Mastering Ethereum

Preface

Writing Mastering Ethereum

Intended audience

Why are there bees on the cover?

The Western honey bee (Apis mellifera) is a species that exhibits highly complex behavior that, ultimately, benefits the hive. Each individual bee operates freely under a set of simple rules and communicate findings of importance by 'dancing'. This dance carries valuable information like the position of the sun and relative geographical coordinates from the hive to the target in question. By interpreting this dance, the bees can relay on this information or act on it, thus, carrying out the will of the 'hive-mind'.

Although bees form a caste-based society and have a queen for producing offspring, there is no central authority or leader in a beehive. The highly intelligent and sophisticated behavior exhibited by a multi thousand member colony is an emergent property from the interaction of the individuals in a social network.

Nature demonstrates that decentralized systems can be resilient and can produce emergent complexity and incredible sophistication without the need for a central authority, hierarchy, or complex parts.

Conventions used in this book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP	This icon signifies a tip or suggestion.

NOTE	This icon signifies a general note.
WARNING	This icon indicates a warning or caution.

Code examples

The examples are illustrated in Solidity, JavaScript and Python, and using the command line of a Unix-like operating system. All code snippets are available in the GitHub repository in the *code* subdirectory of the main repository. Fork the book code, try the code examples, or submit corrections via GitHub:

https://github.com/ethereumbook/ethereumbook/tree/first_edition/

All the code snippets can be replicated on most operating systems with a minimal installation of compilers, interpreters and libraries for the corresponding languages. Where necessary, we provide basic installation instructions and step-by-step examples of the output of those instructions.

Some of the code snippets and code output have been reformatted for print. In all such cases, the lines have been split by a backslash (\) character, followed by a newline character. When transcribing the examples, remove those two characters and join the lines again and you should see identical results as shown in the example.

All the code snippets use real values and calculations where possible, so that you can build from example to example and see the same results in any code you write to calculate the same values. For example, the private keys and corresponding public keys and addresses are all real. The sample transactions, contracts, blocks, and blockchain references have all been introduced in the actual Ethereum blockchain and are part of the public ledger, so you can review them on any Ethereum system.

Using code examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, ISBN, and copyright. For example: "*Mastering Ethereum* by Andreas M. Antonopoulos and Gavin Wood (O'Reilly), 978-1-491-97194-9. Copyright 2018."

Some editions of this book are offered under an open source license, such as CC-BY-NC, in which case the terms of that license apply.

If you feel your use of code examples falls outside fair use or the permission given above, feel free

to contact us at permissions@oreilly.com.

Ethereum addresses and transactions in this book

The Ethereum addresses, transactions, keys, QR codes, and blockchain data used in this book are, for the most part, real. That means you can browse the blockchain, look at the transactions offered as examples, retrieve them with your own scripts or programs, etc.

However, note that the private keys used to construct addresses are either printed in this book, or have been "burned." That means that if you send money to any of these addresses, the money will either be lost forever, or in some cases everyone who can read the book can take it using the private keys printed in here.

WARNING	DO NOT SEND MONEY TO ANY OF THE ADDRESSES IN THIS BOOK. Your money will be taken by another reader, or lost forever.

O'Reilly safari

NOTE	*Safari* (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to contact us

Please address comments and questions concerning this book to the publisher:

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at https://www.oreilly.com.

Find us on Facebook: https://facebook.com/oreilly

Follow us on Twitter: https://twitter.com/oreillymedia

Watch us on YouTube: https://www.youtube.com/oreillymedia

Contacting the authors

Information about "Mastering Ethereum" as well as the Open Edition and translations are available on: https://ethereumbook.info/

Contacting Andreas

You can contact, Andreas M. Antonopoulos, on his personal site: https://antonopoulos.com/

Follow Andreas on Facebook: https://facebook.com/AndreasMAntonopoulos

Follow Andreas on Twitter: https://twitter.com/aantonop

Follow Andreas on Linkedin: https://linkedin.com/company/aantonop

Andreas would also like to thank all of the patrons who support his work through monthly donations. You can follow Andreas' Patreon page here: https://patreon.com/aantonop

Contacting Gavin

Acknowledgments by Andreas

I owe my love of words and books to my mother, Theresa, who raised me in a house with books lining every wall. My mother also bought me my first computer in 1982, despite being a self-described technophobe. My father, Menelaos, a civil engineer who published his first book at 80 years old, was the one who taught me logical and analytical thinking and a love of science and engineering.

Thank you all for supporting me throughout this journey.

Acknowledgments by Gavin

Early release draft (GitHub contributions)

Many contributors offered comments, corrections, and additions to the early-release draft on GitHub. Thank you all for your contributions to this book.

Following is an alphabetically sorted list of notable GitHub contributors, including their GitHub ID in parentheses:

- Abhishek Shandilya (abhishandy)
- Adam Zaremba (zaremba)
- Alejandro Santander (ajsantander)
- Alejo Salles (fiiiu)
- Alex Manuskin (amanusk)
- Alex Van de Sande (alexvandesande)

- Anthony Lusardi (pyskell)
- Assaf Yossifoff (assafy)
- Ben Kaufman (ben-kaufman)
- Brian Ethier (dbe)
- Bryant Eisenbach (fubuloubu)
- Chanan Sack (chanan-sack)
- Christopher Gondek (christophergondek)
- Chris Remus (chris-remus)
- Cornell Blockchain (CornellBlockchain)
    - Alex Frolov (sashafrolov)
    - Brian Guo (BrianGuo)
    - Brian Leffew (bleffew99)
    - Giancarlo Pacenza (GPacenza)
    - Lucas Switzer (LucasSwitz)
    - Ohad Koronyo (ohadh123)
    - Richard Sun (richardsfc)
- Dan Shields (NukeManDan)
- Daniel McClure (danielmcclure)
- Denis Milicevic (D-Nice)
- Dennis Zasnicoff (zasnicoff)
- Diego H. Gurpegui (diegogurpegui)
- Dimitris Tsapakidis (dimitris-t)
- Flash Sheridan (FlashSheridan)
- Franco Daniel Berdun (fMercury)
- Hon Lau (masterlook)
- Hudson Jameson (Souptacular)
- Iuri Matias (iurimatias)
- Ivan Molto (ivanmolto)
- Jason Hill (denifednu)
- Javier Rojas (fjrojasgarcia)
- Joel Gugger (guggerjoel)
- Jonathan Velando (rigzba21)
- Jon Ramvi (ramvi)
- Jules Lainé (fakje)
- Kevin Carter (kcar1)

- Krzysztof Nowak (krzysztof)
- Leo Arias (elopio)
- Luke Schoen (ltfschoen)
- Liang Ma (liangma)
- Martin Berger (drmartinberger)
- Matthew Sedaghatfar (sedaghatfar)
- Mike Pumphrey (bmmpxf)
- Mobin Hosseini (iNDicat0r)
- Nagesh Subrahmanyam (chainhead)
- Nichanan Kesonpat (nichanank)
- Nick Johnson (arachnid)
- Pet3rpan (pet3r-pan)
- Pierre-Jean Subervie (pjsub)
- Pong Cheecharern (Pongch)
- Qiao Wang (qiaowang26)
- Raul Andres Garcia (manilabay)
- Roger Häusermann (haurog)
- Solomon Victorino (bitsol)
- Steve Klise (sklise)
- Sylvain Tissier (SylTi)
- Tim Nugent (timnugent)
- Timothy McCallum (tpmccallum)
- Tomoya Ishizaki (zaq1tomo)
- Vignesh Karthikeyan (meshugah)
- Will Binns (wbnns)
- Xavier Lavayssière (xalava)
- Yash Bhutwala (yashbhutwala)
- Yeramin Santana (ysfdev)
- Zhen Wang (zmxv)
- ztz (zt2)

Quick Glossary

This quick glossary contains many of the terms used in relation to Ethereum. These terms are used throughout the book, so bookmark this for quick reference.

Account

An object containing an address, balance, and nonce, and optional storage and code. An account can be a contract account or an EOA (externally owned account).

Address

Most generally, this represents an EOA or contract, which can receive (destination address) or send (source address) transactions on the blockchain. More specifically, it is the right-most 160 bits of a Keccak hash of an ECDSA public key.

Assert

In Solidity assert(false) compiles to **0xfe**, which is an invalid opcode, using up all remaining gas, and reverting all changes. When an assert() statement fails, something very wrong and unexpected should be happening, and you will need to fix your code. You should use assert to avoid conditions which should never, ever occur.

Big-endian

A positional number representation where the most significant digit is first. The opposite of little-endian, where the least significant digit is first.

BIP

Bitcoin Improvement Proposals. A set of proposals that members of the Bitcoin community have submitted to improve Bitcoin. For example, BIP-21 is a proposal to improve the Bitcoin uniform resource identifier (URI) scheme.

Block

A block is a collection of required information (a block header) about the comprised transactions, and a set of other block headers known as ommers. It is added to the Ethereum network by miners.

Blockchain

A sequence of blocks validated by the proof-of-work system, each linking to its predecessor all the way to the genesis block. This varies from the Bitcoin protocol in that it does not have a block size limit; it instead uses varying gas limits.

Bytecode

Byzantium Fork

Byzantium is the first of two hard forks for Metropolis development stage. It included EIP-649: Metropolis Difficulty Bomb Delay and Block Reward Reduction, where the Ice Age (see below) was delayed by 1 year, and the block reward was reduced from 5 to 3 ether.

Compiling

Converting code written in a high-level programming language (e.g. Solidity) into a lower level language (e.g. EVM bytecode).

Consensus

When numerous nodes, usually most nodes on the network, all have the same blocks in their locally-validated best block chain. Not to be confused with consensus rules.

Consensus rules

The block validation rules that full nodes follow to stay in consensus with other nodes. Not to be confused with consensus.

Constantinople

The second part of the Metropolis stage, planned for mid-2018. Expected to include a switch to hybrid Proof-of-Work/Proof-of-Stake consensus algorithm, among other changes.

Contract account

An account containing code that executes whenever it receives a transaction from another account (EOA or contract).

Contract creation transaction

A special transaction, with the "zero address" as the recipient, that is used to register a contract and record it on the Ethereum blockchain (see "zero address").

DAO

Decentralised Autonomous Organization. Companies and other organizations which operate without hierarchical management. Also may refer to a contract named "The DAO" launched on 30th April 2016, which was then hacked in June 2016 and ultimately motivated a hard fork (codenamed DAO) at block #1,192,000 which reversed the hacked DAO contract, and caused Ethereum and Ethereum Classic to split into two competing systems.

DApp

Decentralised Application. At a minimum, it is a smart contract and a web user-interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services. In addition, many DApps include decentralized storage and/or message protocol and platform.

Deed

Non-fungible token standard introduced in ERC721 proposal. Unlike ERC20 tokens, deeds prove ownership and are not interchangeable, though they are not recognized as legal documents in any jurisdiction, at least not currently.

Difficulty

A network-wide setting that controls how much computation is required to produce a proof of work.

Digital signature

A digital signing algorithm is a process by which a user can produce a short string of data called a "signature" of a document using a private key such that anyone with the corresponding public key, the signature, and the document can verify that (1) the document was "signed" by the owner of that particular private key, and (2) the document was not changed after it was signed.

ECDSA

Elliptic Curve Digital Signature Algorithm, or ECDSA, is a cryptographic algorithm used by Ethereum to ensure that funds can only be spent by their rightful owners.

EIP

Ethereum Improvement Proposals describe proposed standards for the Ethereum platform. An EIP is a design document providing information to the Ethereum community, describing a new feature or its processes or environment. For more information, see https://github.com/ethereum/EIPs (also see the definition for ERC, below).

Entropy

In the context of Cryptography, lack of predictability, or level of randomness. When generating secret information, such as master private keys, algorithms usually rely on a source of high entropy to ensure the output is unpredictable.

ENS

Ethereum Name Service. For more information, see https://github.com/ethereum/ens/.

EOA

Externally Owned Account. Accounts created by or for human users of the Ethereum network.

ERC

Ethereum Request for Comments. Some EIPs are labeled as ERCs, which denotes proposals attempting to define a specific standard of Ethereum usage.

Ethash

A Proof-of-Work algorithm for Ethereum 1.0. For more information, see https://github.com/ethereum/wiki/wiki/Ethash.

Ether

Ether is the native cryptocurrency used by the Ethereum ecosystem which covers gas costs when executing Smart Contracts. Its symbol is Ξ, the Greek uppercase Xi character.

Event

An event allows the use of EVM logging facilities, which in turn can be used to call JavaScript callbacks in the user interface of a DApp, which listen for these events. For more information, see http://solidity.readthedocs.io/en/develop/contracts.html#events.

EVM

Ethereum Virtual Machine, a stack-based virtual machine which executes bytecode. In Ethereum, the execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a formal model of a virtual state machine.

EVM Assembly Language

A human-readable form of EVM bytecode.

Fallback function

A default function, called in the absence of data or a declared function name.

Faucet

A website that dispenses rewards in the form of free test ether for developers who want to do

test on testnets.

Frontier

The initial test development stage of Ethereum, which lasted from July 2015 to March 2016.

Ganache

Personal Ethereum blockchain which you can use to run tests, execute commands, and inspect state while controlling how the chain operates.

TODO: Change for Clarity

Gas

A virtual fuel used in Ethereum to execute smart contracts. The Ethereum Virtual Machine uses an accounting mechanism to measure the consumption of gas and constrain (limit) the consumption of computing resources. See Turing-Complete. Gas is a unit of computation that is incurred per instruction of Smart Contract executed. The gas is pegged at Ether cryptocurrency. The gas is analogous to talk time on a cellular network. Thus, the price of running a transaction in fiat currency is `gas * (ETH/gas) * (fiat currency/ETH)`.

Gas limit

When talking about blocks, they too, have a field called gas limit. It defines the maximum amount of gas all transactions in the whole block combined are allowed to consume.

Genesis block

The first block in the blockchain, used to initialize a particular network and its cryptocurrency.

Geth

Go Ethereum. One of the most prominent implementations of the Ethereum protocol written in Go.

Hard fork

Hard fork, also known as Hard-Forking Change, is a permanent divergence in the blockchain, commonly occurs when non-upgraded nodes can't validate blocks created by upgraded nodes that follow newer consensus rules. Not to be confused with fork, soft fork, software fork or Git fork.

Hash

A fixed-length fingerprint of variable-size input produced by a hash function.

HD wallet

Wallets using the Hierarchical Deterministic (HD Protocol) key creation and transfer protocol (BIP32).

HD wallet seed

HD wallet seed or root seed is a potentially-short value used as a seed to generate the master private key and master chain code for an HD wallet. The wallet seed can be represented by mnemonic words making it easier for humans to copy, backup and restore private keys.

Homestead

The second development stage of Ethereum, launched in March 2016 at block #1,150,000.

Ice Age

A hard fork of Ethereum at block #200,000 to introduce an exponential difficulty increase (aka Difficulty Bomb), motivating a transition to Proof-of-Stake.

IDE (Integrated Development Environment)

An integrated user interface that combines a code editor, compiler, runtime, and a debugger.

Immutable Deployed Code Problem

Once a contract's (or library's) code is deployed it becomes immutable. Being able to fix possible bugs and add new features is key for the software development cycle. This represents a challenge for smart contract development.

Inter exchange Client Address Protocol (ICAP)

An Ethereum Address encoding that is partly compatible with the International Bank Account Number (IBAN) encoding, offering a versatile, checksummed and interoperable encoding for Ethereum Addresses. ICAP addresses can encode Ethereum Addresses or common names registered with an Ethereum name registry. They always begin with XE. The aim is to introduce a new IBAN country code: XE, Ethereum E prefixed with the "extended" X, as used in non-jurisdictional currencies (e.g. XBT, XRP, XCP).

Internal transaction (also "message")

A transaction sent from a contract account to another contract account or an EOA.

Keccak256

Cryptographic hash function used in Ethereum. Keccak256 was standardised to SHA-3.

Key Derivation Function (KDF)

Also known as a password stretching algorithm, it is used by keystore format which to protect against brute-force, dictionary, or rainbow table attacks against the passphrase encryption. It repeatedly hashes the passphrase.

Keystore File

A JSON-encoded file that contains a single (randomly generated) private key, encrypted by a passphrase for extra security.

LevelDB

LevelDB is an open source on-disk key-value store. LevelDB is a light-weight, single-purpose library for persistence with bindings to many platforms.

Library

A library in Ethereum is a special type of contract that has no payable functions, no fallback function, and no data storage. Therefore, it cannot receive or hold ether, or store data. A library serves as previously deployed code that other contracts can call for read-only computation.

Lightweight client

A lightweight client is an Ethereum client that does not store a local copy of the blockchain, or validate blocks and transactions. It offers the functions of a wallet and can create and broadcast transactions.

Merkle Patricia Tree

Message

An internal transaction that is never serialized and only sent within the EVM.

Metropolis Stage

Metropolis is the third development stage of Ethereum, launched in October 2017.

METoken

Mastering Ethereum Token. An ERC20 token used for demonstration in this book.

Miner

A network node that finds valid proof of work for new blocks, by repeated hashing.

Mist

Mist is the first ever Ethereum enabled browser, built by the Ethereum Foundation. It also contains a browser based wallet that was the first ever implementation of the ERC20 token standard (Fabian Vogelsteller, author of ERC20 was also the main developer in Mist). Mist was also the first wallet to introduce the camelCase checksum (EIP-155, see [eip-155]). Mist runs a full node, and offers a full DApp browser with support for Swarm based storage and ENS addresses.

Network

A peer-to-peer network that propagates transactions and blocks to every Ethereum node (network participant).

Node

A software client that is participating in the peer-to-peer network.

Nonce

In cryptography, the term nonce is used to refer to a value that can only be used once. There are two types of nonce used in Ethereum.

- Account nonce - It's simply the transaction count of an account.
- Proof of work nonce - The random value in a block that was used to get the proof of work satisfied (depending on the difficulty at the time).

Ommer

A child block of an ancestor that is not itself an ancestor. When a miner finds a valid block, another miner may have published a competing block which is added to the tip of the blockchain. Unlike bitcoin, orphaned blocks in Ethereum can be included by newer blocks as ommers and receive a partial block reward. The term "ommer" is the preferred gender neutral term for the sibling of a parent node, but is also referred to as an "uncle".

Paralysis Problem

A common powerful approach to key management for cryptocurrencies is multisig transactions,

referred to more generally as secret sharing. But, what would happen if one of the shared keys was lost? The result would be a complete loss of all of the funds.
This isn't the only bad scenario. It's also possible that the key-share holders have different ideas about how the money should be spent, and can't come to an agreement.
We use the term *Paralysis Problem* to denote any of these awkward situations.

Paralysis Proof System

Paralysis Proofs help address a pervasive key-management problem in cryptocurrencies. See **Paralysis Problem**.
A Paralysis Proof System can tolerate system paralysis in settings where players fail to act in concert.
A Paralysis Proof System can be realized relatively easily for Ethereum using a smart contract.

Parity

One of the most prominent interoperable implementations of the Ethereum client software.

Proof-of-Stake (PoS)

Proof-of-Stake is a method by which a cryptocurrency blockchain protocol aims to achieve distributed consensus. Proof-of-Stake asks users to prove ownership of a certain amount of cryptocurrency (their "stake" in the network) in order to be able to participate to the validation of transactions.

Proof-of-Work (PoW)

A piece of data (the proof) that requires significant computation to find. In Ethereum, miners must find a numeric solution to the Ethash algorithm that meets a network-wide difficulty target.

Receipt

Data returned by an Ethereum client to represent the result of a particular transaction, including a hash of the transaction, its block number, the amount of gas used and, in case of deployment of a Smart Contract, the address of the Contract.

Reentrancy Attack

This attack can be reproduced when the Attacker contract calls to a Victim contract function, let's call it victim.withdraw(), in manner that before the original call to that contract function ever finishes, it calls the victim.withdraw() method again which continues to recursively call itself. This recursive call can be implemented from a fallback function of the Attacker contract. The only trick that the attacker has to perform is to break that recursive call before running out of gas and so avoiding the stolen ether be reverted.

Require

In Solidity, require(false) compiles to **0xfd** which is the **REVERT** opcode. The REVERT instruction provides a way to stop execution and revert state changes, without consuming all provided gas and with the ability to return a reason.
The require function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts.
Prior to the **Byzantium** network upgrade there were two practical ways to revert a transaction: running out of gas or executing an invalid instruction. Both of these options consumed all

remaining gas.
When you look up this opcode in the **Yellow Paper** prior to the **Byzantium** network upgrade, you can't find it and because there was no specification for that opcode, when the EVM reached it, it thrown an *invalid opcode error*.

Revert

Use revert() when you need to handle the same type of situations as require() but with more complex logic. For instances, if your code have some nested if/else logic flow, you will find that it makes sense to use require() instead of require().

Reward

An amount, in Ether (ETH), included in each new block as a reward by the network to the miner who found the Proof-of-Work solution.

Recursive Length Prefix (RLP)

RLP is an encoding standard, designed by the Ethereum developers to encode and serialize objects (data structures) of arbitrary complexity and length.

Satoshi Nakamoto

Satoshi Nakamoto is the name used by the person or people who designed Bitcoin and created its original reference implementation, Bitcoin Core. As a part of the implementation, they also devised the first blockchain database. In the process they were the first to solve the double spending problem for digital currency. Their real identity remains unknown.

Singleton

Vitalik Buterin

Vitalik Buterin is a Russian-Canadian programmer and writer primarily known as the co-founder of Ethereum and as the co-founder of Bitcoin Magazine.

Gavin Wood

Gavin Wood is a British programmer who is the co-founder and former CTO of Ethereum. In August 2014 he proposed Solidity, a contract-oriented programming language for writing smart contracts.

Secret key (aka private key)

The secret number that allows Ethereum users to prove ownership of an account or contracts, by producing a digital signature (see public key, address, ECDSA).

SHA

The Secure Hash Algorithm or SHA is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST).

SELFDESTRUCT opcode

Smart contracts will exist and be executable as long as the whole network exists. They will disappear from the blockchain if they were programmed to self destruct or performing that operation using delegatecall or callcode. Once self-destruct operation is performed, the remaining Ether stored at the contract address is sent to another address and the storage and code is removed from the state. Although this is the expected behavior, the pruning of self-

destructed contracts may or may not be implemented by Ethereum clients. SELFDESTRUCT was previously called SUICIDE, with EIP6, SUICIDE was renamed to SELFDESTRUCT.

Serenity

The fourth and final development stage of Ethereum. Serenity does not yet have a planned release date.

Serpent

A procedural (imperative) programming language with syntax similar to Python. Can also be used to write functional (declarative) code, though it is not entirely free of side effects. Used sparsely. First created by Vitalik Buterin.

Smart Contract

A program which executes on the Ethereum's computing infrastructure.

Solidity

A procedural (imperative) programming language with syntax that is similar to JavaScript, C++ or Java. The most popular and most frequently used language for Ethereum smart contracts. First created by Gavin Wood (co-author of this book).

Solidity inline assembly

Inline assembly is contained code within Solidity that use EVM Assembly, which can be seen as the human-readable form of EVM-code. Inline assembly tries to facilitate inherent difficulty and other issues arising when writing manual assembly.

Spurious Dragon

A hard fork at block #2,675,00 to address more denial of service attack vectors, and another state clearing. Also, a replay attack protection mechanism.

Swarm

A decentralised (P2P) storage network. It is used along with Web3 and Whisper to build DApps.

Tangerine Whistle

A hard fork at block #2,463,00 to change the gas calculation for certain IO-heavy operations and to clear the accumulated state from a denial of service attack, which exploited the low gas cost of those operations.

Testnet

A test network (testnet for short) is used to simulate the behavior of the main Ethereum network.

Transaction

Data committed to the Ethereum Blockchain signed by an originating account, targeting a specific address. The transaction contains metadata such as the gas limit for the transaction.

Truffle

One of the most commonly used Ethereum Development Frameworks. It is composed of several NodeJS packages and can be installed using Node Package Manager (NPM).

Turing Complete

In computability theory, a system of data-manipulation rules (such as a computer's instruction set, a programming language, or a cellular automaton) is said to be Turing complete or computationally universal if it can be used to simulate any Turing machine. The concept is named after English mathematician and computer scientist Alan Turing.

Vyper

A high-level programming language, similar to Serpent with Python-like syntax. Intended to get closer to a pure-functional language. First created by Vitalik Buterin.

Wallet

Software that holds all your secret keys. Used as the interface to access and control your Ethereum accounts and interact with Smart Contracts. Notice that keys need not be stored in your wallet and can be retrieved from an offline storage (e.g. USB flash drive or paper) for improved security. Despite the name, wallets never store the actual coins or tokens.

Web3

The third version of the web. First proposed by Gavin Wood, Web3 represents a new vision and focus for web applications: from centrally owned and managed applications, to applications built on decentralized protocols.

Wei

The smallest denomination of ether. 10^{18} wei = 1 ether.

Whisper

A decentralised (P2P) messaging service. It is used along with Web3 and Swarm to build DApps.

Zero address

A special Ethereum address, with all 20-bytes as zeros, that is specified as a destination address in the "contract creation transaction".

What is Ethereum?

Ethereum is "the World Computer". That's one of the more common descriptions of the Ethereum platform. But what does that mean? Let's try to start with a computer science focused description, and then try to decipher that with a more practical analysis of Ethereum's capabilities and characteristics, while comparing it to Bitcoin and other distributed ledger technologies (that - for simplicity's sake - we will often refer to as "blockchains").

From a computer science perspective, Ethereum is a deterministic but practically unbounded state-machine with two basic functions; the first being a globally accessible singleton state, and the second being a virtual machine that applies changes to that state.

From a more practical perspective, Ethereum is an open-source, globally decentralized computing infrastructure that executes programs called *smart contracts*. It uses a blockchain to synchronize and store the system *state* along with a cryptocurrency called *ether* to meter and constrain execution resource cost.

The Ethereum platform enables developers to build powerful decentralized applications with built-in economic functions. While providing continuous uptime, it also reduces or eliminates censorship, third party interface, and counterparty risk.

Compared to Bitcoin

Many people will come to Ethereum with some prior experience of cryptocurrencies, specifically Bitcoin. Ethereum shares many common elements with other open blockchains: a peer-to-peer network connecting participants, a consensus algorithm for synchronization of state (Proof-of-Work), a digital currency (ether) and a global ledger (the blockchain).

Components of a blockchain

The components of an open, public, blockchain are (usually):

- A peer-to-peer network connecting participants and propagating transactions and blocks of verified transactions, based on a standardized "gossip" protocol.
- A set of consensus rules, implemented in the state machine.
- Messages, in the form of transactions, representing state transitions.
- A state machine that processes transactions according to the consensus rules.
- A distributed database, the blockchain, that records a journal of all the state transitions.
- A consensus algorithm (e.g. Proof-of-Work) that decentralizes control over the blockchain, by forcing participants to compete and constraining them by the consensus rules.
- One or more open-source software implementations of the above.

All or most of these components are usually combined in a single software client. For example, in Bitcoin, the reference implementation is developed by the *Bitcoin Core* open source project, and implemented as the bitcoind client. In Ethereum, rather than a reference implementation, there is a *reference specification*, a mathematical description of the system in the [yellowpaper]. There are a number of clients, which are built according to the reference specification.

In the past, we used the term "blockchain" to represent all of the components above, as a short-hand reference to the combination of technologies that encompass all of the characteristics above. Today, however, the term blockchain has become diluted by marketers and profiteers, looking to hype their projects and attain unrealistic valuations for their startups. It is effectively meaningless on its own. We need qualifiers to help us understand the characteristics of the blockchain in question, such as *open, public, global, decentralized, neutral, and censorship-resistant*, to identify the important emergent characteristics of a "blockchain" system that these components allow.

Not all blockchains are created equal. When you are told that something is a blockchain, you have not received an answer, rather you need to start asking a lot of questions to clarify what "blockchain" means. Start by asking for a description of the components above, then ask about whether this "blockchain" exhibits the *open, public, etc.* characteristics.

Development of Ethereum

In many ways, both the purpose and construction of Ethereum are strikingly different from the open blockchains that preceded it, including Bitcoin.

Ethereum's purpose is not primarily a digital currency payment network. While the digital currency *ether* is both integral and necessary for the operation of Ethereum, ether is intended as a *utility currency* to pay for use of the Ethereum platform.

Unlike Bitcoin, which has a very limited scripting language, Ethereum is designed to be a general purpose programmable blockchain that runs a *virtual machine* capable of executing code of arbitrary and unbounded complexity. Where Bitcoin's Script language is, intentionally, constrained to simple true/false evaluation of spending conditions, Ethereum's language is *Turing Complete*, meaning that it is equivalent to a general purpose computer that can run any computation that a theoretical Turing Machine can run.

The birth of Ethereum

All great innovations solve real problems, Ethereum is no exception. Ethereum was conceived at a time when people recognized the power of the Bitcoin model, and were trying to move beyond application of cryptocurrency applications, into other projects. But developers faced a conundrum: they either needed to build on top of Bitcoin or start a new blockchain. Building upon Bitcoin meant living within the intentional constraints of the network and trying to find workarounds. The limited types and sizes of data storage seemed to limit the types of applications that could run on top, as second layer solutions. Programmers needed to build systems that utilized only the limited set of variables, transaction types, and data. For projects that needed more freedom, more flexibility, starting a new blockchain was the only option. But starting a new blockchain meant bootstrapping all the infrastructure elements, testing, etc.

Towards the end of 2013, Vitalik Buterin, a young programmer and Bitcoin enthusiast, started thinking about further extending the capabilities of Bitcoin and Mastercoin (an overlay protocol that extended Bitcoin to offer rudimentary smart contracts). In October of 2013, Vitalik proposed a more generalized approach to the Mastercoin team, one that allowed flexible and scriptable (but not Turing complete) contracts to replace the specialized contract language of Mastercoin. While the Mastercoin team was impressed, this proposal was too radical a change to fit into their development roadmap.

In December 2013, Vitalik started sharing a white paper which outlined the idea behind Ethereum: a Turing complete programmable and general purpose blockchain. A few dozen people saw this early draft and offered feedback to Vitalik, helping him gradually evolve the proposal.

Both of the authors of this book received an early draft copy of the white paper and commented on it. Andreas M. Antonopoulos was intrigued by the idea and asked Vitalik many questions about the use of a separate blockchain to enforce consensus rules on smart contract execution and the implications of a Turing complete language. Andreas continued to follow Ethereum's progress with great interest but was in the early stages of writing his book "Mastering Bitcoin" and did not participate directly in Ethereum until much later. Dr. Gavin Wood, however, was one of the first people to reach out to Vitalik and offer to help with his C++ programming skills. Gavin became

Ethereum's co-founder, co-designer and CTO.

As Vitalik recounts in his "Ethereum Prehistory" post:

> This was the time when the Ethereum protocol was entirely my own creation. From here on, however, new participants started to join the fold. By far the most prominent on the protocol side was Gavin Wood.

...

> Gavin can also be largely credited for the subtle change in vision from viewing Ethereum as a platform for building programmable money, with blockchain-based contracts that can hold digital assets and transfer them according to pre-set rules, to a general-purpose computing platform. This started with subtle changes in emphasis and terminology, and later this influence became stronger with the increasing emphasis on the "Web 3" ensemble, which saw Ethereum as being one piece of a suite of decentralized technologies, the other two being Whisper and Swarm.

Starting in December 2013, Vitalik and Gavin refined and evolved the idea, together building the protocol layer that became Ethereum.

Ethereum's founders were thinking about a blockchain that didn't aim for a specific purpose, but instead could support a broad variety of applications by being *programmed*. The idea was that by using a general purpose blockchain like Ethereum, a developer could program their particular application without having to bootstrap the underlying mechanisms of peer-to-peer networks, blockchains, consensus algorithms, etc. The Ethereum platform was designed to abstract these details and provide a deterministic and secure programming environment for decentralized blockchain applications.

Much like Satoshi, Vitalik and Gavin didn't just invent a new technology, they combined new inventions with existing technologies in a novel way and delivered the prototype code to prove their ideas to the world.

The founders worked for years, building and refining the vision. And on July 30th 2015, the first Ethereum block was mined. The world's computer started serving the world...

> Vitalik Buterin's article "A Prehistory of Ethereum" was published in September 2017 and provides a fascinating first-person view of Ethereum's earliest moments.
>
> You can read it at https://vitalik.ca/general/2017/09/14/prehistory.html

Ethereum's four stages of development

The birth of Ethereum was the launch of the first stage, named "Frontier". Ethereum's development is planned over four distinct stages, with major changes occurring in each new stage. Each stage may include sub-releases, known as "hard forks" that change functionality in a way that is not

backwards compatible.

The four main development stages are codenamed Frontier, Homestead, Metropolis and Serenity. The intermediate hard forks are codenamed "Ice Age", "DAO", "Tangerine Whistle", "Spurious Dragon", "Byzantium", and "Constantinople". They are listed below, by the block number in which the hard fork occurred:

Past transitions

Block #0

"Frontier" - The initial stage of Ethereum, lasted from July 30th 2015 to March 2016.

Block #200,000

"Ice Age" - A hard fork to introduce an exponential difficulty increase, motivating a transition to Proof-of-Stake.

Block #1,150,000

"Homestead" - The second stage of Ethereum, launched in March 2016.

Block #1,192,000

"DAO" - The hard fork that reversed the hacked DAO contract and caused Ethereum and Ethereum Classic to split into two competing systems.

Block #2,463,000

"Tangerine Whistle" - A hard fork to change the gas calculation for certain IO-heavy operations and to clear the accumulated state from a denial of service attack, which exploited the low gas cost of those operations.

Block #2,675,000

"Spurious Dragon" - A hard fork to address more denial of service attack vectors, and another state clearing. Also, a replay attack protection mechanism.

Current state

We are currently in the *Metropolis* stage, which was planned as two sub-release hard forks (see [hard_fork]) codenamed *Byzantium* and *Constantinople*. Byzantium went into effect in October 2017 and Constantinople is anticipated by mid-2018.

Block #4,370,000

"Metropolis Byzantium" - Metropolis is the third stage of Ethereum, current at the time of writing this book, launched in October 2017. Byzantium is the first of two hard forks for Metropolis.

Future plans

After Metropolis' Byzantium hard fork, there is one more hard fork planned for Metropolis. Metropolis is followed by the final stage of Ethereum's deployment, codenamed Serenity.

Constantinople

- The second part of the Metropolis stage, planned for mid-2018. Expected to include a switch to hybrid Proof-of-Work/Proof-of-Stake consensus algorithm, among other changes.

Serenity

The fourth and final stage of Ethereum. Serenity does not yet have a planned release date.

Ethereum: A general purpose blockchain

The original blockchain (Bitcoin's blockchain), tracks the state of units of bitcoin and their ownership. You can think of bitcoin as a distributed consensus *state machine*, where transactions cause a global *state transition*, altering the ownership of coins. The state transitions are constrained by the rules of consensus, allowing all participants to (eventually) converge on a common (consensus) state of the system, after several blocks are mined.

Ethereum is also a distributed state machine. But instead of tracking only the state of currency ownership, Ethereum tracks the state transitions of a general-purpose data store. By general purpose we mean any data that can be expressed as a *key-value tuple*. A key-value data store simply stores any arbitrary value, referenced by some key. For example, storing the value "Mastering Ethereum", referenced by the key "Book Title". In some ways, this serves the same purpose as the data storage model of *Random Access Memory (RAM)* used by a general purpose computer. Ethereum has *memory* that stores both code and data and it uses the Ethereum blockchain to track how this memory changes over time. Like a general-purpose stored-program computer, Ethereum can load code into its state machine and *run* that code, storing the resulting state changes in its blockchain. Two of the critical differences from a general purpose computer are that Ethereum state changes are governed by the rules of consensus and the state is distributed globally on a shared ledger. Ethereum answers the question: "What if we could track any arbitrary state and program the state machine, to create a world-wide computer operating under consensus?".

Ethereum's components

In Ethereum, the components of a blockchain system described in Components of a blockchain are, more specifically:

P2P Network

Ethereum runs on the *Ethereum Main Network*, which is addressable on TCP port 30303, and runs a protocol called *ĐΞVp2p*.

Consensus rules

Ethereum's consensus rules, are defined in the reference specification, the [yellowpaper].

Transactions

Ethereum transactions (see [transactions]) are network messages, that include (among other things) a sender, recipient, value and data payload.

State Machine

Ethereum state transitions are processed by the *Ethereum Virtual Machine (EVM)*, a stack-based

virtual machine that executes *bytecode* (machine-language instructions). EVM programs called "smart contracts" are written in high-level languages (e.g. Solidity) and compiled to bytecode for execution on the EVM.

Blockchain

Ethereum's blockchain is stored locally on each node as a *database* (usually Google's LevelDB), which contains the transactions and system state in a serialized hashed data structure called a *Merkle Patricia Tree*.

Consensus Algorithm

Ethereum currently uses a Proof-of-Work algorithm called *Ethash*, but there are plans to transition to a Proof-of-Stake system codenamed *Casper* in the near future.

Clients

Ethereum has several interoperable implementations of the client software, the most prominent of which are *Go-Ethereum (Geth)* and *Parity*.

Further references

The Ethereum Yellow Paper: https://ethereum.github.io/yellowpaper/paper.pdf

The "Beige Paper": a rewrite of the "Yellow Paper" for a broader audience in less formal language: https://github.com/chronaeon/beigepaper

ÐΞVp2p network protocol: https://github.com/ethereum/wiki/wiki/%C3%90%CE%9EVp2p-Wire-Protocol

Ethereum Virtual Machine - a list of "Awesome" resources: https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-(EVM)-Awesome-List

LevelDB Database (used most often to store the local copy of the blockchain): http://leveldb.org

Merkle Patricia Trees: https://github.com/ethereum/wiki/wiki/Patricia-Tree

Ethash Proof-of-Work Consensus Algorithm: https://github.com/ethereum/wiki/wiki/Ethash

Casper Proof-of-Stake v1 Implementation Guide: https://github.com/ethereum/research/wiki/Casper-Version-1-Implementation-Guide

Go-Ethereum (Geth) Client: https://geth.ethereum.org/

Parity Ethereum Client: https://parity.io/

Ethereum and Turing Completeness

As soon as you start reading about Ethereum, you will immediately hear the term "Turing Complete". Ethereum, they say, unlike Bitcoin, is "Turing Complete". What exactly does that mean?

The term "Turing Complete" is named after English mathematician Alan Turing who is considered the father of computer science. In 1936 he created a mathematical model of a computer consisting

of a state machine that manipulates symbols, by reading and writing them on sequential memory (resembling an infinite-length magnetic tape). With this construct, Alan Turing went on to provide a mathematical foundation to answer (in the negative) questions about *universal computability*, meaning whether all problems are solvable. He proved that there are classes of problems that are uncomputable. Specifically, he proved that the *Halting Problem* (trying to evaluate whether a program will eventually stop running) is not solvable.

Alan Turing further defined a system to be *Turing Complete*, if it can be used to simulate any Turing Machine. Such a system is called a *Universal Turing Machine (UTM)*.

Ethereum's ability to execute a stored program, in a state machine called the Ethereum Virtual Machine, while reading and writing data to memory makes it a Turing Complete system and therefore a Universal Turing Machine. Ethereum can compute any algorithm that can be computed by any Turing Machine, given the limitations of finite memory.

Ethereum's groundbreaking innovation is to combine the general-purpose computing architecture of a stored-program computer with a decentralized blockchain, thereby creating a distributed single-state (singleton) world computer. Ethereum programs run "everywhere", yet produce a common (consensus) state that is secured by the rules of consensus.

Turing Completeness as a "feature"

Hearing that Ethereum is Turing Complete, you might arrive at the conclusion that this is a *feature* that is somehow lacking in a system that is Turing Incomplete. Rather, it is the opposite. It takes a very focused effort to constrain a system so that it is **not** Turing Complete. Turing completeness arises in even the simplest state machines. In fact the simplest Turing Complete state machine known (Rogozhin, 1996) has 4 states and uses 6 symbols, with a state definition that is only 22 instructions long.

Not only is Turing completeness achievable in the simplest of systems, but systems that are designed to be constrained so that they are Turing Incomplete, are often found to be "Accidentally Turing Complete". A constrained system that is Turing Incomplete is harder to design and must be carefully maintained so that it remains Turing Incomplete.

A fun reference of systems that are "Accidentally Turing Complete" can be found here: http://beza1e1.tuxen.de/articles/accidentally_turing_complete.html

The fact that Ethereum is Turing Complete means that any program of any complexity can be computed in Ethereum. But that flexibility brings some thorny security and resource management problems.

Implications of Turing Completeness

Turing proved that you cannot predict whether a program will terminate, by simulating it on a computer. In simple terms, we cannot predict the path of a program without running it. Turing Complete systems can run in "infinite loops", a term used (in oversimplification) to describe a program that does not terminate. It is trivial to create a program that runs a loop that never ends. But unintended never-ending loops can arise without warning, due to complex interactions between the starting conditions and the code. In Ethereum, this poses a challenge: every

participating node (client), must validate every transaction, running any smart contracts it calls. But as Turing proved, Ethereum can't predict if a smart contract will terminate, or how long it will run, without actually running it (possibly running forever). Whether by accident, or on purpose, a smart contract can be created such that it runs forever when a node attempts to validate it, effectively a denial of service attack. Of course, between a program that takes a millisecond to validate and one that runs forever there is an infinite range of nasty, resource hogging, memory-bloating, CPU overheating programs that simply waste resources. In a world computer, a program that abuses resources gets to abuse the world's resources. How does Ethereum constrain the resources used by a smart contract if it cannot predict resource use in advance?

To answer this challenge, Ethereum introduces a metering mechanism called *gas*. As the EVM executes a smart contract, it carefully accounts for every instruction (computation, data access, etc.). Each instruction has a pre-determined cost in units of gas. When a transaction triggers the execution of a smart contract, it must include an amount of gas that sets the upper limit of computation that can be consumed running the smart contract. The EVM will terminate execution if the amount of gas consumed by computation exceeds the gas available in the transaction. Gas is the mechanism Ethereum uses to allow Turing Complete computation while limiting the resources that any program can consume.

In 2015 an attacker exploited an EVM instruction that cost far less gas than it should have. this allowed the attacker to create transactions that use a lot of memory and take several minutes to validate. To fix this attack, Ethereum had to change its gas accounting formula for certain instructions in a backwards incompatible (hard fork) change. Even with this change, however, Ethereum clients have to skip validating these transactions or waste weeks trying to validate them.

From general purpose blockchains to Decentralized Applications (DApps)

Ethereum started as a way to make a general purpose blockchain that could be programmed for a variety of uses. But very quickly, Ethereum's vision expanded to become a platform for programming *Decentralized Applications (DApps)*. DApps represent a broader perspective than "smart contracts". A DApp is, at the very least, a smart contract and a web user-interface. More broadly, a DApp is a web application that is built on top of open, decentralized, peer-to-peer infrastructure services.

A DApp is composed of at least:

- Smart contracts on a blockchain.
- A web front-end user-interface.

In addition, many DApps include other decentralized components, such as:

- A decentralized (P2P) storage protocol and platform.
- A decentralized (P2P) messaging protocol and platform.

TIP You may see DApps spelled as ĐApps. The Đ character is the Latin character called "ETH", alluding to Ethereum. To display this character, use decimal entity #208 in HTML, and Unicode characters 0xCE (UTF-8), or 0x00D0 (UTF-16).

Evolving the World Wide Web

In 2004, the term "Web 2.0" came to prominence, describing an evolution of the web towards user-generated content, responsive interfaces and interactivity. Web 2.0 is not a technical specification, but rather a term describing the new focus of web applications.

The concept of DApps is meant to take the World Wide Web to its next natural evolution, introducing decentralization with peer-to-peer protocols into every aspect of a web application. The term used to describe this evolution is *Web3*, meaning the third "version" of the web. First proposed by Gavin Wood, *web3* represents a new vision and focus for web applications: from centrally owned and managed applications, to applications built on decentralized protocols.

In later chapters we'll explore the Ethereum web3js JavaScript library which bridges JavaScript applications that run in your browser with the Ethereum blockchain. The web3.js library also includes an interface to a P2P storage network called *Swarm* and a P2P messaging service called *Whisper*. With these three components included in a JavaScript library running in your web browser, developers have a full application development suite that allows them to build web3 DApps:

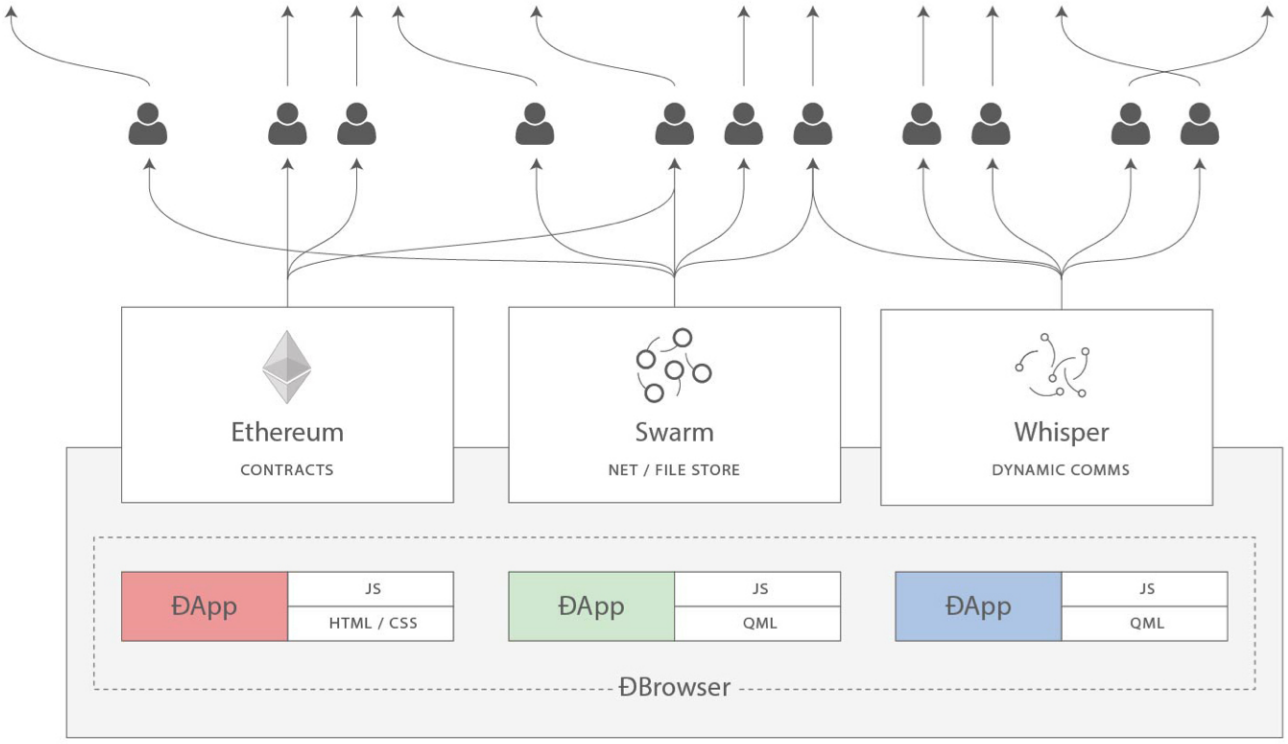

Figure 1. Web3: A suite of decentralized application components for the next evolution of the web

Ethereum's development culture

So far we've talked about how Ethereum's goals and technology differ from other blockchains that

preceded it, like Bitcoin. Ethereum also has a very different development culture.

In Bitcoin, development is guided by conservative principles: all changes are carefully studied to ensure that none of the existing systems are disrupted. For the most part, changes are only implemented if they are backwards compatible. Existing clients are allowed to "opt-in", but will continue to operate if they decide not to upgrade.

In Ethereum, by comparison, the development culture is focused on speed and innovation. The mantra is "move fast and break things". If a change is needed, it is implemented, even if that means invalidating prior assumptions, breaking compatibility, or forcing clients to update. Ethereum's development culture is characterized by rapid innovation, rapid evolution and a willingness to engage in experimentation.

What this means to you as a developer, is that you must remain flexible and be prepared to rebuild your infrastructure as some of the underlying assumptions change. Do not assume anything will be static or permanent. One of the big challenges facing developers in Ethereum is the inherent contradiction between deploying code to an immutable ledger and a development platform that is still rapidly evolving. You can't simply "upgrade" your smart contracts. You must be prepared to deploy new ones, migrate users, apps and funds, and start over.

Ironically, this also means that the goal of building systems with more autonomy and less centralized control cannot be realized. Autonomy and decentralization requires a bit more stability in the platform than you're likely to get in Ethereum, in the next few years. In order to "evolve" the platform, you have to be ready to scrap and restart your smart contracts, which means you have to retain a certain degree of control over them.

But, on the positive side, Ethereum is moving forward very fast. There's very little opportunity for "bike-shedding" - an expression that means arguing over minor details such as how to build the bicycle shed in the back of the building. If you start bike-shedding, you might suddenly discover the rest of the development team changed the plan, and ditched bicycles in favor of autonomous hovercrafts. There are very few sacred principles, final standards, or fixed interfaces in Ethereum.

Eventually, Ethereum core protocol development will slow down and its interfaces will become fixed. But in the meantime, innovation is the driving principle. You'd better keep up, because no one will slow down for you.

Why learn Ethereum?

Blockchains have a very steep learning curve, as they combine multiple disciplines into one domain: programming, information security, cryptography, economics, distributed systems, peer-to-peer networks etc. Ethereum makes this learning curve a lot less steep, so you can get started very quickly. But just below the surface of a deceptively simple environment, lies a lot more. As you learn and start looking deeper, there's always another layer of complexity and wonder.

Ethereum is a great platform for learning about blockchains and it's building a massive community of developers, faster than any other blockchain platform. More than any other blockchain, Ethereum is a *developer's blockchain*, built by developers, for developers. A developer familiar with JavaScript applications can drop into Ethereum and start producing working code very quickly. For the first years of Ethereum, it was common to see t-shirts announcing that you can create a token in

just five lines of code. Of course, this is a double-edged sword. It's easy to write code, but it's very hard to write *good* code and *secure* code.

What this book will teach you?

This book dives into Ethereum and examines every component. You will start with a simple transaction, dissect how it works, build a simple contract, make it better and follow its journey through the Ethereum system.

You will learn how Ethereum works, but also why it is designed the way it is. You will be able to understand how each of the pieces work, but also how they fit together and why.

Ethereum Basics

Control and responsibility

Open blockchains like Ethereum are secure because they are *decentralized*. That means that each user of Ethereum should control their own keys, which control access to their funds and contracts. Some users choose to give up control over their keys by using a third party custodian, such as an exchange wallet. In this book, we will teach you how to take control and manage your own keys.

With that control comes a big responsibility. If you lose your keys, you lose access to funds and contracts. No one can help you regain access - your funds will be locked forever. Here are a few tips to help you manage this responsibility:

- When you are prompted to choose a password: make it strong, back it up and don't share it. If you don't have a password manager, write it down and store it in a locked drawer or safe. This password can be used as long as you have the "keystore" file of your account.
- When you are prompted to back up a key or mnemonic words, use pen and paper to make a physical backup. Do not leave that task for "later", you will forget. These can be used in case you lose all data saved on your system.
- Do not store key material (encrypted or not) in digital documents, digital photos, screenshots, online drives, encrypted PDFs, etc. Don't improvise security. Use a password manager or pen and paper.
- Before transferring any large amounts, first do a small test transaction (e.g., $1 value). Once you receive the test transaction, try sending it from that wallet. If you forgot your password or can't send the funds for any other reason, it is better to find out with a small loss.
- Do not send money to any of the addresses shown in this book. The private keys are listed in the book and someone will immediately take that money.

Ether currency units

Ethereum's currency unit is called *ether*, identified also as ETH or with the symbols Ξ (from the Greek letter "Xi" that looks like a stylized capital E) or (less often) , for example, 1 ether, or 1 ETH, or Ξ1, or 1

TIP Use Unicode character 926 for Ξ and 9830 for ♦.

Ether is subdivided into smaller units, down to the smallest unit possible, which is named *wei*. One *ether* is 1 quintillion *wei* (1 x 10^{18} or 1,000,000,000,000,000,000). You may hear people refer to the currency "Ethereum" too, but this is a common beginner's mistake. Ethereum is the system, ether is the currency.

The value of ether is always represented internally in Ethereum as an unsigned integer value denominated in *wei*. When you transact 1 ether, the transaction encodes 1000000000000000000 wei as the value.

Ether's various denominations have both a *scientific name* using the International System of units (*SI*), and a colloquial name that pays homage to many of the great minds of computing and cryptography.

Table Ether Denominations and Unit Names shows the various units, their colloquial (common) name, and their SI name. In keeping with the internal representation of value, the table shows all denominations in wei (first row), with ether shown as 10^{18} wei in the 7th row:

Table 1. Ether Denominations and Unit Names

Value (in wei)	Exponent	Common Name	SI Name
1	1	wei	wei
1,000	10^3	babbage	kilowei or femtoether
1,000,000	10^6	lovelace	megawei or picoether
1,000,000,000	10^9	shannon	gigawei or nanoether
1,000,000,000,000	10^{12}	szabo	microether or micro
1,000,000,000,000,000	10^{15}	finney	milliether or milli
1,000,000,000,000,000,000	*10^{18}*	*ether*	*ether*
1,000,000,000,000,000,000,000	10^{21}	grand	kiloether
1,000,000,000,000,000,000,000,000	10^{24}		megaether

Choosing an Ethereum wallet

An Ethereum wallet is your gateway to the Ethereum system. It holds your keys and can create and broadcast transactions on your behalf. Choosing an Ethereum wallet can be difficult because there are many different options with different features and designs. Some are more suitable for beginners and some are more suitable for experts. Even if you choose one that you like now, you might decide to switch to a different wallet later on. Ethereum itself is constantly changing and the "best" wallets are often the ones that adapt to them.

But don't worry! If you choose a wallet and don't like how it works, you can change wallets quite easily. All you have to do is make a transaction that sends your funds from the old wallet to the new

wallet, or move the keys by exporting and importing your private keys.

To get started, we will choose three different types of wallets to use as examples throughout the book: a mobile wallet, a desktop wallet, and a web-based wallet. We've chosen these three wallets because they represent a broad range of complexity and features. However, the selection of these wallets is not an endorsement of their quality or security. They are simply a good starting place for demonstrations and testing.

Starter wallets:

MetaMask

MetaMask is a browser extension wallet that runs in your browser (Chrome, Firefox, Opera or Brave Browser). It is easy to use and convenient for testing, as it is able to connect to a variety of Ethereum nodes and test blockchains (see [testnets]).

Jaxx

Jaxx is a multi-platform and multi-currency wallet that runs on a variety of operating systems including Android, iOS, Windows, Mac, and Linux. It is often a good choice for new users as it is designed for simplicity and ease of use.

MyEtherWallet (MEW)

MyEtherWallet is a web-based wallet that runs in any browser. It has multiple sophisticated features we will explore in many of our examples.

Emerald Wallet

Emerald Wallet is designed to work with Ethereum Classic blockchain, but compatible with other Ethereum based blockchains. It's an open source desktop application, works under Windows, Mac and Linux. Emerald wallet can run a full node or connect to a public remote node, working in a "light" mode. It also has a companion tool to do all operations from command line.

We'll start by installing MetaMask on our desktop.

Installing MetaMask

Open the Google Chrome browser and navigate to:

https://chrome.google.com/webstore/category/extensions

Search for "MetaMask" and click on the logo of a fox. You should see the extension's detail page like this:

Figure 2. The detail page of the MetaMask Chrome Extension

It's important to verify that you are downloading the real MetaMask extension, as sometimes people are able to sneak malicious extensions past Google's filters. The real one:

- Shows the ID nkbihfbeogaeaoehlefnkodbefgpgknn in the address bar
- Is offered by https://metamask.io
- Has more than 800 reviews
- Has more than 1,000,000 users

Once you confirm you are looking at the correct extension, click "Add to Chrome" to install it.

Using MetaMask for the first time

Once MetaMask is installed you should see a new icon (head of a fox) in your browser's toolbar. Click on it to get started. You will be asked to accept the terms and conditions and then to create your new Ethereum wallet by entering a password:

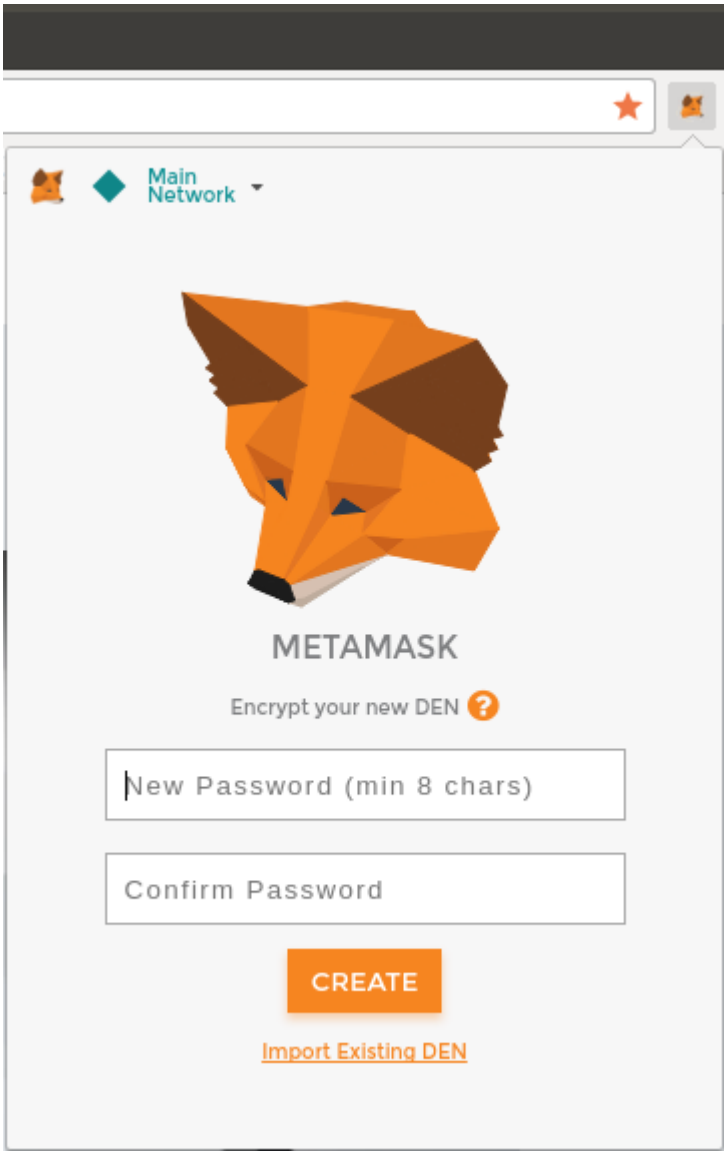

Figure 3. The password page of the MetaMask Chrome Extension

TIP The password controls access to MetaMask, so that it can't be used by anyone with access to your browser.

Once you've set a password, MetaMask will generate a wallet for you and show you a *mnemonic backup* consisting of 12 English words. These words can be used in any compatible wallet to recover access to your funds should something happen to MetaMask or your computer. You do not need the password for this recovery. The 12 words are sufficient.

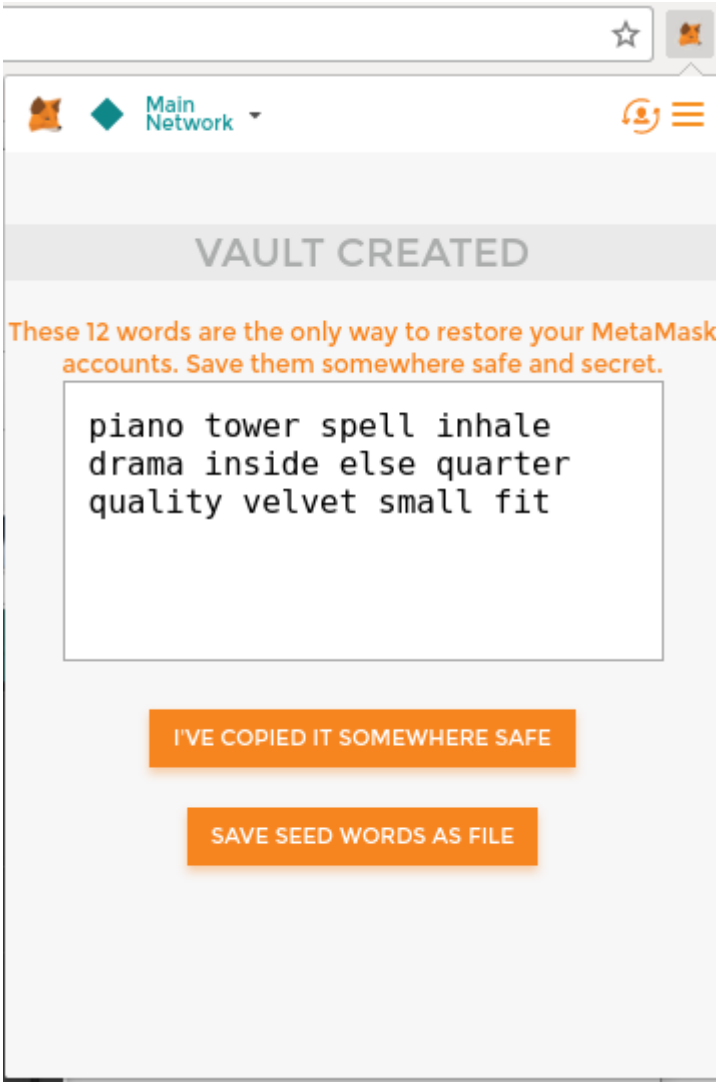

Figure 4. The mnemonic backup of your wallet, created by MetaMask

TIP Backup your mnemonic (12 words) on paper, twice. Store the two paper backups in two separate secure locations, such as a fire resistant safe, a locked drawer or a safe deposit box. Treat the paper backups like cash of equivalent value as what you store in your Ethereum wallet. Anyone with access to these words can gain access and steal your money.

Once you have confirmed that you have stored the mnemonic securely, MetaMask will display your Ethereum account details:

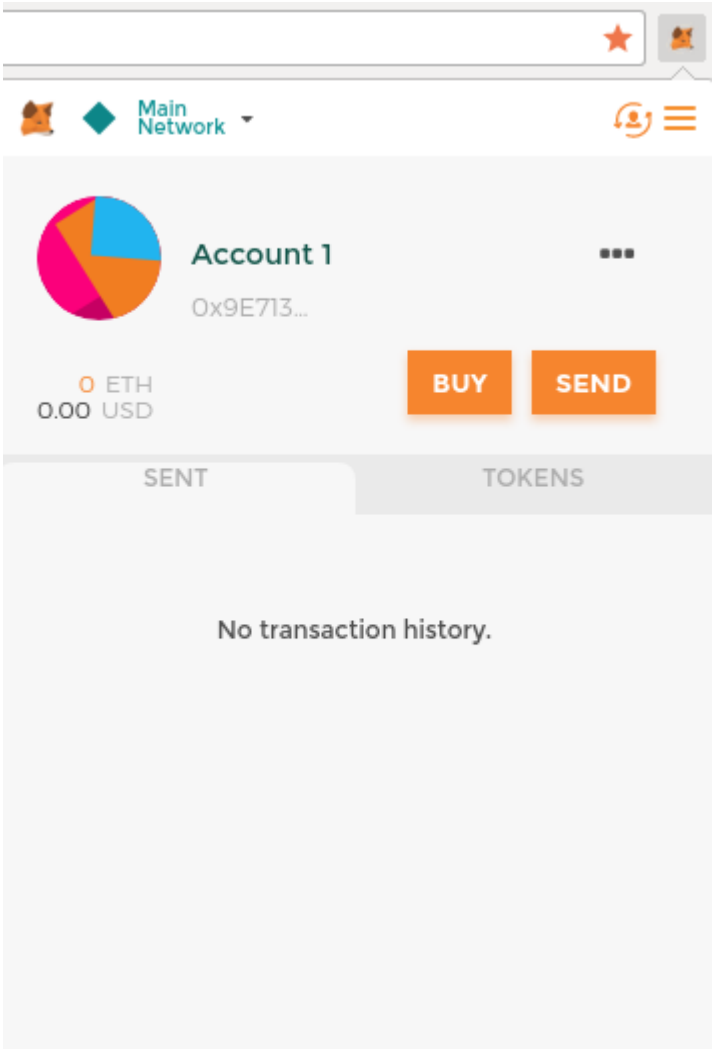

Figure 5. Your Ethereum account in MetaMask

Your account page shows the name of your account ("Account 1" by default), an Ethereum address (0x9E713... in the example) and a colorful icon to help you visually distinguish this account from other accounts. At the top of the account page, you can see which Ethereum network you are currently working on ("Main Network" in the example).

Congratulations! You have set up your first Ethereum wallet!

Switching networks

As you can see on the MetaMask account page, you can choose between multiple Ethereum networks. By default, MetaMask will try to connect to the "Main Network". The other choices are public testnets, any Ethereum node of your choice, or nodes running private blockchains on your own computer (localhost):

Main Ethereum Network

The main, public, Ethereum blockchain. Real ETH, real value, real consequences.

Ropsten Test Network

Ethereum public test blockchain and network, using Proof-of-Work consensus (mining). ETH on this network has no value. The issue with Ropsten was that the attacker minted tens of thousands of blocks, producing huge reorgs and pushing the gas limit up to 9B. A new public

testnet was required then, but later(on 25th March 2017) Ropsten was also revived!

Kovan Test Network

Ethereum public test blockchain and network, using "Aura" protocol for Proof-of-Authority consensus (federated signing). ETH on this network has no value. This test network is supported by "Parity" only. Other Ethereum clients use 'Clique' protocol, which was proposed later, for Proof-of-Authority.

Rinkeby Test Network

Ethereum public test blockchain and network, using "Clique" protocol for Proof-of-Authority consensus (federated signing). ETH on this network has no value.

Localhost 8545

Connect to a node running on the same computer as the browser. The node can be part of any public blockchain (main or testnet), or a private testnet (see [ganache]).

Custom RPC

Allows you to connect MetaMask to any node with a geth-compatible Remote Procedure Call (RPC) interface. The node can be part of any public or private blockchain.

For more information about the various Ethereum testnets and how to choose between them, see [testnets].

TIP	Your MetaMask wallet uses the same private key and Ethereum address on all the networks it connects to. However, your Ethereum address balance on each Ethereum network will be different. Your keys may control ether and contracts on Ropsten, for example, but not on the Main Network.

Getting some test ether

Our first task is to get our wallet funded. We won't be doing that on the Main Network because real ether costs money and handling it requires a bit more experience. For now, we will load our wallet with some testnet ether.

Switch MetaMask to the *Ropsten Test Network*. Then click "Buy", and click "Ropsten Test Faucet". MetaMask will open a new web page:

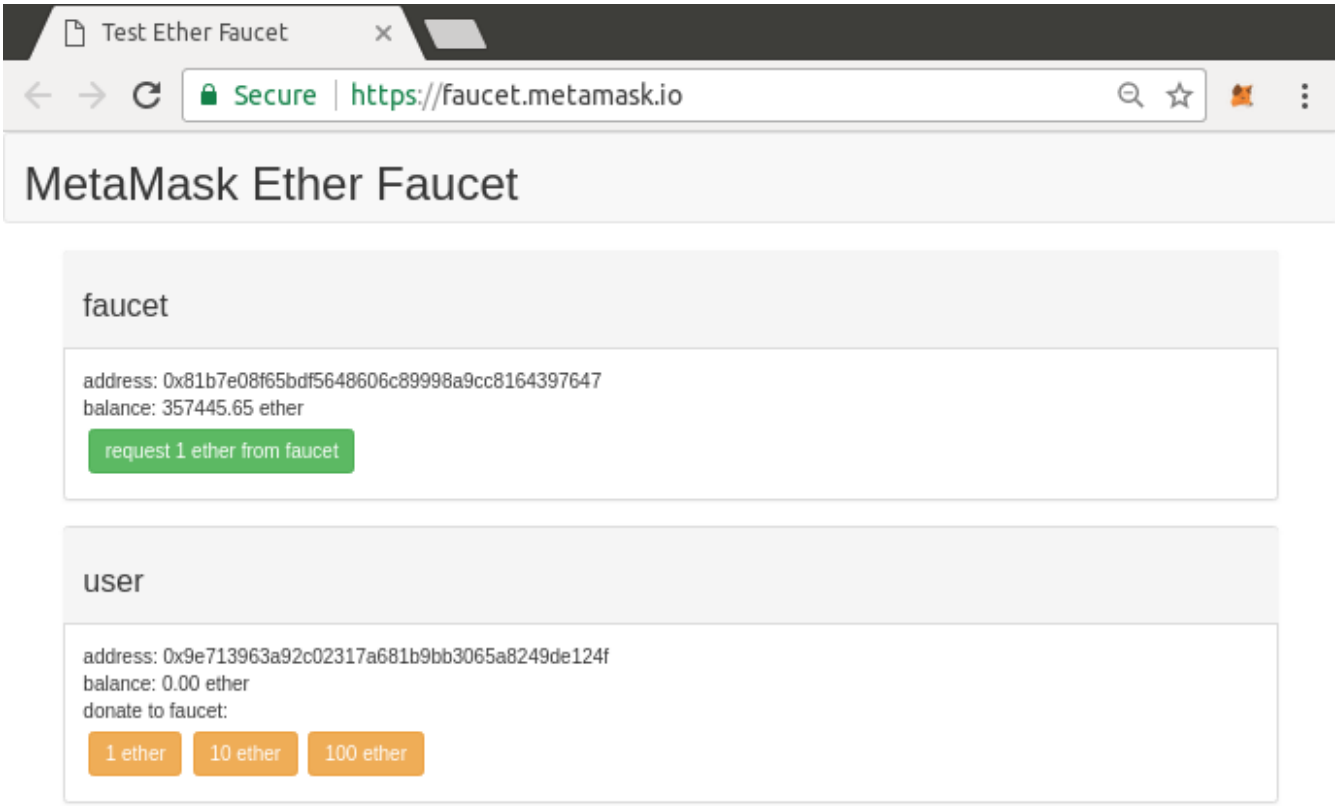

Figure 6. MetaMask Ropsten Test Faucet

You may notice that the web page already contains your MetaMask wallet's Ethereum address. MetaMask integrates Ethereum enabled web pages (see [dapps]) with your MetaMask wallet. MetaMask can "see" Ethereum addresses on the web page, allowing you, for example, to send a payment to an online shop displaying an Ethereum address. MetaMask can also populate the web page with your own wallet's address as a recipient address if the web page requests it. In this page, the faucet application is asking MetaMask for a wallet address to send test ether to.

Press the green "request 1 ether from faucet" button. You will see a transaction ID appear in the lower part of the page. The faucet app has created a transaction - a payment to you. The transaction ID looks like this:

```
0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
```

In a few seconds, the new transaction will be mined by the Ropsten miners and your MetaMask wallet will show a balance of 1 ETH. Click on the transaction ID and your browser will take you to a *block explorer*, which is a website that allows you to visualize and explore blocks, addresses, and transactions. MetaMask uses the etherscan.io block explorer, one of the more popular Ethereum block explorers. The transaction containing our payment from the Ropsten Test Faucet is shown in Etherscan Ropsten Block Explorer

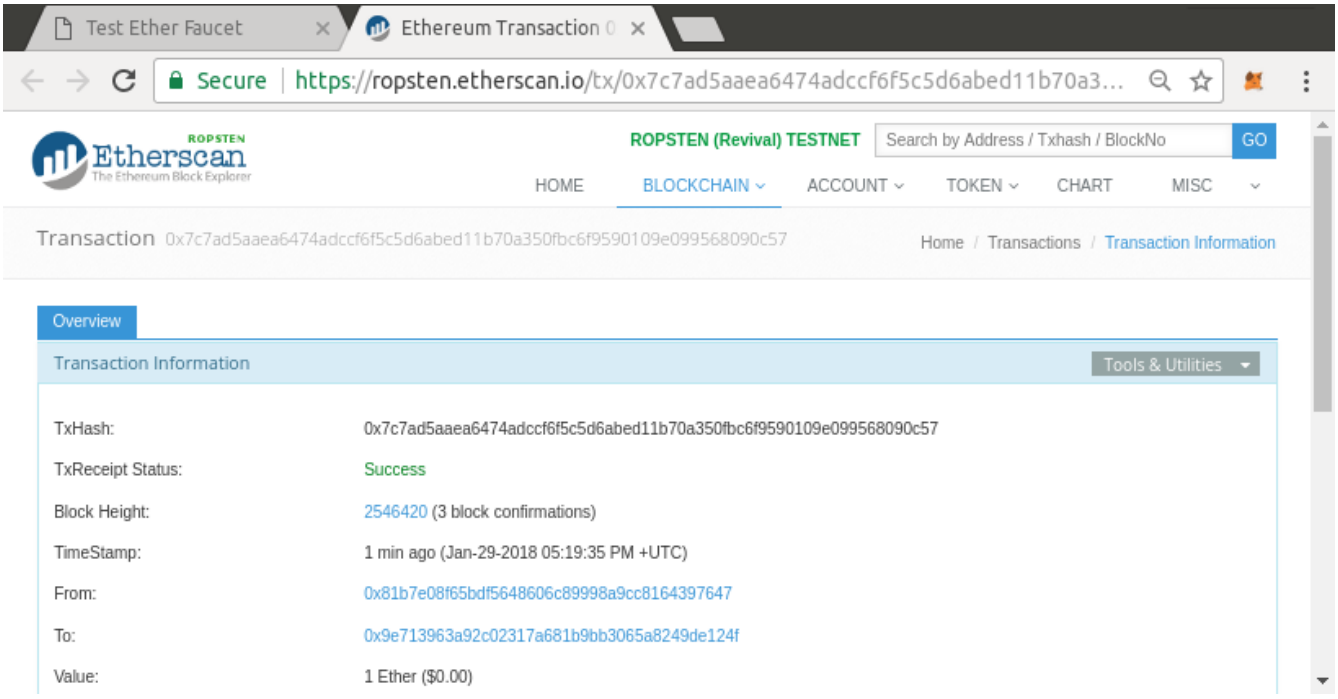

Figure 7. Etherscan Ropsten Block Explorer

The transaction has been recorded on the Ropsten blockchain and can be viewed at any time by anyone, simply by searching for the transaction ID, or visiting the link:

https://ropsten.etherscan.io/tx/0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57

Try visiting that link, or entering the transaction hash into the ropsten.etherscan.io website, to see it for yourself.

Sending ether from MetaMask

Once we've received our first test ether from the Ropsten Test Faucet, we will experiment with sending ether, by trying to send some back to the faucet. As you can see on the Ropsten Test Faucet page, there is an option to "donate" 1 ETH to the faucet. This option is available so that once you're done testing, you can return the remainder of your test ether, so that someone else can use it next. Even though test ether has no value, some people hoard it, making it difficult for everyone else to use the test networks. Hoarding test ether is frowned upon!

Fortunately, we are not test ether hoarders and we want practice sending ether anyway.

Click on the orange "1 ether" button to tell MetaMask to create a transaction paying the faucet 1 ether. MetaMask will prepare a transaction and pop-up a window with the confirmation:

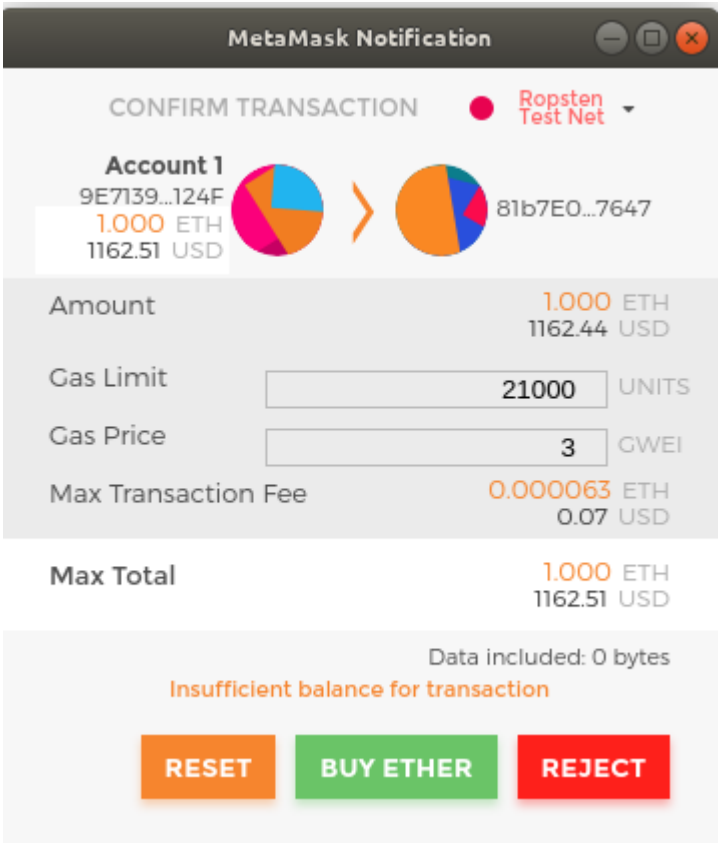

Figure 8. Sending 1 ether to the faucet

Oops! You probably noticed you can't complete the transaction. MetaMask says "Insufficient balance for transaction". At first glance this may seem confusing: we have 1 ETH, we want to send 1 ETH, why is MetaMask saying we have insufficient funds?

The answer is because of the cost of *gas*. Every Ethereum transaction requires payment of a fee, which is collected by the miners to validate the transaction. The fees in Ethereum are charged in a virtual currency called *gas*. You pay the gas with ether, as part of the transaction.

TIP Fees are required on the test networks too. Without fees, a test network would behave differently from the main network, making it an inadequate testing platform. Fees also protect the test networks from denial of service attacks and poorly constructed contracts (e.g. infinite loops), much like they protect the main network.

When you sent the transaction, Metamask calculated the average gas price of recent successful transactions at 3 GWEI, which stands for 3 gigawei. Wei is the smallest subdivision of the ether currency, as we discussed in Ether currency units. The gas cost of sending a basic transaction is 21000 gas units. Therefore, the maximum amount of ETH you spend is 3 * 21000 GWEI = 63000 GWEI = 0.000063 ETH. Be advised that average gas prices can fluctuate as they are predominantly determined by miners. We will see in a later chapter how you can increase/decrease your gas limit to ensure your transaction takes precedence if need be.

All this to say: to make a 1 ETH transaction costs 1.000063 ETH. MetaMask confusingly rounds that *down* to 1 ETH when showing the total, but the actual amount you need is 1.000063 ETH and you only have 1 ETH. Click "Reject" to cancel this transaction.

Let's get some more test ether! Click on the green "request 1 ether from the faucet" button again

and wait a few seconds. Don't worry, the faucet should have plenty of ether and will give you more if you ask.

Once you have a balance of 2 ETH, you can try again. This time, when you click on the orange "1 ether" donation button, you have sufficient balance to complete the transaction. Click "Submit" when MetaMask pops-up the payment window. After all of this, you should see a balance of 0.999937 ETH because you sent 1 ETH to the faucet with 0.000063 ETH in gas.

Exploring the transaction history of an address

By now you have become an expert in using MetaMask to send and receive test ether. Your wallet has received at least two payments and sent at least one. Let's see all these transactions, using the ropsten.etherscan.io block explorer. You can either copy your wallet address and paste it into the block explorer's search box, or you can have MetaMask open the page for you. Next to your account icon in MetaMask, you will see a button showing three dots. Click on it to show a menu of account-related options:

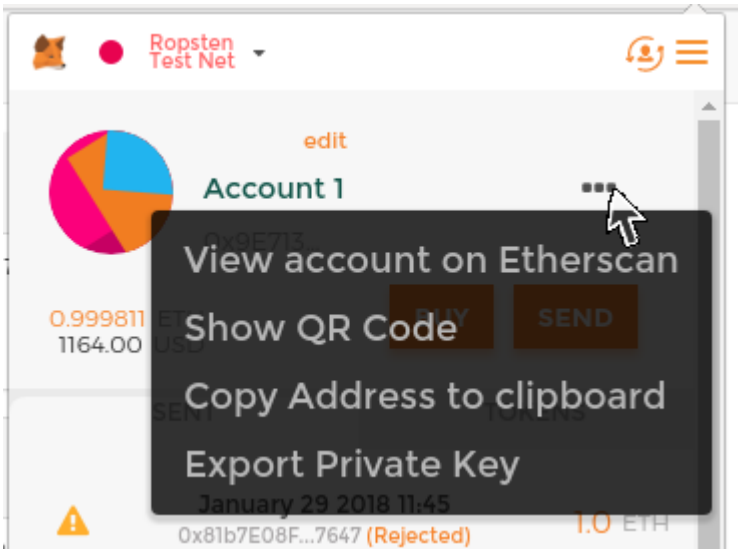

Figure 9. MetaMask Account Context Menu

Select "View Account on Etherscan", to open a web page in the block explorer, showing your account's transaction history:

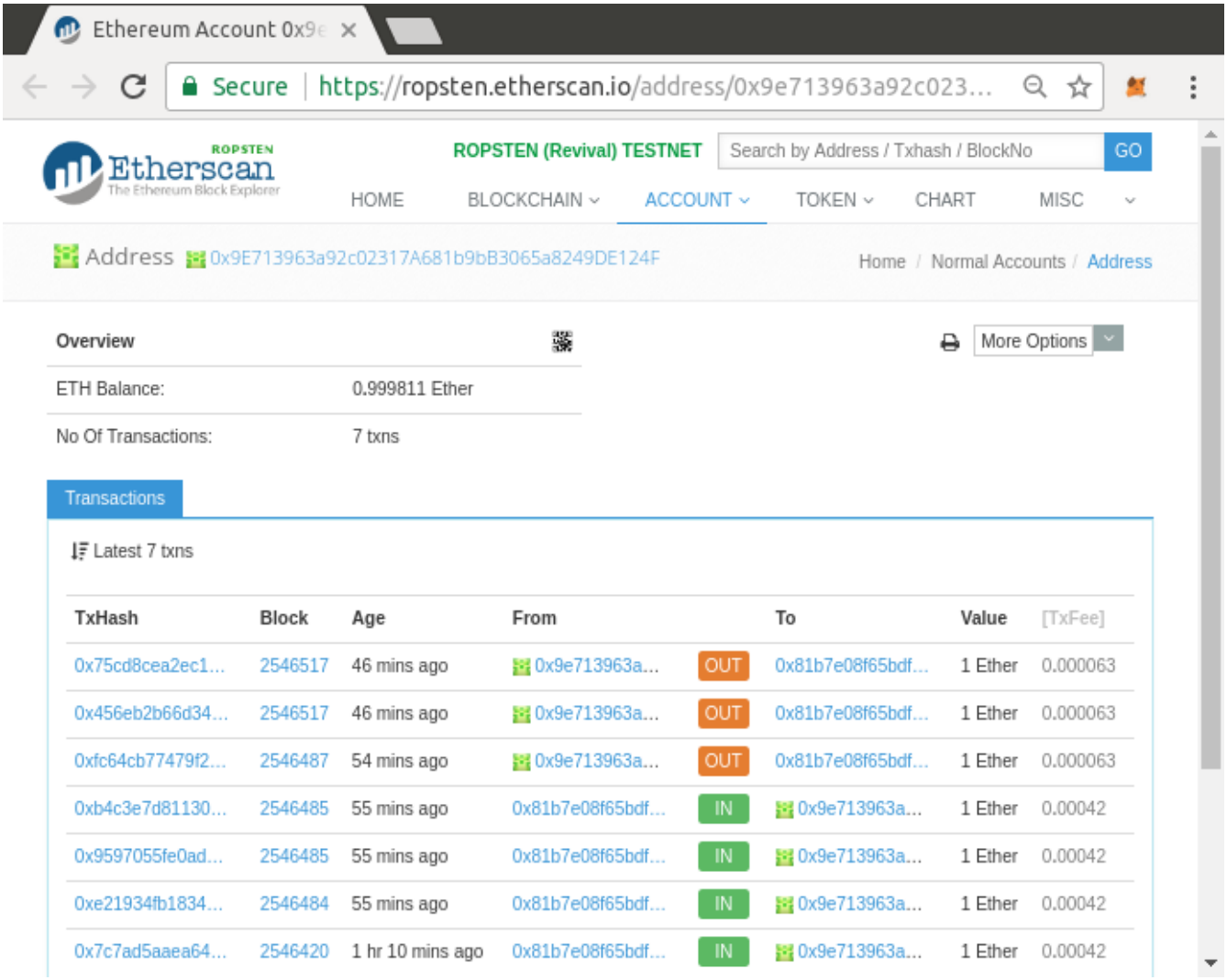

Figure 10. Address Transaction History on Etherscan

Here you can see the entire transaction history of your Ethereum address. It shows all the transactions recorded on the Ropsten blockchain, where your address is the sender or recipient of the transaction. Click on a few of these transactions to see more details.

You can explore the transaction history of any address. See if you can explore the transaction history of the Ropsten Test Faucet address (Hint: it is the "sender" address listed in the oldest payment to your address). You can see all the test ether sent from the faucet to you and to other addresses. Every transaction you see can lead you to more addresses and more transactions. Before long you will be lost in the maze of interconnected data. Public blockchains contain an enormous wealth of information, all of which can be explored programmatically, as we will see in the future examples.

Introducing the world computer

We've created a wallet and we've sent and received ether. So far, we've treated Ethereum as a cryptocurrency. But Ethereum is much, much more. In fact, the cryptocurrency function is subservient to Ethereum's function as a world computer; a decentralized smart contract platform. Ether is meant to be used to pay for running *smart contracts*, which are computer programs that run on an emulated computer called the *Ethereum Virtual Machine (EVM)*.

The EVM is a global singleton, meaning that it operates as if it was a global, single-instance

computer, running everywhere. Each node on the Ethereum network runs a local copy of the EVM to validate contract execution, while the Ethereum blockchain records the changing *state* of this world computer as it processes transactions and smart contracts.

Externally Owned Accounts (EOAs) and contracts

The type of account we created in the MetaMask wallet is called an *Externally Owned Account (EOA)*. Externally owned accounts are those that have a private key, which controls access to funds or contracts. Now, you're probably guessing there is another type of account. The other type of account is a *contract* account. A contract account is owned (and controlled) by the logic of a software program recorded on the Ethereum blockchain and executed by the EVM.

In the future, all Ethereum wallets might be running as Ethereum contracts, blurring the distinction between Externally Owned Accounts and contracts. But the important distinction that will always remain is this: Humans make decisions through EOA's while software make decisions through contracts.

Contracts have an address, just like EOAs (wallets). Contracts can send and receive ether, just like wallets. When a transaction destination is a contract address, it causes that contract to *run* in the EVM, using the transaction as its input.

In addition to ether, transactions can contain *data* indicating which specific function in the contract to run and what parameters to pass to that function. In this way, transactions *call* functions within contracts. Finally, contracts can generate transactions calling other contracts, building complex execution paths. One typical use of this is Contract A calling Contract B in order to maintain a shared state across users of Contract A.

In the next few sections, we will write our first contract. We will then create, fund, and use that contract with our MetaMask wallet and test ether on the Ropsten test network.

A simple contract: a test ether faucet

Ethereum has many different high-level languages, all of which can be used to write a contract and produce EVM bytecode. You can read about many of the most prominent and interesting ones in [high_level_languages]. One high-level language is by far the dominant language for smart contract programming: Solidity. Solidity was created by Gavin Wood, the co-author of this book and has become the most widely used language in Ethereum and beyond. We'll use Solidity to write our first contract.

For our first example, we will write a contract that controls a *faucet*. We've already used a faucet to get test ether on the Ropsten test network. A faucet is a relatively simple thing: it gives out ether to any address that asks and can be refilled periodically. You can implement a faucet as a wallet controlled by a human (or a web server), but we will write a Solidity contract that implements a faucet:

Faucet.sol : A Solidity contract implementing a faucet

```
Unresolved directive in intro.asciidoc - include::code/Faucet.sol[]
```

Download Faucet.sol from: https://github.com/ethereumbook/ethereumbook/blob/first_edition/code/Faucet.sol

This is a very simple contract, about as simple as we can make it. It is also a **flawed** contract, demonstrating a number of bad practices and security vulnerabilities. We will learn by examining all of its flaws in later sections. But for now, let's look at what this contract does and how it works, line by line.

The first line is a comment:

```
// Version of Solidity compiler this program was written for
```

Comments are for humans to read and are not included in the executable EVM bytecode. We usually put them on the line before the code we are trying to explain, or sometimes on the same line. Comments start with two forward slashes //. Everything from the slashes and beyond, until the end of that line, is treated the same as a blank line and ignored.

Ok, the next lines are where our *actual* contract starts:

```
contract Faucet {
```

This line declares a contract object, similar to a class declaration in other object-oriented languages like JavaScript, Java or C++. The contract definition includes all the lines between the curly braces {} which define a scope, much like how curly braces are used in many other programming languages.

Next, we declare the first function of the Faucet contract:

```
function withdraw(uint withdraw_amount) public {
```

The function is named withdraw, which takes one unsigned-integer (uint) argument named withdraw_amount. It is declared as a public function, meaning it can be called by other contracts. The function definition follows between curly braces:

```
require(withdraw_amount <= 100000000000000000);
```

The first part of the withdraw function sets a limit on withdrawals. It uses the built-in Solidity function require to test a precondition, that the withdraw_amount is less than or equal to 100000000000000000 wei, which is the base unit of ether (see Ether Denominations and Unit Names) and equivalent to 0.1 ether. If the withdraw function is called with a withdraw_amount greater than that amount, the require function here will cause contract execution to stop and fail with an *exception*.

This part of the contract is the main logic of our faucet. It controls the flow of funds out of the contract by placing a limit on withdrawals. It's a very simple control but can give you a glimpse of the power of a programmable blockchain: decentralized software controlling money.

Next comes the actual withdrawal:

```
msg.sender.transfer(withdraw_amount);
```

A couple of interesting things are happening here. The msg object is one of the inputs that all contracts can access. It represents the transaction that triggered the execution of this contract. The attribute sender is the sender address of the transaction. The function transfer is a built-in function that transfers ether from the contract to the address it is called from. Reading it backward, this means transfer to the sender of the msg that triggered this contract execution. The transfer function takes an amount as its only argument. We pass the withdraw_amount value that was the parameter to the withdraw function declared a few lines above.

The very next line is the closing curly brace, indicating the end of the definition of our withdraw function.

Below we declare one more function:

```
function () public payable {}
```

This function is a so-called *"fallback"* or *default* function, which is called if the transaction that triggered the contract didn't name any of the declared functions in the contract, or any function at all, or didn't contain data. Contracts can have one such default function (without a name) and it is usually the one that receives ether. That's why it is defined as a public and payable function, which means it can accept ether into the contract. It doesn't do anything, other than accept the ether, as indicated by the empty definition in the curly brackets {}. If we make a transaction that sends ether to the contract address, as if it were a wallet, this function will handle it.

Right below our default function is the final closing curly bracket, which closes the definition of the contract Faucet. That's it!

Compiling the faucet contract

Now that we have our first example contract, we need to use a Solidity compiler to convert the Solidity code into EVM bytecode, so it can be executed by the EVM.

The Solidity compiler comes as a standalone executable, as part of different frameworks, and also bundled in an *Integrated Development Environment (IDE)*. To keep things simple, we will use one of the more popular IDEs, called Remix.

Use your Chrome browser (with the MetaMask wallet we installed earlier) to navigate to the Remix IDE at:

https://remix.ethereum.org/

When you first load Remix, it will start with a sample contract called ballot.sol. We don't need that, so let's close it, clicking on the x on the corner of the tab:

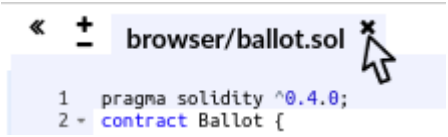

Figure 11. Close the default example tab

Now, add a new tab by clicking on the circular-plus-sign in the left toolbar, naming the new file Faucet.sol:

Figure 12. Click the plus sign to open a new tab

Once you have a new tab open, copy and paste the code from our example Faucet.sol:

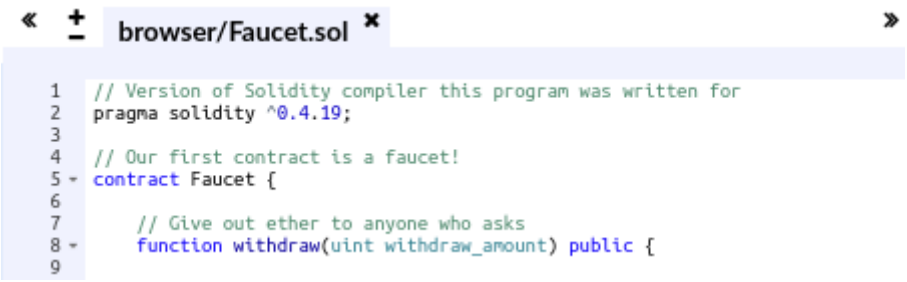

Figure 13. Copy the Faucet example code into the new tab

Now we have loaded the Faucet.sol contract into the Remix IDE, the IDE will automatically compile the code. If all goes well, you will see a green box with "Faucet" in it appear on the right, under the Compile tab, confirming the successful compilation:

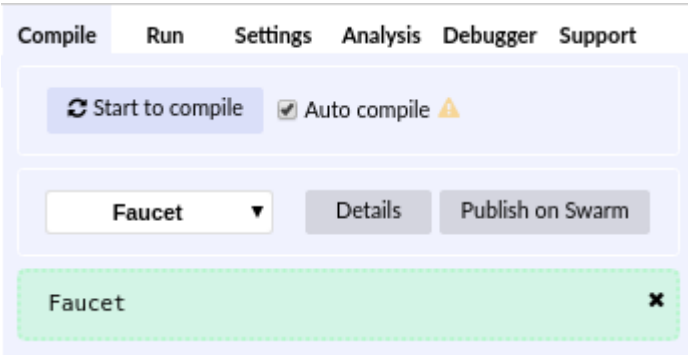

Figure 14. Remix successfully compiles the Faucet.sol contract

If something goes wrong, the most likely problem is that Remix IDE is using a version of the Solidity compiler that is different from 0.4.19. In that case, our pragma directive will prevent Faucet.sol from compiling. To change the compiler version, go to the "Settings" tab, set the compiler version to 0.4.19, and try again.

The Solidity compiler has now compiled our Faucet.sol into EVM bytecode. If you are curious, the bytecode looks like this:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1
0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI PUSH1 0x0
CALLDATALOAD PUSH29 0x100000000000000000000000000000000000000000000000000000 SWAP1
DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI JUMPDEST STOP
JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST PUSH1 0x5F
PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD SWAP1 SWAP2 SWAP1 POP POP PUSH1
0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8 0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO
PUSH1 0x77 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST CALLER PUSH20
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH2 0x8FC DUP3 SWAP1 DUP2 ISZERO MUL
SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP9 DUP9
CALL SWAP4 POP POP POP POP ISZERO ISZERO PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST POP JUMP STOP LOG1 PUSH6 0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75
GASLIMIT CALLVALUE LOG4 0x5f PUSH24 0x7541F409787592C988A079407FB28B4AD000290000000000
```

Aren't you glad you are using a high-level language like Solidity instead of programming directly in EVM bytecode? Me too!

Creating the contract on the blockchain

So we have a contract. We've compiled it into bytecode. Now, we need to "register" the contract on the Ethereum blockchain. We will be using the Ropsten testnet to test our contract, so that's the blockchain we want to record it on.

Registering a contract on the blockchain involves creating a special transaction, whose destination is the address 0x0000000000000000000000000000000000000000, also known as the *zero address*. The zero address is a special address that tells the Ethereum blockchain that you want to register a contract. Fortunately, Remix IDE will handle all of that for you and send the transaction to MetaMask.

First, switch to the "Run" tab and select "Injected Web3" in the "Environment" drop-down selection box. This connects Remix IDE to the MetaMask wallet, and through MetaMask to the Ropsten Test Network. Once you do that, you can see "Ropsten" under Environment. Also, in the Account selection box it shows the address of your wallet:

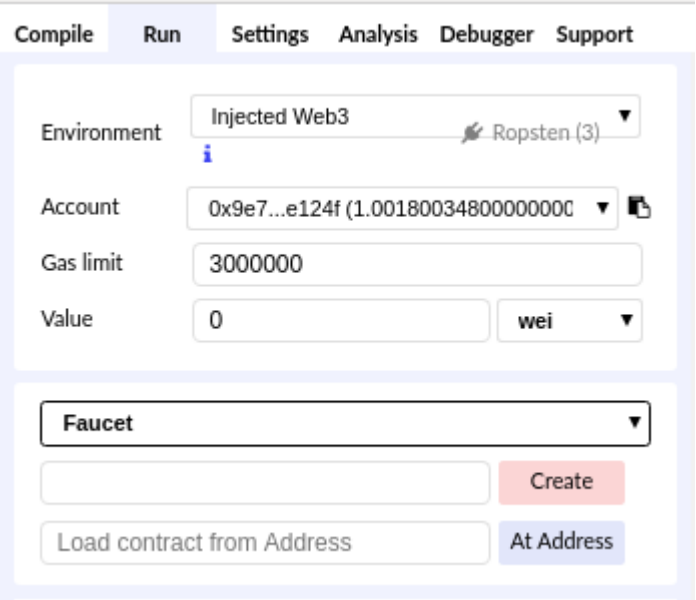

Figure 15. Remix IDE "Run" tab, with "Injected Web3" environment selected

Right below the "Run" settings we just confirmed, is the Faucet contract, ready to be created. Click on the "Create" button:

Figure 16. Click the Create button in the Run tab

Remix IDE will construct the special "creation" transaction and MetaMask will ask you to approve it. As you can see from MetaMask, the contract creation transaction has no ether in it, but it has 258 bytes (the compiled contract) and will consume 10 Gwei in gas. Click "Submit" to approve it:

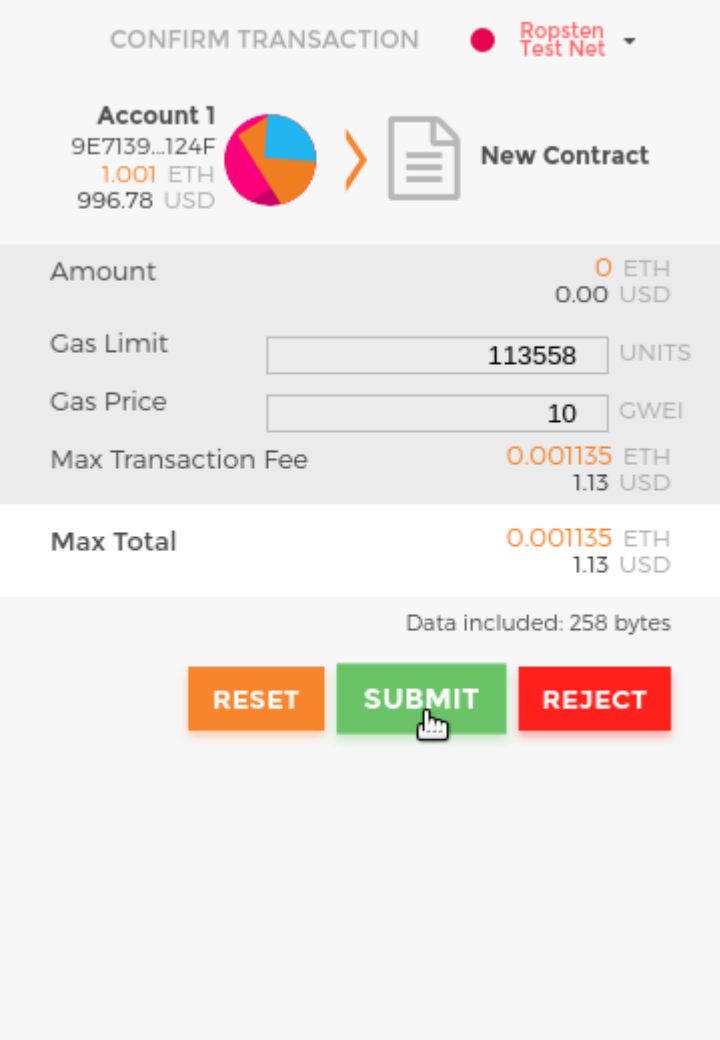

Figure 17. MetaMask showing the contract creation transaction

Now, wait: It will take about 15 to 30 seconds for the contract to be mined on Ropsten. Remix IDE won't appear to be doing much, be patient.

Once the contract is created, it appears at the bottom of the Run tab:

Figure 18. The Faucet contract is ALIVE!

Notice that the Faucet contract now has an address of its own: Remix shows it as Faucet at 0x72e....c7829. The small clipboard symbol to the right allows you to copy the contract address into your clipboard. We will use that in the next section.

Interacting with the contract

Let's recap what we've learned so far: Ethereum contracts are programs that control money, which run inside a virtual machine called the EVM. They are created by a special transaction that submits

their bytecode to be recorded on the blockchain. Once they are created on the blockchain, they have an Ethereum address, just like wallets. Anytime someone sends a transaction to a contract address it causes the contract to run in the EVM, with the transaction as its input. Transactions sent to contract addresses may have ether or data or both in them. If they contain ether, it is "deposited" to the contract balance. If they contain data, the data can specify a named function in the contract and call it, passing arguments to the function.

Viewing the contract address in a block explorer

Now, we have a contract recorded on the blockchain and we can see it has an Ethereum address. Let's check it out on the ropsten.etherscan.io block explorer and see what a contract looks like. Copy the address of the contract by clicking on the clipboard icon next to its name:

Figure 19. Copy the contract address from Remix

Keep Remix open in a tab, we'll come back to it again later. Now, navigate your browser to ropsten.etherscan.io and paste the address into the search box. You should see the contract's Ethereum address history:

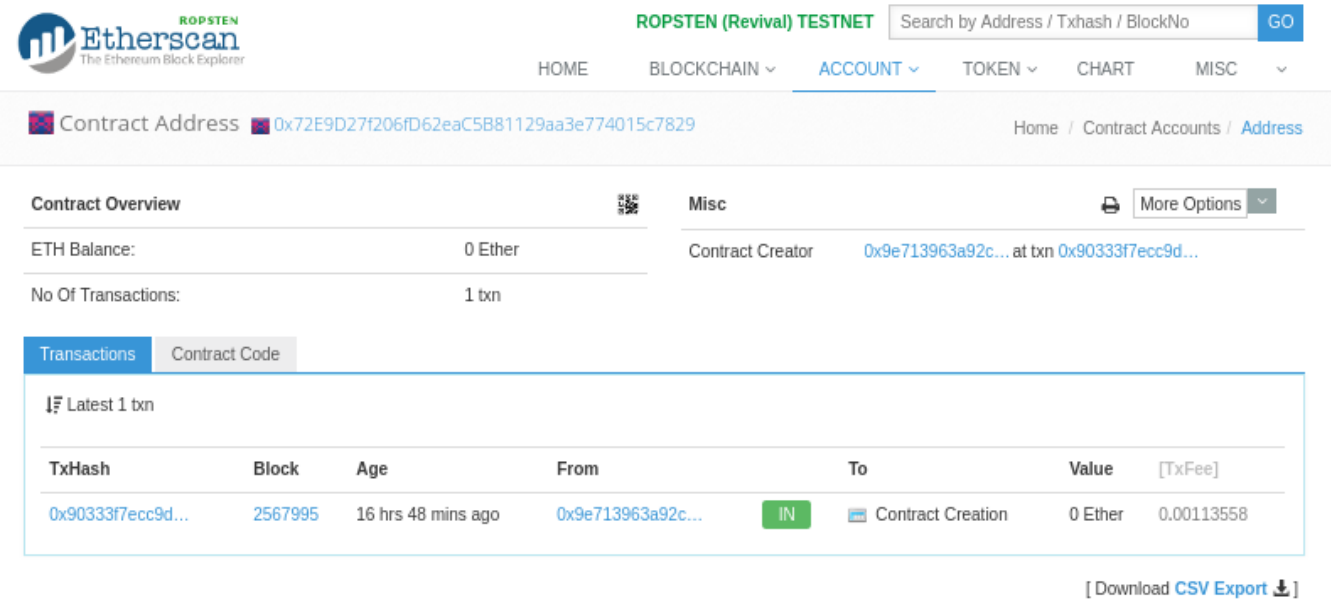

Figure 20. View the Faucet contract address in the etherscan block explorer

Funding the contract

For now, the contract only has one transaction in its history: the contract creation transaction. As you can see, the contract also has no ether (zero balance). That's because we didn't send any ether to the contract in the creation transaction, even though we could have.

Let's send some ether to the contract! You should still have the address of the contract in your clipboard (if not, copy it again from Remix). Open MetaMask, and send 1 ether to it, exactly as you would any other Ethereum address:

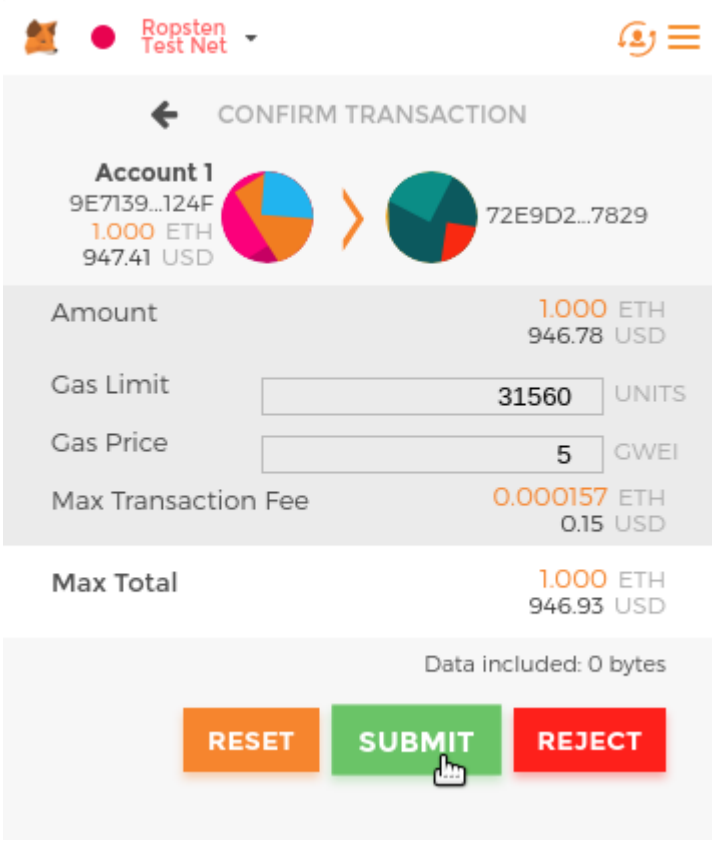

Figure 21. Send 1 ether to the contract address

In a minute, if you reload the etherscan block explorer, it will show another transaction to the contract address and an updated balance of 1 ether.

Remember the unnamed default public payable function in our Faucet.sol code? It looked like this:

```
function () public payable {}
```

When you sent a transaction to the contract address, with no data specifying which function to call, it called this default function. Because we declared it as a payable, it accepted and deposited the 1 ether into the contract account balance. Your transaction caused the contract to run in the EVM, updating its balance. We have funded our faucet!

Withdrawing from our contract

Next, let's withdraw some funds from the faucet. To withdraw, we have to construct a transaction that calls the withdraw function and passes a withdraw_amount argument to it. To keep things

simple for now, Remix will construct that transaction for us and MetaMask will present it for our approval.

Return to the Remix tab and look at the contract under the "Run" tab. You should see a red box labeled withdraw with a field entry labeled uint256 withdraw_amount :

Figure 22. The withdraw function of Faucet.sol, in Remix

This is the Remix interface to the contract. It allows us to construct transactions that call the functions defined in the contract. We will enter a withdraw_amount and click the withdraw button to generate the transaction.

First, let's figure out the withdraw_amount. We want to try and withdraw 0.1 ether, which is the maximum amount allowed by our contract. Remember that all currency values in Ethereum are denominated in wei internally, and our withdraw function expects the withdraw_amount to be denominated in wei too. The amount we want is 0.1 ether, which is 100000000000000000 wei (1 followed by 17 zeros).

TIP Due to a limitation in JavaScript, a number as large as 10^17 cannot be processed by Remix. Instead, we enclose it in double quotes, to allow Remix to receive it as a string and manipulate it as a BigNumber. If we don't enclose it in quotes, the Remix IDE will fail to process it and display "Error encoding arguments: Error: Assertion failed"

Type "100000000000000000" (with the quotes) into the withdraw_amount box and click on the withdraw button:

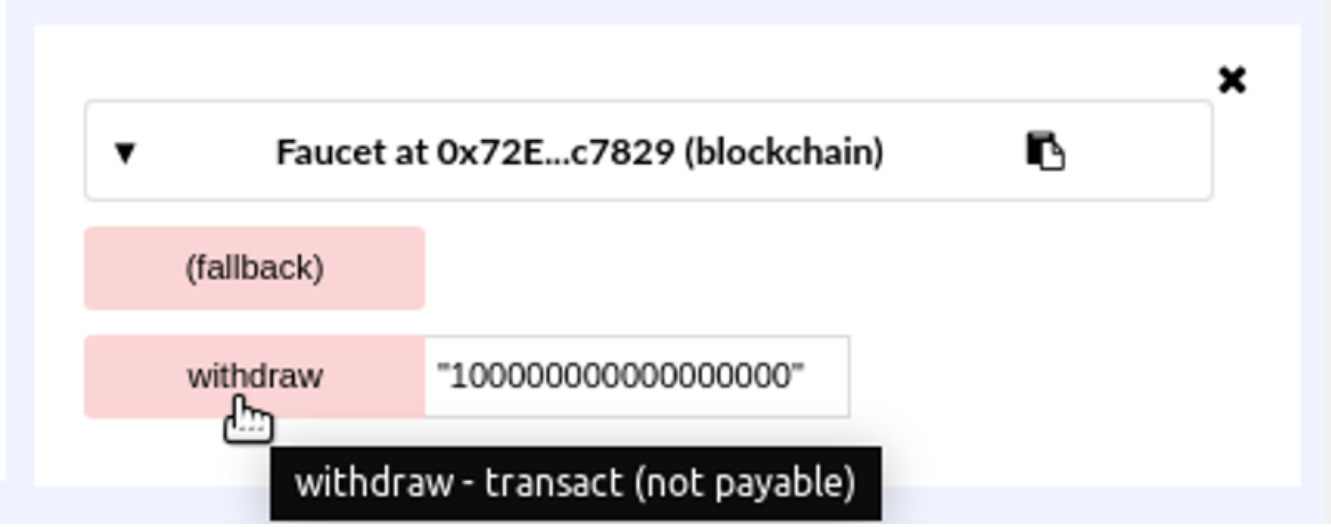

Figure 23. Click "withdraw" in Remix to create a withdrawal transaction

MetaMask will pop-up a transaction window for you to approve. Click "Submit" to send your withdrawal call to the contract:

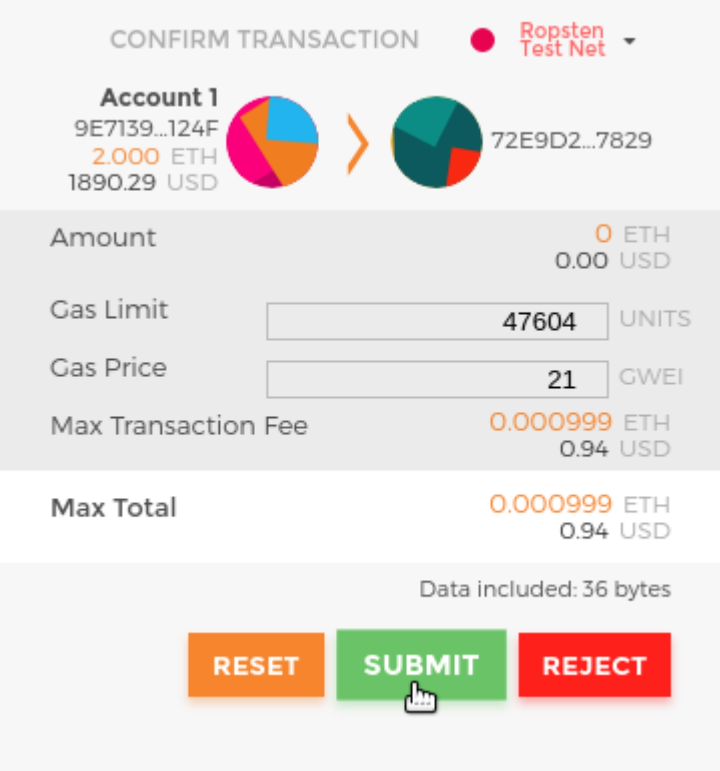

Figure 24. MetaMask transaction to call the withdraw function

Wait a minute and then reload the etherscan block explorer to see the transaction reflected in the Faucet contract address history:

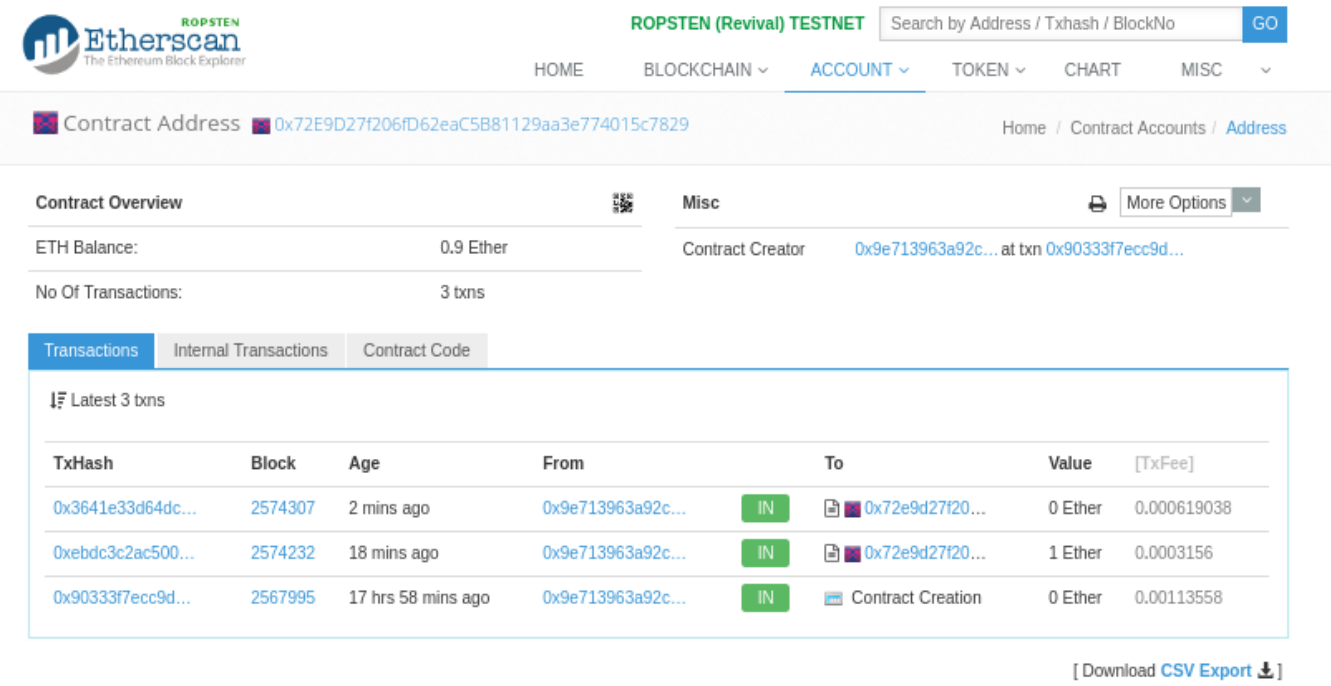

Figure 25. Etherscan shows the transaction calling the withdraw function

We now see a new transaction with the contract address as the destination and zero ether. The contract balance has changed and is now 0.9 ether because it sent us 0.1 ether as requested. But we don't see an "OUT" transaction in the contract address history.

Where's the outgoing withdrawal? A new tab has appeared in the contract's address history page, named "Internal Transactions". Because the 0.1 ether transfer originated from the contract code, it

is an internal transaction (also called a *message*). Click on the "Internal Transactions" tab to see it:

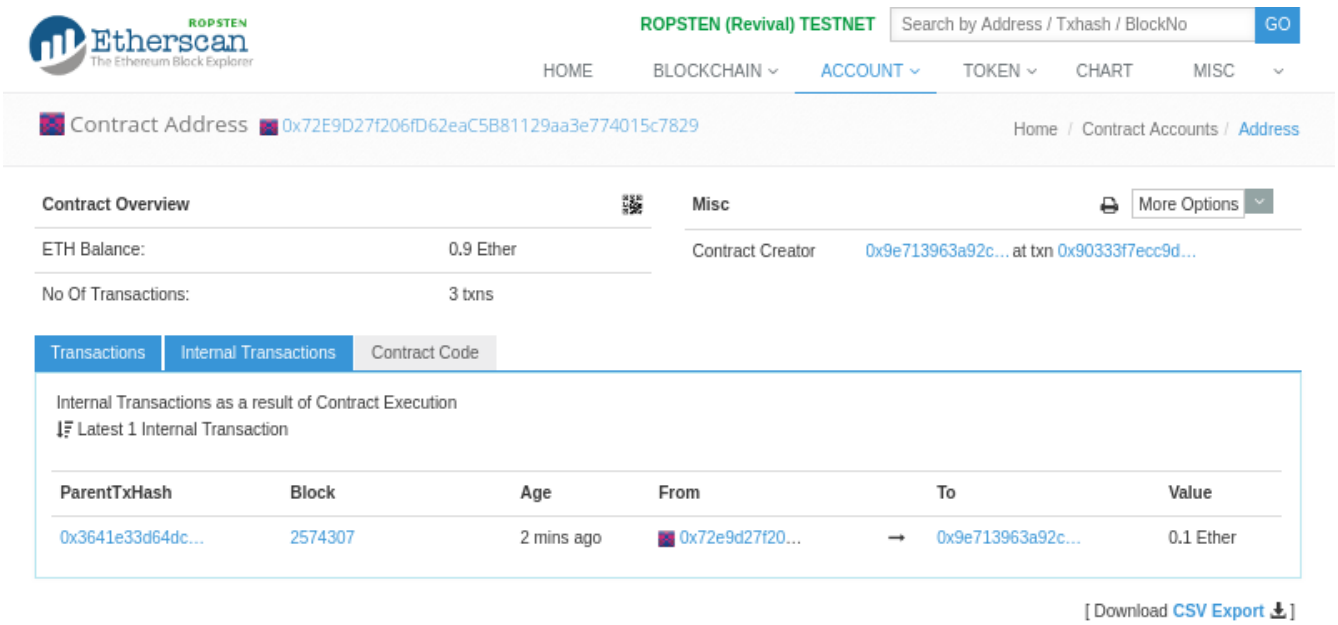

Figure 26. Etherscan shows the internal transaction transferring ether out from the contract

This "internal transaction" was sent by the contract in this line of code (from the withdraw function in Faucet.sol):

```
msg.sender.transfer(withdraw_amount);
```

To recap: We sent a transaction from our MetaMask wallet that contained data instructions to call the withdraw function with a withdraw_amount argument of 0.1 ether. That transaction caused the contract to run inside the EVM. As the EVM ran the Faucet contract's withdraw function, first it called the require function and validated that our amount was less than or equal to the maximum allowed withdrawal of 0.1 ether. Then it called the transfer function to send us the ether. Running the transfer function generated an internal transaction that deposited 0.1 ether into our wallet address, from the contract's balance. That's the one shown in the "Internal Transactions" tab in etherscan.

Conclusion

In this chapter, we've set up a wallet using MetaMask and we've funded it using a faucet on the Ropsten Test Network. We received ether into our wallet's Ethereum address. Then we sent ether to the faucet's Ethereum address.

Next, we wrote a faucet contract in Solidity. We used the Remix IDE to compile the contract into EVM bytecode. We used Remix to form a transaction and recorded the faucet contract on the Ropsten blockchain. Once recorded, the faucet contract had an Ethereum address and we sent it some ether. Finally, we constructed a transaction to call the withdraw function and successfully asked for 0.1 ether. The contract checked our request and sent us 0.1 ether with an internal transaction.

It may not seem like much, but we've just successfully interacted with software that controls money

on a decentralized world computer.

We will do a lot more smart contract programming in [smart contracts] and learn about best practices and security considerations.

Ethereum Clients

An Ethereum client is a software application that implements the Ethereum specification and communicates over the peer-to-peer network with other Ethereum clients. Different Ethereum clients *interoperate* if they comply with the reference specification and the standardized communications protocols. While these different clients are implemented by different teams and in different programming languages, they all "speak" the same protocol and follow the same rules.

Ethereum is an *open source* project and the source code is available under an open (LGPL v3.0) license, free to download and use for any purpose. Open source means more than simply free to use. It also means that Ethereum is developed by an open community of volunteers and can be modified by anyone.

Ethereum is defined by a formal specification called the "Yellow Paper." This is in contrast to, for example, Bitcoin, which is not defined in any formal way. Where Bitcoin's "specification" is the reference implementation Bitcoin Core, Ethereum's specification is documented in a paper that combines an English and a mathematical (formal) specification. This formal specification, in addition to various Ethereum Improvement Proposals, defines the standard behavior of an Ethereum client. The Yellow Paper is periodically updated as major changes are made to Ethereum.

As a result of Ethereum's clear formal specification, there are a number of independently developed, yet interoperable, software implementations of an Ethereum client. Ethereum has a greater diversity of implementations running on the network than any other blockchain.

Ethereum Networks

There exist a variety of Ethereum-based networks which largely conform to the formal specification defined in the Ethereum "Yellow Paper," but which may or may not interoperate with each other.

Among these Ethereum-based networks are: Ethereum, Ethereum Classic, Ella, Expanse, Ubiq, Musicoin, and many others. While mostly compatible on a protocol-level, these networks often have features or attributes that require maintainers of Ethereum client software to make small changes in order to support each network. Because of this, not every version of Ethereum client software runs on every Ethereum-based blockchain.

Currently, there are six main implementations of the Ethereum protocol written in six different languages: Go (Geth), Rust (parity), C++ (cpp-ethereum), Python (pyethereum), Scala (mantis) and Java (harmony).

In this section, we will look at the two most common clients, Geth and Parity. We'll learn to set up a node using each client and explore some of their command-line and application programming interfaces (APIs).

Should I run a full node?

The health, resilience, and censorship resistance of blockchains depend on having many independently operated and geographically dispersed full nodes. Each full node can help other new nodes obtain the block data to bootstrap their operation, as well as offer the operator an authoritative and independent verification of all transactions and contracts.

However, running a full node will incur a significant cost in hardware resources and bandwidth. A full node must download more than 80GB of data (as of April 2018; depending on client) and store it on a local hard drive. This data burden increases quite rapidly every day as new transactions and blocks are added. More on this topic in Hardware Requirements for a Full Node.

A full node running on a live network (mainnet) is not necessary for Ethereum development. You can do almost everything you need to do with a *testnet* node (which stores a copy of the smaller public test blockchain), with a local private blockchain (see [ganache]), or with a cloud-based Ethereum client offered by a service provider (see [infura]).

You also have the option of running a lightweight client which does not store a local copy of the blockchain or validate blocks and transactions. These clients offer the functionality of a wallet and can create and broadcast transactions.

Lightweight clients can be used to connect to existing networks, such as your own full node, a public blockchain, a public or permissioned (PoA) testnet, or a private local blockchain. In practice, you will likely use a lightweight client such as MetaMask, Emerald Wallet, MyEtherWallet or MyCrypto as a convenient way to switch between all of the different node options.

The terms "lightweight client" and "wallet" are used interchangeably, though there are some differences. Usually, a lightweight client offers an API (such as the web3js API) in addition to the transaction functionality of a wallet.

Do not confuse the concept of a lightweight wallet in Ethereum with that of a Simplified Payment Verification (SPV) client in Bitcoin. SPV clients validate block headers and use merkle proofs to validate the inclusion of transactions in the blockchain. Ethereum lightweight clients, generally, do not validate block headers or transactions. They entirely trust a full client operated by a third party to give them RPC access to the blockchain.

Full Node Advantages and Disadvantages

Choosing to run a full node helps the various Ethereum-based networks, but also incurs some mild to moderate costs for you. Let’s look at some of the advantages and disadvantages.

Advantages:

- Supports the resilience and censorship resistance of Ethereum-based networks.
- Authoritatively validates all transactions.
- Can interact with any contract on the public blockchain (without requiring an intermediary).
- Can query (read-only) the blockchain status (accounts, contracts, etc.) offline, if necessary.
- Can query the blockchain without letting a third party know the information you’re reading.

- Can directly deploy your own contracts into the public blockchain (without requiring an intermediary).

Disadvantages:

- Requires significant and growing hardware and bandwidth resources.
- Requires several hours or days to fully sync for the first initial download.
- Must be maintained, upgraded and kept online to remain synced.

Public Testnet Advantages and Disadvantages

Whether or not you choose to run a full node, you will probably want to run a public testnet node. Let's look at some of the advantages and disadvantages of using a public testnet.

Advantages:

- A testnet node needs to sync and store much less data, ~10GB depending on the network (as of April 2018).
- A testnet node can sync fully in a few hours.
- Deploying contracts or making transactions requires test ether, which has no value and can be acquired for free from several "faucets".
- Testnets are public blockchains with many other users and contracts, running "live."

Disadvantages:

- You can't use "real" money on a testnet, it runs on test ether.
- Consequently, you can't test security against real adversaries, as there is nothing at stake.
- There are some aspects of a public blockchain that you cannot test realistically on testnet. For example, transaction fees, although necessary to send transactions, are not a consideration on testnet since gas is free. And the testnets do not experience network congestions like the public network.

Local Instance (TestRPC) Advantages and Disadvantages

For many testing purposes, the best option is to launch a single instance private blockchain, using the testrpc node. TestRPC creates a local-only, private blockchain that you can interact with, without any other participants. It shares many of the advantages and disadvantages of the public testnet, but also has some differences.

Advantages:

- No syncing and almost no data on disk. You mine the first block yourself.
- No need to find test ether, you "award" yourself mining rewards that you can use for testing.
- No other users, just you.
- No other contracts, just the ones you deploy after you launch it.

Disadvantages:

- Having no other users means that it doesn't behave the same as a public blockchain. There's no competition for transaction space or sequencing of transactions.
- No miners other than you means that mining is more predictable, therefore you can't test some scenarios that occur on a public blockchain.
- Having no other contracts means you have to deploy everything that you want to test, including dependencies and contract libraries.
- You can't recreate some of the public contracts and their addresses to test some scenarios (e.g. the DAO contract).

Running an Ethereum client

If you have the time and resources, you should attempt to run a full node, even if only to learn more about the process. In the next few sections we will download, compile, and run the Ethereum clients Go-Ethereum (Geth) and Parity. This requires some familiarity with using the command-line interface on your operating system. It's worth installing these clients whether you choose to run them as full nodes, as testnet nodes, or as clients to a local private blockchain.

Hardware Requirements for a Full Node

Before we get started, you should ensure you have a computer with sufficient resources to run an Ethereum full node. You will need at least 80GB of disk space to store a full copy of the Ethereum blockchain. If you also want to run a full node on the Ethereum testnet, you will need at least an additional 15GB. Downloading 80GB of blockchain data can take a long time, so it's recommended that you work on a fast Internet connection.

Syncing the Ethereum blockchain is very input-output (I/O) intensive. It is best to have a Solid-State Drive (SSD). If you have a mechanical hard disk drive (HDD), you will need at least 8GB of RAM to use as cache. Otherwise, you may discover that your system is too slow to keep up and sync fully.

Minimum Requirements:

- CPU with 2+ cores.
- Solid State Drive (SSD) with at least 80GB free space.
- 4GB RAM minimum, 8GB+ if you have an HDD and not SSD.
- 8 MBit/sec download Internet service.

These are the minimum requirements to sync a full (but pruned) copy of an Ethereum-based blockchain.

At the time of writing (April 2018) the Parity codebase tends to be lighter on resources, if you're running with limited hardware you'll likely see the best results using Parity.

If you want to sync in a reasonable amount of time and store all the development tools, libraries, clients, and blockchains we discuss in this book, you will want a more capable computer.

Recommended Specifications:

- Fast CPU with 4+ cores.
- 16GB+ RAM.
- Fast SSD with at least 500GB free space.
- 25+ MBit/sec download Internet service.

It's difficult to predict how fast a blockchain's size will increase and when more disk space will be required, so it's recommended to check the blockchain's latest size before you start syncing.

Ethereum: https://bitinfocharts.com/ethereum/

Ethereum Classic: https://bitinfocharts.com/ethereum%20classic/

Software Requirements for Building and Running a Client (Node)

This section covers Geth and Parity client software. It also assumes you are using a Unix-like command-line environment. The examples show the output and commands as entered on an Ubuntu Linux operating system running the Bash shell (command-line execution environment).

Typically every blockchain will have their own version of Geth, while Parity provides support for multiple Ethereum-based blockchains (Ethereum, Ethereum Classic, Ellaism, Expanse, Musicoin).

TIP

In many of the examples in this chapter, we will be using the operating system's command-line interface (also known as a "shell"), accessed via a "terminal" application. The shell will display a prompt; you type a command, and the shell responds with some text and a new prompt for your next command. The prompt may look different on your system, but in the following examples, it is denoted by a $ symbol. In the examples, when you see text after a $ symbol, don't type the $ symbol but type the command immediately following it, then press Enter to execute the command. In the examples, the lines below each command are the operating system's responses to that command. When you see the next $ prefix, you'll know it's a new command and you should repeat the process.

Before we get started, we may need to get some prerequisites satisfied. If you've never done any software development on the computer you are currently using, you will probably need to install some basic tools. For the examples that follow, you will need to install git, the source-code management system; golang, the Go programming language and standard libraries; and Rust, a systems programming language.

Git can be installed by following the instructions here: https://git-scm.com/

Go can be installed by following the instructions here: https://golang.org/

NOTE	Geth requirements vary, but if you stick with Go version 1.10 or greater you should be able to compile any version of Geth you want. Of course, you should always refer to the documentation for your chosen flavor of Geth. The version of golang that is installed on your operating system or is available from your system's package manager may be significantly older than 1.10. If so, remove it and install the latest version from golang.org.

Rust can be installed by following the instructions here: https://www.rustup.rs/

NOTE	Parity requires Rust version 1.24 or greater.

Parity also requires some software libraries, such as OpenSSL and libudev. To install these on a Linux (Debian) compatible system:

```
$ sudo apt-get install openssl libssl-dev libudev-dev
```

For other operating systems, use the package manager of your OS or follow the Wiki instructions (https://github.com/paritytech/parity/wiki/Setup) to install the required libraries.

Now you have git, golang, rust, and necessary libraries installed, let's get to work!

Parity

Parity is an implementation of a full node Ethereum client and DApp browser. Parity was written from the "ground up" in Rust, a systems programming language with the aim of building a modular, secure, and scalable Ethereum client. Parity is developed by Parity Tech, a UK company, and is released under a GPLv3 open source license.

NOTE	Disclosure: One of the authors of this book, Gavin Wood, is the founder of Parity Tech and wrote most of the Parity client. Parity represents about 28% of the installed Ethereum client base.

To install Parity, you can use the Rust package manager cargo or download the source code from GitHub. The package manager also downloads the source code, so there's not much difference between the two options. In the next section, we will show you how to download and compile Parity yourself.

Installing Parity

The Parity Wiki offers instructions for building Parity in different environments and containers:

https://github.com/paritytech/parity/wiki/Setup

We'll build Parity from source. This assumes you have already installed Rust using rustup (See Software Requirements for Building and Running a Client (Node)).

First, let's get the source code from GitHub:

```
$ git clone https://github.com/paritytech/parity
```

Now, let's change to the parity directory and use cargo to build the executable:

```
$ cd parity
$ cargo build
```

If all goes well, you should see something like:

```
$ cargo build
    Updating git repository `https://github.com/paritytech/js-precompiled.git`
 Downloading log v0.3.7
 Downloading isatty v0.1.1
 Downloading regex v0.2.1

 [...]

Compiling parity-ipfs-api v1.7.0
Compiling parity-rpc v1.7.0
Compiling parity-rpc-client v1.4.0
Compiling rpc-cli v1.4.0 (file:///home/aantonop/Dev/parity/rpc_cli)
Finished dev [unoptimized + debuginfo] target(s) in 479.12 secs
$
```

Let's try and run parity to see if it is installed, by invoking the --version option:

```
$ parity --version
Parity
  version Parity/v1.7.0-unstable-02edc95-20170623/x86_64-linux-gnu/rustc1.18.0
Copyright 2015, 2016, 2017 Parity Technologies (UK) Ltd
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

By Wood/Paronyan/Kotewicz/Drwięga/Volf
   Habermeier/Czaban/Greeff/Gotchac/Redmann
$
```

Great! Now that Parity is installed, we can sync the blockchain and get started with some basic command-line options.

Go-Ethereum (Geth)

Geth is the Go language implementation, which is actively developed and considered the "official" implementation of the Ethereum client. Typically, every Ethereum-based blockchain will have its own Geth implementation. If you're running Geth, then you'll want to make sure you grab the

correct version for your blockchain using one of the repository links below.

Repository Links

Ethereum: https://github.com/ethereum/go-ethereum (or https://geth.ethereum.org/)

Ethereum Classic: https://github.com/ethereumproject/go-ethereum

Ellaism: https://github.com/ellaism/go-ellaism

Expanse: https://github.com/expanse-org/go-expanse

Musicoin: https://github.com/Musicoin/go-musicoin

Ubiq: https://github.com/ubiq/go-ubiq

NOTE	You can also skip these instructions and install a precompiled binary for your platform of choice. The precompiled releases are much easier to install and can be found at the "release" section of the repositories above. However, you may learn more by downloading and compiling the software yourself.

Cloning the repository

Our first step is to clone the git repository, so as to get a copy of the source code.

To make a local clone of this repository, use the git command as follows, in your home directory or under any directory you use for development:

```
$ git clone <Repository Link>
```

You should see a progress report as the repository is copied to your local system:

```
Cloning into 'go-ethereum'...
remote: Counting objects: 62587, done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 62587 (delta 10), reused 13 (delta 4), pack-reused 62557
Receiving objects: 100% (62587/62587), 84.51 MiB | 1.40 MiB/s, done.
Resolving deltas: 100% (41554/41554), done.
Checking connectivity... done.
```

Great! Now that we have a local copy of Geth, we can compile an executable for our platform.

Building Geth from Source Code

To build Geth, change to the directory where the source code was downloaded and use the make command:

```
$ cd go-ethereum
$ make geth
```

If all goes well, you will see the Go compiler building each component until it produces the geth executable:

```
build/env.sh go run build/ci.go install ./cmd/geth
>>> /usr/local/go/bin/go install -ldflags -X
main.gitCommit=58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c -v ./cmd/geth
github.com/ethereum/go-ethereum/common/hexutil
github.com/ethereum/go-ethereum/common/math
github.com/ethereum/go-ethereum/crypto/sha3
github.com/ethereum/go-ethereum/rlp
github.com/ethereum/go-ethereum/crypto/secp256k1
github.com/ethereum/go-ethereum/common
[...]
github.com/ethereum/go-ethereum/cmd/utils
github.com/ethereum/go-ethereum/cmd/geth
Done building.
Run "build/bin/geth" to launch geth.
$
```

Let's run geth to make sure it works before stopping it and chainging it's configuration:

```
$ ./build/bin/geth version

Geth
Version: 1.6.6-unstable
Git Commit: 58a1e13e6dd7f52a1d5e67bee47d23fd6cfdee5c
Architecture: amd64
Protocol Versions: [63 62]
Network Id: 1
Go Version: go1.8.3
Operating System: linux
GOPATH=/usr/local/src/gocode/
GOROOT=/usr/local/go
```

Your geth version command may show slightly different information, but you should see a version report much like the one above.

Finally, we may want to copy the geth command to our operating system's application directory (or a directory on the command-line execution path). On Linux, we'd use the following command:

```
$ sudo cp ./build/bin/geth /usr/local/bin
```

Don't start running geth yet, because it will start synchronizing the blockchain "the slow way," and

that will take far too long (weeks). The First Synchronization of Ethereum-based Blockchains explains the challenge with the initial synchronization of Ethereum's blockchain.

The First Synchronization of Ethereum-based Blockchains

Normally, when syncing an Ethereum blockchain, your client will download and validate every block and every transaction since the genesis block.

While it is possible to fully sync the blockchain this way, the sync will take a very long time and has high computing resource requirements (much more RAM and faster storage).

Many Ethereum-based blockchains were the victim of a Denial-of-Service (DoS) attack at the end of 2016. Blockchains affected by this attack will tend to sync slowly when doing a full sync.

For example, on Ethereum, a new client will make rapid progress until it reaches block 2,283,397. This block was mined on September 18th, 2016 and marks the beginning of the DoS attacks. From this block and until block 2,700,031 (November 26th, 2016), the validation of transactions becomes extremely slow, memory intensive, and I/O intensive. This results in validation times exceeding 1 minute per block. Ethereum implemented a series of upgrades, using hard forks, to address the underlying vulnerabilities that were exploited in the denial of service. These upgrades also cleaned up the blockchain by removing some 20 million empty accounts created by spam transactions. <<[1]>>

If you are syncing with full validation, your client will slow down and may take several days or longer to validate any blocks affected by this DoS attack.

Most Ethereum clients include an option to perform a "fast" synchronization that skips the full validation of transactions until it has synced to the tip of the blockchain, then resumes full validation.

For Geth, the option to enable fast synchronization is typically called --fast. You may need to refer to the specific instructions for your chosen Ethereum chain.

For Parity, the option is --warp for older versions (< 1.6) and is enabled by default (no need to set a configuration option) on newer versions (>= 1.6).

NOTE	Geth and Parity can only operate fast synchronization when starting with an empty block database. If you have already started syncing without "fast" mode, Geth and Parity cannot switch. It is faster to delete the blockchain data directory and start "fast" syncing from the beginning than to continue syncing with full validation. Be careful to not delete any wallets when deleting the blockchain data!

JSON-RPC Interface

Ethereum clients offer an Application Programming Interface (API) and a set of Remote Procedure Call (RPC) commands, which are encoded as JavaScript Object Notation (JSON). You will see this referred to as the *JSON-RPC API*. Essentially, the JSON-RPC API is an interface that allows us to write

programs that use an Ethereum client as a *gateway* into an Ethereum network and blockchain.

Usually, the RPC interface is offered over as an HTTP service on port 8545. For security reasons it is restricted, by default, to only accept connections from localhost (the IP address of your own computer which is 127.0.0.1).

To access the JSON-RPC API, you can use a specialized library, written in the programming language of your choice, which provides "stub" function calls corresponding to each available RPC command. Or, you can manually construct HTTP requests and send/receive JSON encoded requests. You can even use a generic command-line HTTP client, like curl, to call the RCP interface. Let's try that(ensure that you have Geth configured and running first):

Using curl to call the web3_clientVersion function over JSON-RPC

```
$ curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":1}' \
http://localhost:8545

{"jsonrpc":"2.0","id":1,
"result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

In this example, we use curl to make an HTTP connection to address http://localhost:8545. We are already running geth, which offers the JSON-RPC API as an HTTP service on port 8545. We instruct curl to use the HTTP POST command and to identify the content as Content-Type: application/json. Finally, we pass a JSON-encoded request as the data component of our HTTP request. Most of our command line is just setting up curl to make the HTTP connection correctly. The interesting part is the actual JSON-RPC command we issue:

```
{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":4192}
```

The JSON-RPC request is formatted according to the JSON-RPC 2.0 specification, which you can see here: http://www.jsonrpc.org/specification

Each request contains 4 elements:

jsonrpc

Version of the JSON-RPC protocol. This MUST be exactly "2.0".

method

The name of the method to be invoked.

params

A structured value that holds the parameter values to be used during the invocation of the method. This member MAY be omitted.

id

An identifier established by the Client that MUST contain a String, Number, or NULL value if included. The Server MUST reply with the same value in the Response object if included. This

member is used to correlate the context between the two objects.

TIP

The id parameter is used primarily when you are making multiple requests in a single JSON-RPC call, a practice called *batching*. Batching is used to avoid the overhead of a new HTTP and TCP connection for every request. In the Ethereum context for example, we would use batching if we wanted to retrieve thousands of transactions in one HTTP connection. When batching, you set a different id for each request and then match it to the id in each response from the JSON-RPC server. The easiest way to implement this is to maintain a counter and increment the value for each request.

The response we receive is:

```
{"jsonrpc":"2.0","id":4192,
"result":"Geth/v1.8.0-unstable-02aeb3d7/linux-amd64/go1.8.3"}
```

This tells us that the JSON-RPC API is being served by Geth client version 1.8.0.

Let's try something a bit more interesting. In the next example, we ask the JSON-RPC API for the current price of gas in wei:

```
$ curl -X POST -H "Content-Type: application/json" --data \
'{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":4213}' \
http://localhost:8545

{"jsonrpc":"2.0","id":4213,"result":"0x430e23400"}
```

The response, 0x430e23400, tells us that the current gas price is 1.8 Gwei (gigawei or billion wei).

https://github.com/ethereum/wiki/wiki/JSON-RPC

Parity's Geth Compatibility Mode

Parity has a special "Geth Compatibility Mode", where it offers a JSON-RPC API that is identical to that offered by geth. To run Parity in Geth Compatibility Mode, use the --geth switch:

```
$ parity --geth
```

Lightweight Ethereum Clients

Lightweight clients offer a subset of the functionality of a full client. They do not store the full Ethereum blockchain, so they are faster to setup and require far less data storage.

A lightweight client offers one or more of the following functions:

- Manage private keys and Ethereum addresses in a wallet.
- Create, sign, and broadcast transactions.

- Interact with smart contracts, using the data payload.
- Browse and interact with DApps.
- Offer links to external services such as block explorers.
- Convert ether units and retrieve exchange rates from external sources.
- Inject a web3 instance into the web browser as a JavaScript object.
- Use a web3 instance provided/injected into the browser by another client.
- Access RPC services on a local or remote Ethereum node.

Some lightweight clients, for example mobile (smartphone) wallets, offer only basic wallet functionality. Other lightweight clients are fully-developed DApp browsers. Lightweight clients commonly offer some of the functions of a full node Ethereum client without synchronizing a local copy of the Ethereum blockchain.

Let's look at some of the most popular lightweight clients and the functions they offer.

Mobile (Smartphone) Wallets

All mobile wallets are lightweight clients because smartphones do not have adequate resources to run a full Ethereum client.

Popular mobile wallets include Jaxx, Status, and Trust Wallet. We list these as examples of popular mobile wallets (this is not an endorsement or an indication of the security or functionality of these wallets).

Jaxx

A multi-currency mobile wallet based on BIP39 mnemonic seeds, with support for Bitcoin, Litecoin, Ethereum, Ethereum Classic, ZCash, a variety of ERC20 tokens and many other currencies. Jaxx is available on Android, iOS, as a browser plugin wallet, and desktop wallet for a variety of operating systems. Find it at https://jaxx.io

Status

A mobile wallet and DApp browser, with support for a variety of tokens and popular DApps. Available for iOS and Android smartphones. Find it at https://status.im

Trust Wallet

A mobile Ethereum, Ethereum Classic wallet that supports ERC20, and ERC223 tokens. Trust Wallet is available for iOS and Android smartphones. Find it at https://trustwalletapp.com/

Cipher Browser

A full-featured Ethereum-enabled mobile DApp browser and wallet. Allows integration with Ethereum apps and tokens. Find it at https://www.cipherbrowser.com

Browser wallets

A variety of wallets and DApp browsers are available as plugins or extensions of web browsers such as Chrome and Firefox: lightweight clients that run inside your browser.

Some of the more popular ones are MetaMask, Jaxx, and MyEtherWallet/MyCrypto.

MetaMask

MetaMask was introduced in [intro], and is a versatile browser-based wallet, RPC client, and basic contract explorer. It is available on Chrome, Firefox, Opera, and Brave Browser. Find MetaMask at:

https://metamask.io

At first glance, MetaMask is a browser-based wallet. But, unlike other browser wallets, MetaMask injects a web3 instance into the browser, acting as an RPC client that connects to a variety of Ethereum blockchains (eg. mainnet, Ropsten testnet, Kovan testnet, local RPC node, etc.). The ability to inject a web3 instance and act as a gateway to external RPC services, makes MetaMask a very powerful tool for developers and users alike. It can be combined, for example, with MyEtherWallet or MyCrypto, acting as an web3 provider and RPC gateway for those tools.

Jaxx

Jaxx, which was introduced as a mobile wallet in Mobile (Smartphone) Wallets, is also available as a Chrome and Firefox extension. Find it at:

https://jaxx.io

MyEtherWallet (MEW)

MyEtherWallet is a browser-based JavaScript lightweight client that offers:

- A software wallet running in JavaScript.
- A bridge to popular hardware wallets such as the Trezor and Ledger.
- A web3 interface that can connect to a web3 instance injected by another client (eg. MetaMask).
- An RPC client that can connect to an Ethereum full client.
- A basic interface that can interact with smart contracts, given a contract's address and Application Binary Interface (ABI).

MyEtherWallet is very useful for testing and as an interface to hardware wallets. It should not be used as a primary software wallet, as it is exposed to threats via the browser environment and is not a secure key storage system.

You must be very careful when accessing MyEtherWallet and other browser-based JavaScript wallets, as they are frequent targets for phishing. Always use a bookmark and not a search engine or link to access the correct web URL. MyEtherWallet can be found at:

https://MyEtherWallet.com

MyCrypto

Just prior to publication of the first edition of this book, the MyEtherWallet project split into two competing implementations, guided by two independent development teams: a "fork" as it is called

in open source development. The two projects are called MyEtherWallet (the original branding) and MyCrypto. At the time of the split, MyCrypto offered identical functionality as MyEtherWallet. It is likely that the two projects will diverge as the two development teams adopt different goals and priorities.

As with MyEtherWallet, you must be very careful when accessing MyCrypto in your browser. Always use a bookmark, or type the URL very carefully (then bookmark it for future use).

MyCrypto can be found at:

https://MyCrypto.com

Mist

Mist is the first ever Ethereum enabled browser, built by the Ethereum Foundation. It also contains a browser-based wallet that was the first ever implementation of the ERC20 token standard (Fabian Vogelsteller, author of ERC20 was also the main developer in Mist). Mist was also the first wallet to introduce the camelCase checksum (EIP-155, see [eip-155]). Mist runs a full node, and offers a full DApp browser with support for Swarm based storage and ENS addresses. Find it at:

https://github.com/ethereum/mist

References

- [[[1]]] EIP-161: http://eips.ethereum.org/EIPS/eip-161

Keys, Addresses

One of Ethereum's foundational technologies is *cryptography*, which is a branch of mathematics used extensively in computer security. Cryptography means "secret writing" in Greek, but the science of cryptography encompasses more than just secret writing, which is referred to as encryption. Cryptography can also be used to prove knowledge of a secret without revealing that secret (digital signature), or prove the authenticity of data (digital fingerprint). These types of cryptographic proofs are the mathematical tools critical to Ethereum and most blockchain systems, being extensively used in Ethereum applications. Ironically, encryption is not an important part of Ethereum, as its communications and transaction data are not encrypted and do not need to be encrypted to secure the system. In this chapter we will introduce some of the cryptography used in Ethereum to control ownership of funds, in the form of keys and addresses.

Introduction

Ethereum has two different types of accounts, which can own and control ether: *Externally Owned Accounts* (EOA) and *Contracts*. In this section we will examine the use of cryptography to establish ownership of ether by externally owned accounts, i.e. private keys.

Ownership of ether in EOAs is established through *digital keys*, *Ethereum addresses*, and *digital signatures*. The digital keys are not actually stored on the blockchain or transmitted on the Ethereum network, but are instead created and stored by users in a file, or simple database, called

a *wallet*. The digital keys in a user's wallet are completely independent of the Ethereum protocol and can be generated and managed by the user's wallet software without reference to the blockchain or access to the internet. Digital keys enable many of the interesting properties of Ethereum, including decentralized trust and control, and ownership attestation.

Ethereum transactions require a valid digital signature to be included in the blockchain, which can only be generated with a secret key; therefore, anyone with a copy of that key has control of the ether. The digital signature in an Ethereum transaction proves the true owner of the funds.

Digital keys come in pairs consisting of a private (secret) key and a public key. Think of the public key as similar to a bank account number and the private key as similar to the secret PIN, that provides control over the account. These digital keys are very rarely seen by the users of Ethereum. For the most part, they are stored inside the wallet file and managed by Ethereum wallet software.

In the payment portion of an Ethereum transaction, the intended recipient is represented by an *Ethereum address*, which is used in the same way as the beneficiary name on a check (i.e., "Pay to the order of"). In most cases, an Ethereum address is generated from and corresponds to a public key. However, not all Ethereum addresses represent public keys; they can also represent contracts, as we will see in [contracts]. The Ethereum address is the only representation of the keys that users will routinely see, because this is the part they need to share with the world.

First, we will introduce cryptography and explain the mathematics used in Ethereum. Next, we will look at how keys are generated, stored, and managed. Finally, we will review the various encoding formats used to represent private and public keys, and addresses.

Public key cryptography and cryptocurrency

Public key cryptography is a core concept of modern day information security. First publicly invented in the 1970s by Martin Hellman, Whitfield Diffie and Ralph Merkle, it was a monumental breakthrough which incited the first big wave of public interest into the field of cryptography. Before the 70s, strong cryptographic knowledge was held under governmental control with little public research untill the open publication of public key cryptography research.

Public key cryptography uses unique keys that are used to secure information. These unique keys are based on mathematical functions that have a unique property: they are easy to calculate in one direction, but very difficult to calculate in the inverse direction. Based on these mathematical functions, cryptography enables the creation of digital secrets and unforgeable digital signatures which are secured by the laws of mathematics.

For example, multiplying two large prime numbers together is trivial. But given the product of two large primes, it is very difficult to find the prime factors (a problem called *prime factorization*). Let's say I present the number 6895601 and tell you it is the product of two primes. Finding those two primes is much harder than it was for me to multiply them to produce 6895601.

Some of these mathematical functions can be inverted easily if you know some secret information. In our example above, if I tell you that one of the prime factors is 1931, you can trivially find the other one with a simple division: 6895601 / 1931 = 3571. Such functions are called *trapdoor functions* because given one piece of secret information, you can take a shortcut that makes it trivial to reverse the function.

Another category of mathematical functions that is useful in cryptography is based on arithmetic operations on an elliptic curve. In elliptic curve arithmetic, multiplication modulo a prime is simple but division is impossible (a problem known as the *discrete logarithm problem*). Elliptic curve cryptography is used extensively in modern computer systems and is the basis of Ethereum's (and other cryptocurrencies') digital keys and digital signatures.

TIP

Read more about cryptography and the mathematical functions that are used in modern cryptography:

Cryptography: https://en.wikipedia.org/wiki/Cryptography

Trapdoor Function: https://en.wikipedia.org/wiki/Trapdoor_function

Prime Factorization: https://en.wikipedia.org/wiki/Integer_factorization

Discrete Logarithm: https://en.wikipedia.org/wiki/Discrete_logarithm

Elliptic Curve Cryptography: https://en.wikipedia.org/wiki/Elliptic_curve_cryptography

In Ethereum, we use public key cryptography to create a key pair that controls access to ether and allows us to authenticate to contracts. The key pair consists of a private key, a unique public key, and are considered a "pair" because the public key is derived from the private key. The public key is used to receive funds and the private key is used to create digital signatures to sign transactions to spend the funds. Digital signatures are also used to authenticate owners or users of contracts, as we will see in [contract_authentication].

There is a mathematical relationship between the public and the private key that allows the private key to be used to generate signatures on messages. This signature can be validated against the public key without revealing the private key.

When spending ether, the current owner presents her public key and a signature (different each time, but created from the same private key) in a transaction. Through the presentation of the public key and signature, everyone in the Ethereum system can independently verify and accept the transaction as valid, confirming that the person transferring the ether owned them at the time of the transfer.

TIP

In most wallet implementations, the private and public keys are stored together as a *key pair* for convenience. However, the public key can be trivially calculated from the private key, so storing only the private key is also possible.

Why Use Asymmetric Cryptography (Public/Private Keys)?

Why is asymmetric cryptography used in Ethereum? It's not used to "encrypt" (make secret) the transactions. Rather, the useful property of asymmetric cryptography is the ability to generate *digital signatures*. A private key can be applied to the digital fingerprint of a transaction to produce a numerical signature. This signature can only be produced by someone with knowledge of the private key. However, anyone with access to the public key and the transaction fingerprint can use them to *verify* the signature. This useful property of asymmetric cryptography makes it possible for anyone to verify every signature on every transaction, while ensuring that only the owners of private keys can produce valid signatures.

Private keys

A private key is simply a number, picked at random. Ownership and control over the private key is the root of user control over all funds associated with the corresponding Ethereum address, as well as access to contracts that authorize that address. The private key is used to create signatures that are required to spend ether by proving ownership of funds used in a transaction. The private key must remain secret at all times, because revealing it to third parties is equivalent to giving them control over the ether and contracts secured by that key. The private key must also be backed up and protected from accidental loss. If it's lost, it cannot be recovered and the funds secured by it are lost forever too.

TIP The Ethereum private key is just a number. You can pick your private keys randomly using just a coin, pencil, and paper: toss a coin 256 times and you have the binary digits of a random private key you can use in an Ethereum wallet. The public key and address can then be generated from the private key.

Generating a private key from a random number

The first and most important step in generating keys is to find a secure source of entropy, or randomness. Creating an Ethereum private key is essentially the same as "Pick a number between 1 and 2^{256}." The exact method you use to pick that number does not matter as long as it is not predictable or repeatable. Ethereum software uses the underlying operating system's random number generator to produce 256 bits of entropy (randomness). Usually, the OS random number generator is initialized by a human source of randomness, which is why you may be asked to wiggle your mouse around for a few seconds, or press random keys on your keyboard.

More precisely, the range of possible private keys is slightly less than 2^{256}. The private key can be any number between 1 and n - 1, where n is a constant (n = 1.158 * 10^{77}, slightly less than 2^{256}) defined as the order of the elliptic curve used in Ethereum (see Elliptic curve cryptography explained). To create such a key, we randomly pick a 256-bit number and check that it is less than n - 1. In programming terms, this is usually achieved by feeding a larger string of random bits, collected from a cryptographically secure source of randomness, into a 256-bit hash algorithm such as Keccak-256 or SHA256 (see [cryptographic_hash_algorithm]), which will conveniently produce a 256-bit number. If the result is less than n - 1, we have a suitable private key. Otherwise, we simply

try again with another random number.

WARNING	Do not write your own code to create a random number or use a "simple" random number generator offered by your programming language. Use a cryptographically secure pseudo-random number generator (CSPRNG) with a seed from a source of sufficient entropy. Study the documentation of the random number generator library you choose to make sure it is cryptographically secure. Correct implementation of the CSPRNG is critical to the security of the keys.

The following is a randomly generated private key (k) shown in hexadecimal format (256 bits shown as 64 hexadecimal digits, each 4 bits):

```
f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

TIP	The size of Ethereum's private key space, (2^{256}) is an unfathomably large number. It is approximately 10^{77} in decimal. For comparison, the visible universe is estimated to contain 10^{80} atoms.

Public keys

An Ethereum public key is a *point* on an elliptic curve, meaning it is a set of X and Y coordinates that satisfy the elliptic curve equation.

In simpler terms, an Ethereum public key is two numbers, joined together. These numbers are produced from the private key by a calculation that can *only go one way*. That means that it is trivial to calculate a public key if you have the private key. But you cannot calculate the private key from the public key.

> MATH is about to happen! Don't panic. If you find it hard to read the previous paragraph, you can skip the next few sections. There are many tools and libraries that will do the math for you.

The public key is calculated from the private key using elliptic curve multiplication, which is irreversible: $K = k * G$, where k is the private key, G is a constant point called the *generator point*, and K is the resulting public key. The reverse operation, known as "finding the discrete logarithm"—calculating k if you know K—is as difficult as trying all possible values of k, i.e., a brute-force search.

In simpler terms: arithmetic on the elliptic curve is different from "regular" integer arithmetic. A point (G) can be multiplied by an integer (k) to produce another point (K). But there is no such thing as *division*, so it is not possible to simply "divide" the public key K by the point G to calculate the private key k. This is the one-way mathematical function described in Public key cryptography and cryptocurrency.

TIP

Elliptic curve multiplication is a type of function that cryptographers call a "one way" function: it is easy to do in one direction (multiplication) and impossible to do in the reverse direction (division). The owner of the private key can easily create the public key and then share it with the world knowing that no one can reverse the function and calculate the private key from the public key. This mathematical trick becomes the basis for unforgeable and secure digital signatures that prove ownership of Ethereum funds and control of contracts.

Before we demonstrate how to generate a public key from a private key, let's look at elliptic curve cryptography in a bit more detail.

Elliptic curve cryptography explained

Elliptic curve cryptography is a type of asymmetric or public key cryptography based on the discrete logarithm problem as expressed by addition and multiplication on the points of an elliptic curve.

A visualization of an elliptic curve is an example of an elliptic curve, similar to that used by Ethereum.

TIP

Ethereum uses the exact same elliptic curve, called secp256k1, as Bitcoin. That makes it possible to re-use many of the elliptic curve libraries and tools from Bitcoin.

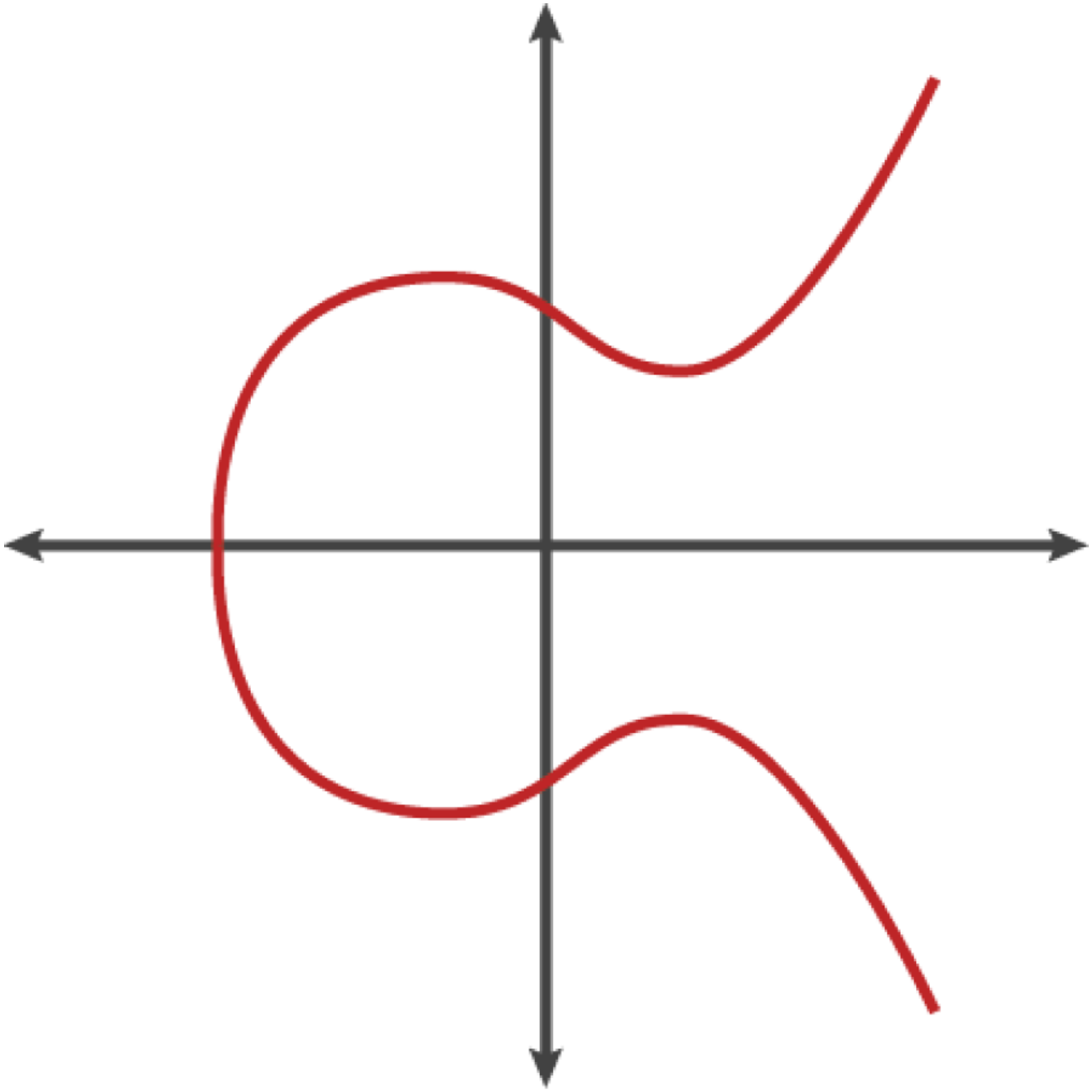

Figure 27. A visualization of an elliptic curve

Ethereum uses a specific elliptic curve and set of mathematical constants, as defined in a standard called secp256k1, established by the National Institute of Standards and Technology (NIST). The secp256k1 curve is defined by the following function, which produces an elliptic curve:

or

The *mod p* (modulo prime number p) indicates that this curve is over a finite field of prime order *p*, also written as \(\mathbb{F}_p \), where p = $2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, a very large prime number.

Because this curve is defined over a finite field of prime order instead of over the real numbers, it looks like a pattern of dots scattered in two dimensions, which makes it difficult to visualize. However, the math is identical to that of an elliptic curve over real numbers. As an example, Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17 shows the same elliptic curve over a much smaller finite field of prime order 17, showing a pattern of dots on a grid. The

secp256k1 Ethereum elliptic curve can be thought of as a much more complex pattern of dots on an unfathomably large grid.

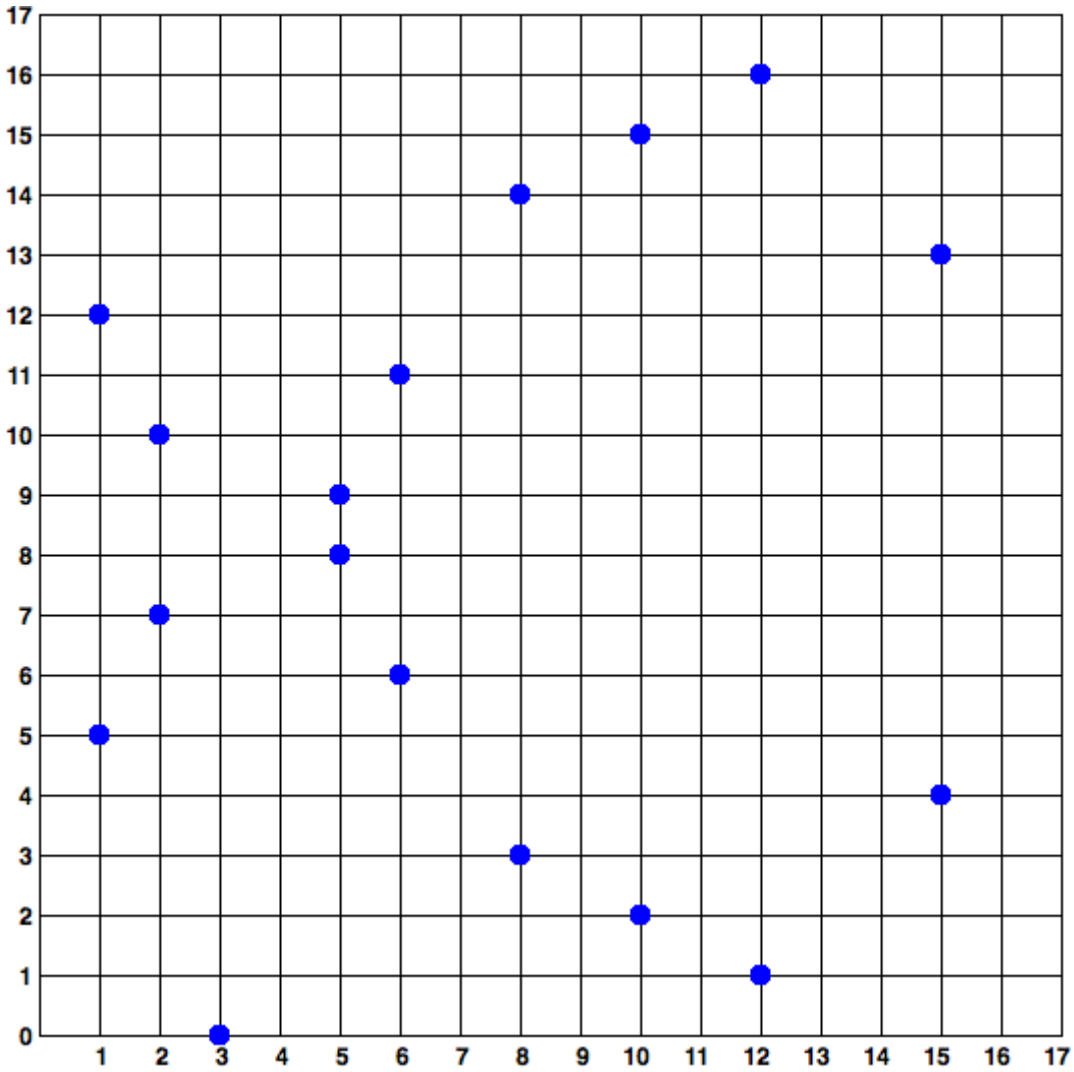

Figure 28. Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17

So, for example, the following is a point Q with coordinates (x,y) that is a point on the secp256k1 curve:

```
Q = (49790390825249384486033144355916864607616083520101638681403973749255924539515,
595741321618999000458620864939210157800321752917558073992840077210503412973 60)
```

Using Python to confirm that this point is on the elliptic curve shows how you can check this yourself using Python. The variables x and y are the coordinates of the point Q as above. The variable p is the prime order of the elliptic curve (the prime that is used for all the modulo operations). The last line of Python is the elliptic curve equation (the % operator in Python is the modulo operator). If x and y are indeed points on the elliptic curve, then they satisfy the equation and the result is zero (0L is a long integer with value zero). Try it yourself, by typing python on a command line and copying each line (after the prompt >>>) from the listing:

Example 1. Using Python to confirm that this point is on the elliptic curve

```
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x =
4979039082524938448603314435591686460761608352010163868140397374925592453951 5
>>> y =
595741321618999000458620864939210157800321752917558073992840077210503412973 60
>>> (x ** 3 + 7 - y**2) % p
0L
```

Elliptic curve arithmetic operations

A lot of elliptic curve math looks and works very much like the integer arithmetic we learned at school. Specifically, we can define an addition operator, which instead of adding numbers is adding points on the curve. Once we have the addition operator, we can also define multiplication of a point and a whole number, such that it is equivalent to repeated addition.

Addition is defined such that given two points P_1 and P_2 on the elliptic curve, there is a third point $P_3 = P_1 + P_2$, also on the elliptic curve.

Geometrically, this third point P_3 is calculated by drawing a line between P_1 and P_2. This line will intersect the elliptic curve in exactly one additional place. Call this point $P_3' = (x, y)$. Then reflect in the x-axis to get $P_3 = (x, –y)$.

In elliptic curve math, there is a point called the "point at infinity," which roughly corresponds to the role of the number zero in addition. On computers, it's sometimes represented by x = y = 0 (which doesn't satisfy the elliptic curve equation, but it's an easy separate case that can be checked). There are a couple of special cases that explain the need for the "point at infinity."

If P_1 and P_2 are the same point, the line "between" P_1 and P_2 should extend to be the tangent on the curve at this point P_1. This tangent will intersect the curve in exactly one new point. You can use techniques from calculus to determine the slope of the tangent line. These techniques curiously work, even though we are restricting our interest to points on the curve with two integer coordinates!

In some cases (i.e., if P_1 and P_2 have the same x values but different y values), the tangent line will be exactly vertical, in which case P3 = "point at infinity."

If P_1 is the "point at infinity," then $P_1 + P_2 = P_2$. Similarly, if P_2 is the point at infinity, then $P_1 + P_2 = P_1$. This shows how the point at infinity plays the role that zero plays in "normal" arithmetic.

It turns out that + is associative, which means that (A + B) + C = A + (B + C). That means we can write A + B + C without parentheses and without ambiguity.

Now that we have defined addition, we can define multiplication in the standard way that extends addition. For a point P on the elliptic curve, if k is a whole number, then k * P = P + P + P + ... + P (k times). Note that k is sometimes confusingly called an "exponent" in this case

Generating a public key

Starting with a private key in the form of a randomly generated number *k*, we multiply it by a predetermined point on the curve called the *generator point G* to produce another point somewhere else on the curve, which is the corresponding public key *K*. The generator point is specified as part of the secp256k1 standard and is always the same for all implementations of secp256k1 and all keys derived from that curve use the same point *G*:

where *k* is the private key, *G* is the generator point, and *K* is the resulting public key, a point on the curve. Because the generator point is always the same for all Ethereum users, a private key *k* multiplied with *G* will always result in the same public key *K*. The relationship between *k* and *K* is fixed, but can only be calculated in one direction, from *k* to *K*. That's why an Ethereum address (derived from *K*) can be shared with anyone and does not reveal the user's private key (*k*).

As we described in Elliptic curve arithmetic operations, the multiplication of k * G is equivalent to repeated addition, so G + G + G + ... + G, repeated k times. In summary, to produce a public key *K*, from a private key *k*, we add the generator point *G* to itself, *k* times.

TIP	A private key can be converted into a public key, but a public key cannot be converted back into a private key because the math only works one way.

Let's apply this calculation to find the public key for the specific private key we showed you in Private keys:

Example private key to public key calculation

```
K = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 * G
```

A cryptographic library can help us calculate K, using elliptic curve multiplication. The resulting public key *K* is defined as a point K = (x,y):

Example public key calculated from the example private key

```
K = (x, y)

where,

x = 6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b
y = 83b5c38e5e2b0c8529d7fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

In Ethereum you may see public keys represented as a hexadecimal serialization of 66 hexadecimal characters (33 bytes). This is adopted from a standard serialization format proposed by the industry consortium Standards for Efficient Cryptography Group (SECG), documented in Standards for Efficient Cryptography (SEC1). The standard defines four possible prefixes that can be used to

identify points on an elliptic curve:

Prefix	Meaning	Length (bytes counting prefix)	
0x00	Point at Infinity	1	
0x04	Uncompressed Point	65	
0x02	Compressed Point with even Y	33	
0x03	Compressed Point with odd Y	33	

Ethereum only uses uncompressed public keys, therefore the only prefix that is relevant is (hex) 04. The serialization concatenated the X and Y coordinates of the public key:

```
04 + X-coordinate (32 bytes/64 hex) + Y coordinate (32 bytes/64 hex)
```

Therefore, the public key we calculated in Example public key calculated from the example private key is serialized as:

```
046e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0c8529d7
fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

Elliptic curve libraries

There are a couple of implementations of the secp256k1 elliptic curve that are used in cryptocurrency related projects:

OpenSSL

The OpenSSL library offers a comprehensive set of cryptographic primitives, including a full implementation of the secp256k1. For example, to derive the public key, the function EC_POINT_mul() can be used. Find it at https://www.openssl.org/

libsecp256k1

Bitcoin Core's libsecp256k1, is a C-language implementation of the secp256k1 elliptic curve and other cryptographic primitives. The libsecp256 of elliptic curve cryptography was written from scratch to replace OpenSSL in Bitcoin Core software, and is considered superior in both performance and security. Find it at: https://github.com/bitcoin-core/secp256k1

Cryptographic hash functions

Cryptographic hash functions are used throughout Ethereum. In fact, hash functions are used extensively in almost all cryptographic systems, a fact captured by cryptographer Bruce Schneier who said "Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography."

In this section we will discuss hash functions, understand their basic properties and how those properties make them so useful in so many areas of modern cryptography. We address hash functions here, because they are part of the transformation of Ethereum public keys into addresses.

In simple terms, "a hash function is any function that can be used to map data of arbitrary size to data of fixed size." Source: Wikipedia. The input to a hash function is called a *pre-image* or *message*. The output is called a *hash*, or *digest*. A special sub-category of hash functions is *cryptographic hash functions*, which have specific properties that are useful to cryptography.

A cryptographic hash function is a *one way* hash function that maps data of arbitrary size to a fixed-size bit string, where it is computationally infeasible to recreate the input if one knows the output. The only way to determine the input is to conduct a brute-force search of possible inputs, checking for a matching output.

Cryptographic hash functions have five main properties (Source: Wikipedia/Cryptographic Hash Function):

Determinism

Any input message always produces the same hash digest.

Verifiability

Computing the hash of a message is efficient (linear performance).

Uncorrelated

A small change to the message (e.g. one bit change) should change the hash output so extensively that it cannot be correlated to the hash of the original message.

Irreversibility (resistance to first pre-image)

Computing the message from a hash is infeasible, equivalent to a brute force search through possible messages.

Collision Protection (resistance to second pre-image)

It should be infeasible to calculate two different messages that produce the same hash output.

Resistance to second pre-image is primarily important to prevent digital signature forgery in Ethereum.

The combination of these properties make cryptographic hash functions useful for a broad range of security applications including:

- Data fingerprinting
- Message integrity (error detection)
- Proof-of-Work
- Authentication (password hashing and key stretching)
- Pseudo-random number generators
- Pre-image commitment
- Unique identifiers

We will find many of these in Ethereum, as we progress through the various layers of the system.

Ethereum's cryptographic hash function - Keccak-256

Ethereum uses the *Keccak-256* cryptographic hash function in many places. Keccak-256 was designed as a candidate for the SHA-3 Cryptographic Hash Function Competition held in 2007 by the National Institute of Science and Technology (NIST). Keccak was the winning algorithm that became standardized as Federal Information Processing Standard (FIPS) 202 in 2015.

However, during the period when Ethereum was developed, NIST standardization was being finalized. NIST adjusted some of the parameters of Keccak after the completion of the standards process, allegedly to improve its efficiency. This was occurring at the same time as heroic whistleblower Edward Snowden revealed documents that imply that NIST may have been improperly influenced by the National Security Agency to intentionally weaken the Dual_EC_DRBG random-number generator standard, effectively placing a backdoor in the standard random number generator. The result of this controversy was a backlash against the proposed changes and a significant delay in the standardization of SHA-3. At the time, the Ethereum Foundation decided to implement the original Keccak algorithm, as proposed by its inventors, rather than the SHA-3 standard as modified by NIST.

WARNING	While you may see "SHA3" mentioned throughout Ethereum documents and code, many if not all of those instances actually refer to Keccak-256, not the finalized FIPS-202 SHA-3 standard. The implementation differences are slight, having to do with padding parameters, but they are significant in that Keccak-256 produces different hash output than FIPS-202 SHA-3 given the same input.

Due to the confusion created by the difference between the hash function used in Ethereum (Keccak-256) and the finalized standard (FIP-202 SHA-3), there is an effort underway to rename all instances of sha3 in all code, opcodes and libraries to keccak256. See ERC-59 for details.

Which hash function am I using?

How can you tell if the software library you are using is FIPS-202 SHA-3 or Keccak-256, if both might be called "SHA3"?

An easy way to tell is to use a *test vector*, an expected output for a given input. The test most commonly used for a hash function is the *empty input*. If you run the hash function with an empty string as input you should see the following results:

Testing whether the SHA3 library you are using is Keccak-256 of FIP-202 SHA-3

```
Keccak256("") =
c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470

SHA3("") =
a7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
```

So, regardless of what the function is called, you can test it to see whether it is the original Keccak-

256, or the final NIST standard FIPS-202 SHA-3, by running the simple test above. Remember, Ethereum uses Keccak-256, even though it is often called SHA-3 in the code.

Next, let's examine the first application of Keccak-256 in Ethereum, which is to produce Ethereum addresses from public keys.

Ethereum addresses

Ethereum addresses are *unique identifiers* that are derived from public keys or contracts using a one-way hash function (specifically Keccak-256).

In our previous examples, we started with a private key and used elliptic curve multiplication to derive a public key:

Private Key *k*:

```
k = f8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315
```

Public Key *K* (X and Y coordinates concatenated and shown as hex):

```
K =
6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0c8529d7fa
3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

WARNING	It is worth noting that the public key is not formatted with the prefix (hex) 04 when the address is calculated.

We use Keccak-256 to calculate the *hash* of this public key:

```
Keccak256(K) = 2a5bc342ed616b5ba5732269001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Then we keep only the last 20 bytes (the least significant bytes in big-endian), which is our Ethereum address:

```
001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Most often you will see Ethereum addresses with the prefix "0x" that indicates it is a hexadecimal encoding, like this:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

Ethereum address formats

Ethereum addresses are hexadecimal numbers, identifiers derived from the last 20 bytes of the Keccak-256 hash of the public key.

Unlike Bitcoin addresses which are encoded in the user interface of all clients to include a built-in checksum to protect against mistyped addresses, Ethereum addresses are presented as raw hexadecimal without any checksum.

The rationale behind that decision was that Ethereum addresses would eventually be hidden behind abstractions (such as name services) at higher layers of the system and that checksums should be added at higher layers if necessary.

In retrospect, this design choice lead to a number of problems, including the loss of funds due to mistyped addresses and input validation errors. Ethereum name services were developed slower than initially expected and alternative encodings such as ICAP were adopted very slowly by wallet developers.

Inter Exchange Client Address Protocol (ICAP)

The *Inter exchange Client Address Protocol (ICAP)* is an Ethereum Address encoding that is partly compatible with the International Bank Account Number (IBAN) encoding, offering a versatile, checksummed and interoperable encoding for Ethereum Addresses. ICAP addresses can encode Ethereum Addresses or common names registered with an Ethereum name registry.

Read about ICAP on the Ethereum Wiki:https://github.com/ethereum/wiki/wiki/ICAP:-Inter-exchange-Client-Address-Protocol

IBAN is an international standard for identifying bank account numbers, mostly used for wire transfers. It is broadly adopted in the European Single Euro Payments Area (SEPA) and beyond. IBAN is a centralized and heavily regulated service. ICAP is a decentralized but compatible implementation for Ethereum addresses.

An IBAN consists of up to 34 alphanumeric characters (case-insensitive) string containing a country code, checksum, and bank account identifier (which is country-specific).

ICAP uses the same structure by introducing a non-standard country code "XE" that stands for "Ethereum", followed by a two-character checksum and 3 possible variations of an account identifier:

Direct

Up to 30 alphanumeric character big-endian base-36 integer representing the least significant bits of an Ethereum address. Because this encoding fits less than 155 bits, it only works for Ethereum addresses that start with one or more zero bytes. The advantage is that it is compatible with IBAN, in terms of the field length and checksum. Example: XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD (33 characters long)

Basic

Same as the "Direct" encoding except that it is 31 characters long. This allows it to encode any

Ethereum address, but makes it incompatible with IBAN field validation. Example: XE18CHDJBPLTBCJ03FE9O2NS0BPOJVQCU2P (35 characters long)

Indirect

Encodes an identifier that resolves to an Ethereum address through a name registry provider. Uses 16 alphanumeric characters, composed of an *asset identifier* (e.g. ETH), a name service (e.g. XREG) and a 9-character name (e.g. KITTYCATS), which is a human-readable name. Example: XE##ETHXREGKITTYCATS (20 characters long), where the "##" should be replaced by the two computed checksum characters.

We can use the helpeth command-line tool to create ICAP addresses. Let's try with our example private key (prefixed with 0x and passed as a parameter to helpeth):

```
$ helpeth keyDetails -p
0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315

Address: 0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
ICAP: XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D
Public key:
0x6e145ccef1033dea239875dd00dfb4fee6e3348b84985c92f103444683bae07b83b5c38e5e2b0c8529d7
fa3f64d46daa1ece2d9ac14cab9477d042c84c32ccd0
```

The helpeth command constructs a hexadecimal Ethereum address as well as an ICAP address for us. The ICAP address for our example key is:

```
XE60HAMICDXSV5QXVJA7TJW47Q9CHWKJD
```

Because our example Ethereum address happens to start with a zero byte, it can be encoded using the "Direct" ICAP encoding method that is valid in an IBAN format. You can tell because it is 33 characters long.

If our address did not start with a zero, it would be encoded with the "Basic" encoding, which would be 35 characters long and invalid as an IBAN format.

TIP

The chances of any Ethereum address starting with a zero byte are 1 in 256. To generate one like that, it will take on average 256 attempts with 256 different random private keys before we find one that works as an IBAN-compatible "Direct" encoded ICAP address.

At this time, ICAP is unfortunately only supported by a few wallets.

Hex encoding with checksum in capitalization (EIP-55)

Due to the slow deployment of ICAP or name services, a new standard was proposed with Ethereum Improvement Proposal 55 (EIP-55). You can read the details at:

https://github.com/Ethereum/EIPs/blob/master/EIPS/eip-55.md

EIP-55 offers a backward compatible checksum for Ethereum addresses by modifying the capitalization of the hexadecimal address. The idea is that Ethereum addresses are case-insensitive and all wallets are supposed to accept Ethereum addresses expressed in capital or lower-case characters, without any difference in interpretation.

By modifying the capitalization of the alphabetic characters in the address, we can convey a checksum that can be used to protect the integrity of the address against typing or reading mistakes. Wallets that do not support EIP-55 checksums simply ignore the fact that the address contains mixed capitalization. But those that do support it, can validate it and detect errors with a 99.986% accuracy.

The mixed-capitals encoding is subtle and you may not notice it at first. Our example address is:

```
0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9
```

with an EIP-55 mixed-capitalization checksum it becomes:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Can you tell the difference? Some of the alphabetic (A-F) characters from the hexadecimal encoding alphabet are now capital, while others are lower case. You might not even have noticed the difference unless you looked carefully.

EIP-55 is quite simple to implement. We take the Keccak-256 hash of the lower-case hexadecimal address. This hash acts as a digital fingerprint of the address, giving us a convenient checksum. Any small change in the input (the address) should cause a big change in the resulting hash (the checksum), allowing us to detect errors effectively. The hash of our address is then encoded in the capitalization of the address itself. Let's break it down, step-by-step:

1. Hash the lower-case address, without the 0x prefix:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0f9")
23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9695d9a19d8f673ca991deae1
```

2. Capitalize each alphabetic address character if the corresponding hex digit of the hash is greater than or equal to 0x8. This is easier to show if we line up the address and the hash:

```
Address: 001d3f1ef827552ae1114027bd3ecf1f086ba0f9
Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Our address contains an alphabetic character d in the fourth position. The fourth character of the hash is 6, which is less than 8. So, we leave the d lower-case. The next alphabetic character in our address is f, in the sixth position. The sixth character of the hexadecimal hash is c, which is greater than 8. Therefore, we capitalize the F in the address, and so on. As you can see, we only use the first 20-bytes (40 hex characters) of the hash as a checksum, since we only have 20-bytes (40 hex

characters) in the address to capitalize appropriately.

Check the resulting mixed-capitals address yourself and see if you can tell which characters were capitalized and which characters they correspond to in the address hash:

```
Address: 001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
Hash   : 23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
```

Detecting an error in an EIP-55 encoded address

Now, let's look at how EIP-55 addresses will help us find an error. Let's assume we have printed out an Ethereum address, which is EIP-55 encoded:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

Now let's make a basic mistake in reading that address. The character before the last one is a capital "F". For this example let's assume we misread that as a capital "E". We type in the (incorrect address) into our wallet:

```
0x001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
```

Fortunately, our wallet is EIP-55 compliant! It notices the mixed capitalization and attempts to validate the address. It converts it to lower case, and calculates the checksum hash:

```
Keccak256("001d3f1ef827552ae1114027bd3ecf1f086ba0e9")
5429b5d9460122fb4b11af9cb88b7bb76d8928862e0a57d46dd18dd8e08a6927
```

As you can see, even though the address has only changed by one character (in fact, only one bit as "e" and "f" are 1-bit apart), the hash of the address has changed radically. That's the property of hash functions that makes them so useful for checksums!

Now, let's line up the two and check the capitalization:

```
001d3F1ef827552Ae1114027BD3ECF1f086bA0E9
5429b5d9460122fb4b11af9cb88b7bb76d892886...
```

It's all wrong! Several of the alphabetic characters are incorrectly capitalized. Remember that the capitalization is the encoding of the *correct* checksum.

The capitalization of the address we input doesn't match the checksum just calculated, meaning something has changed in the address, and an error has been introduced.

Transactions

Transactions are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded (mined) on the Ethereum blockchain. Behind that basic definition, there are a lot of surprising and fascinating details. Another way to look at transactions is that they are the only thing that can trigger a change of state or cause a contract to execute in the EVM. Ethereum is a global singleton state machine, and transactions are the only thing that can make that state machine "tick", changing its state. Contracts don't run on their own. Ethereum doesn't run "in the background". Everything starts with a transaction.

In this section, we will dissect transactions, show how they work, and understand the details.

Structure of Transaction

First let's take a look at the basic structure of a transaction, as it is serialized and transmitted on the Ethereum network. Each client and application that receives a serialized transaction will store it in-memory using its own internal data structure, perhaps embellished with metadata that doesn't exist in the network serialized transaction itself. The network serialization of a transaction is, therefore, the only common standard of a transaction's structure.

A transaction is a serialized binary message that contains the following data:

nonce
: A sequence number, issued by the originating EOA, used to prevent message replay.

gas price
: The price of gas (in wei) the originator is willing to pay.

start gas
: The maximum amount of gas the originator is willing to pay.

to
: Destination Ethereum address.

value
: Amount of ether to send to the destination.

data
: Variable length binary data payload.

v,r,s
: The three components of an ECDSA signature of the originating EOA.

The transaction message's structure is serialized using the Recursive Length Prefix (RLP) encoding scheme (see [rlp]), which was created specifically for accurate and byte-perfect data serialization in Ethereum. All numbers in Ethereum are encoded as big-endian integers, of lengths that are multiples of 8 bits.

Note that the field labels ("to", "start gas", etc.) are shown here for clarity, but are not part of the transaction serialized data, which contains the field values RLP-encoded. In general, RLP does not contain any field delimiters or labels. RLP's length prefix is used to identify the length of each field. Anything beyond the defined length, therefore, belongs to the next field in the structure.

While this is the actual transaction structure transmitted, most internal representations and user interface visualizations embellish this with additional information, derived from the transaction or from the blockchain.

For example, you may notice there is no "from" data in the address identifying the originator EOA. That EOA's public key can easily be derived from the v,r,s components of the ECDSA signature. The address can, in turn, be easily derived from the public key. When you see a transaction showing a "from" field, that was added by the software used to visualize the transaction. Other metadata frequently added to the transaction by client software include the block number (once it is mined) and a transaction ID (calculated hash). Again, this data is derived from the transaction and not part of the transaction message itself.

The transaction nonce

The nonce is one of the most important and least understood components of a transaction. The definition in the Yellow Paper (see [yellow_paper]) reads:

```
nonce: A scalar value equal to the number of transactions sent from this address or,
in the case of accounts with associated code, the number of contract-creations made by
this account.
```

Strictly speaking, the nonce is an attribute of the originating address (it only has meaning in the context of the sending address). However, the nonce is not stored explicitly as part of an account's state on the blockchain. Instead it is calculated dynamically, by counting the number of confirmed transactions that have originated from an address.

The nonce value is also used to prevent erroneous calculations of an account balance. For example, let's say an account has a balance of 10 ether and signs two transactions spending 6 ether each, with nonce 1 and nonce 2 respectively. Which of these two transaction is valid? In a distributed system like Ethereum, nodes may receive the transactions out of sequence. The nonce forces transactions from any address to be processed sequentially, without gaps, regardless of the sequence in which they were received by a node. That way, all nodes calculate the same balance. The transaction paying 6 ether with nonce 1 will be processed successfully, reducing the balance of the account to 4 ether. The transaction paying 6 ether with the nonce 2, will be seen as invalid by all nodes, regardless of when it was received. If the nonce 2 transaction is received first, a node will hold it and not validate it until it has seen, and processed, the nonce 1 transaction.

The use of the nonce to ensure that all nodes calculate the same balance, and sequence transactions properly, is equivalent to the mechanism used to prevent "double spending" in Bitcoin. However, because Ethereum tracks account balances and does not track coins individually (called UTXO in Bitcoin), "double spending" can only happen if the account balance is calculated erroneously. The nonce mechanism prevents that from happening.

Keeping track of nonces

In practical terms, the nonce is an up-to-date count of the number of *confirmed* (mined) transactions that have originated from an account. To find out what the nonce is, you can interrogate the blockchain, for example via the web3 interface:

Retrieving the transaction count of our example address

```
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f")
40
```

TIP The nonce is a zero-based counter, meaning the first transaction has nonce 0. In Retrieving the transaction count of our example address, we have a transaction count of 40, meaning nonces 0 through 39 have been seen. The next transaction's nonce will be 40.

Your wallet will keep track of nonces for each address it manages. It's fairly simple to do that, as long as you are only originating transactions from a single point. Let's say you are writing your own wallet software or some other application that originates transactions. How do you track nonces?

When you create a new transaction, you assign the next nonce in the sequence. But until it is confirmed, it will not count towards the getTransactionCount total.

Unfortunately, the getTransactionCount function will run into some problems if we send a few transactions in a row. There is a known bug where getTransactionCount does not count pending transactions correctly. Let's look at an example:

```
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", "pending")
40
web3.eth.sendTransaction({from: web3.eth.accounts[0], to:
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01, "ether")});
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", "pending")
41
web3.eth.sendTransaction({from: web3.eth.accounts[0], to:
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01, "ether")});
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", "pending")
41
web3.eth.sendTransaction({from: web3.eth.accounts[0], to:
"0xB0920c523d582040f2BCB1bD7FB1c7C1ECEbdB34", value: web3.toWei(0.01, "ether")});
web3.eth.getTransactionCount("0x9e713963a92c02317a681b9bb3065a8249de124f", "pending")
41
```

As you can see, the first transaction we sent increased the transaction count to 41, showing the pending transaction. But when we sent 3 more transactions in quick succession, the getTransactionCount call didn't count them correctly. It only counted one, even though there are 3 pending in the mempool. If we wait a few seconds, once a block is mined, the getTransactionCount call will return the correct number. But in the interim, while there are more than one transactions pending, it does not help us.

When you build an application that constructs transactions, it cannot rely on getTransactionCount for pending transactions. Only when pending and confirmed are equal (all outstanding transactions are confirmed) can you trust the output of getTransactionCount to start your nonce counter. Thereafter, keep track of the nonce in your application until each transaction confirms.

Parity's JSON RPC interface offers the parity_nextNonce function, that returns the next nonce that should be used in a transaction. The parity_nextNonce function counts nonces correctly, even if you construct several transactions in rapid succession, without confirming them.

Parity has a web console for accessing the JSON RPC interface, but here we are using a command line HTTP client to access it:

```
curl --data
'{"method":"parity_nextNonce","params":["0x9e713963a92c02317a681b9bb3065a8249de124f"],
"id":1,"jsonrpc":"2.0"}' -H "Content-Type: application/json" -X POST localhost:8545

{"jsonrpc":"2.0","result":"0x32","id":1}
```

Gaps in nonces, duplicate nonces, and confirmation

It is important to keep track of nonces if you are creating transactions programmatically, especially if you are doing so from multiple independent processes simultaneously.

The Ethereum network processes transactions sequentially, based on the nonce. That means that if you transmit a transaction with nonce 0 and then transmit a transaction with nonce 2, the second transaction will not be mined. It will be stored in the mempool, while the Ethereum network waits for the missing nonce to appear. All nodes will assume that the missing nonce has simply been delayed and that the transaction with nonce 2 was received out-of-sequence.

If you then transmit a transaction with the missing nonce 1, both transactions (nonces 1 and 2) will be mined. Once you fill the gap, the network can mine the out-of-sequence transaction that it held in the mempool.

What this means is that if you create several transactions in sequence and one of them does not get mined, all the subsequent transactions will be "stuck", waiting for the missing nonce. A transaction can create an inadvertent "gap" in the nonce sequence because it is invalid or has insufficient gas. To get things moving again, you have to transmit a valid transaction with the missing nonce.

If on the other hand you accidentally duplicate a nonce, for example by transmitting two transactions with the same nonce, but different recipients or values, then one of them will be confirmed and one will be rejected. Which one is confirmed will be determined by the sequence in which they arrive at the first validating node that receives them.

As you can see, keeping track of nonces is necessary and if your application doesn't manage that process correctly, you will run into problems. Unfortunately, things get even more difficult if you are trying to do this concurrently, as we will see in the next section.

Concurrency, transaction origination, and nonces

Concurrency is a complex aspect of computer science, and it crops up unexpectedly sometimes, especially in decentralized/distributed real-time systems like Ethereum.

In simple terms, concurrency is when you have simultaneous computation by multiple independent systems. These can be in the same program (e.g. threading), on the same CPU (e.g. multi-processing), or on different computers (i.e. distributed systems). Ethereum, by definition, is a system that allows concurrency of operations (nodes, clients, DApps), but enforces a singleton state (e.g. there is only one common/shared state of the system at each mined block).

Now, imagine that we have multiple independent wallet applications that are generating transactions from the same address or addresses. One example of such a situation would be an exchange processing withdrawals for a hot wallet. Ideally, you'd want to have more than one computer processing withdrawals, so that it doesn't become a bottleneck or single point of failure. However, this quickly becomes problematic, as having more than one computer producing withdrawals will result in some thorny concurrency problems, not least of which is the selection of nonces. How do multiple computers generating, signing and broadcasting transactions from the same hot wallet account coordinate?

You could use a single computer to assign nonces, on a first-come first-served basis to computers signing transactions. However, this computer is now a single-point of failure. Worse, if several nonces are assigned and one of them never gets used (because of a failure in the computer processing the transaction with that nonce), all of the subsequent ones get stuck.

You could generate the transactions, but don't sign them or assign a nonce to them. Then queue them to a single node that signs them and also keeps track of nonces. Again, you have a single point of failure. The signing and tracking of nonces is the part of your operation that is likely to become congested under load, whereas the generation of the unsigned transaction is the part you don't really need to parallelize. You have concurrency, but you don't have it in any useful part of the process.

In the end, these concurrency problems, on top of the difficulty of tracking account balances and transaction confirmation in independent processes, force most implementations towards avoiding concurrency and creating bottlenecks such as a single process handling all withdrawal transactions in an exchange.

Transaction gas

We discuss *gas* in detail in [gas]. However, let's cover some basics about the role of the gasPrice and startGas components of a transaction.

Gas is the fuel of Ethereum. Gas is not ether - it's a separate virtual currency with an exchange rate vis-a-vis ether. Ethereum uses gas to control the amount of resources that a transaction can spend, since it will be processed on thousands of computers around the world. The open-ended (Turing complete) computation model requires some form of metering in order to avoid denial of service attacks or inadvertent resource-devouring transactions.

Gas is separate from ether in order to protect the system from the volatility that might arise along

with rapid changes in the value of ether.

The gasPrice field in a transaction allows the transaction originator to set the exchange rate of each unit of gas. Gas price is measured in wei per gas unit. For example, in a transaction we recently created for an example in this book, our wallet had set the gasPrice to 3 Gwei (3 Giga-wei, 3 billion wei).

The popular site ethgasstation.info provides information on the current prices of gas, and other relevant gas metrics for the Ethereum main network:

https://ethgasstation.info/

Wallets can adjust the gasPrice in transactions they originate, to achieve faster confirmation (mining) of transactions. The higher the gasPrice, the faster the transaction is likely to confirm. Conversely, lower priority transactions can carry a reduced price they are willing to pay for gas, resulting in slower confirmation. The minimum gasPrice that can be set is zero, which means a fee-free transaction. During periods of low demand for space in a block, such transactions will get mined.

TIP The minimum acceptable gasPrice is zero. That means that wallets can generate completely free transactions. Depending on capacity, these may never be mined, but there is nothing in the protocol that prohibits free transactions. You can find several examples of such transactions successfully mined in the Ethereum blockchain.

The web3 interface offers a gasPrice suggestion, by calculating a median price across several blocks:

```
truffle(mainnet)> web3.eth.getGasPrice(console.log)
truffle(mainnet)> null BigNumber { s: 1, e: 10, c: [ 10000000000 ] }
```

The second important field related to gas, is startGas. This is explained in more detail in [gas]. In simple terms, startGas defines how many units of gas the transaction originator is willing to spend to complete the transaction. For simple payments, meaning transactions that transfer ether from one EOA to another EOA, the gas amount needed is fixed at 21,000 gas units. To calculate how much ether that will cost, you multiply 21,000 with the gasPrice you're willing to pay:

```
truffle(mainnet)> web3.eth.getGasPrice(function(err, res) {console.log(res*21000)} )
truffle(mainnet)> 210000000000000
```

If your transaction's destination address is a contract, then the amount of gas needed can be estimated but cannot be determined with accuracy. That's because a contract can evaluate different conditions that lead to different execution paths, with different gas costs. That means that the contract may execute only a simple computation or a more complex one depending on conditions that are outside of your control and cannot be predicted. To demonstrate this let's use a rather contrived example: each time a contract is called it increments a counter and on the 100th time (only) it computes something complex. If you call the contract 99 times one thing happens, but on the 100th something completely different happens. The amount of gas you would pay for that

depends on how many other transactions have called that function before your transaction is mined. Perhaps your estimate is based on being the 99th transaction and just before your transaction is mined, someone else calls the contract for the 99th time. Now you're the 100th transaction to call and the computation effort (and gas cost) is much higher.

To borrow a common analogy used in Ethereum, you can think of startGas as the fuel tank in your car (your car is the transaction). You fill the tank with as much gas as you think it will need for the journey (the computation needed to validate your transaction). You can estimate the amount to some degree, but there might be unexpected changes to your journey such as a diversion (a more complex execution path), which increase fuel consumption.

The analogy to a fuel tank is somewhat misleading, however. It's more like a credit account for a gas station company, where you pay after the trip is completed, based on how much gas you actually used. When you transmit your transaction, one of the first validation steps is to check that the account it originated from has enough ether to pay the gasPrice * startGas fee. But the amount is not actually deducted from your account until the end of the transaction execution. You are only billed for gas actually consumed by your transaction at the end, but you have to have enough balance for the maximum amount you are willing to pay before you send your transaction.

Transaction recipient

The recipient of a transaction is specified in the to field. This contains a 20-byte Ethereum address. The address can be an EOA or a contract address.

Ethereum does no further validation of this field. Any 20-byte value is considered valid. If the 20-byte value corresponds to an address without a corresponding private key, or without a corresponding contract, the transaction is still valid. Ethereum has no way of knowing whether an address was correctly derived from a public key (and therefore from a private key).

WARNING	Ethereum cannot and does not validate recipient addresses in a transaction. You can send to an address that has no corresponding private key or contract, thereby "burning" the ether, rendering it forever unspendable. Validation should be done at the user interface level.

Sending a transaction to an invalid address will *burn* the ether sent, rendering it forever inaccessible (unspendable), since no signature can be generated to spend it. It is assumed that validation of the address happens at the user interface level (see [eip-55] or [icap]). In fact, there are a number of valid reasons for burning ether, including as a game-theory disincentive to cheating in payment channels and other smart contracts.

Transaction value and data

The main "payload" of a transaction is contained in two fields: value and data. Transactions can have both value and data, only value, only data, or neither value nor data. All four combinations are valid.

A transaction with only value is a *payment*. A transaction with only data is an *invocation*. A transaction with neither value nor data, well that's probably just a waste of gas! But it is still

possible.

Let's try all of the above combinations:

First, we set the source and destination addresses from our wallet, just to make the demo easier to read:

Set the source and destination addresses

```
src = web3.eth.accounts[0];
dst = web3.eth.accounts[1];
```

Transaction with value (payment), and no data payload

Value, no data

```
web3.eth.sendTransaction({from: src, to: dst, value: web3.toWei(0.01, "ether"), data:
""});
```

Our wallet shows a confirmation screen, indicating the value to send, and no data payload:

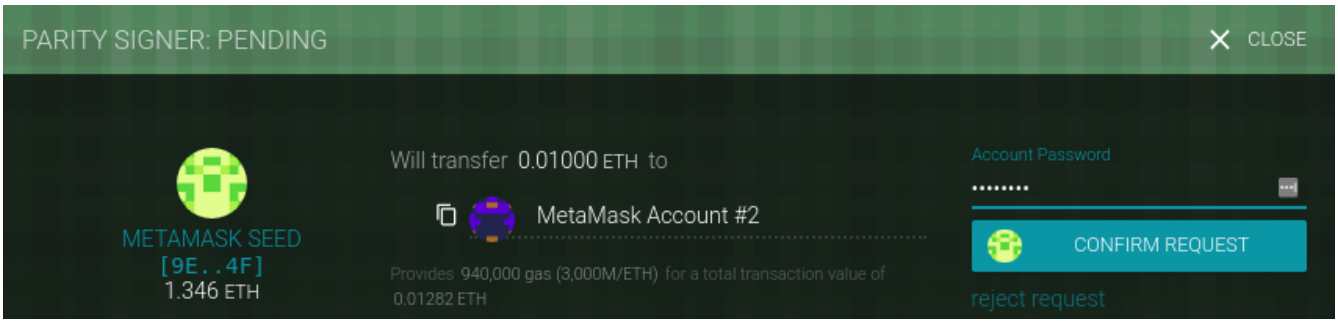

Figure 29. Parity wallet showing a transaction with value, but no data

Transaction with value (payment), and a data payload

Value and data

```
web3.eth.sendTransaction({from: src, to: dst, value: web3.toWei(0.01, "ether"), data:
"0x1234"});
```

Our wallet shows a confirmation screen, indicating the value to send and a data payload:

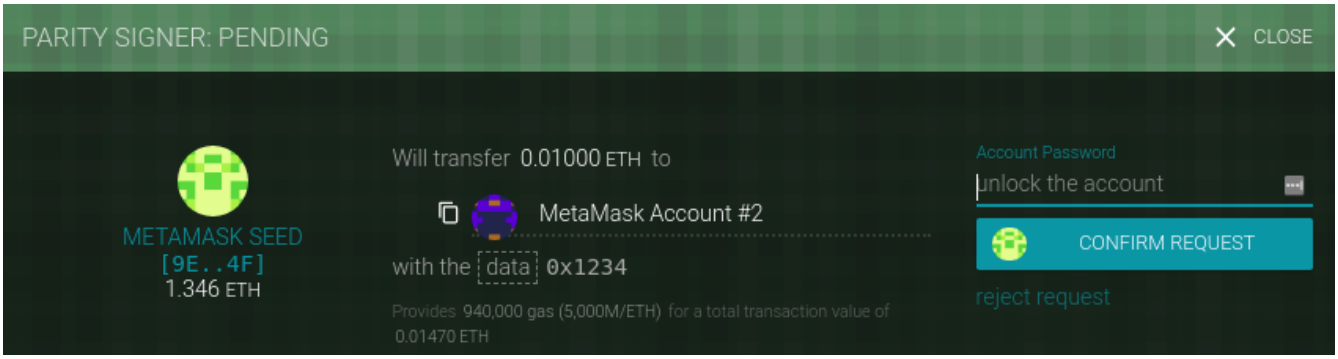

Figure 30. Parity wallet showing a transaction with value and data

Transaction with 0 value, only a data payload

No value, only data

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: "0x1234"});
```

Our wallet shows a confirmation screen, indicating the value as 0 and a data payload:

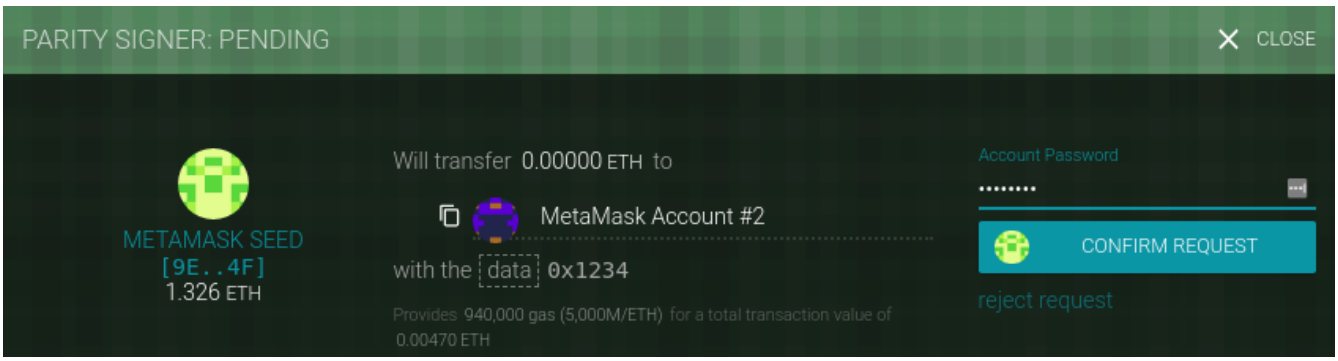

Figure 31. Parity wallet showing a transaction with no value, only data

Transaction with neither value (payment), nor data payload

No value, no data

```
web3.eth.sendTransaction({from: src, to: dst, value: 0, data: ""}));
```

Our wallet shows a confirmation screen, indicating 0 value and no data:

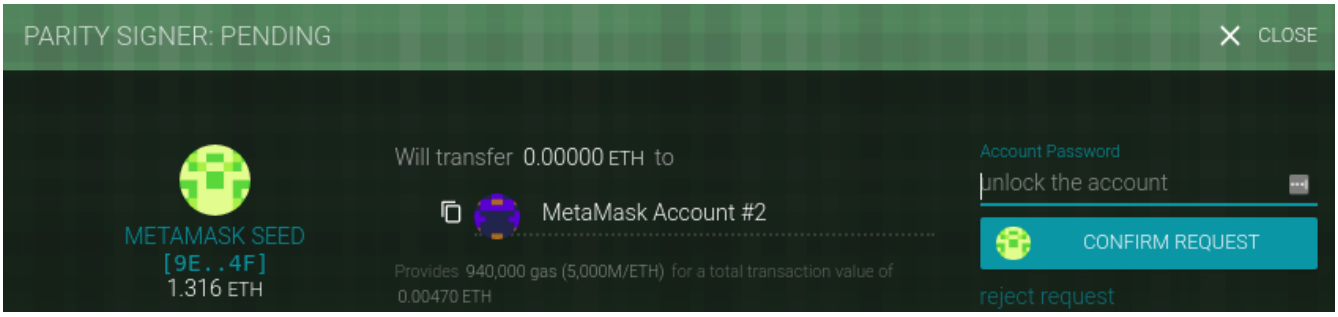

Figure 32. Parity wallet showing a transaction with no value, and no data

Transmitting value to EOAs and contracts

When you construct an Ethereum transaction that contains value, it is the equivalent of a *payment*. These transactions will behave differently depending on whether the destination address is a contract or not.

For EOA addresses, or rather for any address that isn't registered as a contract on the blockchain, Ethereum will record a state change, adding the value you sent to the balance of the address. If the address has not been seen before, it will be created and its balance initialized to the value of your payment.

If the destination address (to) is a contract, then the EVM will execute the contract and attempt to call the function named in the data payload of your transaction (see [invocation]). If there is no data payload in your transaction, the EVM will call the destination contract's *fallback* function and, if that function is payable, will execute it to determine what to do next.

A contract can reject incoming payments by throwing an exception immediately when the payable function is called, or as determined by conditions coded in the payable function. If the payable function terminated successfully (without an exception), then the contract's state is updated to reflect an increase in the contract's ether balance.

Transmitting a data payload to an EOA or contract

When your transaction contains a data payload, it is most likely addressed to a contract address. That doesn't mean you cannot send a data payload to an EOA. In fact, you can do that. However, in that case, the interpretation of the data payload is up to the wallet you use to access the EOA. Most wallets ignore any data payload received in a transaction to an EOA they control. In the future, it is possible that standards may emerge that allow wallets to interpret data payload encodings the way contracts do, thereby allowing transactions to invoke functions running inside user wallets. The critical difference is that any interpretation of the data payload by an EOA, is not subject to Ethereum's consensus rules, unlike a contract execution.

For now, let's assume your transaction is delivering a data payload to a contract address. In that case, the data payload will be interpreted by the EVM as *function invocation*, calling the named function and passing any encoded arguments to the function.

The data payload sent to a contract is a hex-serialized encoding of:

A function selector

The first 4 bytes of the Keccak256 hash of the function's *prototype*. This allows the EVM to unambiguously identify which function you wish to invoke.

The function arguments

The function's arguments, encoded according to the rules for the various elementary types defined by the EVM.

Let's look at a simple example, drawn from our Faucet.sol : A Solidity contract implementing a faucet. In Faucet.sol, we defined a single function for withdrawals:

```
function withdraw(uint withdraw_amount) public {
```

The *prototype* of the withdraw function is defined as the string containing the name of the function, followed by the data type of each of its arguments enclosed in parenthesis and separated by a single comma. The function name is withdraw and it takes a single argument that is a uint (which is an alias for uint256). So the prototype of withdraw would be:

```
withdraw(uint256)
```

Let's calculate the Keccak256 hash of this string (we can use the truffle console or any JavaScript web3 console to do that):

```
web3.sha3("withdraw(uint256)");
'0x2e1a7d4d13322e7b96f9a57413e1525c250fb7a9021cf91d1540d5b69f16a49f'
```

The first 4 bytes of the hash are 0x2e1a7d4d. That's our "function selector" value, which will tell the EVM which function we want to call.

Next, let's calculate a value to pass as the argument withdraw_amount. We want to withdraw 0.01 ether. Let's encode that to a hex-serialized big-endian unsigned 256-bit integer, denominated in wei:

```
withdraw_amount = web3.toWei(0.01, "ether");
'10000000000000000'
withdraw_amount_hex = web3.toHex(withdraw_amount);
'0x2386f26fc10000'
```

Now, we add the function selector to the amount (padded to 32 bytes):

```
2e1a7d4d00000000000000000000000000000000000000000000000002386f26fc10000
```

That's the data payload for our transaction, invoking the withdraw function and requesting 0.01 ether as the withdraw_amount.

Special transaction: Contract registration

There is one special case of a transaction with a data payload and no value. This is a transaction that *registers* a new contract. Contract registration transactions are sent to a special destination address, the zero address. In simple terms, the to field in a contract registration transaction contains the address 0x0. This address represents neither an EOA (there is no corresponding private/public key pair) nor a contract. It can never spend ether or initiate a transaction. It is only used as a destination, with the special meaning "register this contract".

While the zero address is only intended for contract registration, it sometimes receives payments

from various addresses. There are two explanations for this: either it is by accident, resulting in the loss of ether, or it is an intentional *ether burn* (see [burning_ether]). If you want to do an intentional ether burn, you should make your intention clear to the network and use the specially designated burn address instead:

```
0x000000000000000000000000000000000000dEaD
```

WARNING	Any ether sent to the contract registration address 0x0 or the designated burn address 0x0...dEaD above will become unspendable and lost forever.

A contract registration transaction should contain no ether value, only a data payload that contains the compiled bytecode of the contract. The only effect of this transaction is to register the contract.

As an example, we can publish Faucet.sol used in [intro]. The contract needs to be compiled into a binary hexadecimal representation. This can be done with the Solidiy compiler.

```
> solc --bin Faucet.sol
======= Faucet.sol:Faucet =======
Binary:
6060604052341561000f57600080fd5b60e58061001d6000396000f3006060604052600436106053f576000
357c0100000000000000000000000000000000000000000000000000000000900463ffffffff1680632e1a
7d4d146041575b005b3415604b57600080fd5b605f60048080359060200190919050506061565b005b6701
6345785d8a0000811115151560775760080fd5b3373ffffffffffffffffffffffffffffffffffffffff16
6108fc829081150290604051600060405180830381858888f193505050501515560b657600080fd5b505600
a165627a7a72305820d276ddd56041f7dc2d2eab69f01dd0a0146446562e25236cf4ba5095d2ee802f0029
```

The same information can also be obtained from the Remix online compiler. Now we can create the transaction.

```
> src = web3.eth.accounts[0];
> faucet_code =
"0x6060604052341561000f57600080fd5b60e58061001d6000396000f3006060604052600436106053f576
000357c0100000000000000000000000000000000000000000000000000000000900463ffffffff1680632
e1a7d4d146041575b005b3415604b57600080fd5b605f60048080359060200190919050506061565b005b6
7016345785d8a0000811115151560775760080fd5b3373ffffffffffffffffffffffffffffffffffffffff
f166108fc829081150290604051600060405180830381858888f193505050501515560b657600080fd5b505
600a165627a7a72305820d276ddd56041f7dc2d2eab69f01dd0a0146446562e25236cf4ba5095d2ee802f0
029"

> web3.eth.sendTransaction({from: src, data: faucet_code, gas: 113558, gasPrice:
200000000000})

"0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b"
```

There is no need to specify a to parameter, the default zero address will be used. You can specify gasPrice and gas limit. Once, the contract is mined we can see it on etherscan block explorer

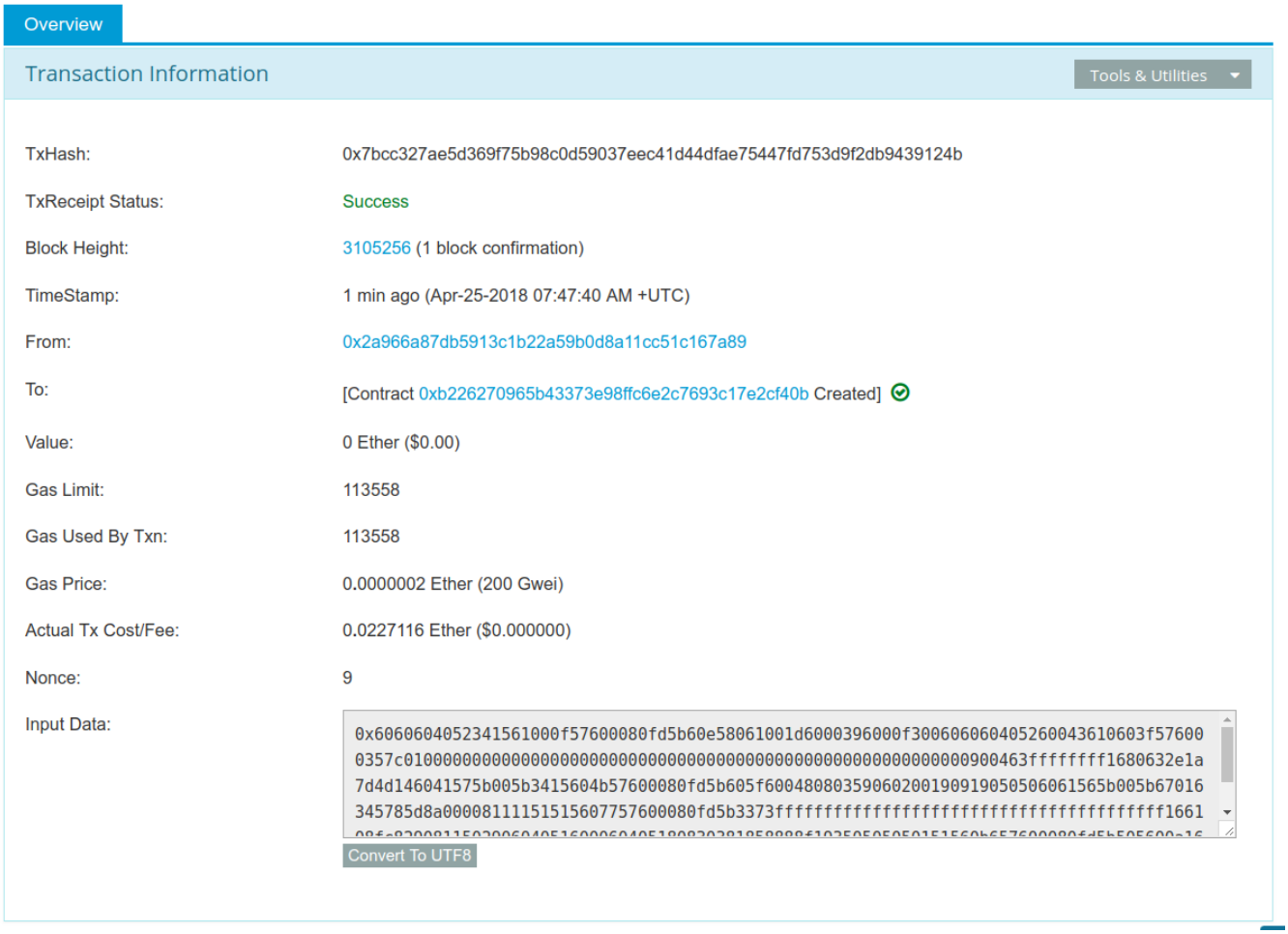

Figure 33. Etherscan showing the contract successully minded

You can look at the receipt of transaction to get information about the contract.

```
>
eth.getTransactionReceipt("0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9
439124b");

{
  blockHash: "0x6fa7d8bf982490de6246875deb2c21e5f3665b4422089c060138fc3907a95bb2",
  blockNumber: 3105256,
  contractAddress: "0xb226270965b43373e98ffc6e2c7693c17e2cf40b",
  cumulativeGasUsed: 113558,
  from: "0x2a966a87db5913c1b22a59b0d8a11cc51c167a89",
  gasUsed: 113558,
  logs: [],
  logsBloom:
"0x00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000"
,
  status: "0x1",
  to: null,
  transactionHash:
"0x7bcc327ae5d369f75b98c0d59037eec41d44dfae75447fd753d9f2db9439124b",
  transactionIndex: 0
}
```

Here we can see the address of the contract. We can send and receive funds from the contract as shown in Transmitting a data payload to an EOA or contract.

```
> contract_address = "0xb226270965b43373e98ffc6e2c7693c17e2cf40b"
> web3.eth.sendTransaction({from: src, to: contract_address, value: web3.toWei(0.1,
"ether"), data: ""});

"0x6ebf2e1fe95cc9c1fe2e1a0dc45678ccd127d374fdf145c5c8e6cd4ea2e6ca9f"

> web3.eth.sendTransaction({from: src, to: contract_address, value: 0, data:
"0x2e1a7d4d0000000000000000000000000000000000000000000002386f26fc10000"});

"0x59836029e7ce43e92daf84313816ca31420a76a9a571b69e31ec4bf4b37cd16e"
```

After a while, both transactions are visible on ethescan

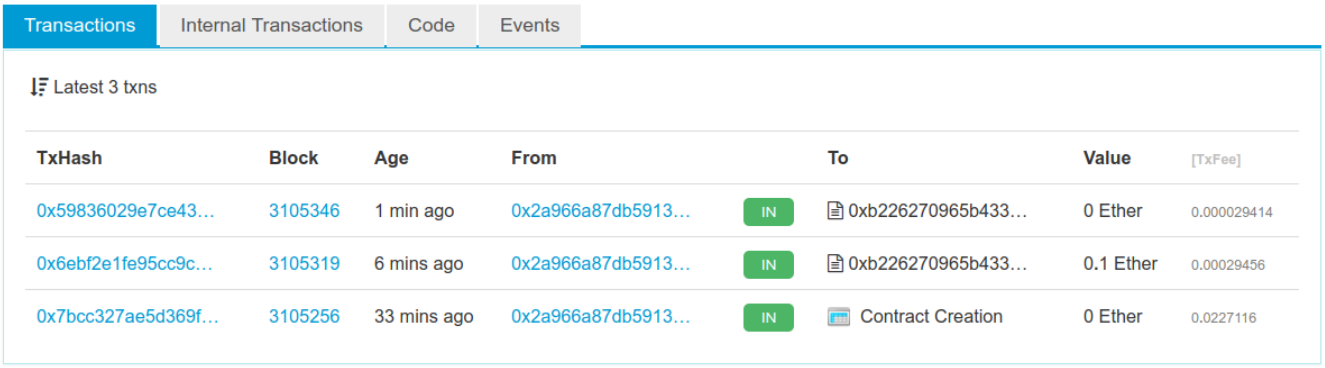

Figure 34. Etherscan showing the transactions for sending and receiving funds

Digital signatures

So far, we have not delved into any detail about "digital signatures." In this section, we look at how digital signatures work and how they can present proof of ownership of a private key without revealing that private key.

Elliptic Curve Digital Signature Algorithm (ECDSA)

The digital signature algorithm used in Ethereum is the *Elliptic Curve Digital Signature Algorithm*, or *ECDSA*. ECDSA is the algorithm used for digital signatures based on elliptic curve private/public key pairs, as described in Elliptic curve cryptography explained.

A digital signature serves three purposes in Ethereum (see the following sidebar). First, the signature proves that the owner of the private key, who is by implication the owner of an Ethereum account, has *authorized* the spending of ether, or execution of a contract. Secondly, the proof of authorization is *undeniable* (non-repudiation). Thirdly, the signature proves that the transaction data have not and *cannot be modified* by anyone after the transaction has been signed.

Wikipedia's Definition of a "Digital Signature"

A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (non-repudiation), and that the message was not altered in transit (integrity).

Source: https://en.wikipedia.org/wiki/Digital_signature

How Digital Signatures Work

A digital signature is a *mathematical scheme* that consists of two parts. The first part is an algorithm for creating a signature, using a private key (the signing key) from a message (the transaction). The second part is an algorithm that allows anyone to verify the signature by only using the message and a public key.

Creating a digital signature

In Ethereum's implementation of ECDSA, the "message" being signed is the transaction, or more accurately, the Keccak256 hash of the RLP-encoded data from the transaction. The signing key is the EOA's private key. The result is the signature:

\(Sig = F_{sig}(F_{keccak256}(m), k)\)

where:

- k is the signing private key
- m is the RLP-encoded transaction
- $F_{keccak256}$ is the Keccak256 hash function
- F_{sig} is the signing algorithm
- Sig is the resulting signature

More details on the mathematics of ECDSA can be found in ECDSA Math.

The function F_{sig} produces a signature Sig that is composed of two values, commonly referred to as R and S:

```
Sig = (R, S)
```

Verifying the Signature

To verify the signature, one must have the signature (R and S), the serialized transaction, and the public key (that corresponds to the private key used to create the signature). Essentially, verification of a signature means "Only the owner of the private key that generated this public key could have produced this signature on this transaction."

The signature verification algorithm takes the message (a hash of the transaction or parts of it), the signer's public key and the signature (R and S values), and returns TRUE if the signature is valid for this message and public key.

ECDSA Math

As mentioned previously, signatures are created by a mathematical function F_{sig} that produces a signature composed of two values R and S. In this section we look at the function F_{sig} in more detail.

The signature algorithm first generates an *ephemeral* (temporary) private/public key pair. This temporary key pair is used in the calculation of the R and S values, after a transformation involving the signing private key and the transaction hash.

The temporary key pair is generated by two input values:

1. A random number q, which is used as the temporary private key
2. and the elliptic curve generator point G

From q and G, we generate the corresponding temporary public key Q (calculated as $Q = q*G$, in the same way Ethereum public keys are derived; see Public keys). The R value of the digital signature is then the x coordinate of the ephemeral public key Q.

From there, the algorithm calculates the S value of the signature, such that:

$$S \equiv q^{-1} (Keccak256(m) + k * R) \pmod{p}$$

where:

- q is the ephemeral private key
- R is the x coordinate of the ephemeral public key
- k is the signing (EOA owner's) private key
- m is the transaction data
- p is the prime order of the elliptic curve

Verification is the inverse of the signature generation function, using the R, S values and the public key to calculate a value Q, which is a point on the elliptic curve (the ephemeral public key used in signature creation):

$$Q \equiv S^{-1} * Keccak256(m) * G + S^{-1} * R * K \pmod{p}$$

where:

- R and S are the signature values
- K is the signer's (EOA owner's) public key
- m is the transaction data that was signed
- G is the elliptic curve generator point
- p is the prime order of the elliptic curve

If the x coordinate of the calculated point Q is equal to R, then the verifier can conclude that the signature is valid.

Note that in verifying the signature, the private key is neither known nor revealed.

TIP	ECDSA is necessarily a fairly complicated piece of math; a full explanation is beyond the scope of this book. A number of great guides online take you through it step by step: search for "ECDSA explained" or try this one: http://bit.ly/2r0HhGB.

Transaction signing in practice

To produce a valid transaction, the originator must apply a digital signature to the message, using the Elliptic Curve Digital Signature algorithm. When we say "sign the transaction", we actually mean "sign the Keccak256 hash of the RLP serialized transaction data". The signature is applied to the hash of the transaction data, not the transaction itself.

TIP	At block # 2,675,000, Ethereum implemented the "Spurious Dragon" hard fork that, among other changes, introduced a new signing scheme that includes transaction replay protection. This new signing scheme is specified in EIP-155 (see [eip155]). This change affects the first step of the signing process, adding three fields (v, r, s) to the transaction before signing.

To sign a transaction in Ethereum, the originator must:

1. Create a transaction data structure, containing the nine fields: nonce, gasPrice, startGas, to, value, data, v, r, s
2. Produce an RLP-encoded serialized message of the transaction
3. Compute the Keccak256 hash of this serialized message
4. Compute the ECDSA signature, signing the hash with the originating EOA's private key
5. Insert the ECDSA signature's computed r and s values in the transaction

Raw transaction creation and signing

Let's create a raw transaction and sign it, using the ethereumjs-tx library. The source code for this example is in raw_tx_demo.js in the GitHub repository:

raw_tx_demo.js: Creating and signing a raw transaction in JavaScript

```
// Load requirements first:
//
// npm init
// npm install ethereumjs-tx
//
// Run with: $ node raw_tx_demo.js
const ethTx = require('ethereumjs-tx');

const txData = {
  nonce: '0x0',
  gasPrice: '0x09184e72a000',
  gasLimit: '0x30000',
  to: '0xb0920c523d582040f2bcb1bd7fb1c7c1ecebdb34',
  value: '0x00',
  data: '',
  v: "0x1c", // Ethereum main net chainID
  r: 0,
  s: 0
};

tx = new ethTx(txData);
console.log('RLP-Encoded Tx: 0x' + tx.serialize().toString('hex'))

txHash = tx.hash(); // This step encodes into RLP and calculates the hash
console.log('Tx Hash: 0x' + txHash.toString('hex'))

// Sign transaction
const privKey =
Buffer.from('91c8360c4cb4b5fac45513a7213f31d4e4a7bfcb4630e9fbf074f42a203ac0b9',
'hex');
tx.sign(privKey);

serializedTx = tx.serialize();
rawTx = 'Signed Raw Transaction: 0x' + serializedTx.toString('hex');
console.log(rawTx)
```

Download it here: https://github.com/ethereumbook/ethereumbook/blob/develop/code/web3js/raw_tx/raw_tx_demo.js

Run the example code:

```
$ node raw_tx_demo.js
RLP-Encoded Tx:
0xe6808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1ecebdb348080
Tx Hash: 0xaa7f03f9f4e52fcf69f836a6d2bbc7706580adce0a068ff6525ba337218e6992
Signed Raw Transaction:
0xf866808609184e72a0008303000094b0920c523d582040f2bcb1bd7fb1c7c1ecebdb3480801ca0ae236e
42bd8de1be3e62fea2fafac7ec6a0ac3d699c6156ac4f28356a4c034fda0422e3e6466347ef6e9796df8a3
b6b05bed913476dc84bbfca90043e3f65d5224
```

Raw transaction creation with EIP-155

The EIP-155 "Simple Replay Attack Protection" standard specifies a replay-attack-protected transaction encoding, which includes a *chain identifier* inside the transaction data, prior to signing. This ensures that transactions created for one blockchain (e.g. Ethereum main network) are invalid on another blockchain (e.g. Ethereum Classic or Ropsten test network). Therefore, transactions broadcast on one network cannot be *replayed* on another, hence the "replay attack protection" name of the standard.

EIP-155 adds three fields to the transaction data structure, v, r, and s. The r and s fields are initialized to zero. These three fields are added to the transaction data *before it is encoded and hashed*. The three additional fields therefore change the transaction's hash, to which the signature is later applied. By including the chain identifier in the data being signed, the transaction signature prevents any changes, as the signature is invalidated if the chain identifier is modified. Therefore, EIP-155 makes it impossible for a transaction to be replayed on another chain, because the signature's validity depends on the chain identifier.

The v signature prefix field is initialized to the chain identifier, with values:

Chain	Chain ID	
Ethereum main net	1	
Morden (obsolete), Expanse	2	
Ropsten	3	
Rinkeby	4	
Rootstock main net	30	
Rootstock test net	31	
Kovan	42	
Ethereum Classic main net	61	
Ethereum Classic test net	62	
Geth private testnets	1337	

The resulting transaction structure is RLP-encoded, hashed and signed. The signature algorithm is modified slightly to encode the chainID in the v prefix, too.

For more details, see the EIP-155 specification: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md

The signature prefix value (v) and public key recovery

As mentioned in Structure of Transaction, the transaction message doesn't include any "from" field. That's because the originator's public key can be computed directly from the ECDSA signature. Once you have the public key, you can compute the address easily. The process of recovering the signer's public key is called a *Public Key Recovery*.

Given the values r and s, that were computed in ECDSA Math, we can compute two possible public keys.

First, we compute two elliptic curve points R and R', from the x coordinate r value that is in the signature. There are two points, because the elliptic curve is symmetric across the x-axis, so that for any value x, there are two possible values that fit the curve, on either side of the x-axis.

From r, we also calculate r^{-1} which is the multiplicative inverse of r.

Finally we calculate z, which is the n-lowest bits of the message hash, where n is the order of the elliptic curve.

The two possible public keys are then:

$K_1 = r^{-1} (sR - zG)$

and

$K_2 = r^{-1} (sR' - zG)$

where:

- K_1 and K_2 are the two possibilities for the signer's public key
- r^{-1} is the multiplicative inverse of signature's r value
- s is the signature's s value
- R and R' are the two possibilities for the ephemeral public key *Q*
- z are the n-lowest bits of the message hash
- G is the elliptic curve generator point

To make things more efficient, the transaction signature includes a prefix value v, which tells us which of the two possible R values are the ephemeral public key. If v is even, then R is the correct value. If v is odd, then R'. That way, we need to calculate only one value for R and only one value for K.

Separating signing and transmission (offline signing)

Once a transaction is signed, it is ready to transmit to the Ethereum network. The three steps of creating, signing, and broadcasting a transaction normally happen in a single function, for example using web3.eth.sendTransaction. However, as we saw in Raw transaction creation and signing, you can create and sign the transaction in two separate steps. Once you have a signed transaction, you can then transmit it using web3.eth.sendSignedTransaction which takes a hex-encoded and signed

transaction message and transmits it on the Ethereum network.

Why would you want to separate the signing and transmission of transactions? The most common reason is security: the computer that signs a transaction must have unlocked private keys loaded in memory. The computer that transmits must be connected to the internet and be running an Ethereum client. If these two functions are on one computer, then you have private keys on an online system, which is quite dangerous. Separating the functions of signing and transmitting is called *offline signing* and is a common security practice.

Depending on the level of security you need, your "offline signing" computer can have varying degrees of separation from the online computer, ranging from an isolated and firewalled subnet (online but segregated) to a completely offline system known as an *air-gapped* system. In an air-gapped system there is no network connectivity at all - the computer is separated from the online environment by a gap of "air". To sign transactions you transfer them to and from the air-gapped computer using data storage media or (better) a webcam and QR code. Of course, this means you must manually transfer every transaction you want signed, and this doesn't scale.

While not many environments can utilize a fully air-gapped system, even a small degree of isolation has significant security benefits. For example, an isolated subnet with a firewall that only allows a message-queue protocol through, can offer a much-reduced attack surface and much higher security than signing on the online system. Many companies use a protocol such as ZeroMQ (0MQ), as it offers a reduced attack surface for the signing computer. With a setup like that, transactions are serialized and queued for signing. The queuing protocol transmits the serialized message, in a way similar to a TCP socket, to the signing computer. The signing computer reads the serialized transactions from the queue (carefully), applies a signature with the appropriate key, and places them on an outgoing queue. The outgoing queue transmits the signed transactions to a computer with an Ethereum client that de-queues them and transmits them.

Transaction propagation

The Ethereum network uses a "flood" routing protocol. Each Ethereum client, acts as a *node* in a *Peer-to-Peer (P2P)*, which (ideally) forms a *mesh* network. No network node is "special", they all act as equal peers. We will use the term "node" to refer to an Ethereum client that is connected to and participates in the P2P network.

Transaction propagation starts with the originating Ethereum node creating (or receiving from offline) a signed transaction. The transaction is validated and then transmitted to all the other Ethereum nodes that are *directly* connected to the originating node. On average, each Ethereum node maintains connections to at least 13 other nodes, called its *neighbors*. Each neighbor node validates the transaction as soon as they receive it. If they agree that it is valid, they store a copy and propagate it to all their neighbors (except the one it came from). As a result, the transaction ripples outwards from the originating node, *flooding* across the network, until all nodes in the network have a copy of the transaction.

Within just a few seconds, an Ethereum transaction propagates to all the Ethereum nodes around the globe. From the perspective of each node, it is not possible to discern the origin of the transaction. The neighbor that sent it to our node may be the originator of the transaction or may have received it from one of its neighbors. To be able to track the origin of transactions, or interfere with propagation, an attacker would have to control a significant percentage of all nodes. This is

part of the security and privacy design of P2P networks, especially as applied to blockchains.

Recording in the chain

While all the nodes in Ethereum are equal peers, some of them are operated by *miners* and are feeding transactions and blocks to *mining farms*, which are computers with high-performance Graphical Processing Units (GPUs). The mining computers add transactions to a candidate block and attempt to find a *Proof-of-Work* that makes the candidate block valid. We will discuss this in more detail in [consensus].

Without going into too much detail, valid transactions will eventually be included in a block of transactions and, thus, recorded in the Ethereum blockchain. Once mined into a block, transactions also modify the state of the Ethereum singleton, either by modifying the balance of an account (in the case of a simple payment), or by invoking contracts that change their internal state. These changes are recorded alongside the transaction, in the form of a transaction *receipt*, which may also include *events*. We will examine all this in much more detail in [evm]

Our transaction has completed its journey from creation to signing by an EOA, propagation, and finally mining. It has changed the state of the singleton and left an indelible mark on the blockchain.

Multiple signatures (multisig) transactions

If you are familiar with Bitcoin's scripting capabilities, you know that it is possible to create a Bitcoin multisig account which can only spend funds when multiple parties sign the transaction (e.g. 2 of 2 or 3 of 4 signatures). Ethereum's value transactions have no provisions for multiple signatures, although arbitrary contracts with any conditions may be deployed to handle the transfer of ether and tokens alike.

To protect your ether under a multisig condition, transfer them to a multisig contract. Whenever you want to send funds to another account, all the required users will need to send transactions to the contract using a regular wallet software, effectively authorizing the contract to perform the final transaction.

These contracts can also be designed to require multiple signatures before executing local code or to trigger other contracts. The security of the scheme is ultimately determined by the multisig contract code.

Discussion and Grid+ reference implementation:
https://blog.gridplus.io/toward-an-ethereum-multisig-standard-c566c7b7a3f6

Tokens

What are tokens?

The word *token* derives from the Old English "tacen" meaning a sign or symbol. Commonly used to mean privately-issued coin-like items that have insignificant value, as used in transportation

tokens, laundry tokens, arcade tokens.

Nowadays, tokens based on blockchains are redefining the word to mean blockchain-based abstractions that can be owned and that represent assets, currency, or access rights.

The association between the word "token" and insignificant value has a lot to do with the limited use of physical tokens. Often restricted to specific businesses, organizations or locations, physical tokens are not easily exchangeable and cannot be used for more than one function. With blockchain tokens, these restrictions are erased. Many of these tokens serve multiple purposes globally and can be traded for each other or for other currencies in global liquid markets. With those restrictions gone, the "insignificant value" expectation is also a thing of the past.

In this section, we look at various uses for tokens and how they are created. We also discuss attributes of tokens such as fungibility and intrinsicality. Finally, we examine the standards and technologies that they are based on an experiment by building our own tokens.

How are tokens used?

The most obvious use of tokens is as digital private currencies. However, this is only one possible use. Tokens can be programmed to serve many different functions, often overlapping. For example, a token can simultaneously convey a voting right, an access right, and ownership of a resource. Currency is just the first "app".

Currency

A token can serve as a form of currency, with a value determined through private trade. For example, ether or bitcoin.

Resource

A token can represent a resource earned or produced in a sharing-economy or resource-sharing environment. For example, a storage or CPU token representing resources that can be shared over a network.

Asset

A token can represent ownership of an intrinsic or extrinsic, tangible or intangible asset. For example, gold, real-estate, a car, oil, energy etc.

Access

A token can represent access rights and even convey access to a digital or physical property, such as a discussion forum, an exclusive website, a hotel room, a rental car.

Equity

A token can represent shareholder equity in a digital organization (e.g. a DAO) or legal fiction (e.g. a corporation)

Voting

A token can represent voting rights in a digital or legal system.

Collectible

A token can represent a digital (e.g. CryptoPunks) or physical collectible (e.g. a painting)

Identity

A token can represent a digital (e.g. avatar) or legal identity (e.g. national ID).

Attestation

A token can represent a certification or attestation of fact by some authority or by a decentralized reputation system (e.g. marriage record, birth certificate, college degree).

Utility

A token can be used to access or pay for a service.

Often, a single token encompasses several of these functions. Sometimes it is hard to discern between them, as the physical equivalents have always been inextricably linked. For example, in the physical world, a driver's license (attestation) is also an identity document (identity) and the two cannot be separated. In the digital realm, previously commingled functions can be separated and developed independently (e.g. an anonymous attestation).

Tokens and fungibility

From Wikipedia:

```
In economics, fungibility is the property of a good or a commodity whose individual
units are essentially interchangeable.
```

Tokens are fungible when we can substitute any single unit of the token for another without any difference in its value or function. For example, ether is a fungible token, as any unit of ether has the same value and use as any other unit of ether.

Strictly speaking, if a token's historical provenance can be tracked, then it is not entirely fungible. The ability to track provenance can lead to blacklisting and whitelisting, reducing or eliminating fungibility. We will examine this further in [privacy].

Non-fungible tokens are tokens that each represent a unique tangible or intangible item and therefore are not interchangeable. For example, a token that represents ownership of a *specific* Van Gogh painting is not equivalent to another token that represents a Picasso. Similarly, a token representing a *specific* digital collectible such as a specific CryptoKitty (see [cryptoKitties]) is not interchangeable with any other CryptoKitty. Each non-fungible token is associated with an unique identifier, such as a serial number.

We will see examples of both fungible and non-fungible tokens later in this section.

Counterparty risk

Counterparty risk is the risk that the *other* party in a transaction will fail to meet their obligations. Some types of transactions create additional counterparty risks because of the addition of more than two parties in the transaction. For example, if you hold a certificate of deposit for a precious

metal and you sell that to someone, there are at least 3 parties in that transaction: the seller, the buyer and the custodian of the precious metal. Someone holds the physical asset and by necessity, they become a party and add counterparty risk to any transaction involving that asset. When an asset is traded indirectly through the exchange of a token of ownership, there is additional counterparty risk from the custodian of the asset. Do they have the asset? Will they recognize (or allow) the transfer of ownership based on the transfer of a token (such as a certificate, deed, title or digital token)? In the world of digital tokens, it is important to understand who holds the asset that is represented by the token and what rules apply to that underlying asset.

Tokens and intrinsicality

The word "intrinsic" derives from the Latin "intra", meaning "from within".

Some tokens represent digital items that are *intrinsic* to the blockchain. Those digital assets are governed by consensus rules, just like the tokens themselves. This has an important implication: tokens that represent intrinsic assets do not carry additional counterparty risk. If you hold the keys to 1 ether, there is no other party holding that ether for you. The blockchain consensus rules apply and your ownership (control) of the private keys is equivalent to ownership of the asset, without any intermediary.

Conversely, many tokens are used to represent *extrinsic* things, like real-estate, corporate voting shares, trademarks, gold bars. The ownership of these items, which are not "within" the blockchain, is governed by law, custom and policy that are separate from the consensus rules that govern the token. In other words, token issuers and owners may still depend on real world non-smart contracts. As a result, these extrinsic assets carry additional counterparty risk because they are held by custodians, recorded in external registries, or controlled by laws and policies outside the blockchain environment.

One of the most important ramifications of blockchain-based tokens is the ability to convert extrinsic assets into intrinsic assets and thereby remove counterparty risk. A good example is moving from equity in a corporation (extrinsic) to an equity or voting token in a *decentralized autonomous organization* or similar (intrinsic) organization.

Using tokens: utility or equity

Almost all projects in Ethereum today are launching with some kind of token. But do all these projects really need a token? Are there any disadvantages to using a token, or will we see the slogan "tokenize all the things" come to fruition?

First, let’s start by clarifying the role of a token in a new project. The majority of projects are using tokens in one of two ways: either as "utility tokens" or as "equity tokens". Very often, those two roles are conflated and difficult to distinguish.

Utility tokens are those where the use of the token is required to pay for a service, application or resource. Examples of utility tokens include tokens that represent resources such as shared storage, access to services such as social media networks, or ether itself as gas for the Ethereum platform. By comparison, equity tokens are those that represent shares in a startup.

Equity tokens can be as limited as non-voting shares for distribution of dividends and profits, or as expansive as voting shares in a decentralized autonomous organization, where management of the platform is through majority votes by the token holders.

It’s not a duck

Just because a token is used to fundraise for a startup, doesn’t mean it has to be used as payment for the service, and vice-versa. Many startups, however, face a difficult problem: tokens are a great fundraising mechanism, but offering securities (equity) to the public is a regulated activity in most jurisdictions. By disguising equity tokens as utility tokens, many startups hope to get around these regulatory restrictions and raise money from a public offering while presenting it as a pre-sale of a utility token. Whether these thinly disguised equity offerings will be able to skirt the regulators remains to be seen.

As the popular saying goes: "If it walks like a duck and quacks like a duck - it’s a duck". Regulators are not likely to be distracted by these semantic contortions, quite the opposite, they are more likely to see such legal sophistry as an attempt to deceive the public.

Utility tokens: who needs them?

The real problem is that utility tokens introduce significant risks and adoption barriers for startups. Perhaps in a distant future "tokenize all the things" becomes a reality. But, at present, the number of people who have access to, an understanding of, and desire to use a token is a subset of the already small cryptocurrency market.

For a startup, each innovation represents a risk and a market filter. Innovation is taking the road least traveled, walking away from the path of tradition. It is already a lonely walk. If a startup is trying to innovate in a new area of technology, such as storage sharing over P2P networks, that is a lonely enough path. Adding a utility token to that innovation and requiring users to adopt tokens in order to use the service compounds the risk and increases the barriers to adoption. It’s walking off the already lonely trail of P2P storage innovation and into the wilderness.

Think of each innovation as a filter. It limits adoption to the subset of the market that can become early adopters of this innovation. Adding a second filter compounds that effect, further limiting the addressable market. You are asking your early adopters to adopt not one but two completely new technologies: the novel application/platform/service you built, and the token economy.

For a startup, each innovation introduces risks that increase the chance of failure of the startup. If you take your already risky startup idea and add a utility token, you are adding all the risks of the underlying platform (Ethereum), broader economy (exchanges, liquidity), regulatory environment (equity/commodity regulators) and technology (smart contracts, token standards). That’s a lot of risk for a startup.

Advocates of "tokenize all the things" will likely counter that by adopting tokens, they are also inheriting the market enthusiasm, early adopters, technology, innovation and liquidity of the entire token economy. That is true too. The question is whether the benefits and enthusiasm outweigh the risks and uncertainties.

Nevertheless, some of the most innovative business ideas are indeed taking place in the crypto

realm. If regulators are not quick enough to adopt laws and support new business models, talents and entrepreneurs will seek to operate in other jurisdictions which are more crypto-friendly. This is actually happening right now.

Finally, at the beginning of this chapter, when introducing tokens we discuss the colloquial meaning of token as "something of insignificant value". The underlying reason for the insignificant value of most tokens is because they can only be used in a very narrow context: one bus company, one laundromat, one arcade, one hotel, one company store. Limited liquidity, limited applicability, and high conversion costs reduce the value of tokens all the way down until it is only a "token" value. So when you add a utility token to your platform, but the token can only be used on your own one platform with a small market, you are re-creating the conditions that made physical tokens worthless. If in order to use your platform, a user has to convert something into your utility token, use it and then convert the remainder back into something more generally useful, you've created company scrip. The switching costs of a digital token are orders of magnitude lower than a physical token without a market. But the switching costs are not zero. Utility tokens that work across an entire industry sector will be very interesting and probably quite valuable. But if you set up your startup to have to bootstrap an entire industry standard in order to succeed, you may have already failed.

One of the benefits of deploying services on general-purpose platforms like Ethereum is exactly being able to connect smart contracts, increasing the potential for liquidity and utility of tokens.

Make this decision for the right reasons. Adopt a token because your application *cannot work without a token* (e.g. Ethereum). Adopt it because the token solves a fundamental market barrier or access problem. Don't introduce a utility token because it is the only way you can raise money fast and you need to pretend it's not a public securities offering.

Token Standards

Blockchain tokens have existed before Ethereum. In some ways, the first blockchain currency, bitcoin, is a token itself. Many token platforms were also developed on Bitcoin and other cryptocurrencies before Ethereum. However, the introduction of the first token standard on Ethereum led to an explosion of tokens.

Vitalik Buterin suggested tokens as one of the most obvious and useful applications of a generalized programmable blockchain such as Ethereum. In fact, in the first year of Ethereum, it was common to see Vitalik and others wearing t-shirts emblazoned with the Ethereum logo and a smart contract sample on the back. There were several variations of this t-shirt, but the most common showed an implementation of a token.

ERC20 Token Standard

The first standard was introduced in November 2015 by Fabian Vogelsteller, as an Ethereum Request for Comments (ERC). It was automatically assigned GitHub issue number 20, giving rise to the name "ERC20 token". The vast majority of tokens are currently based on ERC20. The ERC20 request for comments eventually became Ethereum Improvement Proposal EIP20 but is mostly still referred to by the original name ERC20. You can read the standard here:

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

ERC20 is a standard for *fungible tokens* meaning that different units of an ERC20 token are interchangeable and have no unique properties.

The ERC20 standard defines a common interface for contracts implementing a token, such that any compatible token can be accessed and used in the same way. The interface consists of a number of functions that must be present in every implementation of the standard, as well as some optional functions and attributes that may be added by developers.

ERC20 required functions & events

totalSupply

Returns the total units of this token that currently exist. ERC20 tokens can have fixed or variable supply.

balanceOf

Given an address, returns the token balance of that address.

transfer

Given an address and amount, transfers that amount of tokens to that address, from the balance of the address that executed the transfer.

transferFrom

Given a sender, recipient and amount, transfers tokens from one account to another. Used in combination with approve below.

approve

Given a recipient address and amount, authorizes that address to execute several transfers up to that amount, from the account that issued the approval.

allowance

Given an owner address and a spender address, returns the remaining amount that the spender is approved to withdraw from the owner.

Transfer event

Event triggered upon successful transfer (call to transfer or transferFrom) (even for zero value transfers).

Approval event

Event logged upon a successful call to approve.

ERC20 optional functions

name

Returns a human readable name (e.g. "US Dollars") of the token.

symbol

Returns a human readable symbol (e.g. "USD") for the token.

decimals

Returns the number of decimals used to divide token amounts. For example, if decimals is 2, then the token amount is divided by 100 to get its user representation.

The ERC20 interface defined in Solidity

Here's what an ERC20 interface specification looks like in Solidity:

```
contract ERC20 {
   function totalSupply() constant returns (uint theTotalSupply);
   function balanceOf(address _owner) constant returns (uint balance);
   function transfer(address _to, uint _value) returns (bool success);
   function transferFrom(address _from, address _to, uint _value) returns (bool
success);
   function approve(address _spender, uint _value) returns (bool success);
   function allowance(address _owner, address _spender) constant returns (uint
remaining);
   event Transfer(address indexed _from, address indexed _to, uint _value);
   event Approval(address indexed _owner, address indexed _spender, uint _value);
}
```

ERC20 data structures

If you examine any ERC20 implementation, it will contain two data structures, one to track balances and one to track allowances. In Solidity, they are implemented with a *data mapping*.

The first data mapping implements an internal table of token balances, by owner. This allows the token contract to keep track of who owns the tokens. Each transfer is a deduction from one balance and an addition to another balance.

Balances: a mapping from address (owner) to amount (balance)

```
mapping(address => uint256) balances;
```

The second data structure is a data mapping of allowances. As we will see in ERC20 workflows: "transfer" and "approve & transferFrom", with ERC20 tokens an owner of a token can delegate authority to a spender, allowing them to spend a specific amount (allowance) from the owner's balance. The ERC20 contract keeps track of the allowances with a two-dimensional mapping, with the primary key being the address of the token owner, mapping to a spender address and an allowance amount:

Allowances: a mapping from address (owner) to address (spender) to amount (allowance)

```
mapping (address => mapping (address => uint256)) public allowed;
```

ERC20 workflows: "transfer" and "approve & transferFrom"

The ERC20 token standard has two transfer functions. You might be wondering why?

ERC20 allows two different workflows. The first is a single-transaction, straightforward workflow using the transfer function. This workflow is the one used by wallets to send tokens to other wallets. The vast majority of token transactions happen with the transfer workflow.

Executing the transfer contract is very simple. If Alice wants to send 10 tokens to Bob, her wallet sends a transaction to the token contract's address, calling the transfer function with Bob's address and "10" as the arguments. The token contract adjusts Alice's balance (-10) and Bob's balance (10) and issues a +Transfer event.

The second workflow is a two-transaction workflow that uses approve, followed by transferFrom. This workflow allows a token owner to delegate their control to another address. It is most often used to delegate control to a contract for distribution of tokens, but it can also be used by exchanges. For example, if a company is selling tokens for an ICO, they can approve a crowdsale contract address to distribute a certain amount of tokens. The crowdsale contract can then transferFrom the token contract owner balance to each buyer of the token.

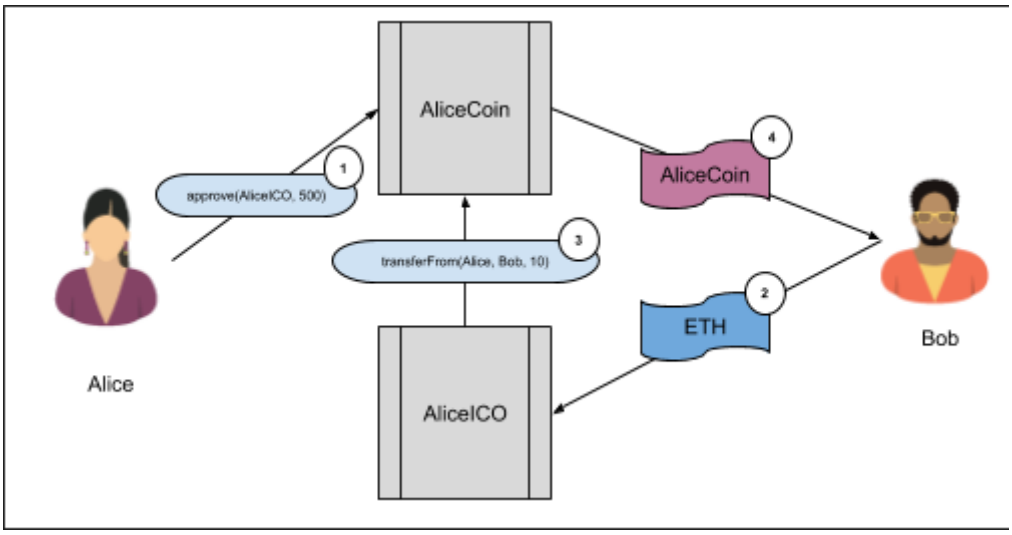

Figure 35. The two-step approve & transferFrom workflow of ERC20 tokens

For the approve & transferFrom workflow, two transactions are needed. Let's say that Alice wants to allow the AliceICO contract to sell 50% of all the AliceCoin tokens to buyers like Bob and Charlie. First, Alice launches the AliceCoin ERC20 contract, issuing all the AliceCoin to her own address. Then, Alice launches the AliceICO contract that can sell tokens for ether. Next, Alice initiates the approve & transferFrom workflow. She sends a transaction to AliceCoin, calling approve, with the address of AliceICO and 50% of the totalSupply. This will trigger the Approval event. Now, the AliceICO contract can sell AliceCoin.

When AliceICO receives ether from Bob, it needs to send some AliceCoin to Bob in return. Within the AliceICO contract is an exchange rate between AliceCoin and ether. The exchange rate that Alice set when she created the AliceICO determines how many tokens Bob will receive for the amount of ether sent to AliceICO. When AliceICO calls the AliceCoin transferFrom function, it sets Alice's address as the sender, Bob's address as the recipient, and uses the exchange rate to determine how many AliceCoin tokens will be transferred to Bob in the "value" field. The AliceCoin contract transfers the balance from Alice's address to Bob's address and triggers a Transfer event. The AliceICO contract can call transferFrom an unlimited number of times, as long as it doesn't exceed the approval limit Alice set. The AliceICO contract can keep track of how many AliceCoin tokens it can sell by calling the allowance function.

ERC20 Implementations

While it is possible to implement an ERC20-compatible token in about thirty lines of Solidity code, most implementations are more complex, to account for potential security vulnerabilities. There are two implementations mentioned in the EIP20 standard:

Consensys EIP20

A simple and easy to read implementation of an ERC20-compatible token.

You can read the Solidity code for Consensys' implementation here: https://github.com/ConsenSys/Tokens/blob/master/contracts/eip20/EIP20.sol

OpenZeppelin StandardToken

This implementation is ERC20-compatible, with additional security precautions. It forms the basis of OpenZeppelin libraries implementing more complex ERC20-compatible tokens with fundraising caps, auctions, vesting schedules and other features.

You can see the Solidity code for OpenZeppelin StandardToken here: https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/ERC20/StandardToken.sol

Launching our own ERC20 token

Let's create and launch our own token. For this example, we will use the truffle framework (see [truffle]). The example assumes you have already installed truffle, configured it, and are familiar with its basic operation.

We will call our token "Mastering Ethereum Token", with symbol "MET".

You can find this example in the book's GitHub repository: https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken

First, let's create and initialize a truffle project directory, the same way we did in [truffle_project_directory]. Run these four commands and accept the default answers to any questions:

```
$ mkdir METoken
$ cd METoken
METoken $ truffle init
METoken $ npm init
```

You should now have the following directory structure:

```
METoken/
 ├──── contracts
 │     └──── Migrations.sol
 ├──── migrations
 │     └──── 1_initial_migration.js
 ├──── package.json
 ├──── test
 ├──── truffle-config.js
 └──── truffle.js
```

Edit the truffle.js or truffle-config.js configuration file to set up your truffle environment, or copy the one we used from:

https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken/truffle-config.js

If you use the example truffle-config.js, remember to create a file .env in the METoken folder containing your test private keys for testing and deployment on public Ethereum test networks, such as ganache or Kovan. You can export your test network private key from MetaMask.

After that your directory look like:

```
METoken/
 ├──── contracts
 │     └──── Migrations.sol
 ├──── migrations
 │     └──── 1_initial_migration.js
 ├──── package.json
 ├──── test
 ├──── truffle-config.js
 ├──── truffle.js
 └──── .env *new file*
```

WARNING	Only use test keys or test mnemonics that are not used to hold funds on the main Ethereum network. Never use keys that hold real money for testing.

For our example, we will import the OpenZeppelin StandardContract, which implements some important security checks and is easy to extend. Let's import that library:

```
$ npm install zeppelin-solidity

+ zeppelin-solidity@1.6.0
added 8 packages in 2.504s
```

The zeppelin-solidity package will add about 250 files under the node_modules directory. The OpenZeppelin library includes a lot more than the ERC20 token, but we will only use a small part of it.

Next, let's write our token contract. Create a new file METoken.sol and copy the example code from GitHub:

https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken/contracts/METoken.sol

Our contract is very simple, as it inherits all the functionality from the OpenZeppelin StandardToken library:

METoken.sol : A Solidity contract implementing an ERC20 token

```
Unresolved directive in tokens.asciidoc -
include::code/METoken/contracts/METoken.sol[]
```

Here, we are defining the optional variables name, symbol, and decimals. We also define an _initial_supply variable, set to 21 million tokens, and two decimals of subdivision (2.1 billion total). In the contract's initialization (constructor) function we set the totalSupply to be equal to _initial_supply and allocate all of the _initial_supply to the balance of the account (msg.sender) that creates the METoken contract.

We now use truffle to compile the METoken code:

```
$ truffle compile
Compiling ./contracts/METoken.sol...
Compiling ./contracts/Migrations.sol...
Compiling zeppelin-solidity/contracts/math/SafeMath.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/BasicToken.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/ERC20.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/ERC20Basic.sol...
Compiling zeppelin-solidity/contracts/token/ERC20/StandardToken.sol...
```

As you can see, truffle incorporated necessary dependencies from the OpenZeppelin libraries and compiled those contracts too.

Let's set up a migration script, to deploy the METoken contract. Create a new file 2_deploy_contracts.js in the METoken/migrations folder. Copy the contents from the example on Github repository:

https://github.com/ethereumbook/ethereumbook/blob/develop/code/truffle/METoken/migrations/2_deploy_contracts.js

Here's what it contains:

2_deploy_contracts: Migration to deploy METoken

```
Unresolved directive in tokens.asciidoc -
include::code/METoken/migrations/2_deploy_contracts.js[]
```

Before we deploy on one of the Ethereum test networks, let's start a local blockchain to test

everything. Start the ganache blockchain, either from the command-line with ganache-cli or from the graphical user interface, as we did in [using_ganache].

Once ganache is started, we can deploy our METoken contract and see if everything works as expected:

```
$ truffle migrate --network ganache
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0xb2e90a056dc6ad8e654683921fc613c796a03b89df6760ec1db1084ea4a084eb
  Migrations: 0x8cdaf0cd259887258bc13a92c0a6da92698644c0
Saving successful migration to network...
  ... 0xd7bc86d31bee32fa3988f1c1eabce403a1b5d570340a3a9cdba53a472ee8c956
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Deploying METoken...
  ... 0xbe9290d59678b412e60ed6aefedb17364f4ad2977cfb2076b9b8ad415c5dc9f0
  METoken: 0x345ca3e014aaf5dca488057592ee47305d9b3e10
Saving successful migration to network...
  ... 0xf36163615f41ef7ed8f4a8f192149a0bf633fe1a2398ce001bf44c43dc7bdda0
Saving artifacts...
```

On the ganache console, we should see that our deployment has created 4 new transactions:

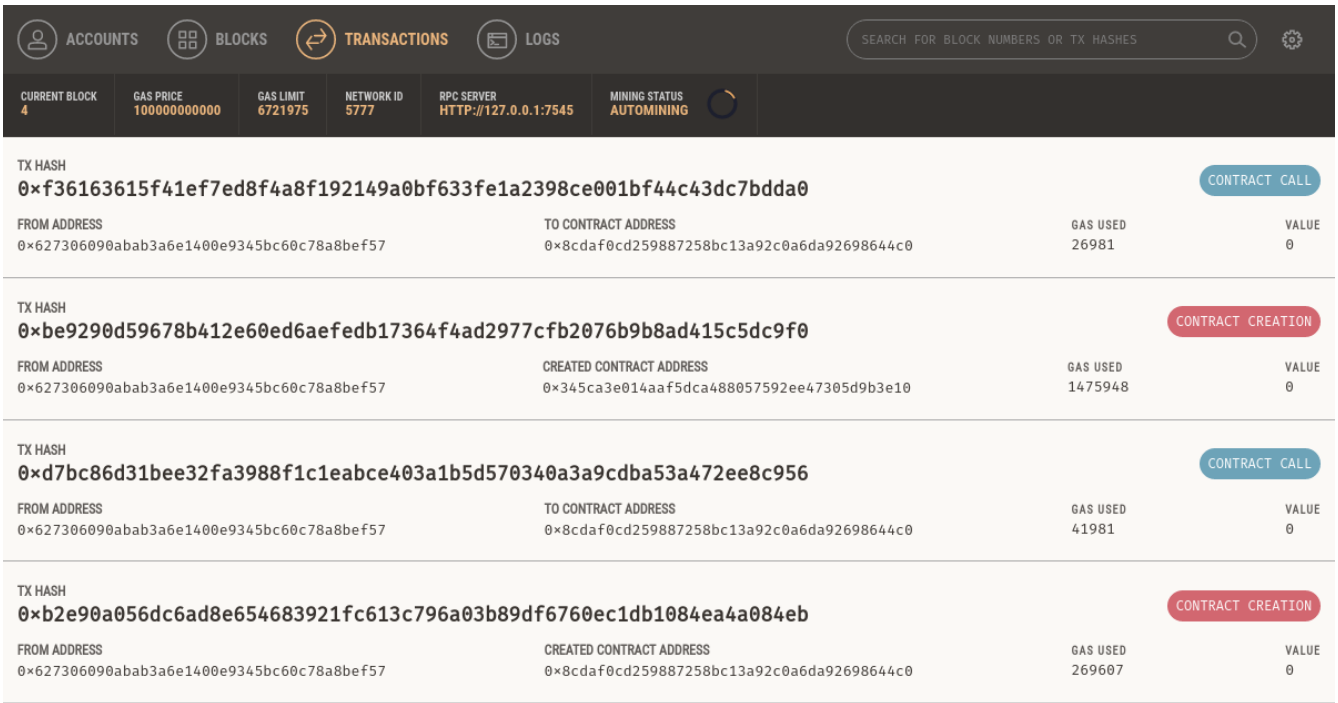

Figure 36. METoken deployment on Ganache

Interacting with METoken using the truffle console

We can interact with our contract on the ganache blockchain, using the truffle console. This is an

interactive JavaScript environment that provides access to the truffle environment and, via Web3, to the blockchain. In this case, we will connect the truffle console to the ganache blockchain:

```
$ truffle console --network ganache
truffle(ganache)>
```

The truffle(ganache)> prompt shows that we are connected to the ganache blockchain and are ready to type our commands. The truffle console supports all the truffle commands, so we could compile and migrate from the console. We've already run those commands, so let's go directly to the contract itself. The METoken contract exists as a JavaScript object within the truffle environment. Type METoken at the prompt and it will dump the entire contract definition:

```
truffle(ganache)> METoken
{ [Function: TruffleContract]
  _static_methods:

[...]

currentProvider:
 HttpProvider {
   host: 'http://localhost:7545',
   timeout: 0,
   user: undefined,
   password: undefined,
   headers: undefined,
   send: [Function],
   sendAsync: [Function],
   _alreadyWrapped: true },
network_id: '5777' }
```

The METoken object also exposes several attributes, such as the address of the contract (as deployed by the migrate command):

```
truffle(ganache)> METoken.address
'0x345ca3e014aaf5dca488057592ee47305d9b3e10'
```

If we want to interact with the deployed contract, we have to use an asynchronous call, in the form of a JavaScript "promise". We use the deployed function to get the contract instance and then call the totalSupply function:

```
truffle(ganache)> METoken.deployed().then(instance => instance.totalSupply())
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

Next, let's use the accounts created by ganache to check our METoken balance and send some METoken to another address. First, let's get the account addresses:

```
truffle(ganache)> let accounts
undefined
truffle(ganache)> web3.eth.getAccounts((err,res) => { accounts = res })
undefined
truffle(ganache)> accounts[0]
'0x627306090abab3a6e1400e9345bc60c78a8bef57'
```

The accounts list now contains all the accounts created by ganache, and account[0] is the account that deployed the METoken contract. It should have a balance of METoken, because our METoken constructor gives the entire token supply to the address that created it. Let's check:

```
truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(accounts[0]).then(console.log) })
undefined
BigNumber { s: 1, e: 9, c: [ 2100000000 ] }
```

Finally, let's transfer 1000.00 METoken from account[0] to account[1], by calling the contract's transfer function:

```
truffle(ganache)> METoken.deployed().then(instance => { instance.transfer(accounts[1],
100000) })
undefined
truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(accounts[0]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 9, c: [ 2099900000 ] }

undefined
truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(accounts[1]).then(console.log) })
undefined
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

TIP METoken has 2 decimals of precision, meaning that 1 METoken is 100 units in the contract. When we transfer 1000 METoken, we specify the value as 100,000 in the transfer function.

As you can see, in the console, account[0] now has 20,999,000 MET, and account[1] has 1000 MET.

If you switch to the ganache graphical user interface, you will see the transaction that called the transfer function:

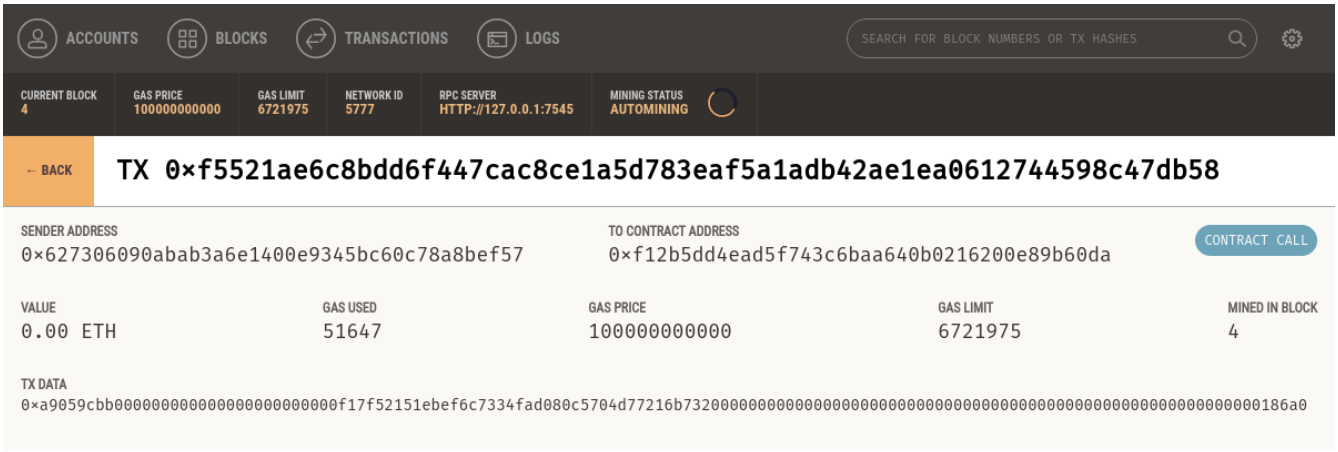

Figure 37. METoken transfer on Ganache

Sending ERC20 tokens to contract addresses

So far we've setup an ERC20 token and transferred from one account to another. All the accounts we used for these demonstrations are externally-owned accounts (EOAs), meaning they are controlled by a private key, not a contract. What happens if we send MET to a contract address? Let's find out!

First, let's deploy another contract into our test environment. For this example, we will use our first contract Faucet.sol. Let's add it to the METoken project by copying it to the contracts directory. Our directory should look like this:

```
METoken/
 ├──── contracts
 │     ├──── Faucet.sol
 │     ├──── METoken.sol
 │     └──── Migrations.sol
```

We'll also add a migration, to deploy Faucet separately from METoken:

```
var Faucet = artifacts.require("Faucet");

module.exports = function(deployer) {
  // Deploy the Faucet contract as our only task
  deployer.deploy(Faucet);
};
```

Let's compile and migrate the contracts, from the truffle console:

```
$ truffle console --network ganache
truffle(ganache)> compile
Compiling ./contracts/Faucet.sol...
Writing artifacts to ./build/contracts

truffle(ganache)> migrate
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x89f6a7bd2a596829c60a483ec99665c7af71e68c77a417fab503c394fcd7a0c9
  Migrations: 0xa1ccce36fb823810e729dce293b75f40fb6ea9c9
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0x28d0da26f48765f67e133e99dd275fac6a25fdfec6594060fd1a0e09a99b44ba
  METoken: 0x7d6bf9d5914d37bcba9d46df7107e71c59f3791f
Saving artifacts...
Running migration: 3_deploy_faucet.js
  Deploying Faucet...
  ... 0x6fbf283bcc97d7c52d92fd91f6ac02d565f5fded483a6a0f824f66edc6fa90c3
  Faucet: 0xb18a42e9468f7f1342fa3c329ec339f254bc7524
Saving artifacts...
```

Great. Now let's send some MET to the Faucet contract:

```
truffle(ganache)> METoken.deployed().then(instance => {
instance.transfer(Faucet.address, 100000) })
truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(Faucet.address).then(console.log)})
truffle(ganache)> BigNumber { s: 1, e: 5, c: [ 100000 ] }
```

Alright, we have transferred 1000 MET to the Faucet contract. Now, how do we withdraw from the Faucet?

Remember, Faucet.sol is a pretty simple contract. It only has one function, withdraw, which is for withdrawing *ether*. It doesn't have a function for withdrawing MET, or any other ERC20 token. If we use withdraw it will try to send ether, but since the faucet doesn't have a balance of ether yet, it will fail. The METoken contract knows that Faucet has a balance, but the only way that it can transfer that balance is if it receives a transfer call from the address of the contract. Somehow we need to make the Faucet contract call the transfer function in METoken.

If you're wondering what to do next, don't. There is no solution to this problem. The MET sent to Faucet is stuck, forever. Only the Faucet contract can transfer it, and the Faucet contract doesn't have code to call the transfer function of an ERC20 token contract.

Perhaps you anticipated this problem. Most likely, you didn't. In fact, neither did hundreds of Ethereum users who accidentally transferred various tokens to contracts that didn't have any

ERC20 capability. According to some estimates, tokens worth more than $2.5 million USD have been "stuck" like this and are lost forever.

One of the ways that users of ERC20 tokens can inadvertently lose their tokens in a transfer, is when they attempt to transfer to an exchange or another service. They copy an Ethereum address from the website of an exchange, thinking they can simply send tokens to it. However, many exchanges publish receiving addresses that are actually contracts! These contracts serve a number of different functions, most often sweeping all funds sent to them to a "cold storage" or another centralized wallet. Despite the many warnings saying "do not send tokens to this address", lots of tokens are lost this way.

Demonstrating the approve & transferFrom workflow

Our Faucet contract couldn't handle ERC20 tokens. Sending tokens to it, using the transfer function results in the loss of those tokens. Let's rewrite the contract and make it handle ERC20 tokens. Specifically, we will turn it into a faucet that gives out MET to anyone who asks.

For this example, we make a copy of the truffle project directory, call it METoken_METFaucet, initialize truffle, npm, install OpenZeppelin dependencies and copy the METoken.sol contract. See our first example Launching our own ERC20 token for the detailed instructions.

Now, let's create a new faucet contract, call it METFaucet.sol. It will look like this:

METFaucet.sol: a faucet for METoken

```
include::code/METoken_METFaucet/contracts/METFaucet.sol
```

We've made quite a few changes to the basic faucet example. Since METFaucet will use the transferFrom function in METoken, it will need two additional variables. One will hold the address of the deployed METoken contract. The other will hold the address of the owner of MET who will provide approval the faucet withdrawals. The METFaucet will call METoken.transferFrom and instruct it to move MET from the owner to the address where the faucet withdrawal request came from.

We declare these two variables here:

```
StandardToken public METoken;
address public METOwner;
```

Since our faucet needs to be initialized with the correct addresses for METoken and METOwner we need to declare a custom constructor:

```
// METFaucet constructor, provide the address of METoken contract and
// the owner address we will be approved to transferFrom
function METFaucet(address _METoken, address _METOwner) public {

    // Initialize the METoken from the address provided
    METoken = StandardToken(_METoken);
    METOwner = _METOwner;
}
```

The next change is to the withdraw function. Instead of calling transfer, METFaucet uses the transferFrom function in METoken and asks METoken to transfer MET to the faucet recipient:

```
// Use the transferFrom function of METoken
METoken.transferFrom(METOwner, msg.sender, withdraw_amount);
```

Finally, since our faucet no longer sends ether, we should probably prevent anyone from sending ether to METFaucet, as we wouldn't want it to get stuck. We change the fallback payable function to reject incoming ether, using the revert function to revert any incoming payments:

```
// REJECT any incoming ether
function () public payable { revert(); }
```

Now that our METFaucet.sol code is ready, we need to modify the migration script to deploy it. This migration script will be a bit more complex, as METFaucet depends on the address of METoken. We will use a JavaScript promise to deploy the two contracts in sequence. Create 2_deply_contracts.js as follows:

[[2_deploy_contracts]]

```
var METoken = artifacts.require("METoken");
var METFaucet = artifacts.require("METFaucet");
var owner = web3.eth.accounts[0];

module.exports = function(deployer) {

    // Deploy the METoken contract first
    deployer.deploy(METoken, {from: owner}).then(function() {
        // then deploy METFaucet and pass the address of METoken
        // and the address of the owner of all the MET who will approve METFaucet
        return deployer.deploy(METFaucet, METoken.address, owner);
    });
}
```

Now, we can test everything in the truffle console. First, we use migrate to deploy the contracts. When METoken is deployed it will allocate all the MET to the account that created it, web3.eth.accounts[0]. Then, we call the approve function in METoken to approve METFaucet to

send up to 1000 MET on behalf of web3.eth.accounts[0]. Finally, to test our faucet, we call METFaucet.withdraw from web3.eth.accounts[1] and try to withdraw 10 MET. Here are the console commands:

```
$ truffle console --network ganache
truffle(ganache)> migrate
Using network 'ganache'.

Running migration: 1_initial_migration.js
  Deploying Migrations...
  ... 0x79352b43e18cc46b023a779e9a0d16b30f127bfa40266c02f9871d63c26542c7
  Migrations: 0xaa588d3737b611bafd7bd713445b314bd453a5c8
Saving artifacts...
Running migration: 2_deploy_contracts.js
  Replacing METoken...
  ... 0xc42a57f22cddf95f6f8c19d794c8af3b2491f568b38b96fef15b13b6e8bfff21
  METoken: 0xf204a4ef082f5c04bb89f7d5e6568b796096735a
  Replacing METFaucet...
  ... 0xd9615cae2fa4f1e8a377de87f86162832cf4d31098779e6e00df1ae7f1b7f864
  METFaucet: 0x75c35c980c0d37ef46df04d31a140b65503c0eed
Saving artifacts...
truffle(ganache)> METoken.deployed().then(instance => {
instance.approve(METFaucet.address, 100000) })
truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(web3.eth.accounts[1]).then(console.log) })
truffle(ganache)> BigNumber { s: 1, e: 0, c: [ 0 ] }
truffle(ganache)> METFaucet.deployed().then(instance => { instance.withdraw(1000,
{from:web3.eth.accounts[1]}) } )
truffle(ganache)> METoken.deployed().then(instance => {
instance.balanceOf(web3.eth.accounts[1]).then(console.log) })
truffle(ganache)> BigNumber { s: 1, e: 3, c: [ 1000 ] }
```

As you can see from the results, we can use the approve and transferFrom workflow to authorize one contract to transfer tokens defined in another token. If properly used, ERC20 tokens can be used by externally-owned addresses and other contracts.

However, the burden of managing ERC20 tokens correctly is pushed to the user interface. If a user incorrectly attempts to transfer ERC20 tokens to a contract address and that contract is not equipped to receive ERC20 tokens, the tokens will be lost.

Issues with ERC20 tokens

The adoption of the ERC20 token standard has been truly explosive. Thousands of tokens have been launched, both to experiment with new capabilities and to raise funds in various "crowdfund" auctions and Initial Coin Offerings (ICOs). However there are some potential pitfalls, as we saw with the issue of transferring tokens to contract addresses.

One of the less obvious issues with ERC20 tokens is that they expose subtle differences between tokens and ether itself. Where ether is transferred by a transaction which has a recipient address as

its destination, token transfers occur within the *specific token contract state* and have the token contract as their destination, not the recipient's address. The token contract tracks balances and issues events. In a token transfer, no transaction is actually sent to the recipient of the token. Instead, the recipient's address is added to a map within the token contract itself. A transaction sending ether to an address changes the state of an address. A transaction transferring a token to an address only changes the state of the token contract, not the state of the recipient address. Even a wallet that has support for ERC20 tokens does not become aware of a token balance unless the user explicitly adds a specific token contract to "watch". Some wallets watch the most popular token contracts to detect balances held by addresses they control, but that's limited to a small fraction of the available ERC20 contracts.

In fact, it's unlikely that a user would *want* to track all balances in all possible ERC20 token contracts. Many ERC20 tokens are more like email spam than usable tokens. They automatically create balances for accounts that have ether activity, to attract users. If you have an Ethereum address with a long history of activity, especially if it was created in the presale, you will find it full of "junk" tokens that appeared out of nowhere. Of course, the address isn't really full of tokens, it's the token contracts that have your address in them. You only see these balances if these tokens contracts are being watched by the block explorer or wallet you use to view your address.

Tokens don't behave the same way as ether. Ether is sent with the send function and accepted by any payable function in a contract or any externally owned address. Tokens are sent using transfer or approve & transferFrom functions that exist only in the ERC20 contract, and do not (at least in ERC20) trigger any payable functions in a recipient contract. Tokens are meant to function just like a cryptocurrency such as ether, but they come with certain subtle distinctions that break that illusion.

Consider another issue. To send ether, or use any Ethereum contract you need ether to pay gas. To send tokens, you *also need ether*. You cannot pay for a transaction's gas with a token and the token contract can't pay the gas for you. This can cause some rather strange user experiences. For example, let's say you use an exchange or Shapeshift to convert some bitcoin to a token. You "receive" the token in a wallet that tracks that token's contract and shows your balance. It looks the same as any of the other cryptocurrencies you have in your wallet. Now try sending the token and your wallet will inform you that you need ether to do that. You might be confused - after all you didn't need ether to receive the token. Perhaps you have no ether. Perhaps you didn't even know the token was an ERC20 token on Ethereum, maybe you thought it was a cryptocurrency with its own blockchain. The illusion just broke.

Some of these issues are specific to ERC20 tokens. Others are more general issues that relate to abstraction and interface boundaries within Ethereum. Some can be solved by changing the token interface, others may need changes to fundamental structures within Ethereum (such as the distinction between EOAs and contracts, and between transactions and messages). Some may not be "solvable" exactly and may require user interface design to hide the nuances and make the user experience consistent regardless of the underlying distinctions.

In the next sections we will look at various proposals that attempt to address some of these issues.

ERC223 - a proposed token contract interface standard

The ERC223 proposal attempts to solve the problem of inadvertent transfer of tokens to a contract

(that may or may not support tokens) by detecting whether the destination address is a contract or not. ERC223 requires that contracts designed to accept tokens implement a function named tokenFallback. If the destination of a transfer is a contract and the contract does not have support for tokens (i.e. does not implement tokenFallback), the transfer fails.

To detect whether the destination address is a contract, the ERC223 reference implementation uses a small segment of inline bytecode, in a rather creative way:

```
function isContract(address _addr) private view returns (bool is_contract) {
    uint length;
    assembly {
          //retrieve the size of the code on target address, this needs assembly
          length := extcodesize(_addr)
    }
    return (length>0);
}
```

You can see the discussion around the ERC223 proposal here:

https://github.com/ethereum/EIPs/issues/223

The ERC223 contract interface specification is:

```
interface ERC223Token {
  uint public totalSupply;
  function balanceOf(address who) public view returns (uint);

  function name() public view returns (string _name);
  function symbol() public view returns (string _symbol);
  function decimals() public view returns (uint8 _decimals);
  function totalSupply() public view returns (uint256 _supply);

  function transfer(address to, uint value) public returns (bool ok);
  function transfer(address to, uint value, bytes data) public returns (bool ok);
  function transfer(address to, uint value, bytes data, string custom_fallback) public
returns (bool ok);

  event Transfer(address indexed from, address indexed to, uint value, bytes indexed
data);
}
```

ERC223 is not widely implemented and there is some debate in the ERC discussion thread about backwards compatibility and trade-offs between implementing changes at the contract interface level versus the user interface. The debate continues.

ERC777 - a proposed token contract interface standard

Another proposal for an improved token contract standard is ERC777. This proposal has several

goals, including:

- To offer an ERC20 compatibility interface
- To transfer tokens using a send function, similar to ether transfers
- To be compatible with ERC820 for token contract registration
- Contracts and addresses can control which tokens they send through a tokensToSend function that is called prior to sending
- Contracts and addresses are notified by calling a tokensReceived function in the recipient
- Token transfer transactions contain metadata in a userData and operatorData field
- To operate in the same way, whether sending to a contract or EOA

The details and ongoing discussion on ERC777 can be found here: https://github.com/ethereum/EIPs/issues/777

The ERC777 contract interface specification is:

```
interface ERC777Token {
    function name() public constant returns (string);
    function symbol() public constant returns (string);
    function totalSupply() public constant returns (uint256);
    function granularity() public constant returns (uint256);
    function balanceOf(address owner) public constant returns (uint256);

    function send(address to, uint256 amount) public;
    function send(address to, uint256 amount, bytes userData) public;

    function authorizeOperator(address operator) public;
    function revokeOperator(address operator) public;
    function isOperatorFor(address operator, address tokenHolder) public constant
returns (bool);
    function operatorSend(address from, address to, uint256 amount, bytes userData,
bytes operatorData) public;

    event Sent(address indexed operator, address indexed from, address indexed to,
uint256 amount, bytes userData, bytes operatorData);
    event Minted(address indexed operator, address indexed to, uint256 amount, bytes
operatorData);
    event Burned(address indexed operator, address indexed from, uint256 amount, bytes
userData, bytes operatorData);
    event AuthorizedOperator(address indexed operator, address indexed tokenHolder);
    event RevokedOperator(address indexed operator, address indexed tokenHolder);
}
```

A reference implementation of ERC777 is linked in the proposal. ERC777 depends on a parallel proposal for a registry contract, specified in ERC820. Some of the debate on ERC777 is about the complexity of adopting two big changes at once: a new token standard and a registry standard. The discussion continues.

ERC721 - non-fungible token (deed) standard

All the token standards we have looked at so far are *fungible* tokens, meaning that each unit of a token is entirely interchangeable. The ERC20 token standard only tracks the final balance of each account and does not (explicitly) track the provenance of any token.

The ERC721 proposal is for a standard for *non-fungible* tokens, also known as *deeds*.

From the Oxford Dictionary:

```
deed: A legal document that is signed and delivered, especially one regarding the
ownership of property or legal rights.
```

The use of the word deed is intended to reflect the "ownership of property" part, even though these are not recognized as "legal documents" in any jurisdiction, at least not currently.

Non-fungible tokens track ownership of a unique thing. The thing owned can be a digital item, such as a game item, or digital collectible. Or, the thing can be a physical item whose ownership is tracked by a token, such as a house, a car, artwork. A deed could also represent things with negative value, such as loans (debt), liens, easements, etc. The ERC721 standard places no limitation or expectation on the nature of the thing whose ownership is tracked by a deed, only that it can be uniquely identified, which in the case of this standard is achieved by a 256-bit identifier.

The details of the standard and discussion are tracked in two different GitHub locations:

Initial proposal: https://github.com/ethereum/EIPs/issues/721

Continued discussion: https://github.com/ethereum/EIPs/pull/841

To grasp the basic difference between ERC20 and ERC721, it is sufficient to look at the internal data structure used in ERC721:

```
// Mapping from deed ID to owner
mapping (uint256 => address) private deedOwner;
```

Whereas ERC20 tracks the balances that belong to each owner, with the owner being the primary key of the mapping, ERC721 tracks each deed ID and who owns it, with the deed ID being the primary key of the mapping. From this basic difference flow all the properties of a non-fungible token.

The ERC721 contract interface specification is:

```
interface ERC721 /* is ERC165 */ {
    event Transfer(address indexed _from, address indexed _to, uint256 _deedId);
    event Approval(address indexed _owner, address indexed _approved, uint256
_deedId);
    event ApprovalForAll(address indexed _owner, address indexed _operator, bool
_approved);

    function balanceOf(address _owner) external view returns (uint256 _balance);
    function ownerOf(uint256 _deedId) external view returns (address _owner);
    function transfer(address _to, uint256 _deedId) external payable;
    function transferFrom(address _from, address _to, uint256 _deedId) external
payable;
    function approve(address _approved, uint256 _deedId) external payable;
    function setApprovalForAll(address _operateor, boolean _approved) payable;
    function supportsInterface(bytes4 interfaceID) external view returns (bool);
}
```

ERC721 also supports two **optional** interfaces, one for metadata and one for enumeration of deeds and owners.

The ERC721 optional interface for metadata is:

```
interface ERC721Metadata /* is ERC721 */ {
    function name() external pure returns (string _name);
    function symbol() external pure returns (string _symbol);
    function deedUri(uint256 _deedId) external view returns (string _deedUri);
}
```

The ERC721 optional interface for enumeration is:

```
interface ERC721Enumerable /* is ERC721 */ {
    function totalSupply() external view returns (uint256 _count);
    function deedByIndex(uint256 _index) external view returns (uint256 _deedId);
    function countOfOwners() external view returns (uint256 _count);
    function ownerByIndex(uint256 _index) external view returns (address _owner);
    function deedOfOwnerByIndex(address _owner, uint256 _index) external view returns
(uint256 _deedId);
}
```

Token standards

In this section, we've reviewed several proposed standards and a couple of widely-deployed standards for token contracts. What exactly do these standards do? Should you use these standards? How should you use them? Should you add functionality beyond these standards? Which standards should you use? We will examine all those questions next.

What are token standards? What is their purpose?

Token standards are a *minimum* specification for an implementation. What that means is that in order to be compliant with, say ERC20, you need to at minimum, implement the functions and behavior specified by ERC20. You are also free to *add* to the functionality by implementing functions that are not part of the standard.

The primary purpose of these standards is to encourage *interoperability* between contracts. Thus, all wallets, exchanges, user interfaces and other infrastructure components can *interface* in a predictable manner with any contract that follows the specification.

The standards are meant to be *descriptive*, rather than *prescriptive*. How you choose to implement those functions is up to you - the internal function of the contract is not relevant to the standard. They have some functional requirements, which govern the behavior under specific circumstances, but they do not prescribe an implementation. An example of this is the behavior of a transfer function if the value is set to zero.

Should you use these standards?

Given all these standards, each developer faces a dilemma: use the existing standards or innovate beyond the restrictions they impose?

This dilemma is not easy to resolve. Standards necessarily restrict your ability to innovate, by creating a narrow "rut" that you have to follow. On the other hand, the basic standards have emerged from the experience with hundreds of applications and often fit well with 99% of the use-cases.

As part of this consideration is an even bigger issue: the value of interoperability and broad adoption. If you choose to use an existing standard, you gain the value of all the systems designed to work with that standard. If you choose to depart from the standard, you have to consider the cost of building all of the support infrastructure on your own, or persuading others to support your implementation as a new standard. The tendency to forge your own path and ignore existing standards is known as "Not Invented Here" and is antithetical to the open source culture. On the other hand, progress and innovation depends on departing from tradition sometimes. It's a tricky choice, so consider it carefully!

Wikipedia "Not Invented Here" (https://en.wikipedia.org/wiki/Not_invented_here)

```
Not invented here is a stance adopted by social, corporate, or institutional cultures
that avoid using or buying already existing products, research, standards, or
knowledge because of their external origins and costs, such as royalties.
```

Security by maturity

Beyond the choice of standard, there is the parallel choice of *implementation*. When you decide to use a standard, such as ERC20, you have to then decide how to implement a compatible token. There are a number of existing "reference" implementations that are broadly used in the Ethereum ecosystem. Or you could write your own from scratch. Again, this choice represents a dilemma that

can have serious security implications.

Existing implementations are "battle tested". While it is impossible to prove that they are secure, many of them underpin millions of dollars of tokens. They have been attacked, repeatedly and vigorously. So far, no significant vulnerabilities have been discovered. Writing your own is not easy - there are many subtle ways that a contract can be compromised. It is much safer to use a well-tested broadly-used implementation. In our examples above, we used the OpenZeppelin implementation of the ERC20 standard, as this implementation is security focused from the ground up.

If you use an existing implementation you can also extend it. Again, be careful with this impulse. Complexity is the enemy of security. Every single line of code you add expands the *attack surface* of your contract and could represent a vulnerability lying in wait. You may not notice a problem until you put a lot of value on top of the contract and someone breaks it.

Extensions to token interface standards

The token standards discussed in this section start with a very minimal interface, with limited functionality. Many projects have created extended implementations, to support features that they need for their application. Some of these include:

Owner Control

Specific addresses, or set of addresses (multi-signature) are given special capabilities, such as blacklisting, whitelisting, minting, recovery etc.

Burning

A token burn is when tokens are deliberately destroyed by transfer to an unspendable address or by erasing a balance and reducing the supply.

Minting

The ability to add to the total supply of tokens, at a predictable rate, or by "fiat" of the creator of the token.

Crowdfunding

The ability to offer tokens for sale, for example through an auction, market sale, reverse-auction, etc.

Caps

Pre-defined and immutable limits on the total supply, the opposite of the "minting" feature.

Recovery "Back Doors"

Functions to recover funds, reverse transfers, or dismantle the token that can be activated by a designated address or set of addresses (multi-signature).

Whitelisting

The ability to restrict token transfers only to listed addresses. Most commonly used to offer tokens to "accredited investors" after vetting by the rules of different jurisdictions. There is usually a mechanism for updating the whitelist.

Blacklisting

The ability to restrict token transfers by disallowing specific addresses. There is usually a function for updating the blacklist.

There are some reference implementations for many of these functions, for example in the OpenZeppelin library. Some of these are use-case specific and only implemented in a few tokens. There are, as of now, no widely accepted standards for the interfaces to these functions.

As previously discussed, the decision to extend a token standard with additional functionality represents a tradeoff between innovation/risk and interoperability/security.

Tokens and ICOs

Tokens have become an explosive development in the Ethereum ecosystem. It is likely that they will be a very important, foundational, component of all smart-contract platforms like Ethereum.

Nevertheless, the importance and future impact of these standards should not be confused with an endorsement of the current token offerings. As in any early stage technology, the first wave of products and companies will almost all fail, and some will fail spectacularly. Many of the tokens on offer in Ethereum today are barely disguised scams, pyramid schemes and money grabs.

The trick is to separate the long-term vision and impact of this technology, which is likely to be huge, from the short term bubble of token ICOs, which is rife with fraud. Both can be true at the same time. The token standards and platform will survive the current token mania, and then they will likely change the world.

Appendix A: Ethereum Standards

Ethereum Improvement Proposals (EIPs)

https://github.com/ethereum/EIPs/blob/master/https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1.md

From EIP-1:

```
EIP stands for Ethereum Improvement Proposal. An EIP is a design document providing
information to the Ethereum community, or describing a new feature for Ethereum or its
processes or environment. The EIP should provide a concise technical specification of
the feature and a rationale for the feature. The EIP author is responsible for
building consensus within the community and documenting dissenting opinions.
```

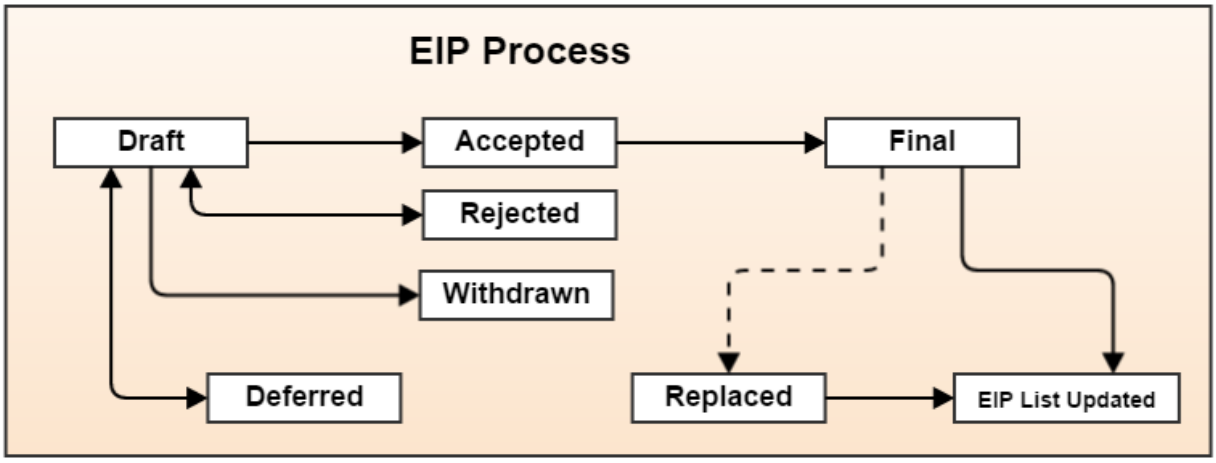

Figure 38. Ethereum Improvement Proposal Workflow

Ethereum Request for Comments (ERCs)

Bitcoin Improvement Proposals (BIPs)

Standards from bitcoin used in Ethereum

Table of Most Important EIPs and ERCs

Table 2. Important EIPs and ERCs

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-1	EIP Purpose and Guidelines	Martin Becze, Hudson Jameson	Meta	Final	
EIP-2	Homestead Hard-fork Changes	Vitalik Buterin	Core	Final	
EIP-5	Gas Usage for `RETURN` and `CALL`	Christian Reitwiessner	Core	Draft	
EIP-6	Renaming Suicide Opcode	Hudson Jameson	Interface	Final	
EIP-7	`DELEGATECALL`	Vitalik Buterin	Core	Final	
EIP-8	devp2p Forward Compatibility Requirements for Homestead	Felix Lange	Networking	Final	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-20	ERC-20 Token Standard. Describes standard functions a token contract may implement to allow DApps and Wallets to handle tokens across multiple interfaces/DApps. Methods include: `totalSupply()`, `balanceOf(address)`, `transfer`, `transferFrom`, `approve`, `allowance`. Events include: `Transfer` (triggered when tokens are transferred), `Approval` (triggered when `approve` is called).	Fabian Vogelsteller, Vitalik Buterin	ERC	Final	Frontier
EIP-55	ERC-55 Mixed-case checksum address encoding	Vitalik Buterin	ERC	Final	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-86	Setting the stage for "abstracting out" account security, and allowing users creation of "account contracts" toward a model where in the long-term all accounts are contracts that can pay for gas, and users are free to defined their own security model (that perform any desired signature verification and nonce checks instead of using the in-protocol mechanism where ECDSA and default nonce scheme are the only "standard" way to secure an account, which is currently hard-coded into transaction processing).	Vitalik Buterin	Core	Deferred (to be replaced)	Constantinople

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-96	Setting the Blockhash and state root refactoring to store blockhashes in the state to reduce protocol complexity and need for client implementation complexity necessary to process the `BLOCKHASH` opcode. Extends range of how far back blockhash checking may go, with the side effect of creating direct links between blocks with very distant block numbers to facilitate much more efficient initial Light Client syncing.	Vitalik Buterin	Core	Deferred	Constantinople
EIP-100	Change formula that computes the difficulty of a block (difficulty adjustment algorithm) to target mean block time and take uncles into account.	Vitalik Buterin	Core	Final	Metropolis Byzantinium

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-101	Serenity Currency and Crypto Abstraction. Abstracting Ether up a level with the benefit of allowing Ether and sub-Tokens to be treated similarly by contracts, reducing level of indirection required for custom-policy accounts such as Multisigs, and purifying the underlying Ethereum protocol by reducing the minimal consensus implementation complexity	Vitalik Buterin	Active	Serenity feature	Serenity Casper
EIP-105	"Sharding scaffolding" EIP to allow Ethereum transactions to be parallelised using a binary tree sharding mechanism, and to set the stage for a later sharding scheme. Research in progress: https://github.com/ethereum/sharding	Vitalik Buterin	Active	Serenity feature	Serenity Casper
EIP-137	Ethereum Domain Name Service - Specification	Nick Johnson	ERC	Final	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-140	Add `REVERT` opcode instruction, which stops execution and rolls back the EVM execution state changes without consuming all provided gas (instead the contract only has to pay for memory) or losing logs, and returning to the caller a pointer to the memory location with the error code or message.	Alex Beregszaszi, Nikolai Mushegian	Core	Final	Metropolis Byzantinium
EIP-141	Designated invalid EVM instruction	Alex Beregszaszi	Core	Final	
EIP-145	Bitwise shifting instructions in EVM	Alex Beregszaszi, Paweł Bylica	Core	Deferred	
EIP-150	Gas cost changes for IO-heavy operations	Vitalik Buterin	Core	Final	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-155	Simple Replay Attack Protection. Replay Attack allows any transaction using a pre-EIP155 Ethereum Node or Client to become signed so it is valid and executed on both the Ethereum and Ethereum Classic chains.	Vitalik Buterin	Core	Final	Homestead
EIP-158	State clearing	Vitalik Buterin	Core	Superseded	
EIP-160	EXP cost increase	Vitalik Buterin	Core	Final	
EIP-161	State trie clearing (invariant-preserving alternative[EIP-161]	Gavin Wood	Core	Final	
EIP-162	ERC-162 ENS support for reverse resolution of Ethereum addresses	Maurelian, Nick Johnson	ERC	Final	
EIP-165	ERC-165 Standard Interface Detection	Christian Reitwiessner	Interface	Draft	
EIP-170	Contract code size limit	Vitalik Buterin	Core	Final	
EIP-181	ERC-181 ENS support for reverse resolution of Ethereum addresses	Nick Johnson	ERC	Final	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-190	ERC-190 Ethereum Smart Contract Packaging Standard	Merriam, Coulter, Erfurt, Catalano, Matias	ERC	Final	
EIP-196	Precompiled contracts for addition and scalar multiplication operations on the elliptic curve alt_bn128, which are required in order to perform zkSNARK verification within the block gas limit	Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-197	Precompiled contracts for optimal Ate pairing check of a pairing function on a specific pairing-friendly elliptic curve alt_bn128 and is combined with EIP 196	Vitalik Buterin, Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-198	Precompile to support big integer modular exponentiation enabling RSA signature verification and other cryptographic applications	Vitalik Buterin	Core	Final	Metropolis Byzantinium

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-211	New opcodes: `RETURNDATASIZE` and `RETURNDATACOPY`. Support for returning variable-length values inside the EVM with simple gas charging and minimal change to calling opcodes using new opcodes `RETURNDATASIZE` and `RETURNDATACOPY`. Handles similar to existing `calldata`, whereby after a call, return data is kept inside a virtual buffer from which the caller can copy it (or parts thereof) into memory, and upon the next call, the buffer is overwritten.	Christian Reitwiessner	Core	Final	Metropolis Byzantinium

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-214	New opcode: `STATICCALL`. Permits non-state-changing calls to itself or other contracts whilst disallowing any modifications to state during the call (and its sub-calls, if present) to increase smart contract security and assure developers that re-entrancy bugs cannot arise from the call. Calls the child with `STATIC` flag set `true` for execution of child, causing exception to be thrown upon any attempts to make state-changing operations inside an execution instance where `STATIC` is set `true`, and resets flag once call returns.	Vitalik Buterin, Christian Reitwiessner	Core	Final	Metropolis Byzantinium
EIP-225	Rinkeby Testnet using Proof-of-Authority where blocks only mined by trusted signers				Homestead

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-234	Add `blockHash` to JSON-RPC filter options	Micah Zoltu	Interface	Draft	
EIP-615	Subroutines and Static Jumps for the EVM	Greg Colvin	Core	Draft	
EIP-616	SIMD Operations for the EVM	Greg Colvin	Core	Draft	
EIP-681	ERC-681 URL Format for Transaction Requests	Daniel A. Nagy	Interface	Draft	
EIP-649	Metropolis Difficulty Bomb Delay and Block Reward Reduction - Delay of the Ice Age (aka the Difficulty Bomb by 1 year), and reduction of the block reward from 5 to 3 ether.	Afri Schoedon, Vitalik Buterin	Core	Final	Metropolis Byzantinium

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-658	Embedding transaction status code in receipts. Fetch and embed status field indicative of success or failure state to transaction receipts for callers, as was no longer able to assume the transaction failed if and only if (iff) it consumed all gas after the introduction of the `REVERT` opcode in EIP-140.	Nick Johnson	Core	Final	Metropolis Byzantinium
EIP-706	DEVp2p snappy compression	Péter Szilágyi	Networking	Final	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-721	ERC-721 Non-Fungible Token (NFT) Standard. It is a standard API that would allow smart contracts to operate as unique tradable non-fungible tokens (NFT) that may be tracked in standardised wallets and traded on exchanges as assets of value, similar to ERC-20. CryptoKitties was the first popularly-adopted implementation of a digital NFT in the Ethereum ecosystem.	William Entriken, Dieter Shirley, Jacob Evans, Nastassia Sachs	Standard	Draft	
EIP-758	Subscriptions and filters for transaction return data	Jack Peterson	Interface	Draft	
EIP-801	ERC-801 Canary Standard	ligi	Interface	Draft	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-827	ERC-827 A extension of the standard interface ERC20 for tokens with methods that allows the execution of calls inside transfer and approvals. This standard provides basic functionality to transfer tokens, as well as allow tokens to be approved so they can be spent by another on-chain third party. Also it allows to execute calls on transfers and approvals.	Augusto Lemble	ERC	Draft	

EIP/ERC #	Title	Author	Layer	Status	Created
EIP-930	ERC-930 The ES (Eternal Storage) contract is owned by an address that have write permissions. The storage is public, which means everyone has read permissions. It store the data on mappings, using one mapping per type of variable. The use of this contract allows the developer to migrate the storage easily to another contract if needed.	Augusto Lemble	ERC	Draft	

Appendix B: Ethereum EVM Opcodes and gas consumption

This appendix is based on the consolidation work done by the people of https://github.com/trailofbits/evm-opcodes as a Reference for Ethereum VM (EVM) Opcodes and Instruction information with the following LICENSE https://github.com/trailofbits/evm-opcodes/blob/master/LICENSE.

Table

Opcode	Name	Description	Extra Info	Gas
`0x00`	STOP	Halts execution	-	0
`0x01`	ADD	Addition operation	-	3
`0x02`	MUL	Multiplication operation	-	5

Opcode	Name	Description	Extra Info	Gas
0x03	SUB	Subtraction operation	-	3
0x04	DIV	Integer division operation	-	5
0x05	SDIV	Signed integer division operation (truncated)	-	5
0x06	MOD	Modulo remainder operation	-	5
0x07	SMOD	Signed modulo remainder operation	-	5
0x08	ADDMOD	Modulo addition operation	-	8
0x09	MULMOD	Modulo multiplication operation	-	8
0x0a	EXP	Exponential operation	-	10***
0x0b	SIGNEXTEND	Extend length of two’s complement signed integer	-	5
0x0c - 0x0f	Unused	Unused	-	
0x10	LT	Less-than comparison	-	3
0x11	GT	Greater-than comparison	-	3
0x12	SLT	Signed less-than comparison	-	3
0x13	SGT	Signed greater-than comparison	-	3
0x14	EQ	Equality comparison	-	3
0x15	ISZERO	Simple not operator	-	3
0x16	AND	Bitwise AND operation	-	3
0x17	OR	Bitwise OR operation	-	3
0x18	XOR	Bitwise XOR operation	-	3
0x19	NOT	Bitwise NOT operation	-	3

Opcode	Name	Description	Extra Info	Gas
0x1a	BYTE	Retrieve single byte from word	-	3
0x20	SHA3	Compute Keccak-256 hash	-	30
0x21 - 0x2f	Unused	Unused	-	
0x30	ADDRESS	Get address of currently executing account	-	2
0x31	BALANCE	Get balance of the given account	-	400
0x32	ORIGIN	Get execution origination address	-	2
0x33	CALLER	Get caller address	-	2
0x34	CALLVALUE	Get deposited value by the instruction/transaction responsible for this execution	-	2
0x35	CALLDATALOAD	Get input data of current environment	-	3
0x36	CALLDATASIZE	Get size of input data in current environment	-	2
0x37	CALLDATACOPY	Copy input data in current environment to memory	-	3
0x38	CODESIZE	Get size of code running in current environment	-	2
0x39	CODECOPY	Copy code running in current environment to memory	-	3
0x3a	GASPRICE	Get price of gas in current environment	-	2
0x3b	EXTCODESIZE	Get size of an account's code	-	700
0x3c	EXTCODECOPY	Copy an account's code to memory	-	700

Opcode	Name	Description	Extra Info	Gas
0x3d	RETURNDATASIZE	Pushes the size of the return data buffer onto the stack	EIP 211	2
0x3e	RETURNDATACOPY	Copies data from the return data buffer to memory	EIP 211	3
0x3f	Unused	-	-	
0x40	BLOCKHASH	Get the hash of one of the 256 most recent complete blocks	-	20
0x41	COINBASE	Get the block's beneficiary address	-	2
0x42	TIMESTAMP	Get the block's timestamp	-	2
0x43	NUMBER	Get the block's number	-	2
0x44	DIFFICULTY	Get the block's difficulty	-	2
0x45	GASLIMIT	Get the block's gas limit	-	2
0x46 - 0x4f	Unused	-	-	
0x50	POP	Remove word from stack	-	2
0x51	MLOAD	Load word from memory	-	3
0x52	MSTORE	Save word to memory	-	3*
0x53	MSTORE8	Save byte to memory	-	3
0x54	SLOAD	Load word from storage	-	200
0x55	SSTORE	Save word to storage	-	0*
0x56	JUMP	Alter the program counter	-	8
0x57	JUMPI	Conditionally alter the program counter	-	10

Opcode	Name	Description	Extra Info	Gas
0x58	GETPC	Get the value of the program counter prior to the increment	-	2
0x59	MSIZE	Get the size of active memory in bytes	-	2
0x5a	GAS	Get the amount of available gas, including the corresponding reduction the amount of available gas	-	2
0x5b	JUMPDEST	Mark a valid destination for jumps	-	1
0x5c - 0x5f	Unused	-	-	
0x60	PUSH1	Place 1 byte item on stack	-	3
0x61	PUSH2	Place 2-byte item on stack	-	3
0x62	PUSH3	Place 3-byte item on stack	-	3
0x63	PUSH4	Place 4-byte item on stack	-	3
0x64	PUSH5	Place 5-byte item on stack	-	3
0x65	PUSH6	Place 6-byte item on stack	-	3
0x66	PUSH7	Place 7-byte item on stack	-	3
0x67	PUSH8	Place 8-byte item on stack	-	3
0x68	PUSH9	Place 9-byte item on stack	-	3
0x69	PUSH10	Place 10-byte item on stack	-	3
0x6a	PUSH11	Place 11-byte item on stack	-	3
0x6b	PUSH12	Place 12-byte item on stack	-	3
0x6c	PUSH13	Place 13-byte item on stack	-	3

Opcode	Name	Description	Extra Info	Gas
0x6d	PUSH14	Place 14-byte item on stack	-	3
0x6e	PUSH15	Place 15-byte item on stack	-	3
0x6f	PUSH16	Place 16-byte item on stack	-	3
0x70	PUSH17	Place 17-byte item on stack	-	3
0x71	PUSH18	Place 18-byte item on stack	-	3
0x72	PUSH19	Place 19-byte item on stack	-	3
0x73	PUSH20	Place 20-byte item on stack	-	3
0x74	PUSH21	Place 21-byte item on stack	-	3
0x75	PUSH22	Place 22-byte item on stack	-	3
0x76	PUSH23	Place 23-byte item on stack	-	3
0x77	PUSH24	Place 24-byte item on stack	-	3
0x78	PUSH25	Place 25-byte item on stack	-	3
0x79	PUSH26	Place 26-byte item on stack	-	3
0x7a	PUSH27	Place 27-byte item on stack	-	3
0x7b	PUSH28	Place 28-byte item on stack	-	3
0x7c	PUSH29	Place 29-byte item on stack	-	3
0x7d	PUSH30	Place 30-byte item on stack	-	3
0x7e	PUSH31	Place 31-byte item on stack	-	3
0x7f	PUSH32	Place 32-byte (full word) item on stack	-	3
0x80	DUP1	Duplicate 1st stack item	-	3

Opcode	Name	Description	Extra Info	Gas
0x81	DUP2	Duplicate 2nd stack item	-	3
0x82	DUP3	Duplicate 3rd stack item	-	3
0x83	DUP4	Duplicate 4th stack item	-	3
0x84	DUP5	Duplicate 5th stack item	-	3
0x85	DUP6	Duplicate 6th stack item	-	3
0x86	DUP7	Duplicate 7th stack item	-	3
0x87	DUP8	Duplicate 8th stack item	-	3
0x88	DUP9	Duplicate 9th stack item	-	3
0x89	DUP10	Duplicate 10th stack item	-	3
0x8a	DUP11	Duplicate 11th stack item	-	3
0x8b	DUP12	Duplicate 12th stack item	-	3
0x8c	DUP13	Duplicate 13th stack item	-	3
0x8d	DUP14	Duplicate 14th stack item	-	3
0x8e	DUP15	Duplicate 15th stack item	-	3
0x8f	DUP16	Duplicate 16th stack item	-	3
0x90	SWAP1	Exchange 1st and 2nd stack items	-	3
0x91	SWAP2	Exchange 1st and 3rd stack items	-	3
0x92	SWAP3	Exchange 1st and 4th stack items	-	3
0x93	SWAP4	Exchange 1st and 5th stack items	-	3
0x94	SWAP5	Exchange 1st and 6th stack items	-	3
0x95	SWAP6	Exchange 1st and 7th stack items	-	3

Opcode	Name	Description	Extra Info	Gas
0x96	SWAP7	Exchange 1st and 8th stack items	-	3
0x97	SWAP8	Exchange 1st and 9th stack items	-	3
0x98	SWAP9	Exchange 1st and 10th stack items	-	3
0x99	SWAP10	Exchange 1st and 11th stack items	-	3
0x9a	SWAP11	Exchange 1st and 12th stack items	-	3
0x9b	SWAP12	Exchange 1st and 13th stack items	-	3
0x9c	SWAP13	Exchange 1st and 14th stack items	-	3
0x9d	SWAP14	Exchange 1st and 15th stack items	-	3
0x9e	SWAP15	Exchange 1st and 16th stack items	-	3
0x9f	SWAP16	Exchange 1st and 17th stack items	-	3
0xa0	LOG0	Append log record with no topics	-	375
0xa1	LOG1	Append log record with one topic	-	750
0xa2	LOG2	Append log record with two topics	-	1125
0xa3	LOG3	Append log record with three topics	-	1500
0xa4	LOG4	Append log record with four topics	-	1875
0xa5 - 0xaf	Unused	-	-	
0xb0	JUMPTO	Tentative libevmasm has different numbers	EIP 615	
0xb1	JUMPIF	Tentative	EIP 615	
0xb2	JUMPSUB	Tentative	EIP 615	
0xb4	JUMPSUBV	Tentative	EIP 615	
0xb5	BEGINSUB	Tentative	EIP 615	
0xb6	BEGINDATA	Tentative	EIP 615	
0xb8	RETURNSUB	Tentative	EIP 615	
0xb9	PUTLOCAL	Tentative	EIP 615	

Opcode	Name	Description	Extra Info	Gas
`0xba`	GETLOCAL	Tentative	EIP 615	
`0xbb` - `0xe0`	Unused	-	-	
`0xe1`	SLOADBYTES	Only referenced in pyethereum	-	-
`0xe2`	SSTOREBYTES	Only referenced in pyethereum	-	-
`0xe3`	SSIZE	Only referenced in pyethereum	-	-
`0xe4` - `0xef`	Unused	-	-	
`0xf0`	CREATE	Create a new account with associated code	-	32000
`0xf1`	CALL	Message-call into an account	-	Complicated
`0xf2`	CALLCODE	Message-call into this account with alternative account's code	-	Complicated
`0xf3`	RETURN	Halt execution returning output data	-	0
`0xf4`	DELEGATECALL	Message-call into this account with an alternative account's code, but persisting into this account with an alternative account's code	-	Complicated
`0xf5`	CALLBLACKBOX	-	-	40
`0xf6` - `0xf9`	Unused	-	-	
`0xfa`	STATICCALL	Similar to CALL, but does not modify state	-	40
`0xfb`	CREATE2	Create a new account and set creation address to `sha3(sender + sha3(init code)) % 2**160`	-	
`0xfc`	TXEXECGAS	Not in yellow paper FIXME	-	-

Opcode	Name	Description	Extra Info	Gas
`0xfd`	REVERT	Stop execution and revert state changes, without consuming all provided gas and providing a reason	-	0
`0xfe`	INVALID	Designated invalid instruction	-	0
`0xff`	SELFDESTRUCT	Halt execution and register account for later deletion	-	5000*

Thanks again to https://github.com/trailofbits/evm-opcodes for their contribution.