Building Games with Ethereum Smart Contracts

Intermediate Projects for Solidity Developers

Kedar Iyer
Chris Dannen

Apress®

Building Games with Ethereum Smart Contracts

Intermediate Projects for Solidity Developers

Kedar Iyer
Chris Dannen

Apress®

Building Games with Ethereum Smart Contracts

Kedar Iyer
Brooklyn, New York, USA

Chris Dannen
Brooklyn, New York, USA

ISBN-13 (pbk): 978-1-4842-3491-4
ISBN-13 (electronic): 978-1-4842-3492-1
https://doi.org/10.1007/978-1-4842-3492-1

Library of Congress Control Number: 2018943122

Managing Director, Apress LLC: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Nancy Chen

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC, and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484234914. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Authors

Kedar Iyer is a software engineer who runs Emergent Phenomena, a blockchain consultancy. He is currently writing blockchain software as a member of the Everipedia team. He has a bachelor's degree in mechanical engineering from UCLA and has worked in the past with microsatellites, robotics, and multiple startups.

Chris Dannen is a cofounder and partner at Iterative Capital, a large-scale cryptocurrency miner, investment manager, and private digital asset exchange. A self-taught programmer, he has written three technical books and holds one computer hardware patent. He was formerly the technical editor at *Fast Company*. He graduated from the University of Virginia and lives in New York.

About the Technical Reviewer

Massimo Nardone has more than 23 years of experience in security, web/mobile development, and cloud and IT architecture. His true IT passions are security and Android.

He holds a master of science degree in computing science from the University of Salerno, Italy.

He currently works as chief information security officer (CISO) for Cargotec Oyj and is a member of ISACA Finland Chapter Board.

Massimo has reviewed more than 40 IT books for various publishing companies and is the coauthor of *Pro Android Games* (Apress, 2015).

Acknowledgments

Thank you to Chris Dannen and Solomon Lederer for getting me into blockchain and introducing me to the NYC blockchain community. To Nancy Chen and James Markham at Apress for putting together this book, and Chris for offering me the opportunity to write it. And to my parents and sister for being supportive of my odd career choices.

—Kedar

Thank you to my team at Iterative Capital for their hard work and support, and to Kedar for traversing the smoky parlors of Las Vegas to make the games in this book especially authentic.

—Chris

What Is Ethereum?

Ethereum is a trusted compute platform with a native currency built on top of a decentralized network. A global network of nodes works together to form a consensus on the state of a shared database.

If Bitcoin offers us a glimpse into the future of money, Ethereum offers the equivalent for private property, financial assets, legal contracts, supply chains, and personal data. Any digital unit that can be owned by someone can be stored in an Ethereum smart contract and transferred between owners without the need for a third party or middleman such as a bank, exchange, or central government.

Ethereum works by successively executing a series of transactions, each of which is a block of code. That code is written in a special language named Solidity. This is the language we will be exploring in this book.

We will start by getting set up (Chapter 2), deploying simple contracts (Chapter 3), and going over the basics of the Solidity language (Chapter 4). Then we will take a brief detour into the theory behind contract security (Chapter 5) and crypto-economics (Chapter 6) before spending the last half of the book walking through a series of sample projects (Chapters 7–11). By the end of this book, you will be comfortable reading and interpreting existing Solidity contracts and ready to write your own original Solidity code.

Prerequisites

Working with Ethereum and Solidity requires some knowledge of computer science concepts and prior experience with another programming language. You don't need to be an expert, though; just the basics will do.

Computing Concepts

The best resource for learning the basics of computer science is the Harvard CS50 lecture series on YouTube (www.youtube.com/user/cs50tv). It's a fast-paced, detailed course. If you can make it through all 10 weeks, by all means do, but the first five lectures will teach you enough to tackle Solidity.

For learning about networking, Linux, or security and hacking, check out the popular uploads for Eli the Computer Guy on YouTube (www.youtube.com/user/elithecomputerguy/videos?shelf_id=26&view=0&sort=p). His videos are much more beginner-friendly than the CS50 lectures, so if you're looking for a soft intro to ease you in, this the place to start.

We will be using UNIX (Linux or Mac) command lines throughout the book. Instructions are given for how to make your Windows system compatible with our commands, but we recommend learning Linux if you can.

Networking and security are less important concepts to know, and you can make it through the book and become a Solidity developer without any prior knowledge of either. Networking is important in the Ethereum protocol under the hood, but is abstracted away at the application level, where we will be writing our code. Security is important because the amount of money passing through our contracts will make them lucrative targets. We spend an entire chapter discussing contact security (Chapter 5), but any additional knowledge you can obtain on the topic will serve you well.

Programming

Before diving into Solidity, you should have previous programming experience with another language. The closest language to Solidity is C, but it is neither beginner-friendly nor easy to set up. Your best bet for a simple programming introduction is Codecademy. The simplest language

to learn is Python, and the simplest Codecademy course is Learn Python (www.codecademy.com/learn/learn-python).

JavaScript, while slightly more confusing with syntax, is still easy to learn and more relevant to Ethereum programming because it is used by most client software for interacting with the blockchain. We will be writing and issuing simple JavaScript scripts and commands in this book. The best resource for JavaScript is the Codecademy Introduction to JavaScript course (www.codecademy.com/learn/introduction-to-javascript).

Suggested Reading

This book is an intermediate-level programming book. Before starting with this book, consider that maybe you should be reading a different one.

Introducing Ethereum and Solidity by Chris Dannen (Apress, 2017) is a great book for getting you up to speed with all things Ethereum. If you just want to understand how Ethereum works without getting deep into the nuances of writing smart contracts, that's the book you should be reading.

In the Beginning…was the Command Line by Neal Stephenson (William Morrow, 1999) is the best book I've come across on the history and metaphysics of software. It reads like a novel, is far better written than this book, and if you're here for anything except learning Solidity, you should probably be there instead.

Protocols, Platforms, and Frameworks

Ethereum is both a protocol and a platform, but not a framework.

A *protocol* is a series of rules used to standardize communication over a network. Basic protocols such as IP and TCP allow the garbled bytes flowing through fiber-optic cables to be routed to their proper destinations and decoded into a meaningful structure. Without protocols, the communication between computers would be random noise, like a Maori and English speaker attempting to hold a conversation.

The Ethereum protocol allows nodes on the Ethereum network to hold a meaningful conversation with each other. Through this conversation, they can broadcast transactions, synchronize nodes, and form the consensus that underpins the network.

Platforms and frameworks are a little more loosely defined. For our purposes, we distinguish them by saying platforms allow applications to be built on top of them, whereas frameworks are (usually software) structures that make building those applications easier.

Ethereum is a platform. We can build and deploy distributed applications, or dapps, onto the Ethereum blockchain. Truffle, which we will encounter in Chapter 2, is a framework. It makes developing, compiling, and deploying Ethereum dapps easy.

CHAPTER 1

Conceptual Introduction

This chapter provides a high-level overview of the Ethereum blockchain. The blockchain is an ordered series of blocks, each of which is an ordered series of transactions. A transaction runs on the Ethereum Virtual Machine and executes code that modifies the state tree. We will explore each of these concepts in more detail in the following sections.

Blocks

As stated previously, a *blockchain* consists of an ordered series of blocks. A *block* consists of a header with meta information and a series of transactions. Blocks are created by miners through the mining process and broadcast to the remainder of the network. Every node verifies received blocks against a series of consensus rules. Blocks that don't satisfy the consensus rules will be rejected by the network.

A *fork* occurs when a network has competing sets of consensus rules. This usually occurs through an update in the official client, which in Ethereum's case is a program called *geth*.

Soft forks occur when the newer set of rules is a subset of the old rules. Clients still using the old rules will not reject blocks created by clients using the new rules, so only block creators (miners) have to update their software.

K. Iyer and C. Dannen, *Building Games with Ethereum Smart Contracts*,
https://doi.org/10.1007/978-1-4842-3492-1_1

Hard forks occur when the new set of rules is incompatible with the old set. In this case, all clients must update their software. Hard forks tend to be contentious. If a group of users refuses to update their software, a *chain split* occurs, and blocks that are valid on one chain will not be valid on the other. There have been six hard forks in Ethereum, one of which led to a chain split and the creation of Ethereum Classic (ETC).

Mining

Mining nodes in the Ethereum network compete to create blocks by using a proprietary proof-of-work algorithm called *Ethash*. The input to the Ethash algorithm is the block header, which includes a randomly generated number called a *nonce*. The output is a 32-byte hex number. Modifying the nonce modifies the output, but in an unpredictable fashion.

For the network to accept a mined block, the Ethash output for the block header must be less than the *network difficulty*, another 32-byte hex number that acts as a target to be beaten. Any miner who broadcasts a block that beats the target difficulty receives a *block reward*. The block reward is awarded by including a *coinbase* transaction in the block. The coinbase transaction is usually the first transaction in the block and sends the block reward to the miner. The current block reward since the Byzantium hard fork is 3 ether.

Sometimes two miners produce a block around the same time, and only one gets accepted into the main chain. The unaccepted block is called an *uncle block*. Uncle blocks are included in the chain and receive a lesser block reward, but their transactions don't modify the state tree.

The security of a blockchain is proportional to the amount of *hashpower* in the network. More hashpower in the network means each individual miner has a smaller percentage of the total hashpower and makes network takeover attacks more difficult (see "51% Attacks" in Chapter 6). Including uncle blocks in the chain increases the security of the chain because the hashpower used to create the unaccepted block doesn't get wasted.

The network difficulty is constantly adjusted so that a block is produced every 15–30 seconds.

Transactions

A *transaction* sends ether, deploys a smart contract, or executes a function on an existing smart contract. Transactions consume *gas*, an Ethereum measurement unit that determines the complexity and network cost of a code operation. The gas cost of a transaction is used to calculate the *transaction fee*. The transaction fee is paid by the address sending the transaction to the miner who mines the block.

Transactions can contain an optional `data` field. For contract deployment transactions, `data` is the bytecode of the contract. For transactions sent to a smart contract, `data` contains the name and arguments for the function to invoke.

Ethereum Virtual Machine (EVM)

A *processor* is an integrated circuit that executes a series of given instructions. Each processor has a set of operations it can perform. An *instruction* consists of an operation code, or *opcode*, followed by input data for the operation. The x86 instruction set is the most common instruction set in use today and has about 1,000 unique `opcodes`.

A *program* is a set of instructions executing blindly in order. All code—be it punchcards, assembly, or a high-level language such as Python—gets compiled or interpreted down to a series of raw bytes. These bytes correspond to a series of processor instructions that the computer can run in order, like a dumb machine. Listing 1-1 shows what a Hello World program looks like in x86 Linux Assembly.

Listing 1-1. Hello World in x86 Linux Assembly[1]

```
section     .text
global      _start      ;must be declared for linker (ld)

_start:                     ;tell linker entry point

    mov     edx,len         ;message length
    mov     ecx,msg         ;message to write
    mov     ebx,1           ;file descriptor (stdout)
    mov     eax,4           ;system call number (sys_write)
    int     0x80            ;call kernel

    mov     eax,1           ;system call number (sys_exit)
    int     0x80            ;call kernel

section     .data

msg     db  'Hello, world!',0xa   ;our dear string
len     equ $ - msg          ;length of our dear string
```

A *virtual machine*, or VM, is a software program that pretends to be a processor. It has its own set of opcodes and can execute a program tailored specifically to its instruction set. The low-level bytes that correspond to VM instructions are referred to as *bytecode*. Programming languages can be written that compile down to bytecode for execution. The Java Virtual Machine (JVM) is the most popular virtual machine in use today. Some of you make a living off it. It supports multiple languages including Java, Scala, Groovy, and Jython.

Because it is an emulation, a virtual machine has the advantage of being agnostic to the hardware it runs on. Once a virtual machine has been ported to a new platform such as Windows, Linux, or the embedded OS

[1]Sourceforge, "Hello World!", http://asm.sourceforge.net/intro/hello.html

in your "smart" refrigerator, programs written for that virtual machine can run equally well on the fridge as on your "smart" TV. Java's "Write Once, Run Anywhere" motto comes to mind.

Ethereum has a VM of its own called the *Ethereum Virtual Machine* (EVM). Ethereum requires its own VM because each opcode in the EVM has an associated gas fee. Fees act as a spam deterrent and allow the EVM to function as a permissionless public resource. Each of the EVM's custom `opcodes` has its own fee, meaning that well-written contracts can be cheaper to execute. For instance, the `SSTORE` operation stores data into the state tree, which is an expensive operation because the data has to be replicated across the whole network

The sum of the gas fees accumulated by a transaction's bytecode determines the transaction fee.

State Tree

The primary Ethereum database is its *state tree*, which consists of key/value pairs that map Keccak256 hash keys to a 32-byte value. Data structures in Solidity use one or multiple state tree entries to create programming constructs that are more conducive to programming. A *simple data type* is 32 bytes or less and can be stored in one state tree entry. A *complex data type* like an array requires multiple state tree entries. See the "Data Types" section in Chapter 4 for more on Solidity data structures.

Because a Keccak256 hash is 256 bits long, the Ethereum state tree is designed to store up to 2^{256} unique entries. However, after about 2^{80} entries, hash collisions will make the tree fairly unusable. Either way, this is more disk space than currently exists across the world, so developers can assume that unlimited storage exists. Paying for that storage is another issue, as storing data in the state tree consumes a significant amount of gas. Contracts should be written carefully to minimize the number of insertions and updates they make to the state tree.

The state tree is modified and built up by executing transactions. Most transactions will modify the state tree.

The state tree is implemented as a Merkle Patricia trie. Understanding this data structure is not essential for Solidity programming, but if you are interested, the details are documented on GitHub at https://github.com/ethereum/wiki/wiki/Patricia-Tree.

Web3 Explained

Many early adopters of blockchain technologies were excited by its potential to usher in a new era of the Internet—Web 3.0. Web 1.0 was the initial phase of the Internet: a platform used mostly for selling goods and posting information. Web 2.0 introduced social networks and collaboration to the Internet. Sites including Facebook, Flickr, and Instagram brought user-created content front and center. Web 3.0 is the hope for a new decentralized Web, where central authorities no longer have the power to conduct censorship or control user data.

DARPA originally designed the Internet to be a decentralized communication network that could not be taken down by attacking any central authority. As the Web became more commercialized in the last 15 years, the degree of centralization has increased as well.

Scoring well on Google's search algorithm has become a must for new sites to gain traffic. Facebook controls a large percentage of user-generated data and content behind its walled garden. Netflix and YouTube combined account for about one-third of Internet traffic. Countries such as China and Turkey take advantage of this by banning sites that do not agree to their censorship rules.

One of the goals of Web 3.0 is to *re-decentralize* the Web so it is harder to censor and control. Ethereum is an exciting platform for Web 3.0 enthusiasts because any application built on top of it is automatically decentralized.

An application on Ethereum is commonly referred to as a *distributed application*, or *dapp*. Unlike traditional Internet applications, they do not need servers for hosting and data storage. The Ethereum network handles all the traditional duties of the server, including authentication, contract data storage, and an API. This means dapps cannot be censored like traditional sites. Censoring a dapp would require blacklisting every node on the Ethereum network—not a trivial task.

The term *Web3* can lead to a bit of confusion among the Ethereum community. Although initially it referred to the idea of Web 3.0, it now also commonly refers to Ethereum's client library, web3.js. We will be using *web3* to refer to the client library in this book.

What's New with Ethereum

As of this writing, the Ethereum development community is largely focused on two initiatives that may be relevant to developers building dapps with this book:

- *Proof-of-stake*: Both Bitcoin and Ethereum's low-transaction throughput make some applications and services impractical at present. At peak, Bitcoin can process 7 transactions per second (TPS); Ethereum tops out around 30. In contrast, Visa and MasterCard boast tens of thousands of TPS at their peaks. In proof-of-stake (PoS), miners are replaced by *validators*. Swapping out the SHA-256 proof-of-work consensus algorithm for a PoS algorithm could greatly reduce block times, helping Ethereum to increase throughput beyond even Visa and MasterCard's limits.

- *Sharding*: Currently, every full archival node on the Ethereum network must download the entire blockchain, which as of this writing stands at over 300GB. Options for "light syncing" are available but are not long-term solutions. *Sharding* splits the account space into subspaces, each with its own validators, removing the requirement for the whole network to process every transaction. Transaction throughput projections for a sharded, PoS-enabled Ethereum network reach as high as 2,000 TPS per shard.

Bitcoin vs. Ethereum

Many of you have received your first exposure to cryptocurrencies and blockchains through Bitcoin. Bitcoin was the first cryptocurrency and is still the largest and most used. It enabled users to send and receive money anywhere in the world without going through a third-party intermediary such as a bank or PayPal. Think of it as counterfeit-proof money for the Internet.

Ethereum's primary innovation over Bitcoin is that it adds a trusted compute framework on top of a blockchain. Ethereum nodes may not necessarily trust each other, but they can trust that the network will execute smart contract code in a deterministic fashion. Combined with the inclusion of a native currency, this allows for a variety of functionality that Bitcoin does not support.

With the exception of hash-locked time contracts, Bitcoin does not support conditional paths. Money is either sent or not sent; the transaction does not depend on the internal state of the system. This may seem trivial, but adding support for conditional paths allows developers *flow of control*,

or the ability to specify the order in which individual statements in their program are evaluated and/or executed. An escrow payment is an example of an exchange that is conditional on both parties' participation. Bets are another example of payouts conditional on an external event. Users don't need to trust each other to trust that the smart contract logic will execute as intended.

In many ways, Ethereum is a leap into the unknown. Bitcoin was built to solve the specific problem of creating a decentralized currency. Ethereum offers programmable value transfer based on arbitrary logic, making it conducive to unimagined blockchain-related solutions of the future. The largest use-case at the moment is crowdfunding, but experiments and applications for betting, escrows, decentralized exchanges, prediction markets, decentralized encyclopedias, user-controlled smart data, and more are underway.

Addresses and Keypairs

Ethereum uses the same *asymmetric key cryptography* methods as Bitcoin to authenticate and secure transactions. Public-private keypairs are generated, and messages signed by the *private key* can be decoded only with the corresponding *public key*, and vice versa. An Ethereum *address* is the last 20 bytes of the Keccak256 hash of a public key. Keccak256 is the standard hash function used by Ethereum.

Ether balances tied to an address can be spent by whichever user can prove ownership of the corresponding private key. To do so, all Ethereum transactions are encrypted with the sender's private key. If the user's public key can be used to decrypt the broadcasted message into a valid transaction, that is proof that the user owns the private key.

Contracts and External Accounts

Ethereum has two types of accounts: external accounts and contracts. *External accounts* are controlled by users, whereas *contracts* are semiautonomous entities on the blockchain that can be triggered by a function call. All accounts have an associated balance and nonce. The nonce is incremented after every transaction and exists to prevent duplicate transactions. In addition to these two fields, contracts have access to storage space where they can store additional data fields as specified in their contract code.

Programs in Ethereum

Programs in Ethereum consist of one or more interacting smart contracts. Smart contracts can call functions in other smart contracts. Individual contracts are similar to classes in a traditional language.

Smart contracts can be written in EVM Assembly, Solidity, Low-Level Lisp (LLL), or Serpent. All contracts are eventually compiled down to EVM Assembly bytecode. Solidity is the most commonly used language and the one we will be using. Serpent has been phased out, and LLL usage is rare. New, experimental languages such as Viper are also under development.

Smart contracts are deployed by sending a transaction to the null address (`0x0...`) with the bytecode as the data.

When Ethereum was designed, its creators envisioned that smart contracts would call upon existing contracts for most of their functionality, with each new smart contract acting as a building block for new contracts on the chain. For example, a contract that wishes to manipulate strings would call on an existing `StringUtils` contract to perform operations like string concatenation that are not supported by Solidity.

Unfortunately, developing in this style requires interacting directly with the Ethereum mainnet for testing and development, which has turned out to be quite expensive. Instead, most developers nowadays would copy a standard `StringUtils` contract into their program so that it's available on a private test chain, and then deploy their own copy of the `StringUtils` contract to use in their program. We will see more examples of this in the game projects in the latter half of the book.

Smart contracts automatically expose an *application binary interface* (ABI), which is the binary or bytecode equivalent of an API. The ABI contains all public and external functions and excludes private and internal functions. ABI functions can be called by either an external account while sending a transaction or by another smart contract while executing its internal logic.

Digging into Solidity

Solidity is the primary programming language for the EVM. Because the EVM has custom `opcodes` that are not used by conventional processors, existing programming languages are an awkward fit for the EVM. Solidity was designed specifically for the task of programming smart contracts on Ethereum.

Solidity receives many comparisons to JavaScript, but its closest relative is C. Solidity is a strongly typed language with minimal functionality that emphasizes limiting storage and CPU usage. It supports 256-bit data types for the EVM, unlike most languages, which support only 32- and 64-bit processors.

Developers who have never worked with a strongly typed language should not find it difficult to adjust to Solidity. Many people actually find typed languages easier to deal with than untyped languages, so don't let that intimidate you. Mobile developers coming from Java, Swift, or Objective-C will find Solidity syntax pretty familiar. JavaScript developers

may require some adaptation, as expressions evaluate more easily in loosely typed languages but introduce undesired (and potentially expensive) ambiguities into a system where computation is fee-based.

In a production setting, all developers will have to adjust to working within the gas constraints that limit storage, memory, and CPU usage. Embedded systems developers used to working with limited resources will likely have the easiest transition to Solidity.

Chapter 4 has much more on the ins and outs of working with Solidity.

Staying Hack-Free

Because smart contracts can maintain an ether balance, they are lucrative targets for hackers. Hacks including the DAO attack and Parity multi-sig attack have led to millions of dollars in losses. Most Solidity application code is open source, so following best practices is essential to avoid leaving glaring security flaws in your contract code. These range from interaction techniques such as using a withdrawal method instead of sending ether within a contract (see "Withdrawal Methods" in Chapter 5) to code techniques such as minimizing conditional paths.

In general, Solidity development should be treated more like building a bridge than building a web site. The process is not iterative. Once deployed, a contract's code and ABI cannot be updated. Transferring balances from one contract to another, especially for contracts that maintain an internal ledger, ranges from difficult to impossible.

Whenever possible, proven legacy code should be used instead of new, untested code. Contracts should be thoroughly tested and vetted before being deployed to the mainnet.

Chapter 5 covers contract security in extensive detail. It is the most important chapter in the book. Make sure to read it before attempting to store any assets or ether on a deployed smart contract.

Block Explorers

Block explorers are web sites that provide an easy-to-use interface for navigating a blockchain. Etherscan (https://etherscan.io/) is currently the best block explorer available for Ethereum (Figure 1-1). We can use it to check the height of the latest block while syncing, monitor a pending transaction, view the final gas fee for a transaction, check the network difficulty, view the source code or ABI for a deployed contract, and more.

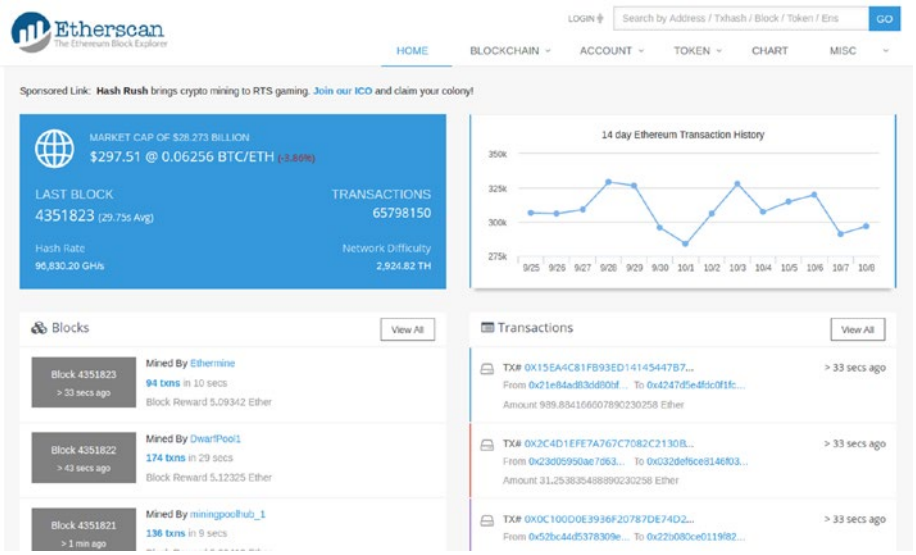

Figure 1-1. *Etherscan block explorer*

We will be using Etherscan extensively in this book to monitor our transactions and wallets. You can search for individual transactions and addresses by using the search box in the upper-right corner.