy, as well as the absolute occurrences and relative scales of each exception type. In particular, we distinguish between two types of out of gas exceptions: those happen during execution (i.e., execute out of gas, EOG) and those after within code deployment (i.e., deploy out of gas, DOG). In other words, EOG happens because of short of execution gas cost, whereas DOG results from lacking of deploy gas cost (see Definition 4 in "The gas mechanism of Ethereum" section).

We claim the importance of considering both external and internal transactions. According to [34], exceptions happened in internal calls may not bubble up if the invoker uses low-level Solidity call functions [34]. In this case, the invoking contract can choose whether to revert itself or simply ignore deep exceptions. In other words, during course of an external transaction, multiple exceptions triggered in internal transactions, while the outside external one still appears to be normal. If not consider these internal exceptions, we may end up underestimating the scale of runtime exceptions and losing sight of some deep factors for their appearances. This fact is perfectly illustrated by the EOG type in Table 2, where every external transaction tends to trigger above 36 instances of exceptions during its execution.

The data shown in Table 2 are obtained by analyzing every historical transaction on Ethereum mainnet from the genesis block (i.e., block #0) towards block #8, 547, 396 (i.e., as of September 14th, 2019). We further divide table columns into two categories, i.e., occurrence

Table 2 Comparing out of gas exception with other exception types in EVM

Exception Type	**Occurence**	**Percentage**			
	All	*External*	*Ratio*	*All*	*External*
1) *Explicit Revert*					
Require-Style Revert (RR)	14,000,856	**11, 456, 103**	1.22	8.12%	**64.91%**
Assert-Style Revert (AR)	990,183	925,701	1.07	0.57%	5.24%
2) *Out of Gas*					
Deploy Out of Gas (DOG)	10,963	10,963	1	0%	0.06%
Execute Out of Gas (EOG)	**155, 373, 273**	4,281,071	**36.29**	**90.09%**	24.25%
3) *Stack Overflow/Underflow*					
Call-Stack Overflow (CSO)	10,032	1,113	9.01	0%	0.01%
Data-Stack Underflow (DSU)	153,445	53,501	2.87	0.09%	0.30%
Data-Stack Overflow (DSO)	152	152	1	0%	0.001%
4) *Illegal Instruction*					
Invalid Jump Destination (IJD)	1,341,130	1,306,785	1.03	0.78%	7.40%
Invalid Opcode (IO)	232,226	189,518	1.23	0.13%	1.07%
5) *Not Enough Ether*					
Insufficient Balance (IB)	359,822	356,517	1.01	0.21%	2.02%
6) *Miscellanea*					
Client Decision, Illegal Write, etc.	1,693	1,692	1.00	0%	0.01%
◊ *Summary*					
Out of Gas (DOG + EOG)	**155, 384, 236**	4,291,945	**36.20**	**90.09%**	24.32%
Explicit Revert (RR + AR)	14,991,039	**12, 137, 417**	1.24	8.69%	**68.77%**
Other Exception Types	2,098,500	1,684,217	1.25	1.22%	9.54%
◊ *Summery (excluding DoS attacks)*					
Out of Gas (DOG + EOG)	4,666,508	4,233,428	1.10	21.86%	24.10%
Explicit Revert (RR + AR)	**14, 987, 686**	**12, 134, 192**	**1.24**	**70.20%**	**69.07%**
Other Exception Types	1,696,017	1,654,643	1.03	7.94%	9.42%

and percentage. In the `Occurrence` column, we show results for three related concepts: 1) number of exception instances, including external and internal transactions; 2) number of external transactions; and 3) average number of exception instances per external transaction. In the `Percentage` column, we show numbers for exception instances and external transactions.

From Table 2, we get the following observations:

- 1) out of gas and explicit revert are two most commonly seen types of exception in Ethereum, which combinedly account for more than 90% of the occurrences in terms of both exception instances (i.e., external transactions plus internal message calls) as well as external transactions.
- 2) When considering all the transactions, out of gas alone accounts for more than 90% of all exceptions, with explicit revert only takes another 8%. However, after excluding the notorious DoS attacks [29, 30] by eliminating transactions from block #2, 250, 000 till block #2, 750, 000 (both inclusive), the result swaps, where gas exception now takes up slightly more than 20%, while explicit revert accounts for another 70%. A closer look at the DoS interval reveals that only 58,517 external transactions in this section contribute to a total number of 150,717,728 exception instances, that's nearly 2,576 instances per external transaction. The finding suggests a gigantic influence of the DoS attacks on our study of gas exception. Hence, in the following of this work, we always exclude transactions (and their exception instances) from block #2, 250, 000 to block #2, 750, 000, in hope of minimizing unfavourable effects from these attacks.
- 3) On average, all the exception types in Table 2 take place more than once in a single external transaction. In other words, there are at least an external transaction that has witnessed more than one exception. Or, some contracts tend to ignore or not fully revert in case of deep runtime exceptions. Besides, transactions may also trigger more than one

type of exceptions. This can be checked by adding all the relative percentage of external transactions for each exception type, which yields around 105%, exceeding the normal 100%.

In summary, explicit revert and out of gas together dominate runtime exceptions, whereas the former appears even three times more frequently (i.e., 70% *vs.* 22%) than the latter, after eliminating the influence internal of DoS attacks. In this work, we focus on the out of gas exception, since we believe in the uniqueness of gas exception for Ethereum smart contracts as compared to regular programs, and that the existence of gas exception is much more subtle and trickier than explicit revert in terms of both the causing factors and the mitigating methods.

Accumulative consequences

Besides popularity, we are also interested in the negative effects (or losses) of gas exceptions, especially as for transaction senders and network miners.

In Fig. 4, we present the accumulative losses of both exceptions, where indices for `EOG` are shown in full lines, and indices for `DOG` are in dashed lines. The results in Fig. 4 are collected and presented in intervals of 1 million blocks, from block #0 till block #8, 547, 396, and we show each value in their logarithmic scale. To evaluate losses, we choose three related indices: 1) number of affected external transactions (shown as `Txs`); 2) accumulated affected gas units (shown as `Gas` in units of 10^9 gas, or giga gas); and 3) corresponding affected `ETH` values (shown as `ETH`).

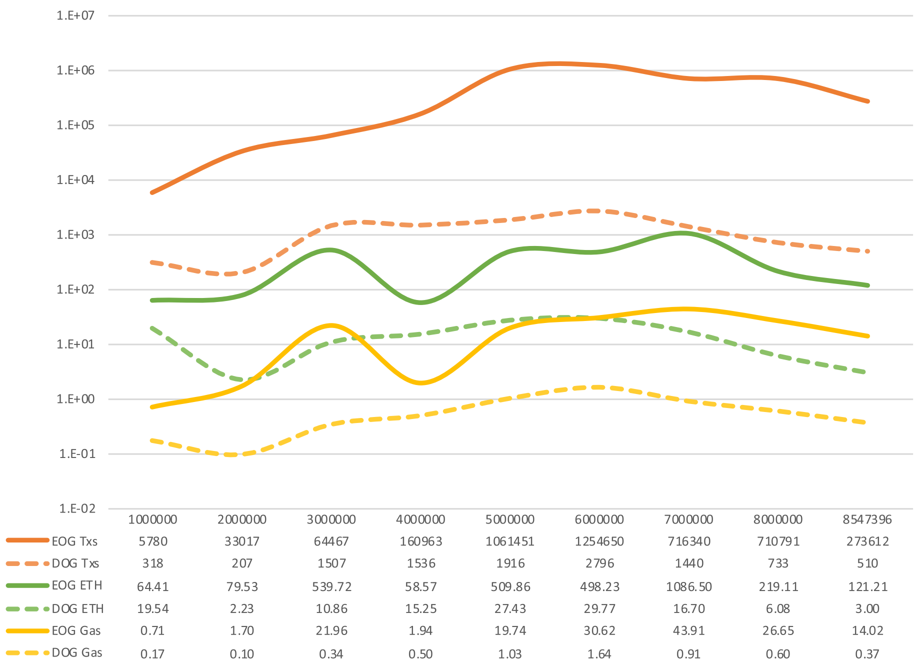
Here, we define the accumulated affected gas as the total of transaction gas limits for each exception instance. In other words, we count the sum of proposed gas limit for every exception instance. While this index systematically overestimates the total losses of gas (as well as corresponding `ETH` values) [6] , it is the simplest and most easily accessible estimator we can get, and we have found some evidences showing the two indices do not differ very significantly, e.g., in orders of magnitude. Besides, we also calculate the corresponding affected `ETH` values with respect to accumulated affected gas by taking each transaction individually with its designated transaction price, i.e., `tx.gasPrice` . Last, the number of accumulated affected gas *does not include* intrinsic gas cost (see Definition 2 in "The gas mechanism of Ethereum" section) as well as mandatory `CALL` execution cost in some cases, i.e., when outer transaction does not trigger an out of gas exception itself. This kind of simplification is reasonable since we're more interested in comparing the pure wasted gas for gas exceptions, whereas the aforementioned costs always exist regardless of any exception.

From Fig. 4, we get the following observations:

- 1) The losses resulted from gas exceptions are huge. In segments of 1 million blocks, as large as some hundred thousand external transactions are affected (that's slightly less than 1 transaction per block), causing a number of several hundreds `ETH` values wasted, or in US dollars, tens of thousands with a fairly low average exchange rate of $150/`ETH`. As for specific type, `EOG` dominates `DOG` in each of the considered indices (i.e., number of external transactions, accumulated affected gas units, and corresponding affected `ETH` values), and the differences are often in orders of magnitude large (i.e., ten times or above).
- 2) In terms of every index, both `EOG` and `DOG` experience similar trends throughout entire transaction history. For example, consider the number of external transactions involved in gas exception, i.e., `EOG Txs` and `DOG Txs` shown in Fig. 4. Both lines begin with a small number, then quickly climb to reach their maxima, and at last stay relatively stable [7] , i.e., about 10^6 external transactions for `EOG` and 10^3 for `DOG`. What's more, both `EOG Txs` and `DOG Txs` reach their maximal value (i.e., 1,254,650 for `EOG` and 2,796 for `DOG`) in between block #5, 000, 000 (Jan, 2018) and #6, 000, 000 (July, 2018), when blockchain and cryptocurrency industry experience their latest hype.
- 3) The lines for accumulated affected gas units and corresponding `ETH` values match very well in shapes, a trend that can be tested by both `EOG` and `DOG` exception types. This suggests a relatively stable long term gas price across large section of blocks.
- 4) In the interval of #2, 000, 000 to #3, 000, 000, we see a salient rise of both affected gas units and `ETH` values for `EOG` exception, then the same indices soon descend to around 1/10 of the original level in interval #3, 000, 000 to #4, 000, 000. Finally, in interval #4, 000, 000 to #5, 000, 000, the exact two indices, i.e., `EOG Gas` and `EOG ETH`, jump again to reach their previous high levels. During these three intervals, we find `EOG Gas` and `EOG ETH` are the only pair of indices showing this kind of trend, i.e., swing up and down in order of magnitude (i.e., ten times or above) in consecutive intervals. By comparing with other indices, we think this results from a sudden dump of affected gas units and `ETH` value for `EOG` during blocks #3, 000, 000 to #4, 000, 000, instead of

[6]Considering an external transaction with deep inter-contract invocation, where the inner most internal transaction runs out of gas, causing each outer transaction to gas exception. When calculating accumulated affected gas, the sum is strictly larger than total loss of gas, since it implicitly adds inner transaction gas limits some multiple times.

[7]Note the last bucket only counts from block #8, 000, 000 to block #8, 547, 396, i.e., slightly more than half of the 1 million blocks interval. If instead using average data for each index, the steady trend will be more obvious.



	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	8547396
EOG Txs	5780	33017	64467	160963	1061451	1254650	716340	710791	273612
DOG Txs	318	207	1507	1536	1916	2796	1440	733	510
EOG ETH	64.41	79.53	539.72	58.57	509.86	498.23	1086.50	219.11	121.21
DOG ETH	19.54	2.23	10.86	15.25	27.43	29.77	16.70	6.08	3.00
EOG Gas	0.71	1.70	21.96	1.94	19.74	30.62	43.91	26.65	14.02
DOG Gas	0.17	0.10	0.34	0.50	1.03	1.64	0.91	0.60	0.37

Fig. 4 Transactions, gas units, and `ETH` values affected by out of gas exceptions

an opposite situation where quick jump between blocks #2, 000, 000 to #3, 000, 000 happens before.

Smart contracts

In order to further understand gas exception, we alter our attention to individual accounts, especially those smart contracts [8] involved in exception transactions.

By grouping exception instances (both external and internal transactions) according to their `sender` and `receiver` addresses, we manage to figure out the most "popular" accounts related to out of gas exceptions. More specifically, we are interested in finding accounts sending and receiving most gas exception transactions (both `EOG` and `DOG`) whether through external transactions or internal message calls.

In Table 3, we show top 10 accounts sending and receiving gas exception transactions. The total number of such accounts for each direction are 1,101,591 and 148,940, respectively. Recall from "Exception taxonomy" section, we deliberately ignore transactions between block #2, 250, 000 and #2, 750, 000 to mitigate the effects of infamous DoS attacks. For each account, we report the number of exception instances (denoted as `Instance`), accumulated affected gas units (denoted as `Gas`, see "Accumulative consequences" sections), and corresponding affected `ETH` values (denoted as `Ether`). Besides, we also estimate the monetary losses of `ETH` using an exchange rate of \$150/`ETH`.

[8]Recall that smart contracts are a kind of accounts with additional code and storage space.

It is worth noting that all of the accounts shown in Table 3 are smart contracts encountering only `EOG` exceptions. Whereas the highest ranked accounts with both `EOG` and `DOG` exceptions ranks #54 (i.e., `0x6090~78Ef`) as transaction sender and #13 (i.e., _, the placeholder address for contract creation transaction) as receiver.

From Table 3, we observe the following facts:

- 1) Smart contracts tend to see more gas exceptions than plain EOAs. This can be easily checked by observing that all the accounts in Table 3 are actually smart contracts, and recall that we mention before the highest ranking EOAs for each direction only take up #54 (as transaction sender) and #13 (as transaction receiver) respectively. There are at least two explanations for this phenomenon. On one hand, smart contracts are more vulnerable to out of gas exceptions. A smart contract, once deployed on Ethereum, can never change its execution code during entire lifetime. This means existing bugs or inappropriate gas limit settings are hard to be fixed then. Thus, if a contract sets a too conservative gas limit for internal message calls, it should have seen more gas exceptions compared to one with a much loose gas limit. On the other hand, smart contracts tend to communicate more frequently between each other than EOAs, creating a large base for unexpected gas exceptions. As a rule of thumb, developers tend to reuse well-tested and verified code libraries while building new applications, where in

Table 3 A list of top 10 accounts sending and receiving most out of gas exceptions

Account Address	**Instance**	**Gas**	**Ether**	
1) *Accounts Sending Most Out of Gas Transactions*				
`0x60bf91ac87fEE5A78c28F7b67701FBCFA79C18EC`	**1,213,760**	9,917,594	0.17	($24.88)
`0x4B9e0d224DABCC96191cacE2D367A8d8B75C9C81`	68,957	209,168	0.002	($0.34)
`0x68C769478002B2E2Db64fE3Be55C943fE4Fbd6b1`	57,257	242,976	0.003	($0.45)
`0xE4c94d45f7Aef7018a5D66f44aF780ec6023378e`	56,388	6,134,966	0.12	($18.26)
`0x0000000000085d4780B73119b644AE5ecd22b376`	25,079	497,667,177	4.60	($689.86)
`0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`	22,847	**1,486,219,348**	**40.37**	($**6055.3**)
`0x7c5Cb1220Bd293Ff9cf903915732e51a71292038`	15,024	639,357,019	8.29	($1243.74)
`0x0000000000013949F288172bD7E36837bDdC7211`	11,212	108,082,558	0.31	($46.91)
`0xd0a6E6C54DbC68Db5db3A091B171A77407Ff7ccf`	10,298	96,311,375	2.49	($373.67)
`0x414FBf684A6426cf6012623f51170a5A86161d52`	10,041	55,541	0.0006	($0.09)
2) *Accounts Receiving Most Out of Gas Transactions*				
`0x0000000000000000000000000000000000000004`	**1,412,148**	4,236,441	0.06	($9.30)
`0x744d70FDBE2Ba4CF95131626614a1763DF805B9E`	81,830	3,826,605,402	49.93	($7489.45)
`0xd0a6E6C54DbC68Db5db3A091B171A77407Ff7ccf`	44,721	**7,658,554,274**	**164.20**	($**24630.2**)
`0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`	38,118	792,031,653	24.70	($3704.29)
`0x8d12A197cB00D4747a1fe03395095ce2A5CC6819`	35,300	975,469,855	23.99	($3598.60)
`0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C`	33,486	1,286,027,958	37.60	($5639.40)
`0x419D0d8BdD9aF5e606Ae2232ed285Aff190E711b`	30,622	578,076,676	5.94	($890.97)
`0xba7435A4b4C747E0101780073eedA872a69Bdcd4`	30,483	1,590,815,838	9.99	($1498.03)
`0x86Fa049857E0209aa7D9e616F7eb3b3B78ECfdb0`	26,523	320,330,157	11.43	($1715.05)
`0x5EdC1a266E8b2c5E8086d373725dF0690af7e3Ea`	23,943	1,437,343,328	13.30	($1995.45)

Ethereum the libraries may be previously deployed contracts which expose the same addresses. Besides, to mitigate the risk of unknown bugs and facilitate better maintainability, it is even widely recommended to build smart contracts using proxy patterns, which again increase the interactions between these contracts. All in all, the communications between smart contracts are much more common than between EOAs, contributing to a much larger surface for runtime exceptions, including gas exceptions.

- 2) All the contracts in Table 3 has experienced a large number of gas exceptions during lifetime, where the top 1 accounts both see above 1 million exceptions as transaction sender and receiver respectively. However, the contracts causing most gas units and `ETH` losses, i.e., `0x0601∼266d` and `0xd0a6∼7ccf`, are not the most frequently involved. In fact, the underlined contract `0xd0a6∼7ccf` (EOSSale) has caused more than 164 `ETH` losses with only 44,721 transaction calls, which is much smaller than contract `0x04` of 1,412,148 invocations instead.
- 3) Contracts at the bottom half of Table 3 (i.e., receiving the most gas exceptions) has caused far more losses than the top half (i.e., sending the most exceptions). Notice that the total amount of losses (both gas and `ETH`) are always identical counting from both directions, which suggests an imbalance or asymmetry between transaction senders and receivers. In other words, a large number of ordinary accounts (both EOAs and smart contracts) tend to interact with a small set of popular accounts (mostly smart contracts) which act as celebrities in the world of smart contracts. For example, EOAs may need to transfer well-known ERC-20 tokens between each other by calling the same token transfer function, and famous contract libraries are often shared for reuse by a large number of smart contracts.
- 4) Consider the contract with address `0x04` which ranks first as the most out of gas exception receiver. According to Ethereum yellow paper [14], this contract is among a set of 8 special "precompiled" contracts that are proposed to facilitate common and preliminary functionalities to the platform, e.g., the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD 160-bit hash scheme, and so forth. As for `0x04`, it acts as an identity function for its input, i.e., by returning the

same input data as its output value. While not figuring out the point to call this contract, an even more appealing fact emerges when we look at the huge amount of exception instances (i.e., 1,412,148) versus a nearly negligible affected gas units (i.e., 4,236,441). Further investigation reveals that all exceptions result from internal message calls, and all but one invocations have set a small gas limit of 3 units. Even mysteriously, the `Parity` trace module seems unable to identify these internal invocations, as well as the resulting gas exceptions. After a careful inspection of relevant traces and contract bytecode, we are confident to confirm the existence of both transactions and exceptions. We guess the leading factor for these large number of small gas limit calls to contract `0x04` is a subtle compiler bug, however, we do not know the intention and mechanism behind currently.

- 5) There are two contracts showing up in both lists that send and receive most gas exception transactions, i.e., the underlined contract `0xd0a6∼7ccf` (EOSSale) and the tilded contract `0x0601∼266d` (KittyCore), which happens to be implementations of two most popular token standards in Ethereum, i.e., ERC-20, ERC-721[9].

Transactions

In this section, we turn our eyes to individual transaction. More specifically, we look at top external transactions with most affected gas units by out of gas exceptions.

In Table 4, we show top 5 external transactions which has: 1) triggered most out of gas exceptions; 2) seen most accumulated affected gas units. The columns from left to right are number of exceptions (denoted as `OG`), number of message calls (including the outmost external transaction, denoted as `Call`), accumulated affected gas units (denoted as `Gas`), available execution gas limit (excluding intrinsic gas cost for the outmost external transaction, denoted as `Limit`), and whether this external transaction runs out of gas itself (denoted as `Ext`). As before, we intentionally exclude transactions between block $\#2,250,000$ and $\#2,750,000$ to mitigate the influence of historical DoS attacks on Ethereum, and that we present wasted gas units as accumulated gas limits of exception transactions.

From Table 4, we get the following observations:

- 1) While most transactions in Table 4 have triggered an impressive number of `OG` exceptions, more than half of them are not externally out of gas themselves. In other words, only looking at the external transaction may lead to serious under-estimation of the frequency for gas exceptions.
- 2) The underlined transaction `0xeffd∼5725` in block $\#3,271,486$ is caught with an extremely large number of gas exceptions, i.e., 1,562 in a single external transaction. By further investigation, we find it a contract creation transaction (with `tx.to` set to empty) with 1,561 delegatecalls to the same contract `0x7f6E∼86F3`, each time with 0 gas limit (and thus doomed to failed as `EOG`). Further study on the transaction input data, here act as contract initiation code, reveals that the code performs nothing meaningful but only continuously generating out of gas delegatecalls through an infinite loop, and that the bytecode seems not have been produced by standard `solc` compiler, but instead coded manually to perform the instructed tasks. While we do not have access to the source code of this init code or of the delegated contract `0x7f6E∼86F3`, we believe it is not intended to do something good, and may be linked to previous DoS attacks.
- 3) The last three transactions in the bottom half of Table 4 (i.e., `0xd0f8∼7458`, `0x0180∼b2d8`, and `0xf52c∼aa55`) each causes a tiny amount of gas losses, i.e., 3 units per message call in average, whereas Etherscan seems not reporting any gas exception in them. However, by carefully inspection of the data, as well as using online debugger of Etherscan, we are quite sure about their existence, and that we find all these exceptions are direct results of invoking the identity contract (i.e., `0x04`) with inadequate gas limit, as described in "Smart contracts" section. What's more, all these exceptions happens to be triggered by the contract `0x60bf∼18EC`, which appears in the top half of Table 3 as the contract sending most out of gas exceptions.

Blockchain-based cloud applications

In this section, we investigate the relationship between out of gas exception and blockchain-based cloud application. In general, out of gas exception may cause three negative consequences to the successful of blockchain-based cloud application.

First of all, as part of the blockchain-based cloud application, any exception (including gas exception) happened during smart contract execution means a stop of the normal application logic, and should eventually cause the overall operation to revert. This kind of midway reversion will inevitably lead to a poor QoS (Quality of Service), especially when the backbone blockchain is experiencing a busy traffic and thus with higher latency. Compared with other types of exceptions, out of gas exceptions are even more troublesome as few developers could have anticipated the

[9] Also known as standard fungible and non-fungible token protocols in Ethereum.

Table 4 A list of top 5 external transactions that: 1) triggering the most out of gas exceptions; as well as 2) with the most accumulated wasted gas

Transaction	**OG**	**Call**	**Gas**	**Limit**	**Ext**
1) *Transactions Triggering Most Gas Exceptions*					
`0xeffd72d2245acd53020c519d2be29bd82f83e184730e3b2dbd0015ea0a425725`	**1,562**	**1562**	1,116,188	1,116,188	`YES`
`0x5108feb5a8f8988227a5d107c470828f9174f41aaa9bfd3682988836413eeedd`	130	326	**1,737,970**	**7,779,026**	`NO`
`0xd0f8611733461d33df72e8b1561078db4882a681a7299916caf47fb2b72b7458`	82	740	246	7,776,764	`NO`
`0x018077d47b8a3b478b8f0ea99baa0e2bbe2cf90fccf6b4c50fa4c5d5b276b2d8`	82	739	246	7,601,156	`NO`
`0xf52c87299dd303541436f30093ecb1a28f22629ec6d4ed50da76bb12b168aa55`	80	721	240	7,054,030	`NO`
2) *Transactions with Most Accumulated Affected Gas Units*					
`0x448b49f72d23ecdb281bf1a92d94ab63ef3efc58937d80f51fa2dadd02591bdb`	**52**	146	**43,741,354**	4,978,408	`NO`
`0x48f47b8f8b3a4b9b169079760d53e1711ad22e6305304fa0d40252c797765e0c`	39	83	21,199,332	1,714,298	`NO`
`0xc4f7e0cd07e96e40d0ea47747de1f452bc9ad01d5a3628cce0182a4b3814cc36`	2	4	9,649,680	4,978,408	`YES`
`0xaf96bc199f1a041e76c2698ba14f0e3540f6bf3aa981be3961f9468431943e0a`	2	**2,999**	7,980,206	7,978,728	`YES`
`0x65d8fe350eb49c227d8fb4fa0b86e2a9e179c4cb820ba225040b2204a0248e6b`	1	3	7,979,000	**7,979,000**	`YES`

occurrence of them and enforced the correct protection accordingly.

Second, blockchain-based cloud applications with hidden gas exceptions are difficult to identify, debug, and fix appropriately. When integrated with browser-side frontend code and/or server-side backend code (see Fig. 2), developers face increasingly difficulties in testing and debugging out of gas exception related issues. In addition, the tamper-proof characteristic of smart contract often means hard to fix bugs or upgrading contract code. Hence, blockchain-based cloud application developers should always follow best-practices like proxy pattern at first place to prevent from jeopardizing themselves with hard-to-fix vulnerabilities.

Last, unlike traditional cloud applications, blockchain-based cloud applications often have to deal with digital assets that have intrinsic value (i.e., monetary value) with them. For example, a DEX (decentralized exchange) blockchain-based cloud application must implement at least one function for crypto-currency pair exchange, which will internally call the transfer functions of both crypto-currencies. If the DEX contract fails to identify and properly handle gas exceptions during this process, users may result in losing money while performing exchange. In fact, according to [35], more than three quarters of the blockchain-based cloud applications have functionalities in managing or operating on high monetary value density data. Thus, developers should always prepare themselves to unexpected gas exceptions, or they may accidentally cause monetary loss to their customers.

In the following part, we look at a specific blockchain-based cloud application, i.e., the very popular CryptoKitties game, and perform a case study on out of gas exception issues with regard to it. We choose to focus on the smart contract part of CryptoKitties, as it is the directly influenced component by gas exceptions, and that other components (like the server-side backend component) are not accessible to us at the time of writing.

In Table 5, we present a basic gas exception summary about smart contract components for CryptoKitties as of block #8, 547, 396, the list of smart contracts and their source code are accessible from Etherscan[10]. For each smart contract, we provide the contract address (denoted as `Contract`), contract name (denoted as `Name`), and statistics of both sending and receiving out of gas transactions, denoted as `Sending OG` and `Receiving OG` respectively. For the last two indices, we further divide them into three sub-indices for each: 1) number of `EOG` exceptions; 2) accumulated affected gas units; and 3) corresponding affected `ETH` values.

Note, we do not show number of `DOG` exceptions for both transaction direction, i.e., sending and receiving, since we find no smart contract in Table 5 has ever triggered such an exception. In fact, there are even no contract creation instructions inside any of these contract's source code.

From Table 5, we observe the following facts:

- 1) The distribution of gas exceptions among contracts is asymmetric. In other words, different contracts in the same application seem trigger an uneven number of gas exceptions. Among these 5 smart contracts of CryptoKitties, `KittyCore` and `SaleClockAuction` have triggered the most gas exceptions, both in terms of exception numbers as well as affected gas units. In particular, `KittyCore` is the most vulnerable contract for sending and receiving gas exceptions in terms of number, i.e., 22,847 and 38,118. Whereas `KittyCore` and

[10]See https://etherscan.io/accounts/label/cryptokitties.

Table 5 Basic gas exception summary of smart contracts from CryptoKitties dApp

Contract	**Name**	**Sending OG**				**Receiving OG**	
		EOG	*Gas*	*Ether*	*EOG*	*Gas*	*Ether*
`0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`	KittyCore	**22,847**	**1,486,219,348**	**40.37**	**38,118**	792,031,653	24.70
`0xb77FeddB7e627a78140a2a32CAC65A49eD1DBa8E`	GeneScience	0	0	0	23	1,242,399	0.12
`0x57831A0C76Ba6b4FDcbadd6cb48cB26e8fc15e93`	Offers	0	0	0	4	111,456	0.0004
`0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C`	SaleClockAuction	6,813	133,555,374	4.40	33,486	**1,286,027,958**	**37.60**
`0xC7af99Fe5513eB6710e6D5f44F9989dA40F27F26`	SiringClockAuction	2,770	49,120,889	1.35	7,566	385,080,223	10.28

`SaleClockAuction` ranks first in sending and receiving most gas affected `OG` transactions, respectively.

- 2) For a single smart contract, the gas exception distribution between transaction directions (i.e., sending or receiving) is also often imbalanced. For example, the `SaleClockAuction` contract has seen 4 times more gas exceptions for receiving transaction calls than sending out.
- 3) `GeneScience` and `Offers` have seen much less out of gas exceptions during lifetime than the rest of contracts in Table 5. Besides, they are not recorded as sending out a single gas exception transaction. The reason for this lie in the function decomposition of different contracts, where `GeneScience` and `Offers` are never expected to invoke other contract's functions during course of execution, so will never trigger exceptions in the outward direction (i.e., as transaction sender). What's more, the two contracts also have relatively fixed behaviour, so it is much easier to predict or even bound the maximal gas consumptions before transactions.

RQ2: causing factor

In this section, we study the common causing factors for out of gas exceptions, in hope to help both developers and dApp users to prevent potential gas exceptions.

Common causing factors

By manually inspecting exceptional transactions, their execution traces, as well as related contracts, we have found some common causing factors for out of gas exceptions, as summarized below:

- 1) *Misunderstanding Transaction Mechanism* This is a commonly seen and most trivial causing factor for out of gas exceptions, especially *w.r.t.* to external transactions. In particular, according to the transaction processing mechanism, if the transaction target/destination (`tx.to`) is a smart contract, Ethereum will load that contract's code and starting running along with transaction input (`tx.input`) in EVM. Note, this process is automatically triggered by Ethereum without user intervention. Thus if the user overlooks or ignores the aforementioned contract execution mechanism, and sets transaction gas limit to its minimal viable value (i.e., the very basic intrinsic gas cost for a valid external transaction, 21,000 for normal transfer and 53,000 for contract creation), there will always be out of gas exception since not a single gas unit is available for further contract execution. In our data set, we have found a total number of 542,193 external transactions having this kind of problem, accounting for nearly one fifths of such transactions. Besides, the problem does not see a clearly decreasing in terms of transaction numbers as time passes by. In particular, we have found 41,820 external transactions suffering from the problem from block $\#8,000,000$ to $\#8,547,396$, whereas the highest number per one million blocks is just 175,204 for interval $\#4,000,000$ to $\#5,000,000$.
- 2) *Conservative Gas Limit* This kind of problem stems from the fact that the transactions can terminate without any exception but are otherwise set with a lower gas limit than needed. For example, the transaction `0xf31d∼9557` in block $\#8,547,387$ happens to run out of gas with a relative small gas limit 30,000. By setting a much higher gas limit, we find the actual gas needed for the transaction is only 37,112, or 7,112 more units compared to original gas limit. In other words, the user could have saved a gas loss of 30,000 units by merely paying 7,112 units more, that's a 22,888 units net earning.
- 3) *Compiler Derived Bug* Sometimes, the problem for out of gas exception may stem from hidden bugs or flaws of the contract compiler (in most cases the `solc` Solidity compiler.) An example of this kind is the under-gas call to precompiled identity contract `0x04` [14], where the message call only gets 3 units of gas for execution. This accounts for about 2% of all the exception instances found in our data set. According to [14], the gas cost for identity contract is 15 units plus 3 per input word. In other words, the cost is always large or equal to 15, where a gas limit of 3 is doomed out of gas. In fact, we have seen a large number of such instances during our investigation. (Table 3), like the transaction `0xd0f8∼7458` shown in "Transactions" section (Table 4). While we do not know the cause of this problem, and it may not be a big problem for users, it at least reflects the fact that Solidity compilers are not mature right now, and should be carefully checked in production environment.
- 4) *Unbounded Mass Operation* The authors of [21] have revealed several gas-related contract vulnerabilities which may trigger unexpected behaviours, e.g., locking specific functions forever, or running into a doomed out of gas loop. This phenomenon is confirmed in our investigation by transaction `0x448b49f72d23ecdb281bf1a92d94ab63ef3efc58937d80f51fa2dadd02591bdb`, where two contracts mutually call each other recursive, lead to out of gas.
- 5) *Others* Due to the large size of our dataset, i.e., more than 56 million transaction traces from nearly 150,000 unique smart contracts, we are unable to cover every contract and its execution traces. The factors shown above are discovered by manual

inspection of top accounts, transactions, and contracts involved in out of gas exceptions found in our data set. We plan to check the rest of our data in further, looking for both transactions and smart contracts. We believe there are more hidden factors waiting for discovery.

RQ3: tool evaluation

Data set

In this section, we are devoted to investigating effectiveness of existing tools or methods, *w.r.t.* preventing out of gas exceptions. For this purpose, we have collected a data set of 1,596,145 different out of gas transactions (including internal message calls), belonging to 449 different smart contracts. The data set is built by selecting smart contracts with most receiving gas exceptions (see "Smart contracts" section). In particular, we first take those accounts receiving more than 1,000 gas exceptions, then filter out non smart contracts and some special addresses (e.g., the `NULL` representing contract creation and the `0x04` precompiled contract with no source code available). Since some evaluated tools only accept source code as input, we further checked and retrieved source code of these contracts using Etherscan `getsourcecode` API, making sure all these contracts have corresponding verified source code available. In Table 6, we show a list of 10 example contracts from the data set, offering information like contract address, contract name, number of exception instances, and important compiler parameters.

Note, the contracts shown in Table 6 share some similarities, e.g., all of them are token related contracts compiled with Solidity compiler version `v0.4.x`. Besides, 6 of the contracts turn gas optimization option `on`, and all with an expected execution run (by `-optimize-runs` option) of 200 times.

Gas estimator

Gas estimators are tools or services that can report an estimated gas cost for proposed transactions or contract functions. Depending on required input data and action timing, gas estimators can be further divided into two categories: 1) *offline gas estimators* that only need contract's code as input (depending on specific tool, the code may be source code or bytecode), and are only needed to run once, then to be used arbitrage number of times (provided the contract's code are not modified after); 2) *online gas estimators* which utilize Ethereum client's transaction simulation capability to execute transactions on top of current world state without writing back, users need to provide both contract code as well as proposed transactions, and have to run the tool each time when a new transaction or world state is available.

In principle, online gas estimators (e.g., the standard `eth_estimateGas` JSON-RPC API exposed by `Geth`) can return more accurate gas cost estimations as compared to offline gas estimators. After all, the "estimations" returned by online gas estimators are actually real gas costs of the transactions, based on the current world state seen by the tools. If we believe users will always stick to using online gas estimators before proposing transactions, the possibility that these transactions running out of gas will be negligibly low, and thus we should not have found so many gas exceptions as in our study. The point is that, it suggests Ethereum users are not always using online gas estimators before submitting their transactions. The reasons behind are manifold, perhaps they just do not know of these tools, or maybe users are unable to get accessible to these tools because they do not have direct control over their accounts (e.g., users host their accounts on third-party platforms like cryptocurrency exchanges and do not possess their own Ethereum clients). In any case, we are sure there is some room for offline gas estimators.

In this section, we investigate the potential benefits of using offline gas estimators. In particular, we test the `solc` native gas estimator (`-gas`) on a data set of 10 contracts. We leave other similar tools to further studies.

In Table 7, we show the potential improvements of gas estimator in two groups: 1) with respect to public functions (`Function`); and 2) with respect to message calls (`Instance`).

For each group, from left to right, the values are read: 1) number of instances in our data set (*All*); 2) number of instances `solc` helps to prevent (*Solve*); 3) the extent `solc` can help (*Ratio*). In this experiment, we use a `v0.4.25` version `solc` compiler since both contracts in Table 7 only accepts compiler version `v0.4.x`. In doing this test, we assume the contract code is fixed and we want to refer to `solc` gas estimator to properly set transaction gas limits.

We have the following observations:

- 1) As for public functions, `solc` can help in preventing nearly half of the gas exceptions. In other words, considering an average contract, `solc` gives meaningful estimations for about half of the public functions. Note, the `solc` gas estimator is so conservative that it rejects any function with any kind of loops (e.g., reading from a dynamic array) or unbounded calls. Thus the results it returns should be always exact upper bound for certian functions[11].
- 2) When considering transaction distribution, `solc` seems do not have any applaudable effects. In particular, as shown by contract `0xd0a6~7ccf`, not

[11]Except for some corner cases where `solc` may return underestimate readings because it ignores potential gas costs because of additional storage requirements.

Table 6 A selected list of 10 smart contracts in our sample data set with verified source code available in Etherscan, ranking by number of receiving out of gas exceptions

Contract	**Name**	**Instance**	**Compiler**		
			Version	*Optimize*	*Run*
`0x744d70FDBE2Ba4CF95131626614a1763DF805B9E`	SNT	81,830	`v0.4.11`	`YES`	200
`0xd0a6E6C54DbC68Db5db3A091B171A77407Ff7ccf`	EOSSale	44,721	`v0.4.11`	`NO`	200
`0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`	KittyCore	38,118	`v0.4.18`	`YES`	200
`0x8d12A197cB00D4747a1fe03395095ce2A5CC6819`	EtherDelta	35,300	`v0.4.9`	`YES`	200
`0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C`	SaleClockAuction	33,486	`v0.4.18`	`YES`	200
`0x419D0d8BdD9aF5e606Ae2232ed285Aff190E711b`	Token	30,622	`v0.4.11`	`YES`	200
`0x86Fa049857E0209aa7D9e616F7eb3b3B78ECfdb0`	DSToken	26,523	`v0.4.11`	`NO`	200
`0x5EdC1a266E8b2c5E8086d373725dF0690af7e3Ea`	YottaCoin	23,943	`v0.4.24`	`NO`	0
`0xB68042de5B3dA08a80C20d29aEFab999D0848385`	IDAGToken	20,165	`v0.4.23`	`NO`	200
`0x331d077518216c07C87f4f18bA64cd384c411F84`	EToken2	20,079	`v0.4.8`	`YES`	200

a single of the exception instance can be saved with help `solc`. The reason for this is these instances all calls to functions that are not covered by `solc` (so it cannot give any useful information *w.r.t.* gas cost). Note, the test instances are all collected from our previously found out of gas transactions, so the result shown here is skewed towards hard cases where loops and unbounded calls exist, and may not be fair to `solc`. However, what is clear is that if we want to solve those real-world out of gas problems, `solc` estimator alone is far from useful, and we need more powerful tools for this purpose.

Code optimizer

Besides a built-in gas cost estimator, the `solc` compiler also provides a native code optimizer which could be turned on with `-optimize` option. This optimizer is designed to work on the assembly level, trying to reduce redundancies and rearrange bytecode in hope that the output code could be lighter and more gas efficient. In general, this smart contract optimization problem is a multi-objective optimization, so that the result bytecode is both small in size as well as cheap in execution (i.e., consumes less gas units when called). To help make the right balance between these two targets, users can provide an additional parameter to the optimizer with `-optimize-runs` option (which defaults to 200), representing the expected average number of invocation for each function. Thus, by setting larger `-optimize-runs` parameters, users expect more frequent function executions, and the optimizer should produce code more suitable for these high-frequency use cases. In contrast, smaller `-optimize-runs` parameters represent less active invocations, and should produce code optimized to initial deployments (which cost gas units when the contract is deployed).

In this section, we investigate the use of `solc` native optimizer to help prevent out of gas exceptions. In general,

Table 7 `solc` native gas estimator in use of preventing out of gas exceptions

Address	**Contract**	**Function**	**Instance**				
		All	*Solve*	*Ratio*	*All*	*Solve*	*Ratio*
`0x744d70FDBE2Ba4CF95131626614a1763DF805B9E`	SNT	26	10	38.5%	**81,830**	0	0%
`0xd0a6E6C54DbC68Db5db3A091B171A77407Ff7ccf`	EOSSale	30	17	56.7%	44,651	0	0%
`0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`	KittyCore	**60**	**46**	**76.7%**	38,118	1,899	5.0%
`0x8d12A197cB00D4747a1fe03395095ce2A5CC6819`	EtherDelta	27	17	63.0%	33,738	**2,187**	**6.5%**
`0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C`	SaleClockAuction	19	11	57.9%	33,041	0	0%
`0x419D0d8BdD9aF5e606Ae2232ed285Aff190E711b`	Token	28	9	32.1%	30,622	0	0%
`0x86Fa049857E0209aa7D9e616F7eb3b3B78ECfdb0`	DSToken	22	9	40.9%	26,011	24	0.1%
`0x5EdC1a266E8b2c5E8086d373725dF0690af7e3Ea`	YottaCoin	23	10	43.5%	23,943	0	0%
`0xB68042de5B3dA08a80C20d29aEFab999D0848385`	IDAGToken	20	12	60.0%	20,165	0	0%
`0x1F0480a66883De97d2b054929252aaE8F664c15c`	NePay	22	12	54.5%	16,333	0	0%

when configured correstly, the optimizer should provide a more gas efficient bytecode that consumes less gas when executed, thus lowering the risk of out of gas exceptions. We plan to evaluate other similar contract code optimizers in future work.

In Table 8, we show the improvements of turning on `-optimize` option for the same data set of 10 contracts as in "Gas estimator" section. The compiler we use is of version `v0.4.25` and the `-optimize-runs` parameter is set to 200. Note, in doing this test, we assume the gas limit of each transaction is fixed, and to see if we can avoid gas exceptions by using optimized contract code.

As before, we divide the results into two parts: 1) improvements with respect to public functions (`Function`); and 2) improvements with respect to individual transactions (`Instance`). In the `Function` part, we present three values, i.e., total number of public functions (*All*), number of public functions with increasing gas consumptions (*Up*), and number of public functions with decreasing gas consumptions (*Down*). Whereas in the `Instance` part, we choose the same format as in Table 7, reporting total number of exception transactions (*All*), number of transactions that can be fixed (*Solve*), and the relative scale of fixed transactions (*Ratio*). Note, as can be seen in Table 6, six out of the ten contracts in this test have already turned on `-optimize` option when deployed and that the `-optimize-runs` parameters are all set to 200 just as in our experiment.

From Table 8, we get the following observations:

- 1) All four smart contract with `-optimize` option previously turned off have seen changes of each public function's gas cost. For example, the `0xd0a6~7ccf` contract experience a rise of costs in half of the public functions, whereas it only sees cost reduction in two functions. Oppositely, the `0xB680~8385` contract will have half of the functions cutting down gas costs, and with one exception to increase cost. This suggest that code optimizer can at least modify gas costs for different functions, and this may lead to a trade-off between adding costs to some functions and at the same time reducing to some others.
- 2) While the code optimizer do have some effects in changing function's gas cost, it seems have little effect to really prevent gas exceptions. Again, look the four underlined contracts, these contracts all have seen some reduction of gas costs (at least for some functions), but it turns out that not a single exception transaction can be fixed just because of gas cost reduction. The reasons may be that previous transactions have set a too lower gas limit that out optimization can not manage to save, or that the functions with significant cost reduction are just not those hotspots for out of gas exceptions.
- 3) When look at the `Instance` part, we find that only one contract (the `0x06012~266d`) seems to be sensitive to the use of code optimization techniques. In fact, not a single function in this contract has seen changes in gas cost, and the appearance of these transactions is just a byproduct of the underestimate of `solc` gas estimator. In Table 8, we calculate the *Solve* of `Instance` by comparing `solc` gas estimations with actual gas limits. Since the estimator may return underestimated reading in certain cases, these transactions show up as false positives.

Other approaches

Besides estimating transaction gas cost before submitting, other methods exist to help users prevent out of gas exceptions. One promising approach is to generate more gas-optimized bytecode so contracts could use less gas during their execution. For example, `solc`

Table 8 `solc` built-in code optimizer in use of preventing out of gas exceptions, with `-optimize` option turned on and the `-optimize-runs` parameter set to 200

Address	Contract	Function	Instance				
		All	*Up*	*Down*	*All*	*Solve*	*Ratio*
`0x744d70FDBE2Ba4CF95131626614a1763DF805B9E`	SNT	26	0	0	81,830	0	0%
`0xd0a6E6C54DbC68Db5db3A091B171A77407Ff7ccf`	EOSSale	30	**15 ↑**	**2 ↓**	44,651	0	0%
`0x06012c8cf97BEaD5deAe237070F9587f8E7A266d`	KittyCore	60	0	0	38,118	**375**	**1.0%**
`0x8d12A197cB00D4747a1fe03395095ce2A5CC6819`	EtherDelta	27	0	0	33,738	0	0%
`0xb1690C08E213a35Ed9bAb7B318DE14420FB57d8C`	SaleClockAuction	19	0	0	33,041	0	0%
`0x419D0d8BdD9aF5e606Ae2232ed285Aff190E711b`	Token	28	0	0	30,622	0	0%
`0x86Fa049857E0209aa7D9e616F7eb3b3B78ECfdb0`	DSToken	22	**4 ↑**	**5 ↓**	26,011	0	0%
`0x5EdC1a266E8b2c5E8086d373725dF0690af7e3Ea`	YottaCoin	23	**8 ↑**	**2 ↓**	23,943	0	0%
`0xB68042de5B3dA08a80C20d29aEFab999D0848385`	IDAGToken	20	**1 ↑**	**11 ↓**	20,165	0	0%
`0x1F0480a66883De97d2b054929252aaE8F664c15c`	NePay	22	0	0	16,333	0	0%

provides an option to perform code optimization, i.e., the `-optimize` option, accompany with a modifiable empirical optimization parameter, i.e., the `-optimize-runs` which specifies the expected number of invocation for each contract function. Other useful tools are proposed to detect and rectify under-optimized code fragments [23], or to generate gas-optimization-centric code from exists bytecode [25]. Last but not least, another approach for defending out of gas exceptions is to find ill-coded contracts before the deployment, so that deployed contracts will not contain any potential vulnerabilities that may trigger out of gas exceptions [21]. We propose to investigate these tools in further studies.

Summaries and implications

Based on the prior results and discussions, we summarize the important findings as well as further implications in Table 9. Besides, we also point out relative parties or stakeholders who may have interest in each finding and implication, e.g., BCP developers, BCP end users, development tool producers, blockchain researchers, etc.

Related work

Decentralized application

Decentralized application (dApp) and blockchain-based cloud application (BCP) can roughly be seen as the same type of application, where blockchain and smart contract implement part of the critical program logic.

The study of decentralized application (dApp), or blockchain-based cloud application, has recently grown popular in academia [5, 10, 35–43], a trend accompany with increasing public interests, extensive social media exposures, as well as phenomenal applications continuously coming out, where notably popular dApps are like CryptoKitties, Ethereum Name Service, My Crypto Heros, MakerDAO, and etc.

Wu et al. [35, 43] conducted an empirical study on blockchain-based decentralized applications (i.e., dApp) in Ethereum with 995 dApps across 17 different categories. According to their study, Ethereum dApps with financial implications (i.e., Exchange, Finance, and Gambling) are much more popular than others. The same phenomenon repeatedly occurs considering of both user accounts as well as transaction communications. The authors also investigated the degree of open source for Ethereum dApps. Results show that only a small fraction of dApps (15.7%) are fully open source in terms of both project code and smart contracts, whereas slightly less than half (43.5%) have all smart contracts open sourced. Even considering smart contracts, we notice that more than half of dApps do not provide all the source code, while some may publish part of the code. The fact suggests there are much room for open source movement in the dApp ecosystem to reach the promising future of blockchain-based decentralized applications. Last, the same work also summaried common design patterns for dApp smart contracts and gas cost related issues involving dApps.

There are some work considering the development methodology of dApps. Marchesi et al. [40] proposed an agile software development methodology for dApps, a process to gather requirement, analyze, design, develop, test, and deploy these applications. The authors presented detailed processes, design considerations, and tooling amendments with suitable tutorials. Ellul et al. [37] presented a unified programming model for dApp development, allowing developers to build such systems through a single code artifact.

Other work dedicates the application of dApp in various use cases. Taş et al. [42] use an example dApp to explain architectural considerations and useful tools for dApp development. Tian et al. [10] proposed a secure decentralized framework for truth discovery in the filed of crowdsourcing with a privacy preserving and reliable implementation. Johnson et al. [38] showed a new dApp solution for secure biomedical data sharing based on the Oasis Devnet, a privacy preserving blockchain compatible with EVM. The authors also compared traditional solutions with dApp solution, showing both advantages and disadvantages of their dApp. Chen et al. [36] presented a lottery dApp with multiple randomness sources (i.e., contract state, blockchain state, and off-chain commitments), which are more secure (in terms of predictability of random values) than existing similar dApps. Lee et al. [39] showed an Android APK forgery discrimination dApp on Hyperledger Fabric blockchain, leveraging the tamper-proof characteristic of blockchain Chen et al. [5].

Ethereum gas mechanism and out of gas exception

The gas mechanism is an important feature of Ethereum, which is designed as a solution to the general liveness problem of smart contract enabled blockchain system. By limiting the maximal available gas unit of individual transaction, this gas mechanism can effectively prevent Ethereum from being stuck by (whether benign or malicious) slow-running or never-ending transactions. However, on the other side of the coin, when insufficient gas units are provided, transactions are doom to a kind of runtime exceptions, i.e., the out of gas exceptions.

There are a number of related studies concerning the Ethereum gas mechanism and out of gas exceptions, ranging from code optimization, vulnerability identifying, gas estimation, and cost adjustment.

Wu et al. [35] investigated the actual gas cost *w.r.t.* different dApps. Specifically, they found dApps using the single contract architecture tend to consume less gas in average, as compared to leader-member, equivalent, and factory patterns. Besides, as for deployment gas cost,

Table 9 Summary of important findings and implications

Findings	**Implications**	**Relevant Parties**
`I.` Out of gas and explicit revert are most commonly seen exceptions in Ethereum, which together account for 90% of all occurrences both in terms of exception instances as well as external transactions.	Insufficient gas limit is very common to encounter, as well as both `require` and `assert` failures. Developers and end users should pay special attention to potential gas exceptions.	BCP developers BCP end users
`II.` During the infamous DoS attacks between block #2, 250, 000 and #2, 750, 000, an exploit of 58,517 transactions has triggered 150,717,728 gas exceptions (or 2,576 per transaction), which has significantly skewed the ordinary distribution of different exception types.	One transactions may trigger multiple number of gas exceptions, with use of restricted gas limit per internal call. Concentration of gas exceptions may be evidence to deliberate attacks against the platform or smart contracts. Investigation on gas exceptions should intentionally distinguish between attack related instances and other cases.	Blockchain Researchers Security Researchers
`III.` Since inception, out of gas exceptions alone have caused more than 3,000 `ETH` losses, or approximately several hundred thousand US dollars in worth. On average, every block sees an instance of gas exception. In other words, precious transaction slots are wasted in a one-slot-per-block manner.	The accumulated negative effects of gas exceptions are huge enough that developers, end users, and operators cannot ignore. By following appropriate guidance, it is possible to save money and time for blockchain-based cloud application participants.	BCP developers BCP end users Blockchain Researchers
`IV.` Even until very recently (block #8, 547, 396, or Spet. 14th, 2019), the frequency of gas exceptions do not see significant changes, especially in the most recent times. In other words, gas exceptions do appear in a relatively steady rate regardless of new methods or best-practices proposed for out of gas exception mitigation.	There may be several explanations. First of all, new tools or practices in gas exception mitigation are not applied broadly among relevant participants, which may results from lack of acceptance or delayed adoption. Second, smart contract code is not frequently updated, so that existing gas issues take action again and again. Thus, improve the acceptance of new approaches as well as regularly updates of contract code should be very important.	BCP developers BCP end users Blockchain Researchers
`V.` By comparing smart contracts with externally owned accounts, we find the former are more susceptible to out of gas exceptions, in the sense that gas exceptions are more concentrated on smart contracts than externally owned accounts. Besides, the receivers of gas exception transactions are more concentrated on small set of contracts, whereas the senders tend to be more diverse.	A few popular smart contracts tend to send and receive large number of gas exception transactions, suggesting developers to pay more attention to gas exception related issues during contract development, such as set a larger gas limit to inter-contract invocations or add additional safeguards to unexpected gas exceptions, especially when integrating with popular established libraries.	BCP developers
`VI.` The precompiled smart contract with address `0x04` (which act as identity function for inputs) is responsible for a large number of gas exceptions, although each with very little gas units, typically 3 units per (internal) transaction. Considering the mass scale and small influence, we believe this is linked to some issue of the Solidity compiler.	While we do not know the overall mechanism of this finding, it still suggests the critical role of smart contract compilers and other development tools in the cause and prevention of gas exceptions. Specifically, the developers of these tools should pay more attention to the potentially negative effect of their decisions on gas consumption issues.	Dev-tool developers
`VII.` There are transactions which trigger a large number of gas exceptions during execution, whereas the external transactions themselves do not run out of gas. In other words, gas exceptions happened deep in the call stack may not cause a cascading of exceptions in certain cases, e.g., the calling contract has set a fixed small gas limit to internal transactions.	Hidden gas exceptions are of particular interest to developers and researchers. On one hand, developers should be careful when calling other contract's functions, by setting appropriate gas limits and adding relative safeguards. On the other hand, hidden gas exceptions may be byproduct of critical vulnerabilities or attacks (like in the infamous Ethereum DoS attacks [29, 30]).	BCP developers Blockchain researchers

Table 9 Summary of important findings and implications (*Continued*)

Findings	**Implications**	**Relevant Parties**
`VIII.` A recurring reason of gas exceptions is that the transactions are given too few gas units. This can further be divided into two categories: 1) leaving no gas units for any code execution; 2) setting conservative gas limits than actual needs.	When calling smart contracts (whether from EOA or other smart contract), try to provide more gas units than it seems to consume. For example, always add an additional 5,000 units to the gas consumption result of transaction simulations, or use a sophisticated gas estimator that is proven to return a strict overestimate reading for gas consumption.	BCP developers BCP end users
`IX.` According to experiment, the native gas estimator of `solc` tend to provide estimations of limited use in gas exception mitigation. The tool fails to produce meaningful output when encounters loops or unbounded calls, which however are the exact causes for many real-world out of gas transactions. On the other hand, online estimators should provide satisfactory results if used before each transaction, which is unfortunately not strictly followed, as shown by our results.	Always use Ethereum client's online gas estimation functionality before submitting new transaction, and if possible, consult more tools in providing gas cost estimations. Besides, there is a need for developing and promoting new tools for gas exception mitigation, like gas-oriented code optimization as well as sophisticated gas cost estimators.	BCP developers BCP end users

they found number of functions (NoF) and line of code (LoC) both contributes to the larger deployment gas cost, whereas number of functions is more related than line of code. As for execution gas cost, they reported that half of the transactions for dApps tend to provide less than 100,000 additional execution gas limit (i.e., these transactions end with 100,000 or less gas units available). And by setting transaction gas limit to 141,213 units, users are 80% sure that their transactions with end up without out of gas exceptions. Compared with et al. [35], our work are different in that we have a much larger dataset, i.e., all gas exception transactions from genesis block till very recently, whereas their work only considers a one-year time segment (i.e., the year of 2018) and the contract set are limited to those chosen 995 dApps. What's more, our work focuses on a comprehensive investigation on gas exception and related issues (i.e., compared with other runtime exceptions, identify the causing factors, and test related tools in prevent of gas exceptions), while [35] is more about finding the right gas limit in terms of dApp transactions.

Chen et al. [23] studied the use of Solidity language in writing smart contract. They identified seven gas-costly source code level patterns where the official Solidity compiler (`solc`) failed to optimize. These patterns are further classified into two groups: useless-code related patterns and loop-related patterns. They then built a tool called `GASPER` which can find three of these seven patterns using contract bytecode. In [25], the same author reported 24 bytecode level anti-patterns, and then built a contract optimizer named `GasReducer` baed on these anti-patterns. Unlike [23] and [25], our work focuses on out of gas exceptions, and we use an empirical analysis oriented methodology to find their consequences, their reasons, as well as challenges to existing tools or methods. While gas-costly patterns or anti-patterns may lead to out of gas exceptions, they are neither decisive nor complete.

Grech et al. [21] studied three smart contract vulnerabilities that are directly related to Ethereum gas mechanism. In particular, all these three vulnerabilities can be exploited by hackers to lock a target contract down, effectively making it unusable forever. The authors then devised a static analysis tool named `MadMax` to help find these gas-related vulnerabilities. Compared with [21], our work is more focused on out of gas exceptions themselves and the causing factors, whereas their work deal with identifying and preventing vulnerabilities stemming from out of gas exceptions. Besides, we also show that failing to specify an appropriate `tx.gasLimit` can also contribute to gas exceptions.

Albert et al. [22] proposed a gas analyzer for smart contracts named `GASTAP`, which can infer an upper bound for each function's gas cost. Experiments showed that `GASTAP` outperforms `solc`'s native gas estimators as it can deal with more complex situations where `solc` lacks support of. At the same time, Marescotti et al. [44] proposed a worst-case gas consumption estimation technique inspired by bounded model-checking techniques. Their method was built on top of the so-called gas consumption paths (GCPs), then they used SMT solver and EVM's gas consumption capabilities to retrieve concrete gas limits. However, since [44] lacks a tool implementation as well as subsequent experiments, we do not know its effectiveness on real world smart contracts.

Ma et al. [45] proposed a fuzzing-based approach to gas estimation and limit setting, which they give a name GasFuzz. The same approach can also be used to detect gas-related vulnerabilities. Compared with general fuzzing techniques, GasFuzz build itself on gas weighted control flow graph (CFG) and gas consumption guided selection and mutation strategies. Experiments show that GasFuzz significantly outperforms `solc` in gas cost estimation by reducing the risk of underestimation and out of gas exception. We are interested in GasFuzz as well as other fuzzing-based approaches to gas exception mitigation problem, and plan to compare them in the following work.

There is a gas cost alignment problem in Ethereum, which states that if the gas mechanism assigns much less gas cost for a certain instruction, then hackers could utilize the instruction to launch a DoS attack against the Ethereum network. Both Chen et al. [24] and Yang et al. [32] concluded that Ethereum's current gas mechanism, despite been changed many times, still left considerable rooms for misuse and DoS attacks. Besides, [24] also proposed an adaptive gas cost mechanism aiming at defending these potential DoS attacks.

Conclusion

In this work, we investigate gas exceptions on Ethereum blockchain-based cloud applications. By using instrumented Ethereum client, we collect a large data set of exception transactions as well as their execution traces. We then start by looking at the prevalence of different exceptions, where out of gas stood out with a large number of occurrences as well as money losses. Moreover, we summarize common causing factors for out of gas exceptions, with an emphasis on misunderstanding of transaction mechanism, conservative gas limit, and compiler derived bugs. At last, we investigate the effectiveness of existing tools in helping prevent out of gas exceptions. The results suggest further research and study on this topic.

Abbreviations

BCP: Blockchain-based cloud application; CFG: Control flow graph; dApp: Decentralized application; DeFi: Decentralized finance; DEX: Decentralized exchange; DOG: Deploy out of gas; DoS: Denial of service; ENS: Ethereum name service; EOA: Externally owned account; EOG: Execute out of gas; ETH: Ethereum (the smart contract platform) or Ether (the native cryptocurrency of Ethereum); EVM: Ethereum virtual machine; GCP: Gas consumption path; HBCP: Hybrid blockchain-based cloud application; IoT: Internet-of-things; LoC: Line of code; NoF: Number of functions; OG: Out of gas; PBCP: Pure blockchain-based cloud application; QoS: Quality of service; RQ: Research question; TTP: Trusted third-party; UTXO: Unspent transaction output

Acknowledgments

We thank Zhi Guan and Qingshan Li from Peking University for providing the computation resources, and for comments and suggestions that greatly improved this manuscript.

Authors' contributions

Chao Liu, Huihui Wang, and Zhong Chen conceived, designed, and directed this research. Chao Liu, Jianbo Gao, and Yue Li implemented the experiments, collected, analyzed, and checked the presented results. The author(s) read and approved the final manuscript.

Authors' information

Chao Liu Ph.D. candidate of Electronics Engineering and Computer Science at Peking University. He obtained his bachelor's degree of Computer Science and Technology at Peking University in 2014. His research interests include blockchain, program analysis, software engineering, network and information security, artificial intelligence.
Jianbo Gao Ph.D. student of Electronics Engineering and Computer Science at Peking University. He obtained his bachelor's degree of Computer Science and Technology at Peking University in 2016. His research interests include blockchain, program analysis, software engineering, network and information security.
Yue Li Ph.D. student of Electronics Engineering and Computer Science at Peking University. She obtained her bachelor's degree of Computer Science and Technology at University of Electronic Science and Technology of China in 2018. Her research interests include blockchain, program analysis, network and information security, cryptography.
Huihui Wang received her Ph.D. degree in electrical engineering from the University of Virginia, Charlottesville, VA, USA, in August 2013. In August 2013, she joined the Department of Engineering, Jacksonville University, Jacksonville, FL, USA, where she is currently an Associate Professor and the Founding Chair of the Department of Engineering. In 2011, she was an Engineering Intern with Qualcomm, Inc. She is the author of more than 50 articles and holds one U.S. patent. Her research interests include cyber-physical systems, Internet of Things, healthcare and medical engineering based on smart materials, robotics, haptics based on smart materials/structures, ionic polymer metallic composites, and microelectromechanical system.
Zhong Chen Ph.D., Professor of School of Electronics Engineering and Computer Science at Peking University, and Director of MoE Key Lab of Network and Software Assurance, Director of Financial Information Research Center of Peking University. He has awarded Beijing Excellent Teacher Award in 1996, 1st Prize National Higher Education Teaching Achievements Award in 2005, 2nd Prize National Science and Technology Award in 2010. His research interests include domain-specifc software engineering, network and information security, blockchain.

Funding

This work is fully funded by National Natural Science Foundation of China under the grant No.: 61672060.

Availability of data and materials

As described in "Methodology" section, we collected all the exceptional transactions (i.e., transactions encountering any runtime exceptions) of Ethereum mainnet from genesis block till block #8, 547, 396. The collected data contains detailed transaction information, e.g., source, destination, value, input, gas limit, block number, etc. Besides, we also collected transaction traces as complementary. All these data are stored and analyzed in a MongoDB instance running on the same server as described in "Methodology" section. We plan to publish the above data (not including transaction traces) online later when we finish follow-up tasks like data extraction and uploading. Besides the above transaction data collected by ourselves, we also consult historical Ether price data provided by Etherscan which can be downloaded from link [46]. As for the tested tools, we used `solc` of version `v0.4.25` which can be downloaded from Github through link [47], and the compiling and installation instructions can be referenced from [48]. The dataset used to test existing tools will also be available online soon after follow-up tasks.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]School of Electronics Engineering and Computer Science, Peking University, No.5 Yiheyuan Road, 100871, Beijing, China. [2]Department of Engineering, Jacksonville University, Jacksonville, FL 32211, USA.

Received: 24 February 2020 Accepted: 21 May 2020
Published online: 30 June 2020

References

1. Butun I, Österberg P, Song H (2019) Security of the internet of things: vulnerabilities, attacks and countermeasures. In: IEEE Communications Surveys & Tutorial. https://doi.org/10.1109/COMST.2019.2953364
2. Song H, Srinivasan R, Sookoor T, Jeschke S (2017b) Smart cities: foundations, principles, and applications. Wiley, Hoboken. pp. 1–106

3. Song H, Fink GA, Jeschke S (2017) Security and Privacy in Cyber-Physical Systems: Foundations, Principles and Applications. Wiley-IEEE Press, Chichester. pp. 1–472
4. Alharby M, Aldweesh A, van Moorsel A (2018) Blockchain-based smart contracts: A systematic mapping study of academic research 2018. In: Proceedings of the 2018 International Conference on Cloud Computing, Big Data and Blockchain. https://doi.org/10.1109/iccbb.2018.8756390
5. Chen J, Lv Z, Song H (2019a) Design of personnel big data management system based on blockchain. Futur Gener Comput Syst 101:1122–1129. https://doi.org/10.1016/j.future.2019.07.037
6. Kosba A, Miller A, Shi E, Wen Z, Papamanthou C (2016) Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE symposium on security and privacy (SP). IEEE. pp 839–858. https://doi.org/10.1109/sp.2016.55
7. Kumaresan R, Bentov I (2014) How to use bitcoin to incentivize correct computations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 30–41. https://doi.org/10.1145/2660267.2660380
8. Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM. pp 254–269. https://doi.org/10.1145/2976749.2978309
9. Miers I, Garman C, Green M, Rubin AD (2013) Zerocoin: Anonymous distributed e-cash from bitcoin. In: 2013 IEEE Symposium on Security and Privacy. IEEE. pp 397–411. https://doi.org/10.1109/sp.2013.34
10. Tian Y, Yuan J, Song H (2019) Secure and reliable decentralized truth discovery using blockchain. In: 2019 IEEE Conference on Communications and Network Security (CNS). IEEE. pp 1–8. https://doi.org/10.1109/cns.2019.8802712
11. (2019b) Home - enterprise ethereum alliance. https://entethalliance.org
12. Androulaki E, Barger A, Bortnikov V, Cachin C, Christidis K, De Caro A, Enyeart D, Ferris C, Laventman G, Manevich Y, et al. (2018) Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. ACM. p 30. https://doi.org/10.1145/3190508.3190538
13. Cheng R, Zhang F, Kos J, He W, Hynes N, Johnson N, Juels A, Miller A, Song D (2019) Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE. pp 185–200. https://doi.org/10.1109/eurosp.2019.00023
14. Wood G, et al. (2014) Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj Yellow Pap 151(2014):1–32
15. (2019) Bitcoin cash - peer-to-peer electronic cash. https://www.bitcoincash.org
16. (2019) Litecoin - open source p2p digital currency. https://litecoin.org
17. (2019) Privacy-protecting digital currency - zcash. https://z.cash
18. (2019) Libra - a new global currency. https://libra.org/en-US/
19. Nakamoto S, et al. (2008) Bitcoin: A peer-to-peer electronic cash system. https://doi.org/10.2139/ssrn.3440802
20. (2019c) A next-generation smart contract and decentralized application platform, ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper
21. Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y (2018) Madmax: Surviving out-of-gas conditions in ethereum smart contracts. Proc ACM Program Lang 2(OOPSLA):116
22. Albert E, Gordillo P, Rubio A, Sergey I (2018) Gastap: A gas analyzer for smart contracts. arXiv preprint. arXiv:181110403
23. Chen T, Li X, Luo X, Zhang X (2017a) Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. pp 442–446. https://doi.org/10.1109/saner.2017.7884650
24. Chen T, Li X, Wang Y, Chen J, Li Z, Luo X, Au MH, Zhang X (2017b) An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In: International Conference on Information Security Practice and Experience. Springer. pp 3–24. https://doi.org/10.1007/978-3-319-72359-4_1
25. Chen T, Li Z, Zhou H, Chen J, Luo X, Li X, Zhang X (2018) Towards saving money in using smart contracts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER). IEEE. pp 81–84. https://doi.org/10.1145/3183399.3183420
26. (2019) Defi - best decentralized finance projects | what is defi in crypto. https://defiprime.com
27. Lamport L (1978) The implementation of reliable distributed multiprocess systems. Comput Netw (1976) 2(2):95–114
28. (2019a) Contracts - solidity 0.5.11 documentation: Creating contracts. https://solidity.readthedocs.io/en/v0.5.11/contracts.html#creating-contracts
29. (2016a) Ethereum continues to suffer from ddos attacks. https://www.ethnews.com/ethereum-continues-to-suffer-from-ddos-attacks
30. (2016b) Transaction spam attack: Next steps. https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/
31. (2019) Security alert: Ethereum constantinople postponement. https://blog.ethereum.org/2019/01/15/security-alert-ethereum-constantinople-postponement/
32. Yang R, Murray T, Rimba P, Parampalli U (2019) Empirically analyzing ethereum's gas mechanism. arXiv preprint arXiv:190500553. https://doi.org/10.1109/eurospw.2019.00041
33. (2019) Go ethereum - official go implementation of the ethereum protocol. https://geth.ethereum.org
34. (2019b) Expressions and control structures - solidity 0.5.11 documentation: Error handling: Assert, require, revert and exceptions. https://solidity.readthedocs.io/en/v0.5.11/control-structures.html#error-handling-assert-require-revert-and-exceptions
35. Wu K, Ma Y, Huang G, Liu X (2019) A first look at blockchain-based decentralized applications. Software: Practice and Experience. https://doi.org/10.1002/spe.2751
36. Chen YC, Hsu SY, Chang TW, Wu TW (2019b) Lottery dapp from multi-randomness extraction. In: 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE. pp 78–80. https://doi.org/10.1109/bloc.2019.8751323
37. Ellul J, Pace G (2019) Towards a unified programming model for blockchain smart contract dapp systems. In: 2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW). IEEE. pp 55–56. https://doi.org/10.1109/srdsw49218.2019.00017
38. Johnson M, Jones M, Shervey M, Dudley JT, Zimmerman N (2019) Building a secure biomedical data sharing decentralized app (dapp): Tutorial. J Med Internet Res 21(10):e13,601
39. Lee HW, Lee H (2019) Consortium blockchain based forgery android apk discrimination dapp using hyperledger composer. J Internet Comput Serv 20(5):9–18
40. Marchesi L, Marchesi M, Tonelli R (2019) Abcde–agile block chain dapp engineering. arXiv preprint arXiv:191209074
41. Scholten OJ, Hughes NGJ, Deterding S, Drachen A, Walker JA, Zendle D (2019) Ethereum crypto-games: Mechanics, prevalence, and gambling similarities. In: Proceedings of the Annual Symposium on Computer-Human Interaction in Play. pp 379–389. https://doi.org/10.1145/3311350.3347178
42. Taş R, Tanrıöver ÖÖ (2019) Building a decentralized application on the ethereum blockchain. In: 2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). IEEE. pp 1–4. https://doi.org/10.1109/ismsit.2019.8932806
43. Wu K (2019) An empirical study of blockchain-based decentralized applications. arXiv preprint arXiv:190204969
44. Marescotti M, Blicha M, Hyvärinen AE, Asadi S, Sharygina N (2018) Computing exact worst-case gas consumption for smart contracts. In: International Symposium on Leveraging Applications of Formal Methods. Springer. pp 450–465. https://doi.org/10.1007/978-3-030-03427-6_33
45. Ma F, Fu Y, Ren M, Sun W, Liu Z, Jiang Y, Sun J, Sun J (2019) Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. arXiv preprint arXiv:191002945
46. (2019a) Ether daily price (usd). https://etherscan.io/chart/etherprice?output=csv
47. (2019c) Github - solidity version 0.4.25. https://github.com/ethereum/solidity/releases/tag/v0.4.25
48. (2019d) Solidity - solidity 0.4.25 documentation. https://solidity.readthedocs.io/en/v0.4.25/

Publisher's Note