As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service

Kai Li* Jiaqi Chen* Xianghong Liu* Yuzhe Tang* ✉ XiaoFeng Wang† Xiapu Luo‡
* Syracuse University. Emails: {kli111,jchen217,xliu167,ytang100}@syr.edu
† Indiana University Bloomington. Email: xw7@indiana.edu
‡ Hong Kong Polytechnic University. Email: csxluo@comp.polyu.edu.hk

***Abstract*—Modern blockchains have evolved from cryptocurrency substrates to trust-decentralization platforms, supporting a wider variety of decentralized applications known as DApps. Blockchain remote procedure call (RPC) services emerge as an intermediary connecting the DApps to a blockchain network. In this work, we identify the free contract-execution capabilities that widely exist in blockchain RPCs as a vulnerability of denial of service (DoS) and present the DoERS attack, a Denial of Ethereum RPC service that incurs zero Ether cost to the attacker.**

To understand the DoERS exploitability in the wild, we conduct a systematic measurement study on nine real-world RPC services which control most DApp clients' connection to the Ethereum mainnet. In particular, we propose a novel measurement technique based on orphan transactions to discover the previously unknown behaviors inside the blackbox RPC services, including load balancing and gas limiting. Further DoERS strategies are proposed to evade the protection intended by these behaviors.

We evaluate the effectiveness of DoERS attacks on deployed RPC services with minimal service interruption. The result shows that all the nine services tested (as of Apr. 2020) are vulnerable to DoERS attacks that can result in the service latency increased by $2.1X \sim 50X$. Some of these attacks require only a single request. In addition, on a local Ethereum node protected by a very restrictive limit of 0.65 block gas, sending 150 DoERS requests per second can slow down the block synchronization of the victim node by 91%.

We propose mitigation techniques against DoERS without dropping service usability, via unpredictable load balancing, performance anomaly detection, and others. These techniques can be integrated into a RPC service transparently to its clients.

I. Introduction

With the advent of operational blockchains, decentralized applications (*DApps*) running atop these systems are gaining popularity, providing decentralized finances (DeFi), online gaming, information-security infrastructures, etc. A typical DApp, as illustrated in Figure 1, is architectured in three layers: *DApp clients* running inside web browsers send requests to a *Remote Procedure Call* (*RPC*) service that translates the clients' requests to cryptocurrency transactions or queries to a *blockchain P2P network*. Such a RPC service operates on a blockchain full node maintained directly by the DApp owner (i.e., an in-house RPC node) or a set of nodes hosted by a third party (i.e., a third-party RPC service) intended to ease DApp deployment. Given the ever-growing blockchain states (e.g., 130 GB and 1.8 TB for a fully synced and an archived Ethereum node, respectively, as of 2018), the RPC service plays an increasingly important role in the DApp ecosystem, scaling DApp clients to low-end mobile devices and web browsers. Major blockchains today flock to roll out RPC supports, which spawn a good number of services in practice, including nine service providers (as is evaluated in this work) supporting the Ethereum's JSON-RPC interface [16], blockchain.info [5] with Bitcoin's JSON-RPC [2], dfuse.io [13] and greymass.com [39] with EOSIO's Chain API [6], stellar.org [46] with Stellar Horizon [23], etc. These services host the majority of DApps; for instance, at least 63% of Ethereum based DApps use one RPC service [10].

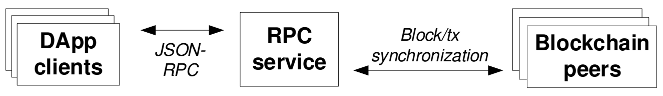


Fig. 1: The system of blockchain RPC service

Despite its importance, the RPC service is less decentralized (one to hundreds of nodes) than the blockchain network (of tens to hundreds of thousands of nodes) and therefore could become a single point of failure should a denial of service (DoS) attack happen, which could lead to the collapse of the whole DApp ecosystem. It is important to note that DoS is known to pose a significant threat to the blockchain ecosystem, particularly in Bitcoin exchanges and mining pools [56], [64], [52]. We believe such an attack can also be launched against a victim RPC service, allowing a service competitor to steal customers from the victim. Besides, the perpetrator who denies a RPC service can illicitly manipulate the transaction order of a financial DApp [51], [49] and gain profit. For instance, in an auction for registering an Ethereum domain [17] or purchasing a CryptoKitty [7], a bidder can delay others' bidding transactions through denying the RPC service they use to win the auction at an unfairly low price. As another example, a client depositing to a hash-time-lock contract (HTLC), as widely used in blockchain applications (e.g., atomic intra-chain or cross-chain swaps [55] and payment channels [31], [59]), can defer the withdrawal of the deposit after the expiration of

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23108
www.ndss-symposium.org

the lock (so-called grieving [51]) by denying the RPC service used by the withdrawal, thus retaining the deposit. With such significant implications, this security risk, however, has never been studied before.

Menace of DoERS. Our research shows that indeed this risk is realistic and serious: today's RPC services are vulnerable and can be easily disabled by a new type of DoS attacks (which we call *DoERS* or Denial of Ethereum Rpc Service) that exploit the free execution capability they expose. More specifically, blockchain systems support the Gas-free execution of a smart contract on an individual RPC node, such as Ethereum's `eth_call` RPC [28] (and `eth_estimateGas` [29] running the same code path). An `eth_call` can be triggered to run any smart contracts including those reading and/or updating their states. Unlike the transaction-triggered smart contract execution, the `eth_call`-triggered execution occurs locally on the recipient RPC node, and its state update, if any, will not be propagated to or reflected in the global blockchain state. The purpose of such a capability is to enable a variety of real applications in pre-production contract testing (e.g., estimating the smart-contract cost before DApp deployment by `estimateGas`), "stored-procedure" like database analytics on blockchain (e.g., the GraphQL queries [12] and decentralized financial analysis [20]), and others. Most importantly, `eth_call` is often free: the charge for its execution is not mandated by Ethereum and instead left to individual RPC services, which tend to waive the expense for attracting DApp clients, as observed in practice. Therefore, the adversary can deploy on the blockchain an attack smart-contract involving a resource-consuming procedure (e.g., an infinite loop of hashing computations) and then trigger it through `eth_call`. This attack is shown in our research to effectively stop a node from performing critical operations for all DApps it hosts, including block/transaction synchronization, serving RPC requests, etc.

Notably, DoERS is different from other DoS attacks on the blockchain network, as studied in the prior research [62], [42], [21], [60]. First, it aims at disrupting the communication channel between a blockchain and its DApps by blocking third-party RPC services (in Figure 1), not taking down the blockchain itself as the other attacks do. Second, our attack exploits a unique weakness – Gas-free contract execution on RPC-enabled Ethereum nodes (Section II), while existing DoS attacks seek under-priced instructions for attacking replicated smart-contract execution [62], [42], [21] or misusing mining mechanisms [60].

The attack is non-trivial to carry out. We need to overcome the protection already in place on each Ethereum node, such as limiting each call's Gas (i.e., Gas limit) and time (i.e., timeout), through strategically delivering continuous queries at an alarmingly low rate below the victim's rate limit (§ III). Also it is less clear how extensively RPC interfaces are open to the public on the nodes operated by the DApp owners. Even more challenging is the use of third-party RPC services, which typically run a load balancer in front of RPC nodes. Such a balancer hides the node(s) serving a specific DApp and spread out its clients' requests using undisclosed strategies. Understanding how it works is critical to the success of an attack targeting a specific DApp or a specific client of the DApp. For this purpose, we performed an analysis and measurement study on Ethereum.

Measurement and findings. More specifically, there are three types of nodes in an Ethereum network: the nodes not accepting any RPC requests (non-RPC nodes), the nodes with public RPC ports, responding to the requests from any web clients (public RPC nodes), and the nodes with private RPC ports, only communicating with specific web servers (private RPC nodes). The majority of the private nodes are the backends of third-party RPC services such as ServiceX6[1] and ServiceX5. Since most DApps rely on these well established services to connect to Ethereum [10], such private nodes play the critical role in controlling connections between DApp clients and the Ethereum network. Thus, our research focuses on measuring these private nodes.[2]

We first detected the presence of Gas limits on the private RPC nodes in nine leading RPC services. We proposed a detection technique that makes `eth_call`s with varying Gas amounts and performs a binary search to find out the Gas limit. The measurement reveals that five out of the nine major RPC services do not configure Gas limits of any kind and the other four set a rather nonrestrictive limit of more than 1.5 block gas.

Further we looked into the load balancers deployed by the nine third-party RPC services. To reverse-engineer their operations, we developed a novel probing technique based upon *orphan transaction*, which stays on one node without being propagated to others. Our approach delivers one orphan transaction through a given RPC service and then sends in the second one that attempts to double-spend the first. If both are assigned to the same node by the balancer, the second transaction will fail (as it double-spends the first one), and otherwise, it will also go through (as the two transactions reside on different nodes). By observing the transaction outcome, we systematically analyzed 9 popular services (§ IV-A). Our study reveals different load-balancing strategies, assigning requests to nodes according to the client's IP (e.g., ServiceX5), service API key (e.g., ServiceX4) or timings of RPC calls (e.g., ServiceX6). Based upon the discoveries, a DoERS attack can be adjusted to target a DApp, a client or the client's visit to a given DApp, depending on their RPC services' balancing strategies (§ IV-A). Among all 8 services, we conclude that 5 can be exploited to block a specific client or a DApp, without fully taking down the whole services. Our study of Gas limit reveals that another five RPC services out of the nine don't configure any Gas limit. Also interesting is our measurement of rate limits, which turns out to be less aligned with the

[1]In this paper, we refer to the nine services by numbered names from ServiceX1 to ServiceX9. We intentionally avoid use their real names to protect their identities and operations.

[2]Nevertheless, we also measure public RPC nodes and uncover 348 of them vulnerable in today's Ethereum mainnet running 8927 nodes (as of Apr. 2020). This additional measurement study is described in Appendix A.

public information (e.g., the measured rate limit of ServiceX5 is twice the one published on their website).

Attacks. In addition to the above targeted attacks, we design DoERS strategies that are specific to the measurement findings. For nodes without Gas limits, DoERS strategically sends a single RPC request exploiting the Ethereum Virtual Machine's (EVM's) `CODECOPY` instruction such that it evades all other known protections (in timeout, load balancing and rate limiting) until it crashes the victim node. This is caused by the EVM design of atomically executing instructions that even a thrown timeout cannot interrupt. For nodes with Gas limits, DoERS sets the "payload" size of each RPC call below the specific Gas limit and increases the request rate to cause visible damage. Because of the innate computation asymmetry between sending a RPC on the client and executing programs on the RPC node, the required request rate remains low as observed in our evaluation.

To verify the impacts DoERS will have on real Ethereum peers[3] and services, we performed a carefully-designed experiment on these services, in a way the effectiveness of the attacks can be observed without significantly downgrading their services (§ V-A). Our study reveals that all nine tested services are vulnerable to the DoERS attacks that cause noticeable performance degradation, increasing the service latency by $2.1X \sim 50X$. Notably, sending a single DoERS request can cause the latency to increase by $10X$ ($30X$) on ServiceX5 (ServiceX3), without triggering any exception. In addition, the evaluation study verifies that the proposed attack strategies can effectively evade the deterministic load balancing in services, such as ServiceX5 and ServiceX4.

We conduct experiments on an Ethereum peer under our control. The controlled experiments allow us to explore more extensively the combinations of attack parameters and to evaluate the effectiveness of attacks in the presence of out-of-gas, timeout and other exceptions. The study shows the attack can slow down block synchronization, in addition to causing latency increases. Particularly, we found sending DoERS RPCs at a rate as low as 150 per second under a very restrictive limit of 0.65 block gas was adequate to slow down block synchronization on a blade-server class machine by 91%.

Mitigation. Mitigating the DoERS threat without undermining the usability of the RPC service is nontrivial. For example, setting a low Gas limit for each request *alone* does not work well, as the protection can still be evaded by the attack using multiple DoERS requests. An exceedingly low Gas limit could also complicate the design of a DApp and downgrade its usability [19]. Indeed, we observed in our measurement study that no RPC service adopts a Gas limit below 1.5 block gas. So instead of capping the Gas usage for each request, we propose to limit the Gas for each service client, to defeat the multi-request attack. For this purpose, we addressed the challenge of identifying the requests from a specific client in an open-membership scenario using its performance profile and further developed the technique to capture performance anomalies.

[3]We will use "nodes" and "peers" interchangeably in this paper.

Another direction is to make the behavior of the frontend load balancer unpredictable (independently assigning each request to a randomly selected peer) so as to weaken a DoERS attack on a specific set of backend peers. The challenge here is that the balancer for today's DApp processes both RPC and transaction requests, and the latter may require certain dependency relations to be preserved: e.g., an ERC20 token's `approve` call and a subsequent `transferFrom` call should be assigned to the same peer to keep their order. This challenge has been addressed in our research with a DoERS-secure load balancer that differentiates transaction requests from RPC queries (including `eth_call`), so that the queries that could lead to DoERS are distributed independently across peers while the transaction requests are handled under their dependency constraints (e.g, order preservation).

In general, DoERS is enabled by *an open-membership RPC service that allows for free execution of arbitrary smart-contract programs on its peers shared by different DApps*. Invalidating any condition here can defeat the attack, but may also affect the fundamental security-usability trade-off expected from practical protection. Such trade-offs have been extensively discussed in the paper. Our proposed mitigation techniques can be built into a RPC service and are transparent to its clients (§ VI).

Contributions. The contributions of the paper are outlined as follows:

• *New attack*. We identify a new denial of service weakness in today's blockchain, showing that the widely existing free query calls enable a potential resource depletion attacks on RPC services, a weakest link of the DApp ecosystem. Also we implemented the attack on Ethereum incurring zero Ether costs and demonstrated the real-world impact of the threat across leading RPC services.

• *New understanding*. We performed a systematic measurement study on the nine leading RPC services that control the connection between most DApp clients and the Ethereum network. Our measurement on leading RPC services' load balancers, using a novel orphan-transaction based prober, has brought to light the hidden strategies they take, which enables targeted attacks on DApps and clients they serve.

• *Mitigation*. We also studied the potential mitigation on the new threat, identifying a few promising solutions, including the ones that selectively penalize the DApps or clients consuming a large amount of resources on a node.

Roadmap. The rest of the paper is organized as follows: the background is introduced in Section II; the research formulation is presented in Section III; the measurement of DoERS exploitability among RPC services are presented in Section IV; attack evaluation is presented in Section V; countermeasures are described in Section VI. Related works are presented in Section VII before discussion on responsible disclosure in Section VIII and conclusion in Section IX.

II. Background: Blockchain as a DApp Platform

Public blockchain is a distributed system that stores a ledger of "transaction" history on a peer-to-peer network. The P2P network is designed to scale, by admitting anyone on the Internet without identification (i.e., open membership) and by providing incentives in cryptocurrency reward to the nodes who "mine" in the network. Mining means that all participating nodes race to solve a puzzle and to decide which transactions to be included in the next block. Based on these mechanisms, real-world blockchains, including Bitcoin, Ethereum, EOS, etc., see a large operational P2P network (e.g., thousands to hundreds of thousands of peers) and enjoy a higher degree of trust decentralization than conventional systems.

Smart contracts and Gas: A smart contract is a user program running on a blockchain. While Bitcoin's contract, Script [4], is domain specific, more extensible blockchains including Ethereum and EOS support running Turing-complete smart contracts. On Ethereum, a client can request to run a smart contract with the provided arguments, if she pays a certain amount of fees known as Gas. The purpose of the Gas mechanism is to prevent denial of service to any Ethereum full nodes (as will be described in related work in § VII), which should be differentiated from this work whose goal is to DoS the RPC nodes. A smart contract (more precisely, the bytecode of the contract) is replicated to all Ethereum nodes and executing the contract is triggered by a transaction that propagates the invocation information to the Ethereum network. This transaction also specifies Gas price which indicates how much the client is willing to pay for each unit of the computation carried out in a smart contract. The higher Gas price, the faster the transaction gets propagated to the blockchain network.

DApp platform: The blockchain is widely used as a source of trust decentralization and underpins today's decentralized web applications, known as DApps. A DApp is typically a javascript program residing in a webpage that accesses information on the blockchain by invoking DApp-specific smart contracts. For instance, the CryptoKitties DApp [7] is a market for digital pets sales. Its system consists of an off-chain website that runs DApp javascript code [7] and five smart contracts on the blockchain [8]. Likewise, Melon terminal [33] is a financial DApp that runs financial-analysis in smart contracts on Ethereum and presents statistics in an off-chain webpage.

RPC services: To bridge the DApp web clients and blockchain, remote procedure call (RPC) services are essential. A RPC service accepts the JSON requests sent from a DApp client inside a web browser and translates them into queries or transactions. To do so, a RPC service internally runs one or a group of blockchain full nodes. Ethereum's RPC interface [16] includes 43 open queries and 3 privileged operations such as `sendTransaction`. A valid privileged operation must be sent from a cryptocurrency owner for signing the operation using her private key, while a RPC query can be open membership in that it can be sent by anyone on the Internet without identification. Unlike transactions propagated to the blockchain network, a RPC query is served locally within the service.

Speculative smart-contract execution (`eth_call`): The particular capability of interest to DoERS is Ethereum's speculative smart-contract execution in `eth_call` (and `eth_estimateGas` on the same code path in both Ethereum clients `Geth` [32] and `Parity` [27]). `eth_call` speculatively runs any smart contracts, in a different way from the conventional contract execution triggered by transactions. The difference is two-fold: 1) `eth_call`-triggered execution runs only on the serving RPC node and is not being propagated. This capability applies not only to the so-called `pure`/`view` functions [41], but also to any state-updating functions, as we tested in Ethereum Virtual Machine (EVM). The updated state however is not propagated and not reflected in the global blockchain state (hence the name, speculative execution). 2) `eth_call` does not mandate charging fees from a contract execution. Instead, such a decision is left to the hands of a service provider. In most practical RPC services, `eth_call` is offered free, as a means to attract DApp clients and developers, which is essential for growing their customer base.

Gas limit is a feature in Ethereum clients (e.g., `Geth` and `Parity`) that bounds the amount of Gas an individual `eth_call` invocation can consume.

III. The DoERS Attack and Research Formulation

Our threat model involves three actors: *an attacker* sends one or multiple malicious, crafted RPC requests to *an Ethereum RPC service* that also serves the regular RPC requests from *a benign client*. In practice, the benign client can be a DApp. The goal of the attacker is to deny the RPC service to the benign client, for instance, increasing its RPC response time. The Ethereum RPC service can be a single Ethereum node choosing to accept RPC requests (the basic model) or a a group of Ethereum nodes behind a frontend infrastructure (e.g., load balancing) to accept RPC requests (the third-party service model). This section considers the basic setting while the third-party service model is presented in § IV.

```
contract DoERS-C {
  function exhaustCPU(uint256 payload_size1) public returns
     (bool){
    bytes32 target=0xf...f;
    for (uint256 i=0; i<payload_size1; ++i){
      target = keccak256(abi.encodePacked(target));}
   return true;}
  bytes32[] storage;
  function exhaustIO(uint256 payload_size1) public returns(
     bool){
    for (uint256 j=0; j<payload_size2; ++j) {
       storage.push(0xf...f);}
    return true;}
  function exhaustMem(uint256 payload_size3) pure public
     returns(bool) {
    bytes32[] memory mem = new bytes32[](payload_size3);
    mem[payload_size3-1] = 0xf...f;//"CODECOPY" allocate
     memory
    return true;}}
```

Fig. 2: The exploitable smart contract to exhaust the computing resources (in CPU, memory allocation, etc.) of the victim node

The DoERS attack is constructed based on an exploitable smart contract that contains resource-consuming procedures. In this paper, we use the `DoERS-C` contract in Figure 2 as an example, while there can be many alternative designs — how to design the most "effective" smart contract for the attack is out of the scope of this paper. Contract `DoERS-C` includes three exploitable functions that aim at depleting CPU, memory and IO resources, respectively, on the victim node. Specifically, function `exhaustCPU` runs a loop of hashing computation. Function `exhaustIO` runs a loop of storage updates in order to incur IO operations; note that Variable `storage` is persisted in the smart-contract's storage. Function `exhaustMem` runs a single operation (EVM instruction `CODECOPY`) to allocate a large array in memory. The three functions all take an argument called payload size, which controls the number of iterations of the loop (in `exhaustCPU` and `exhaustIO`) and the size of the array (in `exhaustMem`). This argument is essentially a knob for tuning the level of resource consumption incurred by the smart contract.

The DoERS attack is executed in two steps: 1) The attacker client deploys the `DoERS-C` smart contract to Ethereum by sending a transaction. This step costs a small amount of Ether. 2) The attacker sends one or multiple `eth_call` RPCs to the victim node to trigger one of the three `exhaustXX` functions in `DoERS-C`. By specifying a large payload size, the execution of these functions incurs a large amount of resource consumption on the victim node. The purpose here is to cripple the node's functionality in block/transaction synchronization, serving co-siding RPCs, blockchain mining, etc. Since `eth_call` does not charge Ether (the main currency unit of Ethereum), the cost of the attack is low. We also describe a zero-Ether DoERS in § VI-A2 that eliminates the Ether cost in the first step.

In practice, the configurations of Ethereum nodes may thwart the above basic attack. For instance, Ethereum's Gas limit, if configured, would limit the amount of computation that can be incurred by each DoERS request. To evade the protection, a sophisticated attacker should lower the payload size to avoid triggering the Gas limit, and instead send multiple such smaller DoERS requests at a certain rate to make the service unavailable to the victim. Also, other protective measures could be in place to raise the bar for a successful DoERS attack, such as timeout, rate limiting, load balancing, as well as other unknown mechanisms inside the black box RPC services (e.g., performance isolation, hypothetically). Based upon this observation, we set the goal of our research as follows:

The goal of our research is to understand the risk of DoERS across deployed Ethereum RPC nodes and services. Particularly, we analyzed *the private nodes serving the backend of third-party RPC services* to measure their susceptibility to the attack, motivated by the fact that most DApp clients are connected to the Ethereum network through such third-party RPC services [10].

Towards the goal, 1) we conducted a systematic measurement study on nine leading RPC services on the market to analyze the behaviors of their load balancers, Gas limits and rate limits; 2) we designed the strategies that evade the protection discovered, in order to make the attack more effective (§ IV) and 3) we evaluated the impacts of our low-cost strategies on existing services and local nodes (§ V).

IV. Exploitability Measurements on RPC Services

This section describes our measurement study including methodology and results on real-world third-party services. Our goal is to understand the internal of a blackbox RPC service by measuring service features in load balancing, Gas limits and rate limiting, as modeled next.

Modeling a RPC service: A RPC service runs web servers on the frontend to accept JSON-RPC requests and run several Ethereum RPC nodes on the backend to process those requests. Each frontend web server may run rate-limiting and load balancing on the received requests. The service model is illustrated in Figure 3.

A. Measuring Blackbox Load Balancers: Methodology

1) Goals: To characterize a load balancer, we first describe a detailed model. A load balancer receives JSON-RPC requests sent from DApp clients' web browsers. The DApp of a JSON-RPC request is identified by one or a few API keys. The JSON-RPC request can also be identified by the IP address where the browser resides. Given an incoming request, the load balancer makes a decision regarding which RPC peer on the service backend should the request be forwarded to. The goal here is to characterize a load balancer in terms of its forwarding policy. Specifically, we aim at answering the following questions:

LB0. Given two RPC queries from the same IP and with the same API key, does the load balancer forward them to the same RPC peer?

LB1. Given two RPC queries with different API keys, does the load balancer forward them to the same RPC peer?

LB2. Given two RPC queries from different IPs, does the load balancer forward them to the same RPC peer?

LB3. Given two RPC queries with the same API keys and same IP but sent with TT seconds apart, does the load balancer forward them to the same RPC peer?

2) Methods: The key technique to enable answering the above questions is whether one can detect the presence of a load balancer. Specifically, given two incoming RPC requests, the presence of a load balancer entails the two requests are forwarded to different RPC peers in the service backend.

Design rational: To detect load balancing in a blackbox service, our *key idea* is to exploit the way that Ethereum clients including both `Geth` and `Parity` handle orphan transactions. Recall that each Ethereum transaction is associated with a count, called $nonce$, from its issuing client. Given the $nonce$ of the latest transaction of a client, an orphan transaction is a transaction sent from the same client and with a nonce no smaller than $nonce{+}2$. An Ethereum peer receiving an orphan transaction handles it in the following manner: It will store the transaction locally and evict it under one of the two conditions: 1) If a transaction with $nonce + 1$ is received, the orphan

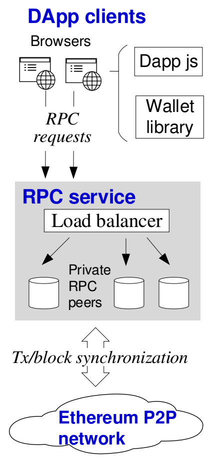


Fig. 3: RPC service model

```
1  bool detectLB_byOrphan(URL srv, int stall){
2    //current nonce plus two is orphan tx
3    int txHash=srv.sendTransaction(fromAddr,
       toAddr,Ether,nonce+2,gasPrice);
4    try{srv.sendTransaction(fromAddr,toAddr,
       Ether,nonce+2,gasPrice-1);
5    }catch(Exception e){
6      //no load balancing for sendTx RPC
7      Tx tx1 = srv.getTransaction(txHash);
8      waitTime(stall);
9      Tx tx2 = srv.getTransaction(txHash);
10     //no load balancing for RPC queries
11     return !(tx1 != null && tx2 != null);}
12   return true;}
13
14 bool detectLB_byBlockNo(URL srv, int time){
15   for(int i=0;i<BOUND;i++&&sleep(time)){
16      records.add(srv.getBlockNumber());}
17   return !isMonotonicIncreasing(records);}
```

Fig. 4: Benchmarks to characterize load balancing in a RPC service

Fig. 5: Characterizing the load balancing of RPC services (✗ means no load balancing detected or no gas limit, either making the service exploitable. XIP-Ykey means sending two requests from X IPs and with Y API keys to detect load balancing.)

Type	RPC services	1IP-1key (LB0)	1IP-2key (LB1)	2IP-1key (LB2)	Gas limit
i	ServiceX1	✗	✗	✗	✗
	ServiceX2	✗	✗	✗	✗
	ServiceX3	✗	✗	✗	50
ii	ServiceX4	✗	✓	✗	✗
	ServiceX5	✗	✗	✓	✗
iii	ServiceX6	✓	✓	✓	10
	ServiceX7	✓	✓	✓	✗
	ServiceX9	✓	✓	✓	5
	ServiceX8	✓	✓	✓	1.5

transaction becomes unorphaned and, together with transaction of $nonce+1$, will be propagated to the entire P2P network. 2) If no transaction with $nonce+1$ is received, the peer will drop the transaction after a timeout, say O_t. A subtle fact is that an orphan transaction can be replayed and updated before it becomes unorphaned. Specifically, an Ethereum peer, upon receiving two orphan transactions of the same nonce, will fail the second one if its $gas\ price$ is lower than 110% of the $gas\ price$ of the first transaction. In a RPC service, if one can send such two orphan transactions and observe the success of the second transaction, she can infer whether there is a load balancer forwarding the second transaction to a different backend peer than the first one.

Measurement mechanisms: Based on the above idea, we design a benchmark program to detect the presence of a load balancer inside a blackbox service. The program, namely `detectLB_byOrphan` in Figure 4, works as follows: It first sends an orphan transaction with $nonce+2$ and $gas\ price$ to a target RPC service ($nonce$ is the nonce of latest confirmed transaction) and observes the returned hash $txHash$ (Line 3). It then sends the second orphan transaction, with the same $nonce+2$ but paying $gas\ price-1$. If the second transaction fails (Line 4), it implies the second transaction is forwarded to the same backend peer with the first transaction; no load balancing is detected. Then, it further sends RPC queries to `eth_getTransactionByHash(txHash)` (Line 7). After waiting for a time period specified in the argument $stall$, it sends the second `transition(txHash)` RPC query (Line 9). If both `getTransaction(txHash)` RPCs return successfully, it means no load balancing is detected for RPC queries. What's noteworthy in our benchmark design is that we additionally require sending two RPC queries at different time points to confirm the absence of load balancing. In our preliminary design without the two `getTransaction(txHash)` queries, we found certain RPC services may exercise different load-balancing policies for different types of RPCs (i.e., privileged RPCs like `sendTransaction()` and open queries like `getTransaction(txHash)`).

To double-check the measurement result, we design a second detection mechanism based on RPC queries `getBlockNumber`. The benchmark, namely `detectLB_byBlockNo` in Figure 4 works as following: It sends a series of `getBlockNumber` queries, one every two seconds, to a target RPC service and observes the sequence of block numbers returned. If an "anomaly" is detected, it implies the presence of a load balancer in RPC queries. Here, the anomaly is defined as a `getBlockNumber` query sent earlier in time returns a block number larger than a later query. This reasoning here is that if all `getBlockNumber` queries are forwarded to the same RPC peer, the block number returned should monotonically increase with time.

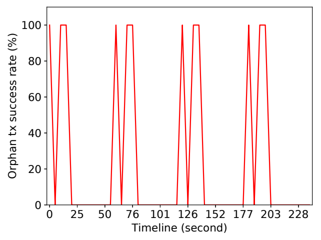
The purpose of using two benchmarks is to complement each other (either confirm or dispute the results of each other), as the `detectLB_byOrphan` can be accurate on the case of asserting no load balancing and `detectLB_byBlockNo` can be accurate on the case of detecting load balancing.

B. Measurement Results: Load Balancers

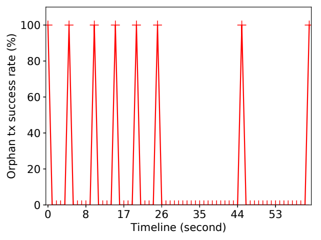
We conducted a series of experiments in order to answer questions LB0, LB1, LB2 and LB3.

For LB0, we set up the benchmark programs in such a way that all RPC requests are sent out with a single API key and from a single client (of a single IP). We run both benchmark programs, `detectLB_byOrphan` and `detectLB_byBlockNo`, against the nine RPC services. For each service, we collect the results (true or false) of the two benchmarks and crosscheck them before determining whether load balancing is present. In particular, we set the RPC rate (`interval` in benchmark `detectLB_byBlockNo`) to the maximal value right below the service rate limit (see the measurement result in § IV-E).

In experiments, the two benchmarks mostly agree with each other. That is, when `detectLB_byOrphan` detects load balancing (no load balancing), `detectLB_byBlockNo` confirms the same. The only exception is ServiceX9, on which `detectLB_byOrphan` detects load balancing but



(a) ServiceX6



(b) ServiceX9

Fig. 6: Load balancing w.r.t. request timing: The Y axis is the success rate of orphan transaction sent in the benchmark; it implies whether the load balancer forwards the RPC request to a new peer.

`detectLB_byBlockNo` does not (i.e., no observed case of decreased block numbers). `detectLB_byBlockNo` on ServiceX9 also return many failed results (i.e., block numbers being 0) which may be the culprit of the inaccuracy.

For LB1, we send RPC requests with two API keys and from the same client IP. **For LB2**, the RPC queries are sent with the same API key and from two client IPs. In the two cases, the different RPC requests apply to Line 7 and Line 9 in Figure 4 for `detectLB_byOrphan`. In `detectLB_byBlockNo`, each step would send out two different RPC queries at the same time, and the "block-number-decreased" anomaly is detected on the combined sequences of the two query results.

Results: The measurement results are presented in Table 5. We find three types of load-balancing behaviors w.r.t. LB0, LB1 and LB2: **Type i)** No load balancing of any sort, represented by ServiceX1, ServiceX3 and ServiceX2, (ii) Deterministic load balancing that entails two subtypes: **Type ii-a)** No load balancing detected when RPC queries are sent with the same API key and from different IPs; this is represented by ServiceX4, **Type ii-b)** No load balancing detected when RPC queries are sent from the same IP but with different API keys; this is represented by ServiceX5, and **Type iii)** Comprehensive load balancing detected, when RPCs are sent from the same IP and with the same API key; this is represented by ServiceX6, ServiceX9, ServiceX7 and ServiceX8.

LB3: We further conduct a measurement study for LB3; we design a simple benchmark to do so which issues a series of `sendTransaction` RPCs, with each two TT seconds apart. We run the benchmark against all services with load balancing (i.e., except for Type-i services). Most load balancers do not exhibit dependency to timing. The exceptions are ServiceX6 and ServiceX9 gateway.

Results: In ServiceX6, with $TT = 5$ seconds, we consistently observe the following behavior of ServiceX6's load balancer: The decision which backend peer to forward a request to is made based on the timing of the request. The experimental result is reported in Figure 6a. The result shows a four-minute period (where our raw result is more than an hour) when a series of transactions from the same `from` address are sent to ServiceX6. Every minute,[4] ServiceX6's load balancer will forward a transaction to a new peer if the transaction is the first one after the 0th, 10th or 15th second in the minute. All transactions sent between 15th and 60th seconds of a minute will end up with the same three backend peers in that minute.

The result of ServiceX9 is illustrated in Figure 6b, which is measured under 10 RPCs per second. In the first 25 seconds of any minute, there is a successful orphan transaction every 5 seconds, implying a new backend peer is allocated. After that, orphan transactions keep failing until the 40th second.

The deterministic behavior of real load balancers discovered in our research apparently serves an important purpose – maintaining consistency across dependent transactions of DApps: for instance, the order between an `approval` and a subsequent `transferFrom` of an ERC20 token should be preserved, so the load balancer always forwards these requests to the same backend node. This property, however, can be exploited to concentrate DoERS attack payloads on a small set of nodes to enhance their effectiveness, as elaborated in § IV-C2.

C. Attack Strategies Evading Load Balancer

An attacker can leverage the above measurement results to adjust an DoERS attack to specific services. Here, we present sample attack strategies specific to Type-ii and Type-iii services (note that Type-i service essentially runs no load balancer and can be attacked in a straightforward way).

1) Targeted Attacks to Type-ii Services: The deterministic behavior of a Type-ii load balancer can be exploited to launch a DoERS attack targeted at specific DApp victims. We propose strategies that a DoERS attacker can use to select victims adaptively to the service types, namely Type-ii(a) and Type-ii(b) services.

Targeted attack to Type-ii(a) services: Recall that a DApp web client commonly sends requests to a RPC service, using API keys. As the API key has to be disclosed on DApp websites to all visiting browsers, the DoERS attacker can easily obtain the API key. The attacker then sends DoERS requests with the API key to the service. Recall that a Type-ii(a) service forwards requests with the same API key to the same peer, despite of which IPs they are sent from. Thus, the DoERS requests will be processed by the same nodes serving other requests of the same API key. By this means, the attacker can disable the RPC node and further delay the service to other clients of the same DApp. Therefore, the DoERS attack can disable *all clients of a victim DApp*.

Targeted attack to Type-ii(b) services: Initially, the attacker prepares a "malware" token contract, called M-Token, which encodes the `exhaustXX` programs in `DoERS-C`. For instance, the `balanceOf` function in the token internally calls `exhaustCPU(1000000)`.

The attacker distributes the malware token M-Token to victim DApp clients. To do so, the attacker can set up a token faucet similar to gitcoin [18], that gives away free M-Tokens and, as a honeypot, attracts victim owners.

[4]Here, it requires the timeline is aligned with the Unix timestamp.

Later the victim owner may open her wallet DApp as usual. She will be surprised to find her DApp webpage unresponsive, because the webpage sending RPC requests to a service would make the service run M-Token's `balanceOf` function and get stuck. Further more, not only M-Token's balance is not viewable on the victim owner's webpage, but also the balances of other benign tokens are not responding. Because both the benign RPCs (to run benign tokens' `balanceOf`) and the M-Token's RPCs are sent from the same browser, thus the same IP, the Type-ii(b) service forwards them to the same backend peer. By this means, the M-Token RPCs can denial-of-service the benign tokens' RPCs.

2) Exploiting Timing Dependency to Attack Type-iii Services: Recall our measurement results in § IV-A that the load balancers of RPC services exhibit timing dependency: if two requests are sent close in time, the balancer forwards them to the same backend peer, for purposes such as preserving the ordering between the two requests. This predictable behavior can be exploited to direct DoERS requests to just a few peers, undermining their services to some DApps without saturating the entire service backend. This results in a low cost and more effective attack on multi-node RPC services.

Specifically, consider ServiceX6 as an example. As revealed from our measurement study (§ IV-A), ServiceX6's load balancer forwards all incoming requests received within a minute (with time aligned) to at most three distinct backend peers. So an attack can exploit this timing dependency to send all its DoERS requests in one minute to land on three specific nodes, which can effectively deny their services to DApps. Particularly, if the attacker knows when a specific DApp or its client issues requests (e.g., through eavesdropping on its communication or aiming at a known auction deadline when bids would come in), he could produce a few attack requests within the 1-minute window to block the three backend peers serving the DApp. This strategy enables a low-cost attack in which one does not need to overload hundreds of backend peers (e.g., more than 192 peers behind `ServiceX6`, as measured in Appendix C), which is very expensive, to undermine the service to some DApps and their clients. Note that such a "flash attack" (e.g., one minute for ServiceX6) can still have serious consequences, e.g., frontrunning a competing bid in a decentralized auction. The effectiveness of this attack is evaluated in § V-A2.

D. Measuring Gas Limits: Methodology

Given a RPC service, the goal is to test the presence of any Gas limit configured on the service's backend peers. Our test program, named by `rpc_gasLimit`, is in List A.15. The goal of the test program is to find the maximal argument (`arrayLength` in function `exhaustMem()`) that does not trigger the out-of-gas exception, a value that implies the Gas limit. To do so, the program starts with an initial guess on the target `arrayLength` value, then grows the guess exponentially until the first exception is observed. It then enters the second phase that binary-search the Gas-limit corresponding value of `arrayLength`. After the target value V is obtained,

```
float rpc_gasLimit(IP rpcNode){
  int lengthLower=0; int lengthUpper=500;//0/500 block gas
  while (lengthUpper - lengthLower > 1){
    arrayLength = (lengthLower + lengthUpper) / 2;
    try{
      rpcNode.eth_call(exhaustMem,arrayLength);
    } (Exception e) {
      if(e instanceOf OutofGasException){
        lengthUpper = arrayLength;
      } else { //no gas limits
        return 0;}
    } else {
      lengthLower = arrayLength;}}
  return localNode.estmateGas(exhaustMem,arrayLength);}
```

Fig. 7: Measure Gas limit of an RPC node

the program then uses a local RPC node (under our control) to run `estimateGas()` with function `exhaustMem` under V. The returned value is the Gas limit. Note that our design uses `exhaustMem` function which consumes Gas faster than the other two `exhaustXXs` and can finish before Ethereum's default 5-second timeout.

E. Measurement Results: Gas & Rate Limits

We measure the Gas limits of the backend peers by using program `rpc_gasLimit` in Figure A.15. Here, we assume that different nodes in the same RPC service have the same Gas limit. The results are illustrated in Table 5 which shows that four out of nine services configure the Gas limit: ServiceX3/ServiceX6/ServiceX9/ServiceX8 respectively set Gas limit at 50/10/5/1.5 block gas.[5] The services without Gas limits are particularly vulnerable to our DoERS attacks.

In our extended study, we also measure the rate limits deployed in many services' frontend. The rate limits are intended to protect the service against distributed DoS. However, rate limiting without real-world identities can be easily bypassed by a Sybil attacker who registers multiple service accounts and accumulates much higher rate limits. Yet, requiring the DApp clients (e.g., a web browser surfing a DApp page) to expose real-world identities is impractical. We thus don't consider rate limiting as an effective protection against DoERS. One can essentially bypass all RPC services' rate limit by using as many API keys or IPs as needed. Nevertheless, we measured rate limits and observe the measured limits are commonly inconsistent with the published rate limits on their website. The measurement result is deferred to Appendix B.

F. Attack Strategies Evading Gas Limit

For the RPC node without Gas limit, we design a single-request DoERS attack that has the power of evading all other protective measures we will observe in the next section (including rate limiting and load balancing). The attack sends a single request with a very large payload size (e.g., 10^9) to run the `exhaustMem` function in the `DoERS-C` smart contract. The key observation here is this: `exhaustMem` runs a single EVM instruction, namely `CODECOPY`, to allocate a large memory. Running a single EVM instruction is atomic and is

[5]Through our responsible disclosure, after our study, ServiceX5 has set a limit of 10 block gas.

not interrupted, even when there is a timeout. Thus, the DoERS attacker can increase the payload size of an `exhaustMem` invocation to evade the 5-second timeout, causing a higher resource consumption and more sever service damage, as will be evaluated in § V-B.

For the RPC node with Gas limit, the attacker can send multiple DoERS requests, each with a medium payload size under the Gas limit. If the requests are sent at a sufficiently high rate, there will be visible service interference, as will be evaluated in § V-A and § V-B.

G. Summary of Attack Strategies

We summarize what an actual DoERS attacker can do, with respect to different real-world situations. We consider the goal of the attacker is to cause maximal damage to the DApp ecosystem, while minimizing her cost.

C1) For nodes or services without Gas limit, the DoERS strategy is to send a single request invoking `exhaustMem` with a big-number payload size (e.g., 2^{64}). If this crashes the EVM on the victim node, the attacker waits for 30 seconds and pings the node before sending the request again. This strategy also applies to any services without Gas limits — the single-request attack evades the protection of a load balancer.

C2) For nodes with Gas limit, the DoERS strategy is to set the payload size of an individual request under the Gas limit and to send multiple such requests at a certain rate. In the case of a very low Gas limit, the attacker can tune up the request rate; because there is innate asymmetry between the service and the DoERS attacker, the attacker can expect to cause significant damage to an individual node at low cost (as will be evaluated in § V). This strategy applies to public RPC peers and Type-i services without load balancers.

C3) For Type-iii services with Gas limit, there can be two DoERS strategies. One (C3a) is to follow the C2 strategy and to increase the rate as necessary to DoS all backend peers in the service. Given the small service scale (tens of peers), the DoERS asymmetry still helps keep attacker's cost low (see § VI-A1 for an analysis). The other strategy (C3b) is to predict load-balancing behaviors and design specific attacks, as will be demonstrated in ServiceX6.

C4) For Type-ii services with Gas limit, the DoERS strategy is to mount targeted attacks. As described in § IV-C, the target can be a specific DApp client, a DApp, or a web3 library. The targeted attacks can evade the deterministic load-balancing behaviors in Type-ii services.

V. Evaluation of DoERS Attacks

In this section, we evaluate the effectiveness and cost of DoERS attacks. The attack effectiveness will be measured by service performance degradation in latency increase, block synchronization slowdown, and others. The attack cost will be measured by the attack rate. Note that an DoERS attack costs zero Ether by design as described in § VI-A2. Specifically, the evaluation aims to answer the following questions:

- Are real-world RPC services and peers exploitable under DoERS attacks? How much increase in response time will be caused by DoERS with "minimal" payload and rate (i.e., without causing any exception) on the real services? § V-A answers these questions.
- On a local Ethereum node, how much damage can DoERS cause with payload and rate large enough to trigger and bypass exceptions? The damage is measured not only in response-time increase, but also in block synchronization slowdown, mining rate slowdown, etc. § V-B answers these questions.

A. Evaluation on Deployed Services

1) Ethics-Driven Evaluation: Methodology: The goal is to verify whether a deployed RPC service is exploitable under DoERS attack. The main challenge comes from designing an effective test on the target services, without attacking them — The intensity of the test needs to be high enough to cause observable effects while it should be low enough to minimize actual performance degradation. The key idea here lies in discovering what we call the "minimally effective" parameters of the DoERS test. A DoERS test is minimally effective if 1) the difference between the response time of regular RPC requests under the test and that without the test is statistically significant, and 2) the response time of regular RPCs increase with the payload size and request rate.

More concretely, in the evaluation, we set up a virtual-machine (VM) instance in Google Cloud Platform (GCP) [24] for probing (a probing node) and another VM instance in Amazon EC2 [1] for measurement (a measurement node). With this setup, the two nodes do not share anything on their paths to a RPC service, and hence minimize performance interference between probing and measurement. During an experiment, we warm up the measurement node (in its network connection) by sending out three regular RPC requests (e.g., `eth_getBlockNumber`) to the target service. Then the measurement node sends out regular RPC requests at a rate of one request every two seconds. The response time of these regular RPCs is recorded. From the 30-th second after the measurement node starts, we launch the probing node which sends DoERS requests with minimally effective parameters. The probing node lasts for t_a seconds and the measurement node continues for another 60 seconds after that.

In order to discover the minimally effective DoERS parameters, we did a series of carefully designed pre-tests in a local machine: We set up the local Ethereum node and conduct local tests to find the DoERS parameters such that the response time with and without the test differ by $5\times$ times. During the pre-tests, we vary the attack parameters in payload size, probe rate and contract type. Note that $5\times$ will be an estimate as the hardware spec on the local node is different that on deployed services. The local pre-tests produce several sets of candidate DoERS parameters, each set is a triplet $\langle type, p, r_x\rangle$ where *type*/*p*/r_x is contract type/payload size/attack rate. For instance, $\langle \texttt{CPU}, 20M, 10\rangle$ means a DoERS attack exploit `exhaustCPU` function with payload size being $20M$[6] and attack rate being 10 requests per second. In particular, $r_x = 0$ means that a single DoERS request is sent out in the entire test process. Based on the above design, on each test, we would send a total of $60*2/2+3 = 63$ regular RPC requests plus at most $t_a * r_x$ DoERS requests. We set the attack duration t_a such that the number of DoERS requests can be upper-bounded before the test.

Besides, we avoid directly using large attack parameters (e.g., attack rates, payload sizes), which would have resulted in severe damages to the service. Instead, we test each service with a sequence of smaller and gradually increasing parameters, with the intention to discover the "trend" or how the server response time grows with increasing parameters. Such a trend allows to predict service response time under large parameters without causing the actual damage (see Figure 8b). With such measures, we expect each of our tests to affect no more than three nodes (out of hundreds) on the backend of each service for a short period of several minutes, to minimize the impact on its normal operations.

2) Evaluation Results: We follow the above methodology and test all nine services. Note that both measurement and probing VM instances do not run `Geth`, but instead run Curl [9] to send RPC requests. We first describe the experiment with ServiceX2 as an example. We run a series of tests described above with different minimally-effective parameters. Each test produces a timeline of RPC response times. For instance, Figure 8a reports such a timeline on ServiceX2 under DoERS attacks exploiting `exhaustCPU` with $30,000$ payload and at the rate of 30 requests per second. The result shows a moderate $5\times$ slowdown under the specific attack setting.

From there, we vary attack rate with payload size fixed at $0.07M$ (vary payload sizes with attack rate fixed at 18 per second). In each test, we define the attack-effective period by the period that the response time increases by at least $1.2\times$ than the response time without attacks. Then we calculate the average response time during the attack-effective phase and report it in Figure 8b. The result clearly shows that the response time grows with increasing payload size and attack rates. For ethical reasons, we stop our test at maximal payload size $0.15M$ <u>or</u> maximal rate 30 per second, resulting in maximal response time at about 100 milliseconds. Also, our experiments observe no timeout or other exceptions thrown. The trend revealed in the figure implies that an actual attacker can use larger parameters than ours to cause a much longer RPC delay towards crashing the service.

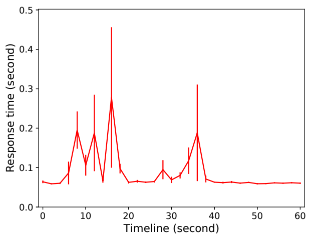
DoERS attacks to Type-iii services: We conduct experiments on ServiceX6 as an example Type-iii service. In the experiment, DoERS requests are sent to `exhaustCPU` with payload size $1.5M$ at the rate of 200 requests per second. The $1.5M$ payload size makes the per-request Gas right below the Gas limit of ServiceX6 service. The attack lasts for 20 seconds, and we observe a protective measure taken by ServiceX6–15 seconds after the attack starts, the DoERS requests are returned with null. The timeline of measured response time is illustrated in Figure 8d. The RPC response time increases from 40 milliseconds before the attack to 160 milliseconds after the attack, leading to a $5\times$ increase. We suspect two causes: First, ServiceX6's load balancing depends on the timing of requests: all DoERS requests sent within one minutes are collocated to the same three RPC nodes. Second, there are hundreds of peers on the backend of ServiceX6 and all peers are saturated by the DoERS attacks.

Targeted attacks to Type-ii services: Among Type-ii services, ServiceX5 is a representative service whose load balancer distinguishes requests based on IPs; recall Table 5. We conduct two tests that differ only by where the DoERS requests are sent. The specific result is in Figure 9a: If the DoERS requests are sent from a different IP from where the measurement requests are sent (as in our original setup), no increase of response time can be spotted. However, if we send the DoERS requests from the same IP with the measurement requests, the response time clearly increases right after the attack starts at the 5th second in Figure 9a. To eliminate the possibility of performance interference between probing and measuring, we conducted an extra test by sending DoERS requests at the same rate but with a much small payload size (e.g., 3 iterations in a loop) and no response-time increase can be observed. The result corroborates our measured load-balancing behavior and directly shows that the adaptive attack strategy (recall § IV-C) is effective on ServiceX5. Note that in Figure 9a, the attack sends only a single request exploiting `exhaustMem` with $20M$ payload size. The 10X increase of response time is caused by this single DoERS request. We also conduct similar experiments on the other Type-ii service, ServiceX4, where DoERS requests are sent with the same/different API key with the measurement requests. The result, presented in Figure 9b, similarly show the effectiveness of our attack strategies – under the DoERS with the same API key, a $6\times$ slowdown (from 0.4 seconds to 2.4 seconds) is caused while under the DoERS of the same API key, there is no visible service slowdown.

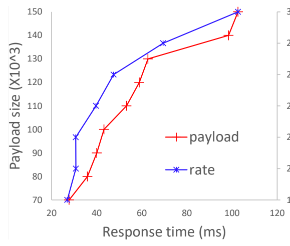
Single-request memory DoERS: For RPC services with no gas limits, the DoERS attacker can send a single RPC request to execute the `exhaustMem` that bypasses any load balancing. On ServiceX2, we send a single request with parameters `eth_call(exhaustMem(`$5*10^7$`))`, and we report the response times in Figure 10a. After the attack starts at the 5th second, the response time grows up by $20\times$ (from 0.1 seconds to 2 seconds). On ServiceX5, we similarly a single request with parameters `eth_call(exhaustMem(`$1*10^9$`))`. From Figure 10b, the response time is increased by $150\times$ (from 0.2 seconds to 30 seconds).

Summary of attack parameters: Table I summarizes the effective attack parameters we found on these services. It can be seen that most existing services, with or without Gas limits, can be successfully attacked, causing an observable response-time increase by at least $3.8\times$. On ServiceX2, for instance, the parameters to cause $3.8\times$ increase are $\langle \texttt{CPU}, 0.15M, 30\rangle$; note that payload size $0.15M$ amounts to 0.2 block gas. Currently ServiceX2 does not set Gas limits; but our result implies that

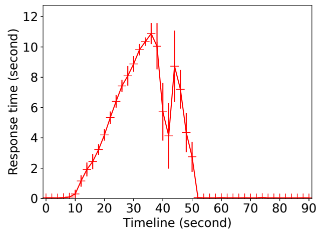
[6]We use M and K to denote a million and a thousand, respectively.



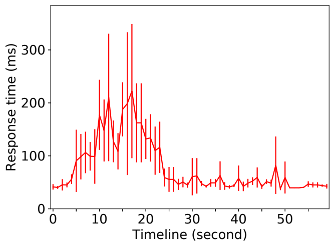
(a) ServiceX2 (i) $\langle \texttt{CPU}, 30K, 30\rangle$



(b) ServiceX2 (i) w. varying rates & payloads

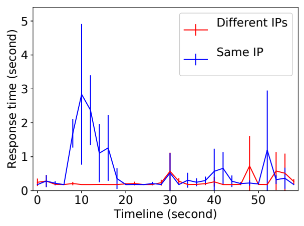


(c) ServiceX8 (iii) $\langle \texttt{CPU}, 0.6M, 200\rangle$

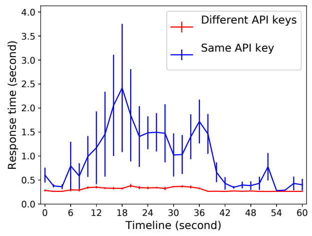


(d) ServiceX6 (iii) $\langle \texttt{CPU}, 1.5M, 200\rangle$

Fig. 8: `exhaustCPU` attacks to RPC services (Type-i and iii)

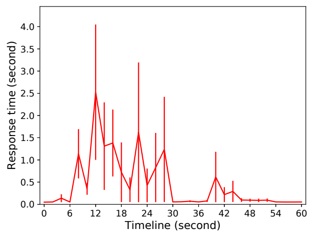


(a) ServiceX5 $\langle \texttt{Mem}, 50M, 0\rangle$

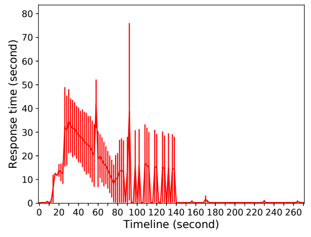


(b) ServiceX4 $\langle \texttt{CPU}, 40k, 30\rangle$

Fig. 9: Targeted attacks to RPC services (Type-ii)



(a) ServiceX2 $\langle \texttt{Mem}, 50M, 0\rangle$



(b) ServiceX5 $\langle \texttt{Mem}, 1000M, 0\rangle$

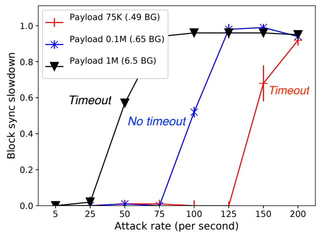
Fig. 10: A single-request attack exploiting `exhaustMem` to nodes without gas limits

TABLE I: Minimally effective attack parameters: Gas* in the number of block gas. In parenthesis are Gas limits.

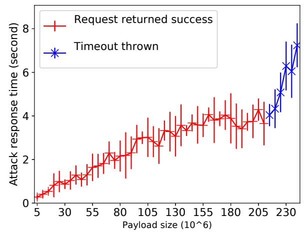
Services	⟨type,payload,rate⟩	Time	Gas*
ServiceX1	$\langle \texttt{CPU}, 2M, 10\rangle$	16×	13
ServiceX2	$\langle \texttt{CPU}, 0.15M, 30\rangle$	3.8×	0.2
ServiceX3	$\langle \texttt{CPU}, 3M, 0\rangle$	30×	19.5 (50)
ServiceX5	$\langle \texttt{Mem}, 50M, 0\rangle$	10×	5000
ServiceX4	$\langle \texttt{CPU}, 0.04M, 30\rangle$	4×	0.3
ServiceX6	$\langle \texttt{CPU}, 1.5M, 200\rangle$	5×	10 (10)
ServiceX7	$\langle \texttt{CPU}, 5M, 10\rangle$	15×	32.5
ServiceX9	$\langle \texttt{CPU}, 0.04M, 30\rangle$	2.1×	0.3 (5)
ServiceX8	$\langle \texttt{CPU}, 0.6M, 200\rangle$	110×	1.5 (1.5)

even if they set a very low Gas limit, like 0.2 block gas (which by the way is unlikely because of interference with service usability as will be discussed in § VI), it is still not enough to defend against DoERS attacks. ServiceX3 can be effectively attacked at a Gas limit as low as 19.5 block gas, which is much lower than their current Gas limit (50 block gas). ServiceX6 can be attacked with their current Gas limit (of 10 block gas), causing 5× response-time increase. We notice that the minimally effective payload sizes differ from different services and this can be caused by different hardware specs of the machines run in these services.

B. Evaluation on a Local Full Node



(a) Block sync. slowdown under `exhaustCPU`



(b) `exhaustMem` and timeout

Fig. 11: Evaluate DoERS attacks on a local node

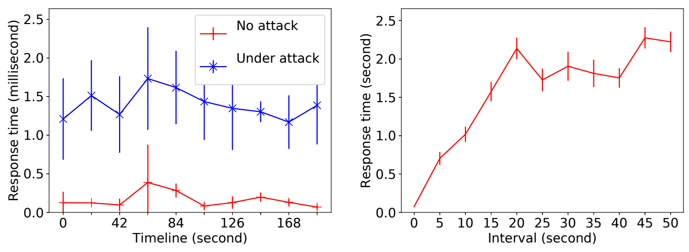
In order to evaluate the damage caused by DoERS more extensively, we conduct experiments on a local machine under our control. The machine is a blade server with a 32-core 2.60GHz Intel(R) Xeon(R) CPU (E5-2640 v3), 256 GB RAM and 4 TB SSD disk. We set up a Geth v1.99 client on the server and fully synchronize it with the Ethereum mainnet. We turn on the RPC on this full node with default settings. The probing node and measurement node are run on the same commodity computer as before (§ V-A2).

The first experiment evaluates the DoERS's impact on the block synchronization rate on the victim. In the experiments, we measure the local victim node's current block height, denoted by B_v. To do so, the measurement node sends `eth_getBlockNumber` RPCs to the victim. We also monitor the block height of a regular mainnet node by B_r and record the initial block height before the attack by B_0. From there, we report a metric that we call block synchronization slowdown: $\frac{B_r(10)-B_v(10)}{B_r(10)-B_0}$ where $B_r(10)/B_v(10)$ is the block height 10 minutes after the attack starts. In the experiment, we vary the payload size and the attack rate, and report the slowdown in Figure 11a.

The result shows that block synchronization slowdown reaches as high as 96% with attach parameter $\langle \texttt{CPU}, 1M, 100\rangle$. When the payload size is $0.1M$ which amounts to a Gas limit of 0.65 block gas, the DoERS attacker can cause synchronization slowdown by 91%, at the rate of 150 RPCs per second. Note that 0.65 block gas is very restrictive and is lower than any Gas limits we observe on all real RPC services and peers. In the figure, each point is labeled by whether a timeout is triggered during the test. It can be seen the DoERS attack of parameters $\langle \texttt{CPU}, 0.1M, 100\rangle$ does not trigger timeout yet still causes a 50% synchronization slowdown.

The second experiment shows how timeout can be effectively evaded by `exhaustMem` on nodes without Gas limits; recall the attack strategy C1 in § IV-G. In this experiment, we conduct a series of tests, each of which sends a single `exhaustMem` request with increasing payload sizes. We report the response time of the attack request as in Figure 11b. When the payload size increases, the response time grows, first without timeout (in the red line) and then with timeout (in the blue line). It is clear that after timeout occurs, increasing payload sizes still leads to the increase of response time. This implies that the `exhaustMem`-based DoERS can essentially evade the timeout and increases payloads to crash the machine. The severe damage is applicable to the 348 public RPC nodes and the five RPC services that do not configure Gas limits. The explanation for this attack is the following: `exhaustMem` contains an EVM instruction `CODECOPY` that runs a loop inside EVM to allocate memory of arbitrary length. Executing the instruction is atomic and can not be interrupted in between by a timeout; throwing a timeout has to wait until the completion of the instruction.

C. Evaluating DApp Response Time under DoERS

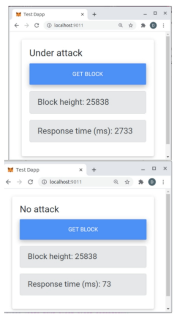


(a) DApp response time w/wo attacks $\langle \text{CPU}, 800K, 200\rangle$ against ServiceX9

(b) DApp response time w/wo attacks $\langle \text{CPU}, 800K, 200\rangle$ against ServiceX9

Fig. 12: DoERS attacks on a metamask-based DApp

Our objective is to evaluate the impacts of DoERS on real-world DApps. Recall Figure 3; a typical DApp architecture includes user-facing web pages, wallet clients running as browser extensions such as metamask [35], the RPC service it uses and a remote blockchain network. It is known that most DApp webpages rely on metamask and third-party RPC services to communicate with Ethereum [35], which has also been confirmed in a simple measurement study we conduct: Among the top 26 DApps (in terms of active user number)



Under attack

Block height: 25838

Response time (ms): 2733

No attack

Block height: 25838

Response time (ms): 73

Fig. 13: Screenshot of DApp

from `dappradar.com`[7], 20 (with combined 201,500 active users in 30 days) use metamask. So we focus on the response time for metamask-based RPC clients.

Specifically, our experiment is based upon a browser running a sample DApp that we develop on top of metamask. The sample DApp is a web button to get the latest block from the Ethereum network, through an RPC query `eth_blockNumber`. Here, metamask is configured to connect to a sample RPC service, namely ServiceX9.

Also, we run a javascript code that issues a metamask query every X seconds. We first set $X = 25$ seconds, since RPC results cached by metamask expire every 20 seconds (based on our experiment results). In the experiment, we measure and compare the response times of the "getBlock" button with and without a DoERS attack on the RPC node, as illustrated in Figure 12a, under the following parameters: 200 requests per second and a payload of $8*10^5$. As we can see from the figure, the response time perceived by the client becomes significantly longer in the presence of the attack, causing a $10\times$ slowdown. We present the screenshots of our DApp with/without the DoERS attack in Figure 13 which are taken from the full video demo shared on our public website.[8]

We then vary the interval X between 5 seconds and 50 seconds. We rerun the above experiment for three times, and report the average response time and their variance in Figure 12b. The result shows a shorter average response time can be observed if the internal is below 20 seconds, which matches the conjectured effect of result caching in metamask.

VI. Countermeasure

A. Analyzing Known Countermeasures

1) Effectiveness of Gas Limits: In the Ethereum community, Gas limits are provided as the primary defense to denial of RPC service. Both Geth and Parity provide configuration knobs to set the Gas limit on a RPC instance. Ideally, the service provider should set a Gas limit low enough to protect their nodes from DoS attacks.

In practice, finding a "meaningful" value for the Gas limit is non-trivial if not impossible at all. There are two restrictions/challenges: 1) Setting a low Gas limit could negatively affect the service usability. For instance, a benign DApp wants to send a Google BigQuery-style RPC [3] to the blockchain service which would be blocked by a low Gas limit. This intention between a security-concerning service provider who wants to set a lower Gas limit and a usability-desiring client rooting for a higher Gas limit is real and has been observed [19]. In the end, the service provider often puts customer experience over the service security, by increasing the Gas limit, such as from 2 to 10 block gas in [19]. 2) More fundamentally, blockchain RPCs supporting Turing-complete programs cause asymmetry of computing cost between the client side and service side. That is, in a usable setup, the client-side cost in sending a

[7]https://dappradar.com/rankings/protocol/eth
[8]https://sites.google.com/view/doersdemo/

RPC request is supposed to much lower than the server-side cost of executing the smart contract. The "perfect" DoERS security will entail equating the client-side cost and server-side cost, which will lead to very restrictive loops (e.g., fewer than ten iterations) and would be detrimental to the service usability.

Empirically, the table in Figure I shows mixed results: On the one hand, some services, notably ServiceX2 and ServiceX4, can be effectively attacked even if the Gas limits are set as low as 0.2 and 0.3 block gas. Let alone that a low Gas limit is unlikely to be deployed in practice due to the impacts to service usability. Our experiments with local nodes described in § V-B also suggest there are effective attack parameters even with low Gas limits as 0.65 block gas. On the other hand, there are services which could mitigate DoERS vulnerability by deploying a reasonably low Gas limit. For instance, if ServiceX5 deploys the Gas limit of 10 block gas (which is the case after our disclosure of the problem to them in May, 2020), it would make the DoERS harder to succeed.

We believe setting a Gas limit is necessary but not sufficient; in other words, complementary defensive measures to Gas limiting are needed to provide effective DoERS protection.

2) Contract Banning and Zero-Ether DoERS: Recall that the first step in DoERS (in § III) requires the attacker to deploy a smart contract at her own cost. From our experience with ServiceX7, a service provider who monitors the RPC performance can correlate the latency spikes to a malicious smart contract; they can take measures to ban all subsequent RPCs accessing the malicious contract. This will force the attacker to deploy the `DoERS-C` smart contract to a new address which could increase her cost in Ether.

We propose a zero-Ether DoERS that incurs zero monetary cost to the attacker, as a technique to evade a contract-banning service provider. The zero-Ether DoERS exploits the "state override" extension of `eth_call` in the recent Geth release [14], [25]. This feature allows a client to upload a smart contract at the invocation time of `eth_call`, instead of using a separate transaction. Specifically, the `eth_call` request carries the bytecode of a smart contract in its "state override" argument and invoke to run a certain function in the bytecode on the RPC node.

With this capability, the attacker can mount the DoERS attack in one step without paying any Ether. The attack works by the attacker sending a crafted `eth_call` request that includes the code of `exhaustXX` in its "state override" object and specify the invoked function to be `exhaustXX`. We have tested this zero-Ether DoERS attack on our local RPC node running Geth v1.9.2 [25].

B. Proposed Countermeasures

The root cause of DoERS is *an open-membership RPC service that allows for free execution of arbitrary smart-contract programs on its peers shared by different DApps*. Intuitively, "falsifying" any condition in this root cause should harden the security against DoERS attacks, such as removing open-membership (e.g., by authenticating DApp clients based on their true identities), charging the contract execution triggered by `eth_call`, limiting the computation expressiveness (e.g., prohibiting loops) and avoiding any sharing of a RPC node among DApps. Along these design directions, we encounter a fundamental trade-off between DoERS security and service usability. For instance, the service provider can simply refuse to admit any `eth_call` triggering to run loops, which, while eliminating DoERS, comes at the expense of not being able to serve the benign DApps that do rely on loops; there are real-world smart contracts like this, such as financial analysis [20]. Also, requiring DApp clients to present real-world identities would be impractical or against the design of blockchain information transparency. We believe eliminating the DoERS vulnerability without affecting service usability is fundamentally difficult, if not impossible at all. Beyond simply Gas limiting, we propose a variety of *mitigation* techniques without dropping service usability, by performance anomaly detection, requiring security deposit, secure load balancing, atomic EVM execution (as will be described next), and other feasible defenses such as performance isolation. These techniques can be engineered in a RPC service at the layers of both service frontend and the underlying EVM.

Unpredictable yet consistency-preserving load balancing: We design a secure and practical RPC load balancer that serves two purposes. First, it is expected to preserve the order between dependent transactions, which is important to ensure the correctness and fairness of the target DApp's operations. Specifically, two transactions issued sequentially from the RPC client need to keep that order in the blockchain's final transaction history. For instance, for an ERC20 token contract, the call `approve` needs to be followed by `transferFrom`, or otherwise, the execution will fail. Second, the load balancer's behavior should be unpredictable in the sense that it independently forwards different incoming requests to randomly selected backend peers. Any determinism in load balancing can be exploited to direct the DoERS payloads to a few victim peers, allowing the attacker to overload them at a low cost.

However, preserving cross-request consistency could be in conflict with achieving load-balancing unpredictability. For instance, independent assignment of the `approve` and `transferFrom` calls could cause the calls to be handled by different backend peers, which will send them independently to miners, rendering the order of their reception on the miners hard to maintain. Note that since `approve` and `transferFrom` are transactions issued from different sender accounts, Ethereum's builtin nonce mechanism does not apply here. Our research shows that the load balancers in existing RPC services are designed to favor consistency preservation (DApp semantics) over unpredictability (§ IV-B).

We believe that this challenge is fundamentally caused by the use of the a single balancer to process both transactions (write to a block chain) and RPC queries (read from the chain). The former requires cross-request consistency while the latter does not. Since DoERS targets the RPC queries, we could simply separate them from transaction requests, through two load balancers, to protect the RPC peers through unpredictable assignment of the queries. More specifically, one balancer handles only transactions while the other forwards only RPC queries (including the `eth_call`'s), independently and randomly selecting a peer from the RPC service for each query (through a uniform distribution). To this end, the load balancer can internally maintain a secret true-random number or the current workload that decides the destination backend peer a RPC query should be forwarded to. In the meantime, the transaction-only balancer distributes the requests under the constraint of preserving consistency, just like what has been done by ServiceX6 today (transactions with temporal locality given to the same backend peer).

A limitation of this dual-balancer solution is that it does not ensure transaction-query consistency: that is, the order between a transaction and a RPC query related to the transaction may not be preserved. One way to address this issue could be simply handing over such a transaction-query pair to the transaction balancer, so they can be assigned to the same peer and propagated to the blockchain in the right order.

Performance anomaly detection plus security deposit: As we analyzed, simple performance monitoring with contract banning can be evaded by our zero-Ether DoERS. We propose a countermeasure against the zero-Ether DoERS. The key idea is for the service provider to require security deposit from any potential clients, such that a benign client's deposit will be refunded and a malicious client's deposit will be confiscated to discourage any further attacks. In the proposed framework, 1) the service provider only processes RPCs from a client having made security deposits. 2) The service provider monitor the performance and detect DoERS requests as performance anomalies. 3) After identifying attackers, the service provider confiscates deposits from attackers and refunds benign clients.

The success of the countermeasure hinges on whether the performance monitor can distinguish malicious DoERS requests from benign RPCs. Here, our assumption is that a DoERS attacker who wants to keep her cost low and to evade existing DDoS protections has to make each malicious `eth_call` cause a significant amount of computations much more than a benign RPC.

Interruptible EVM instructions: The success of single-request DoERS can be attributed to atomic EVM instructions that timeout cannot interrupt. To avoid this attack vector, EVM should allow the "long-lasting" execution of a single instruction (e.g., `CODECOPY` in `exhaustMem`) to be interrupted by timeout. This may require engineering to change EVM's instruction scheduling algorithm and to enforce the maximal memory size allocated by a single `CODECOPY` call.

VII. Related Works

Blockchain DoS security: Since the advent, public blockchains have been a target of DoS attacks. A variety of DoSes have been designed and practiced on the different layers of a blockchain system in smart-contract execution [44], [30], transaction processing [51], [34], [40], mining-based consensus [26], [60], and the underlying P2P network [54], [58], [43], [63]. For instance, in the P2P network layer, an eclipse attack [54], [58] aims to isolate a DoS-victim peer from the network and a routing attack [43], [63] employs BGP hijacking to intercept network traffic towards partitioning it. Among these attack vectors, of particular relevance are the DoSes that evade the Gas-based mechanism for smart-contract execution. Under-priced EVM instructions, notably `EXCODESIZE` [42] and `SUICIDE` [21], have been identified and exploited in practice DoS attacks. Ethereum EIP150 [44] fixes the bugs by increasing the Gas associated with these instructions. Broken metering [62] further exploits the runtime variation of an EVM instruction, with the goal to lower contract-execution throughput (gas per second) at low cost. Defensive mechanisms [47] have been proposed to punish contracts that excessively execute a particular (vulnerable) instruction. Unlike existing DoS attacks, DoERS uniquely targets the RPC-service layer of a blockchain node. DoERS is extremely low-cost and does not incur no Gas or Ether), which differs from existing DoSes that incur significant Gas.

Blockchain RPC attacks: In the existing literature, the only research work on the attacks exploiting blockchain's RPC is a measurement of currency stealing attacks [48]. In the currency-stealing attack, an adversarial client exploits the time window between an account-unlocking RPC request and a transaction-send request, such that she can gain unauthorized access to an account unlocked on a RPC service. DoERS differs from the RPC-based currency stealing attack in that it does not exploit the privileged RPCs (e.g., account unlocking and transaction sending) but focus on the open RPC queries that allow smart-contract execution.

Blockchain measurements: Passive measurement [57] reveals various deployment information in Ethereum network (e.g., node distribution, network sizes, etc.). The approach taken is to launch several Ethereum nodes and collect the messages they exchange with their neighbors, which are analyzed to uncover network information. There are other measurement works focusing on Bitcoin network topology [50], [53], Monero P2P network [45], ERC20 token networks [65], etc. The measurement studies in this work focus on the DoERS security and leverage a novel measurement method based on orphan transactions that are not taken in existing works.

VIII. Responsible Disclosure

We have disclosed the DoERS vulnerability to the developer communities of Geth [15] and Parity/OpenEthereum [38], as well as all tested service providers. The bug reports are sent in May, 2020, leaving tested services at least 9 months to fix the bug before disclosing the vulnerability publicly (in Feb. 2021).

We have received a total of $260 bounty in Ether and are informed by the RPC services that bug fixing is in progress. For instance, our bug report has been acknowledged in Geth v1.9.16 release (July 10, 2020), which sets a new default limit to $25 * 10^6$ Gas. Also after our reporting, ServiceX5 sets a new limit to their service at $25 * 10^6$ Gas, and invites us for further testing.

IX. Conclusion

This paper presents the first measurement study on the security of Ethereum's RPC-enabled nodes under denial of service attacks. The results reveal that five out of the nine popular services (as of Apr. 2020) have turned on RPCs without configuring any Gas limits. These peers are particularly vulnerable and can be crashed by the proposed DoERS attack that sends as few as a single `eth_call` request at zero Ether cost. While the four other services including ServiceX6 have configured Gas limits, the limits are so nonrestrictive that a properly configured DoERS attack can cause a latency increase by $2.1\times \sim 50\times$, as verified in our probes. On a local node protected by a very restrictive limit of 0.65 block gas, sending 150 RPC requests per second can slow down the block synchronization of the victim by 91%.

This work addresses the challenge of eliminating the DoERS vulnerability without affecting service usability. We propose mitigation beyond simply limits the Gas; these techniques include unpredictable load balancing, performance anomaly detection, and interruptible EVM instructions. They are easy to be engineered in a RPC service at the layers of both service frontend and the underlying EVM.

Acknowledgment

The authors thank anonymous NDSS reviewers. The first four authors are partially supported by NSF Grant CNS1815814. XiaoFeng Wang is supported in part by the NSF CNS-1618493, 1838083 and 1801432. Xiapu Luo is supported by Hong Kong RGC Project (No. 152193/19E).

References

[1] Amazon ec2 - amazon web services. https://aws.amazon.com/ec2/.
[2] Api reference (json-rpc). https://en.bitcoin.it/wiki/API_reference_(JSON-RPC).
[3] Bigquery: Serverless, highly scalable, and cost-effective cloud data warehouse. https://curl.haxx.se/.
[4] Bitcoin script. https://en.bitcoin.it/wiki/Script.
[5] Blockchain explorer. https://www.blockchain.com/explorer.
[6] Chain api — eosio developer docs. https://developers.eos.io/manuals/eos/latest/nodeos/plugins/chain_api_plugin/api-reference/index#operation/get_account.
[7] Cryptokitties: Collect and breed digital cats! https://www.cryptokitties.co/.
[8] Cryptokitties on dapp radar (showing five smart contracts deployed on ethereum). https://dappradar.com/app/3/cryptokitties.
[9] curl, command line tool and library for transferring data with urls. https://curl.haxx.se/.
[10] Dapp survey results 2019. https://medium.com/fluence-network/dapp-survey-results-2019-a04373db6452.
[11] Devp2p library (used in ethereum). https://github.com/ethereum/devp2p/.
[12] Eip 1767: Graphql interface to ethereum node data. https://eips.ethereum.org/EIPS/eip-1767.
[13] Eosio on dfuse. https://docs.dfuse.io/guides/eosio/.
[14] Eth namespace: Geth provides "state override" extensions to the standard "eth_call". https://geth.ethereum.org/docs/rpc/ns-eth.
[15] Ethereum bounty program. https://bounty.ethereum.org/.
[16] Ethereum json rpc. https://ethereumbuilders.gitbooks.io/guide/content/en/ethereum_json_rpc.html.
[17] Ethereum name service.
[18] Faucet in gitcoin. https://gitcoin.co/faucet.
[19] Gas limit on eth_call? https://community.infura.io/t/gas-limit-on-eth-call/1115/.
[20] getfunddetails in the melon protocol. https://github.com/melonproject/protocol/blob/f5bc07870c0f6a88a1b0ec855752dbd9cc6a23f5/src/contracts/factory/FundRanking.sol#L8.
[21] Geth nodes under attack again (reddit). https://www.reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively/.
[22] Geth requires manual configuration to enable rpc. https://github.com/ethereum/wiki/wiki/JSON-RPC#json-rpc-endpoint.
[23] Go sdk — stellar developer — rest api. https://www.stellar.org/developers/horizon/reference/index.html.
[24] Google cloud: Cloud computing services. https://cloud.google.com/.
[25] Internal/ethapi: allow eth_call with custom code. https://github.com/ethereum/go-ethereum/issues/19836.
[26] Irreversible transactions: Finney attack. https://en.bitcoin.it/wiki/Irreversible_Transactions#Finney_attack.
[27] Json rpc api - wiki. https://wiki.parity.io/JSONRPC.
[28] Json-rpc in ethereum wiki (eth_call). https://github.com/ethereum/wiki/wiki/json-rpc#eth_call.
[29] Json-rpc in ethereum wiki (eth_estimategas). https://github.com/ethereum/wiki/wiki/json-rpc#eth_estimateGas.
[30] Known attacks - ethereum smart contract best practices. https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-block-gas-limit.
[31] Ligntning network, scalable, instant bitcoin/blockchain transactions.
[32] Management apis by go ethereum. https://github.com/ethereum/go-ethereum/wiki/Management-APIs.
[33] Melon terminal. https://melon.avantgarde.finance/.
[34] Memoria 700 million stuck in 115,000 unconfirmed bitcoin transactions. https://www.ccn.com/700-million-stuck-115000-unconfirmed-bitcoin-transactions/.
[35] Metamask: A crypto wallet & gateway to blockchain apps. https://metamask.io/.
[36] Nmap: the network mapper - free security scanner. https://nmap.org/.
[37] Node discovery protocol in devp2p. https://github.com/ethereum/devp2p/blob/master/discv4.md.
[38] Parity bug bounty program. https://www.parity.io/bug-bounty/.
[39] Public apis on greymass, an eosio block producer. https://greymass.com/en/apis.
[40] Report: Bitcoin (btc) mempool shows backlogged transactions, increased fees if so? https://goo.gl/LsU6Hq.
[41] Solidity in depth – contracts, section: View function and pure function. https://solidity.readthedocs.io/en/v0.4.24/contracts.html.
[42] Transaction spam attack: Next steps. https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/.
[43] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In IEEE Symposium on SP 2017, pages 375–392, 2017.
[44] Vitalik Buterin. Eip150: Gas cost changes for io-heavy operations.
[45] Tong Cao, Jiangshan Yu, Jérémie Decouchant, Xiapu Luo, and Paulo Veríssimo. Exploring the monero peer-to-peer network. IACR Cryptology ePrint Archive, 2019:411, 2019.
[46] Raphael Lefbvre Renaud Larsen Cathy Pill, Sarah Levin Weinberg. Stellar: Smart influencer marketing platform. stellar.io.
[47] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks. In ISPEC 2017, pages 3–24, 2017.
[48] Zhen Cheng, Xinrui Hou, Runhuai Li, Yajin Zhou, Xiapu Luo, Jinku Li, and Kui Ren. Towards a first step to understand the cryptocurrency stealing attack on ethereum. In RAID 2019, pages 47–60, 2019.
[49] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. CoRR, abs/1904.05234, 2019.
[50] Sergi Delgado-Segura, Surya Bakshi, Cristina Pérez-Solà, James Litton, Andrew Pachulski, Andrew Miller, and Bobby Bhattacharjee. Txprobe: Discovering bitcoin's network topology using orphan transactions. In Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers, pages 550–566, 2019.
[51] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: Front-running attacks on blockchain. In Financial Cryptography and Data Security - FC 2019 International

Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers, pages 170–189, 2019.

[52] Amir Feder, Neil Gandal, J. T. Hamrick, and Tyler Moore. The impact of ddos and other security shocks on bitcoin currency exchanges: evidence from mt. gox. J. Cybersecur., 3(2):137–144, 2017.

[53] Matthias Grundmann, Till Neudecker, and Hannes Hartenstein. Exploiting transaction accumulation and double spends for topology inference in bitcoin. In Aviv Zohar, Ittay Eyal, Vanessa Teague, Jeremy Clark, Andrea Bracciali, Federico Pintore, and Massimiliano Sala, editors, Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers, volume 10958 of Lecture Notes in Computer Science, pages 113–126. Springer, 2018.

[54] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In Jaeyeon Jung and Thorsten Holz, editors, USENIX Security 2015, Washington, D.C., USA, pages 129–144. USENIX Association, 2015.

[55] Maurice Herlihy. Atomic cross-chain swaps. In ACM Symposium on PODC 2018, pages 245–254, 2018.

[56] Benjamin Johnson, Aron Laszka, Jens Grossklags, Marie Vasek, and Tyler Moore. Game-theoretic analysis of ddos attacks against bitcoin mining pools. In Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers, pages 72–86, 2014.

[57] Seoung Kyun Kim, Zane Ma, Siddharth Murali, Joshua Mason, Andrew Miller, and Michael Bailey. Measuring ethereum network peers. In Proceedings of IMC 2018, pages 91–104, 2018.

[58] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. IACR Cryptology ePrint Archive, 2018:236, 2018.

[59] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. CoRR, abs/1702.05812, 2017.

[60] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. Bdos: Blockchain denial of service, 2019.

[61] OpenEthereum. Parity opens rpc by default only to requests from localhost. https://github.com/openethereum/openethereum/blob/597cbc2d6c62cae6374f8e6fce6eb954de3504cb/parity/cli/tests/config.full.toml.

[62] Daniel Pérez and Benjamin Livshits. Broken metre: Attacking resource metering in EVM. CoRR, abs/1909.07220, 2019.

[63] Muoi Tran, Inho Choi, Gi Jun Moon, Anh V. Vu, and Min Suk Kang. A Stealthier Partitioning Attack against Bitcoin Peer-to-Peer Network. In To appear in Proceedings of IEEE Symposium on Security and Privacy (IEEE S&P), 2020.

[64] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In Financial Cryptography and Data Security - FC 2014 Workshops, BITCOIN and WAHC 2014, Christ Church, Barbados, March 7, 2014, Revised Selected Papers, pages 57–71, 2014.

[65] Friedhelm Victor and Bianca Katharina Lüders. Measuring ethereum-based ERC20 token networks. In Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers, pages 113–129, 2019.

Appendix

A. Exploitability Measurements on Ethereum Peers

The initial state of the measurement study is a list of Ethereum peers' IPs collected from the mainnet (based on a passive measurement method [57]; see details in § A2). Given the peers' IPs, our first measurement module (§ A1a) is to classify whether each peer is a valid, public RPC peer. Validity means that the peer should not be a honeypot. Each identified public RPC peer is a potential victim under DoERS, as its IP and RPC port is known. The second measurement module (§ A1b) is to profile these potential victim peers and to obtain their gas limit. Recall that a gas limit can prevent a naive DoERS attack. With the knowledge of the IP, RPC port, and gas limits of an Ethereum peer, a practical DoERS attack that evades the gas limit can be readily adjusted and mounted. The measurement results and security implications are discussed respectively in § A2 and in § IV-F.

1) Measurement Methodology:

```
int TestPublicRPC(Peer peerIP){
  int res;
  int[] ports = nmap(peerIP);
  for(int port : ports) {
    try{ res = getRPC(peerIP,port).getBlockNumber();
    } catch (Exception e) { if (e instanceOf Timeout)
     continue; }
    return port; }
  return -1;} // not a RPC node

bool TestHoneypot(Peer peerIP){
  //tx is a double-spend
  try{ txhash = peerIP.sendTransaction(tx);
  } catch(Exception e){ return false; }
  if (txhash > 0) return true; //is a honeypot
  else return false;}
```

Fig. A.14: Benchmark programs to characterize a public RPC peer

a) Module 1: Classify Ethereum Peers: A public RPC peer is an Ethereum peer who accepts RPC requests from anyone on the Internet. To distinguish a public RPC peer from a non-RPC peer, we aim to answer the following questions: 1) if a peer has a port that responds to incoming RPC requests, and 2) if a RPC peer is a honeypot.

To test if a peer supports RPC, we set up a measurement node to scan the ports of a target peer, using `nmap` [36] and starting with the default ports, 8545. For each open port identified, it then sends a RPC request (e.g., `eth_getBlockNumber()`) and observes any response before timeout.

A honeypot in this work is an Ethereum peer who does not follow the Ethereum protocol and conceals its derailing behavior to appear as a honest node. For instance, a honeypot peer falsely returns success to any received transactions without actually validating them, or it does validate a transaction but without propagating it. In practice, honeypot nodes exist for measurement or attacking purposes (e.g., useful to attract actual attackers without affecting the mainnet [48]). From the DoERS's point of view, a honeypot may not be a preferred target as it does not serve real DApp workloads. To test if a peer is a honeypot, our measurement node first sends to the target peer a double-spending transaction and observes the response. If the response is a success (meaning falsely admitting a double-spending transaction), the peer must be a honeypot. Otherwise, our measurement node then sends a valid transaction, and observe if the transaction will appear in other Ethereum peers. If not, the target peer is a honeypot. In our approach, the first step is free and the second step uses very low gas prices to reduce the cost.

b) Module 2: Test Gas Limits: Given a public Ethereum RPC peer identified by the previous Module 1, the goal of Module 2 is to test if there is any Gas limit configured on the peer, and if so how much the limit is.

Our test program, named by `rpc_gasLimit`, is in

```
float rpc_gasLimit(IP rpcNode){
  int lengthLower=0; int lengthUpper=500;//0/500 block gas
  while (lengthUpper - lengthLower > 1){
    arrayLength = (lengthLower + lengthUpper) / 2;
    try{
      rpcNode.eth_call(exhaustMem,arrayLength);
    } (Exception e) {
      if(e instanceOf OutofGasException){
        lengthUpper = arrayLength;
      } else { //no gas limits
        return 0;}
    } else {
      lengthLower = arrayLength;}}
  return localNode.estmateGas(exhaustMem,arrayLength);}
```

Fig. A.15: Measure Gas limit of an RPC node

List A.15. The goal of the test program is to find the maximal argument (`arrayLength` in function `exhaustMem()`) that does not trigger the out-of-gas exception, a value that implies the Gas limit. To do so, the program starts with an initial guess on the target `arrayLength` value, then grows the guess exponentially until the first exception is observed. It then enters the second phase that binary-search the Gas-limit corresponding value of `arrayLength`. After the target value V is obtained, the program then uses a local RPC node (under our control) to run `estimateGas()` with function `exhaustMem` under V. The returned value is the Gas limit. Note that our design uses `exhaustMem` function which consumes gas faster than the other two `exhaustXXs` and can finish before Ethereum's default 5-second timeout.

2) Measurement Results: **Public RPC peers**: We conduct a passive measurement study on the Ethereum DevP2P network [11] for a 96-hour period (from April. 08 to April. 12 2020). This passive measurement method is inspired by [57]. We launch eight measurement nodes and remove the default maximal number of the neighbors allowed on them, such that each node can be connected by as many Ethereum peers as possible. The dynamic nature of Ethereum and its peer discovery mechanism [37] ensures that our measurement nodes can be constantly discovered by and connected to new Ethereum peers. In the measurement period, we record the IPs of all the neighbors of our eight measurement nodes, and remove the duplicated ones. Additionally, for each connected neighbor, our measurement node gets engaged in the Ethereum handshake process[9] through which the network identifier of the neighbor is revealed (more specifically, in the genesis block header exchanged through the STATUS message). We then select the neighbor peers with the "mainnet" identifier. During this 96-hour period, the measurement node finds a total of 8924 distinct Ethereum peers in the mainnet.

Based on the 8924 mainnet peers, we run our benchmark scripts in Figure A.14 against each peer. It finds 439 public RPC peers, among which 348 (91) use the default (non-default) RPC ports (the default RPC port is 8545). Among the 439 public RPC peers in the mainnet, 436 are non-honeypots.

Gas limits: We then move forward to measure the Gas limits

[9]The handshake process runs a series of protocols including RLPX, DevP2P and Ethereum subprotocol, and the STATUS message is in the Ethereum subprotocol.

TABLE II: Measurement results of public RPC peers in Ethereum

Ethereum-mainnet peers	8927
RPC peers in mainnet (default/non-default port)	439 (348/91)
Public RPC peers (i.e., w. a RPC port, in mainnet and non-honeypot)	436
Public RPC peers without gas limits	348
Public RPC peers with gas limits	88

of these non-honeypot, mainnet, RPC peers (which we will simply refer to as public RPC peers). We used the script in Figure A.15. We found out of 436 public RPC peers, 88 have a non-zero Gas limit, among which two have a limit of 782.5 block gas, one has a limit of 214 block gas and all others running `Parity` have 50 block gas as the limit.[10]

Timeout: On nodes without gas limits, we measure the response time of a RPC call throwing timeout exception and observe the measured time on all nodes is consistent with the default 5 second timeout in EVM.

3) Attack Strategies and Security Implication: This result reveals two types of RPC nodes on Ethereum mainnet: 348 RPC peers without gas limits and 88 RPC peers configured with a quite loose gas limit (mostly 50 block gas).

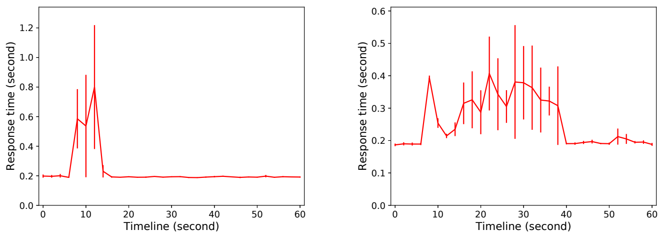
For the RPC node without gas limit, we design a single-request DoERS attack that has the power of evading all other protective measures we will observe in the next section (including rate limiting and load balancing). The attack sends a single request with a very large payload size (e.g., 10^9) to run the `exhaustMem` function in the `DoERS-C` smart contract. The key observation here is this: `exhaustMem` runs a single EVM instruction, namely `CODECOPY`, to allocate a large memory. Running a single EVM instruction is atomic and is not interrupted, even when there is a timeout. Thus, the DoERS attacker can increase the payload size of an `exhaustMem` invocation to evade the 5-second timeout, causing a higher resource consumption and more sever service damage, as will be evaluated in § V-B.

For the RPC node with gas limit, the gas limit, which is mostly 50 block gas, is rather loose. The attacker can send multiple DoERS requests, each with a medium payload size under the gas limit. If the requests are sent at a sufficiently high rate, there will be visible service interference, as will be evaluated in § V-A and § V-B.

The insecurity of RPC nodes further imply the insecurity of DApps. Because by default, RPC open to public clients is turned off on `Geth` [22] and `Parity` [61], the fact that these identified nodes have RPC manually turned on could mean they are intentionally used to host some DApps, as an in-house solution. The insecurity of these RPC nodes could affect the DApps hosted in house.

4) Evaluation on DoERS Effectiveness: We select a public node from our measurement result (i.e., the mainnet scan of IPs/ports) that did not set Gas limits. On this node, we mount a DoERS attack with following parameters:

[10]The default gas limit in `Parity` is 50 block gas.



(a) $\langle \texttt{Mem}, 80M, 0\rangle$ (b) $\langle \texttt{CPU}, 50K, 60\rangle$

Fig. A.16: Evaluate DoERS on a Public RPC Node (18.179.10.63:8545)

`eth_call(exhaustCPU(`$5 * 10^4$`))` at the rate of 60 requests per second. The result is reported in Figure A.16b. After the attack starts at the 5th second, the response time is increased by $2\times$ (from 0.2 seconds to 0.4 seconds).

On the same node, we then send the following single-request attack: `eth_call(exhaustMem(`$8 * 10^7$`))`. The result is in Figure A.16a which shows a $4\times$ increase of response time after the attack starts (from 0.2 seconds to 0.8 seconds).

B. Measuring Rate Limits on RPC Services

Many RPC services have deployed rate limiting on their frontend. We write a simple test program that sends RPCs at a certain rate for a period of time. During the measurement, we increase rates and vary the measurement duration, to observe if an "max rate reached" exception is thrown or if the response returns null. The exception means a maximal rate is reached. Here, what's sent are normal RPCs, such as `eth_getBlockNumber`.

The results in Table III show that the measured rate limits are often inconsistent with the numbers published on services' websites.

C. Estimating Peer Count

Recall (§ IV-A) that some services' load balancers depend on the timing of the requests. Based on the timing dependency, we can design further measurements and estimate the peer count of the service. To do so, we first measure the expiration time of an orphan transaction. To do so, we send an orphan

TABLE III: Characterizing Ethereum RPC services in gas and rate limits (in red are the detected absence of gas limits which poses vulnerability. Also can be seen is the inconsistency between the rate limits published on their websites and the rates revealed thru. the measurements.)

RPC services	Client	Rate limits (free tier)		Mining
		Published	Measured	
ServiceX1	N/A	3/sec.	$1 \sim 2$/sec.	✗
ServiceX2	Geth ethshared	Unlimited	> 7200/min.	✗
ServiceX3	Parity	2/sec.	< 2/sec.	✓
ServiceX4	Geth	Unlimited	$6000 \sim 6060$/min.	✗
ServiceX5	N/A	5/sec./IP	$2.33 \sim 2.66$/sec.	N/A
ServiceX6	Geth-omnibus	10^5/day	$(40.60 \sim 40.75) * 10^5$/day	✗
ServiceX7	Geth	400/min	$240 \sim 300$/min	✗
ServiceX9	Anonymized	Unlimited	$750 \sim 900$/min.	✗

transaction with $nonce + 2$ ($nonce$ is the nonce of the latest confirmed transaction) and wait t seconds before sending the second transaction with $nonce + 1$. The second transaction makes the orphan transaction become un-orphaned. Thus by checking whether the orphan transaction was propagated (thru. `eth_getTransaction`), one can know that if the orphan transaction has lived t seconds. In other words, the lifetime of an orphan transaction must be longer than t seconds. By varying t, one can get the exact expiration time of an orphan transaction. By this means, we measure ServiceX6 and obtain that an orphan transaction expires in 64 minutes.

With the knowledge of orphan expiration time t and the timing behavior, one can infer the peer count in a RPC service. We design the following experiment: In a period of 64 minutes, we send three orphan transactions every minute (in its first 20 seconds). We observe that all 192 orphan transactions succeeded. This result implies that there are at least 192 RPC peers in the backend of ServiceX6.[11]

We have measured ServiceX9's orphan expiration time in the similar fashion with ServiceX6 and obtained 40 second. This result implies there are at least 6 nodes in the backend of ServiceX9 service.

[11]Otherwise, with fewer than 192 peers, it would cause at two orphans to collide on the same peer, one of which must fail.