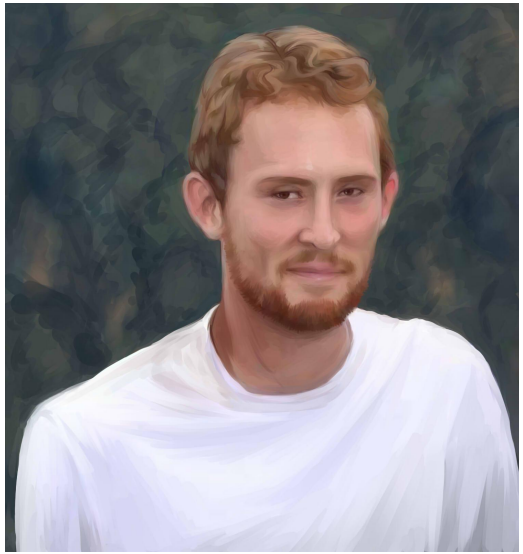
Ethereum at BitGo

Securing and Running A Multisig Ethereum Wallet

Alex Melville

Alex Melville (Technical Lead on the Coins Team)



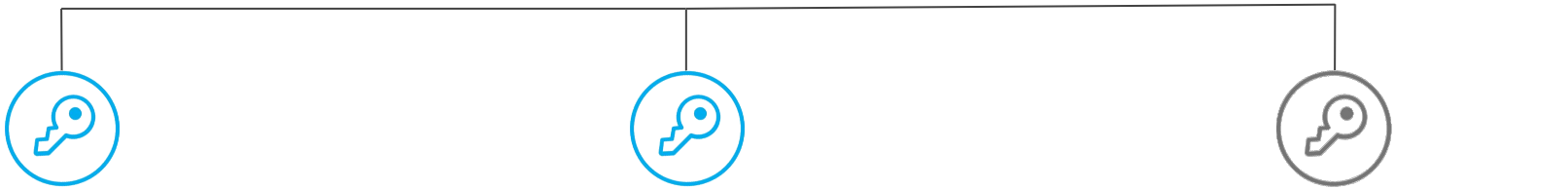
github.com/Melvillian

Today's Plan

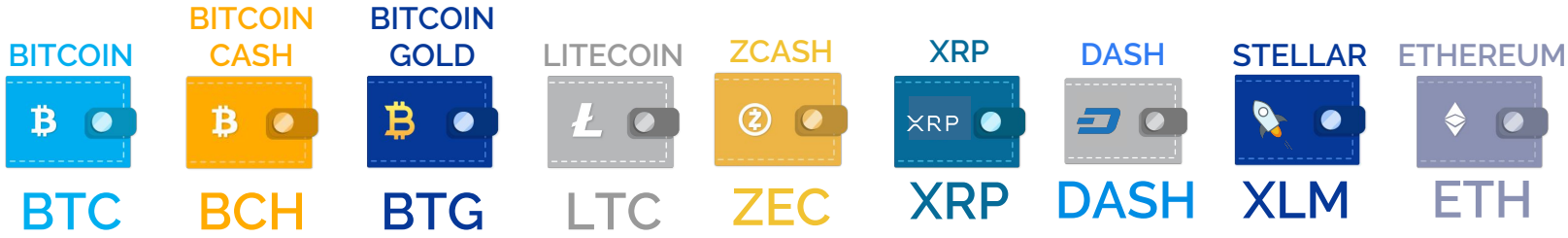
Questions!

BitGo: 2-of-3 Multi-Signature Wallets

No single point of failure!



Supported Coins and Tokens



- 0x (ZRX)
- Aelf (ELF)
- AION (AION)
- AirSwap (AST)
- ANA (ANA)
- AppCoins (APPC)
- Aragon (ANT)
- Augur (REP)
- Bancor (BNT)
- Basic Attention Token (BAT)
- Bread (BRD)
- Celsius (CEL)
- Change (CAG)
- Civic (CVC)
- CoinLion (LION)
- Colu Local Network (CLN)
- Cryptopay (CPAY)
- DAOstack (GEN)
- Decision Token (HST)
- Dent (DENT)
- Gemini Dollar (GUSD)
- Gnosis (GNO)
- Golem (GNT)
- Hold (HOLD)
- Indorse (IND)
- iShook (SHK)
- Kin (KIN)
- Kyber Network (KNC)
- Linker Coin (LNC)
- Maker (MKR)
- Numeraire (NMR)
- OmiseGo (OMG)
- OPTin Token (OPT)
- Paxos (PAX)
- PlusCoin (PLC)
- Polymath (POLY)
- Populous (PPT)
- Power Ledger (POWR)
- Propy (PRO)
- Pundi X (NPXS)
- Rebglo (REB)
- Rialto (XRL)
- Salt (SALT)
- Sentinel Protocol (UPP)
- Serenity (SRNT)
- Snovio (SNOV)
- Status Network Token (SNT)
- Storj (STORJ)
- Storm (STORM)
- SwissBorg (CHSB)
- TrueUSD (TUSD)
- WaltonChain (WTC)
- WeTrust (TRST)
- Worldwide Asset eXchange (WAX)
- Zilliqa (ZIL)

...with more being added all the time.

Ethereum Basics (Addresses vs. Contracts)

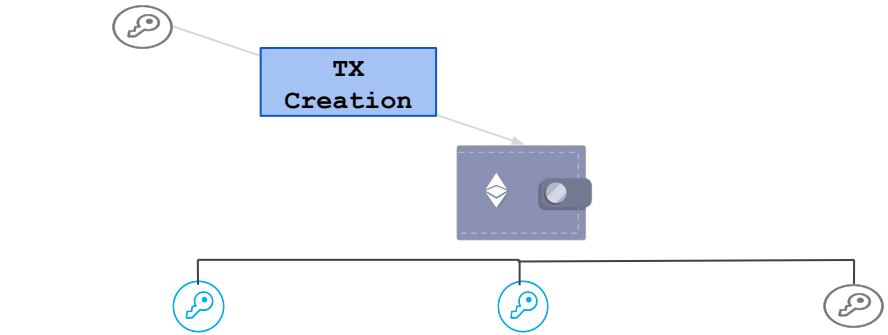
Addresses:

- Basically like Bitcoin Addresses
- Single Signature
- Initiate Contract Creation/Execution
- Pay for Fees! (EIP 101)

Contracts

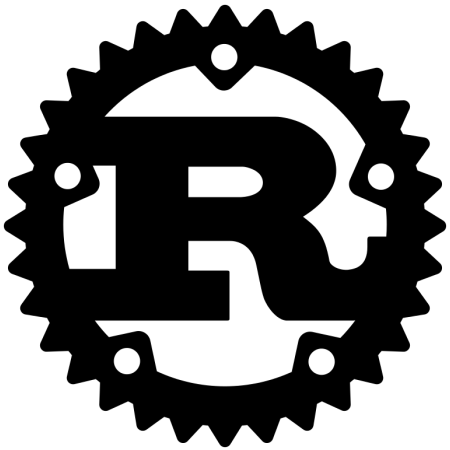
- Turing-Complete Execution
- Extremely Tricky to Reason About

External (Fee) Address



Ethereum Basics (Two Full Node Implementations)

Parity



Rust

Geth

Golang

Writing and Deploying A Multisig Wallet Smart Contract

The Code:

- Multisignature Wallet
- 12 functions
  - Sending
  - Safe Mode
  - Helper Functions
- Events
  - Transacted
  - Deposited
- web3.js (we don't use it!)

```
/**
 * Execute a multi-signature transaction from this wallet using 2 signers: one from msg.sender and the other from ecrecover.
 * Sequence IDs are numbers starting from 1. They are used to prevent replay attacks and may not be repeated.
 *
 * @param toAddress the destination address to send an outgoing transaction
 * @param value the amount in Wei to be sent
 * @param data the data to send to the toAddress when invoking the transaction
 * @param expireTime the number of seconds since 1970 for which this transaction is valid
 * @param sequenceId the unique sequence id obtainable from getNextSequenceId
 * @param signature see Data Formats
 */
function sendMultiSig(
    address toAddress,
    uint value,
    bytes data,
    uint expireTime,
    uint sequenceId,
    bytes signature
) public onlySigner {
  // Verify the other signer
  var operationHash = keccak256("ETHER", toAddress, value, data, expireTime, sequenceId);

  var otherSigner = verifyMultiSig(toAddress, operationHash, signature, expireTime, sequenceId);

  // Success, send the transaction
  if (!(toAddress.call.value(value)(data))) {
    // Failed executing transaction
    revert();
  }
  Transacted(msg.sender, otherSigner, operationHash, toAddress, value, data);
}
```

Writing and Deploying A Multisig Wallet Smart Contract

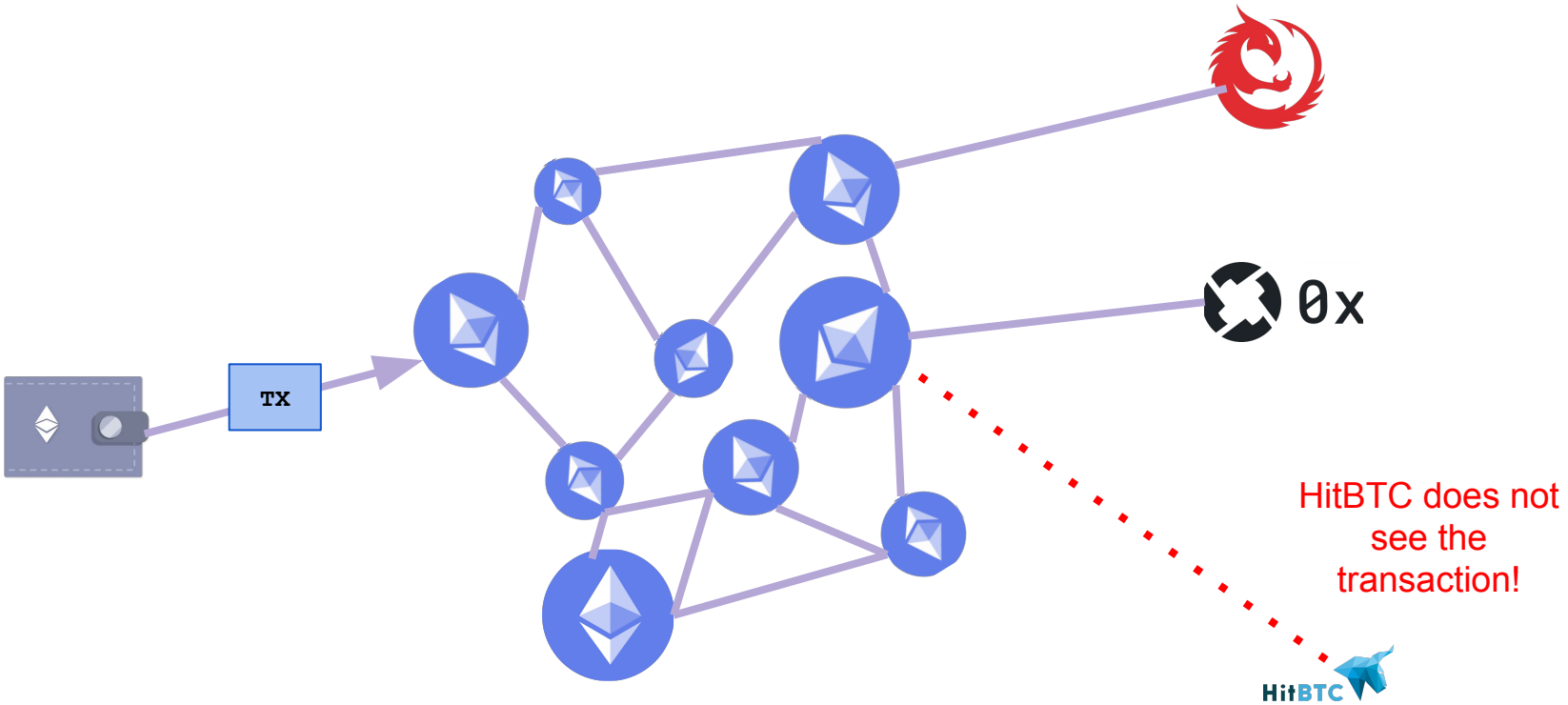
```
/**
    * Do common multisig verification for both eth sends and erc20token transfers
    *
    * @param toAddress the destination address to send an outgoing transaction
    * @param operationHash see Data Formats
    * @param signature see Data Formats
    * @param expireTime the number of seconds since 1970 for which this transaction is valid
    * @param sequenceId the unique sequence id obtainable from getNextSequenceId
    * returns address that has created the signature
    */
  function verifyMultiSig(
      address toAddress,
      bytes32 operationHash,
      bytes signature,
      uint expireTime,
      uint sequenceId
  ) private returns (address) {

    var otherSigner = recoverAddressFromSignature(operationHash, signature);

    // Verify if we are in safe mode. In safe mode, the wallet can only send to signers
    if (safeMode && !isSigner(toAddress)) {
      // We are in safe mode and the toAddress is not a signer. Disallow!
      revert();
    }
    // Verify that the transaction has not expired
    if (expireTime < block.timestamp) {
      // Transaction expired
      revert();
    }

    // Try to insert the sequence ID. Will revert if the sequence id was invalid
    tryInsertSequenceId(sequenceId);

    // Try to insert the sequence ID. Will revert if the sequence id was invalid
    tryInsertSequenceId(sequenceId);

    if (!isSigner(otherSigner)) {
      // Other signer not on this wallet or operation does not match arguments
      revert();
    }
    if (otherSigner == msg.sender) {
      // Cannot approve own transaction
      revert();
    }

    return otherSigner;
  }
```

Writing and Deploying A Multisig Wallet Smart Contract

The Deploy:

- Similar Care as Hardware
- Audits
  - OpenZeppelin
  - Coinspect
  - ABDK Consulting
- Fee Address
  - It Costs to Deploy Wallet and Receive Addresses

The Ethereum Multisig Problem: Fungibility and Usability



Multiparty ECDSA Signatures to the Rescue!

Fast Secure Two-Party ECDSA Signing*

Yehuda Lindell**

Dept. of Computer Science
Bar-Ilan University, ISRAEL
lindell@biu.ac.il

Abstract. ECDSA is a standard digital signature schemes that is widely used in TLS, Bitcoin and elsewhere. Unlike other schemes like RSA, Schnorr signatures and more, it is particularly hard to construct efficient threshold signature protocols for ECDSA (and DSA). As a result, the best-known protocols today for secure distributed ECDSA require running heavy zero-knowledge proofs and computing many large-modulus exponentiations for every signing operation. In this paper, we consider the specific case of two parties (and thus no honest majority) and construct a protocol that is approximately *two orders of magnitude faster* than the previous best. Concretely, our protocol achieves good performance, with a single signing operation for curve P-256 taking approximately 37ms between two standard machine types in Azure (utilizing a single core only). Our protocol is proven secure under standard assumptions using a game-based definition. In addition, we prove security by simulation under a plausible yet non-standard assumption regarding Paillier.

1 Introduction

1.1 Background

In the late 1980s and the 1990s, a large body of research emerged around the problem of *threshold cryptography*; cf. [3,8,10,11,14,26,25,22]. In its most general form, this problem considers the setting of a private key shared between n parties with the property that any subset of t parties may be able to decrypt or sign, but any set of less than t parties can do nothing. This is a specific example of secure multiparty computation, where the functionality being computed is

Fast Multiparty Threshold ECDSA with Fast Trustless Setup

Rosario Gennaro
City University of New York
rosario@cs.ccny.cuny.edu

Steven Goldfeder
Princeton University
stevenag@cs.princeton.edu

ABSTRACT

A threshold signature scheme enables distributed signing among n players such that any subgroup of size $t+1$ can sign, whereas any group with t or fewer players cannot. While there exist previous threshold schemes for the ECDSA signature scheme, we are the first protocol that supports multiparty signatures for any $t \leq n$ with an efficient dealerless key generation. Our protocol is faster than previous solutions and significantly reduces the communication complexity as well. We prove our scheme secure against malicious adversaries with a dishonest majority. We implemented our protocol, demonstrating its efficiency and suitability to be deployed in practice.

ACM Reference Format:
Rosario Gennaro and Steven Goldfeder. 2018. Fast Multiparty Threshold ECDSA with Fast Trustless Setup. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243859

1 INTRODUCTION

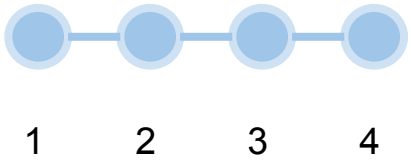
A threshold signature scheme enables n parties to share the power to issue digital signatures under a single public key. A threshold t is specified such that any subset of $t+1$ players can jointly sign, but any smaller subset cannot. Generally, the goal is to produce that $t < n/2$. Moreover, their scheme requires $2t+1$ players to participate to generate a signature. This is not ideal for several reasons. Firstly, it rules out the possibility of an n-of-n threshold signing scheme. Secondly, it provides an attacker with additional targets: while an attacker only needs to compromise $t+1$ servers, the scheme requires $2t+1$ servers to generate a signature.

As Gennaro *et al.*'s scheme did not suport the n-of-n case, Mackenzie and Reiter built a scheme specifically for the 2-of-2 case (i.e. $t=1$ and $n=2$) [27]. Recently much improved 2-out-of-2 schemes have been presented [12, 26]. However 2-out-of-2 sharing is very limited and can't express more flexible sharing policies that might be required in certain applications.

Gennaro and others in [17] (improved in [4]) address the more general (t, n) case in the *threshold optimal* case, meaning $n \geq t+1$ and that only $t+1$ players are needed to sign. However, their scheme too has a setback in that the distributed key generation protocol is very costly and impractical.

Our Result: We present a new threshold-optimal protocol for ECDSA that improves in many significant ways over [4, 17]. Our protocol supports a highly efficient distributed key generation; it also supports faster signing than [4, 17], and requires far less data to be transmitted between the parties (details of the comparison appear below).

Transaction Nonce Holes: Full Node



TX #1
1 ETH

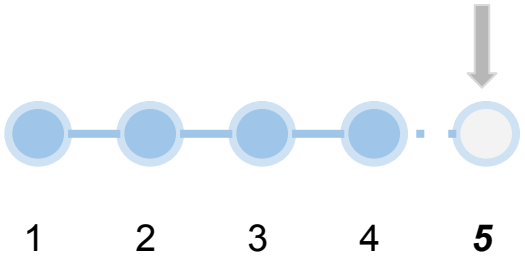
TX #2
3 ETH

TX #3
5 ETH

TX #4
7 ETH

= Confirmed

Transaction Nonce Holes: Full Node



TX #1
1 ETH

TX #2
3 ETH

TX #3
5 ETH

TX #4
7 ETH

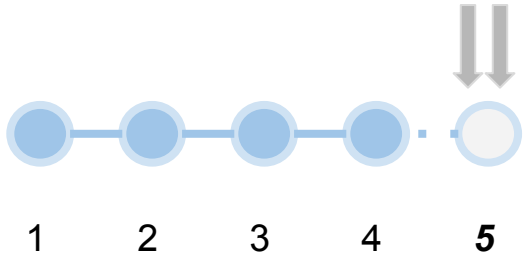
TX #5
9 ETH

= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Full Node



TX #1
1 ETH

TX #2
3 ETH

TX #3
5 ETH

TX #4
7 ETH

TX #5
9 ETH

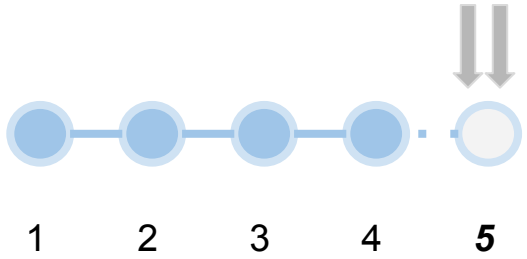
TX #6
11 ETH

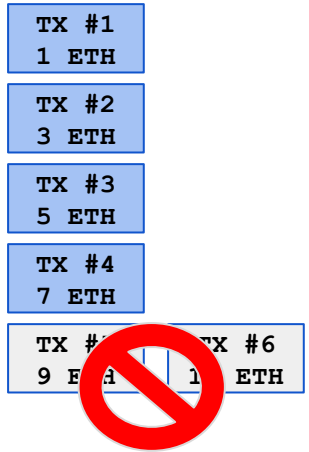
= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Full Node



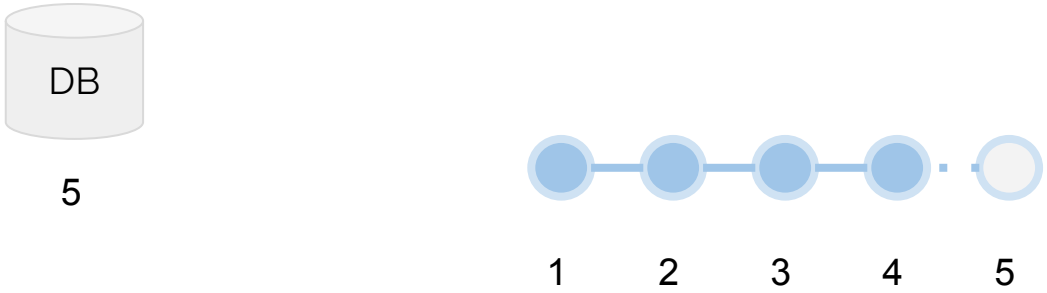


= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Database

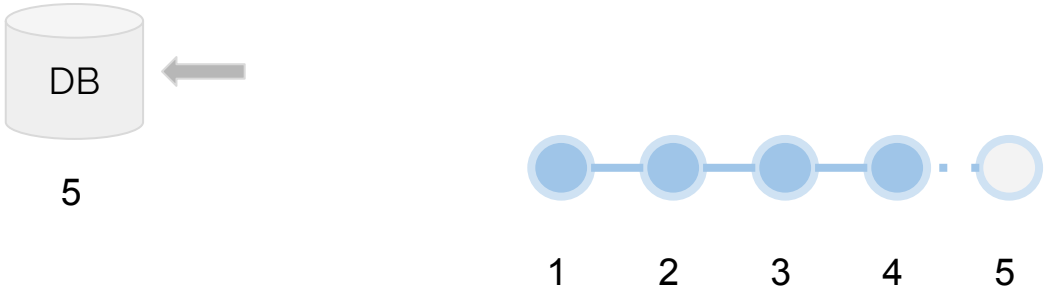


= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Database

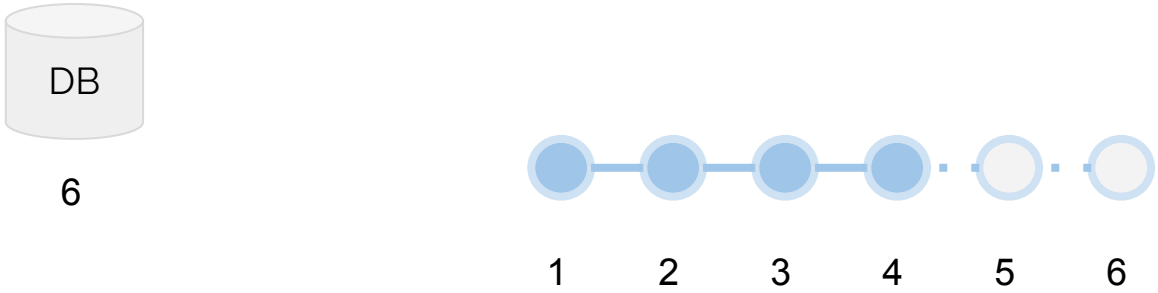


= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Database

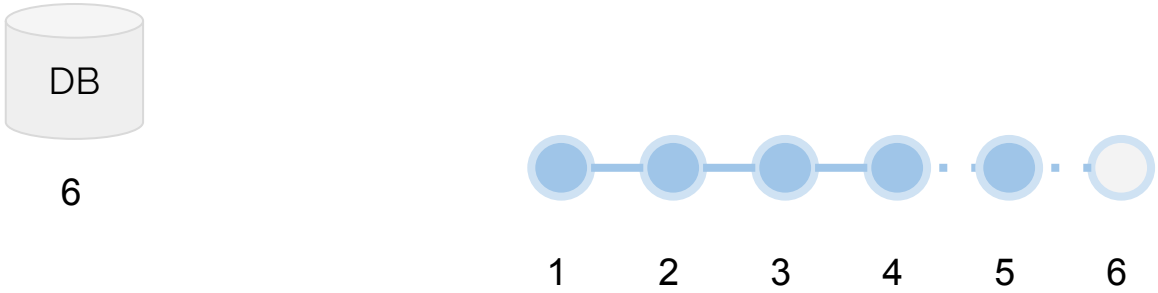


= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Database

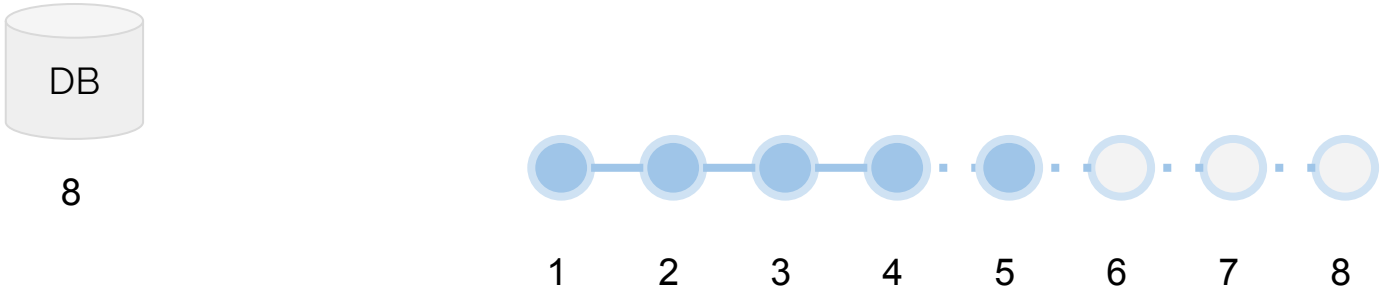


= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Database

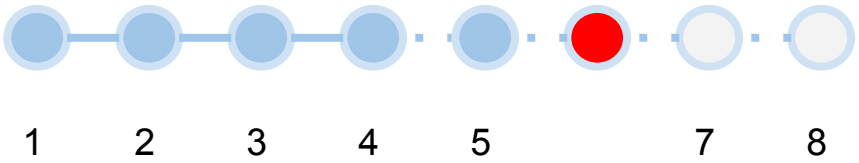


= Unconfirmed

= DB Query

= Confirmed

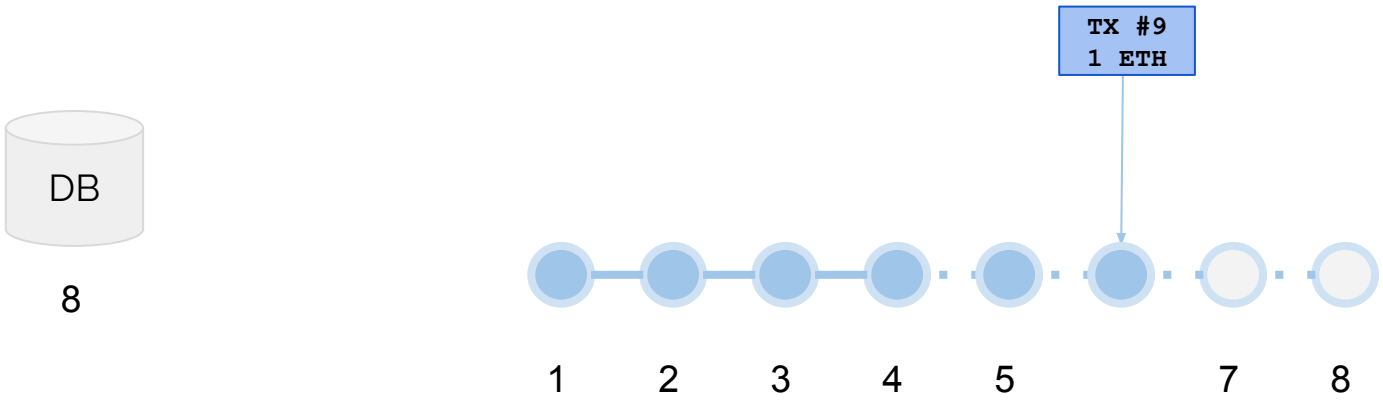
Transaction Nonce Holes: Database



= DB Query

= Nonce Confirmed

Transaction Nonce Holes: Database

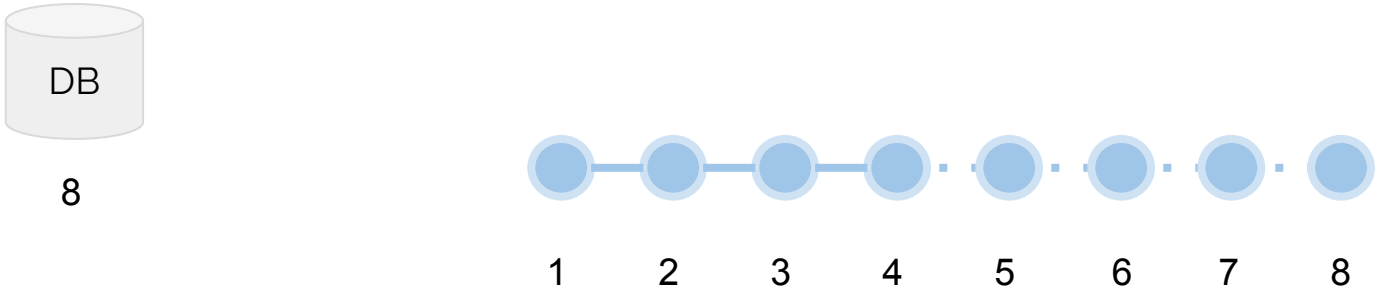


= Unconfirmed

= DB Query

= Confirmed

Transaction Nonce Holes: Database

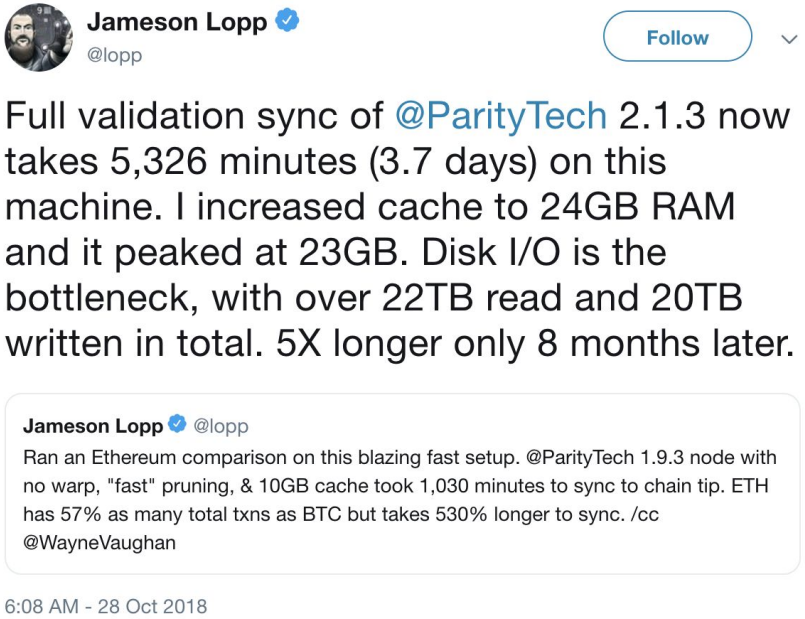


= Unconfirmed

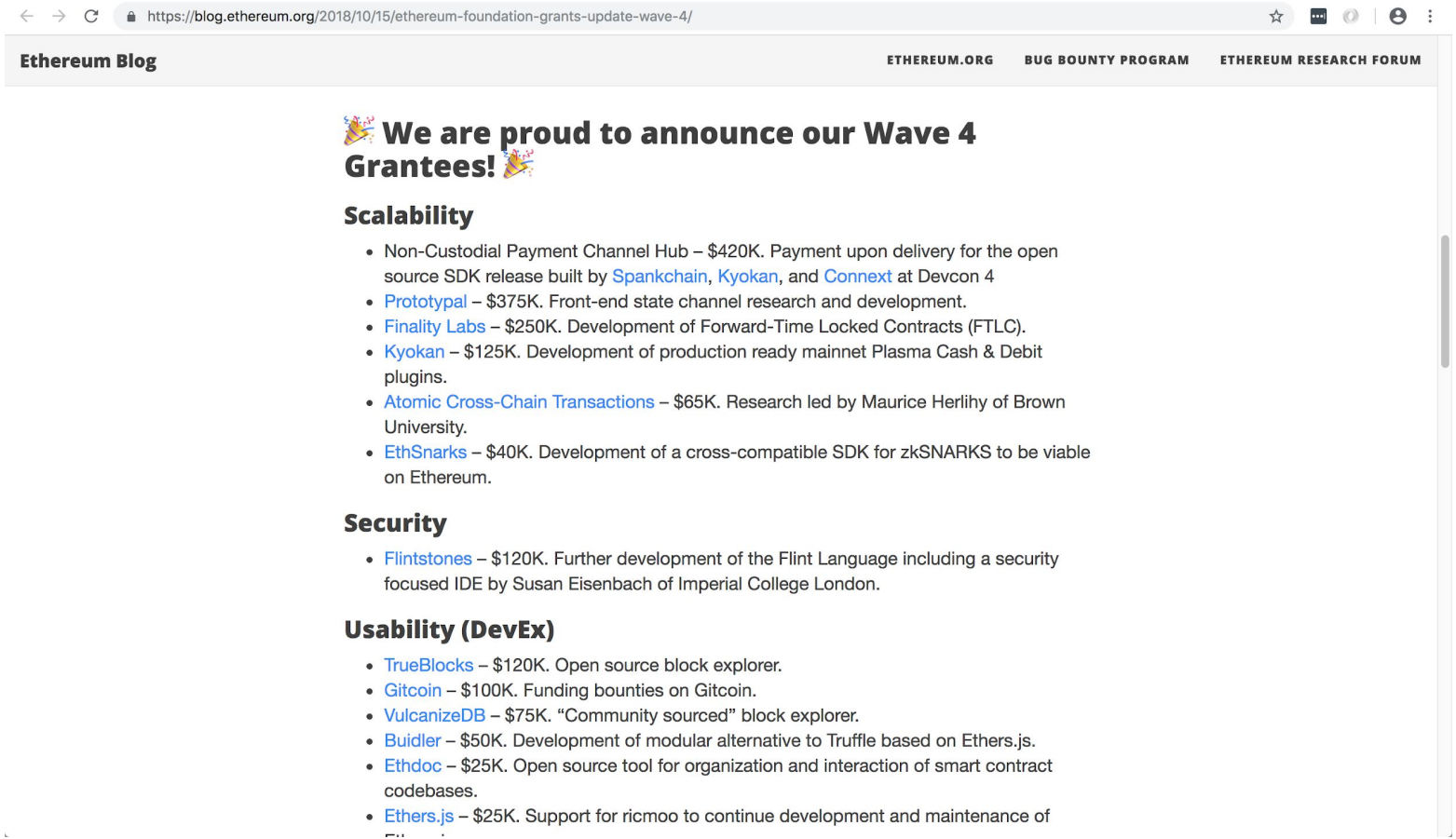
= DB Query

= Confirmed

Learning to Love/Hate Parity



Right At Your Fingertips



https://blog.ethereum.org/2018/10/15/ethereum-foundation-grants-update-wave-4/

Ethereum Blog ETHEREUM.ORG BUG BOUNTY PROGRAM ETHEREUM RESEARCH FORUM

🎉 We are proud to announce our Wave 4 Grantees! 🎉

Scalability

- Non-Custodial Payment Channel Hub – $420K. Payment upon delivery for the open source SDK release built by Spankchain, Kyokan, and Connext at Devcon 4
- Prototypal – $375K. Front-end state channel research and development.
- Finality Labs – $250K. Development of Forward-Time Locked Contracts (FTLC).
- Kyokan – $125K. Development of production ready mainnet Plasma Cash & Debit plugins.
- Atomic Cross-Chain Transactions – $65K. Research led by Maurice Herlihy of Brown University.
- EthSnarks – $40K. Development of a cross-compatible SDK for zkSNARKS to be viable on Ethereum.

Security

- Flintstones – $120K. Further development of the Flint Language including a security focused IDE by Susan Eisenbach of Imperial College London.

Usability (DevEx)

- TrueBlocks – $120K. Open source block explorer.
- Gitcoin – $100K. Funding bounties on Gitcoin.
- VulcanizeDB – $75K. “Community sourced” block explorer.
- Buidler – $50K. Development of modular alternative to Truffle based on Ethers.js.
- Ethdoc – $25K. Open source tool for organization and interaction of smart contract codebases.
- Ethers.js – $25K. Support for ricmoo to continue development and maintenance of

Thank You!