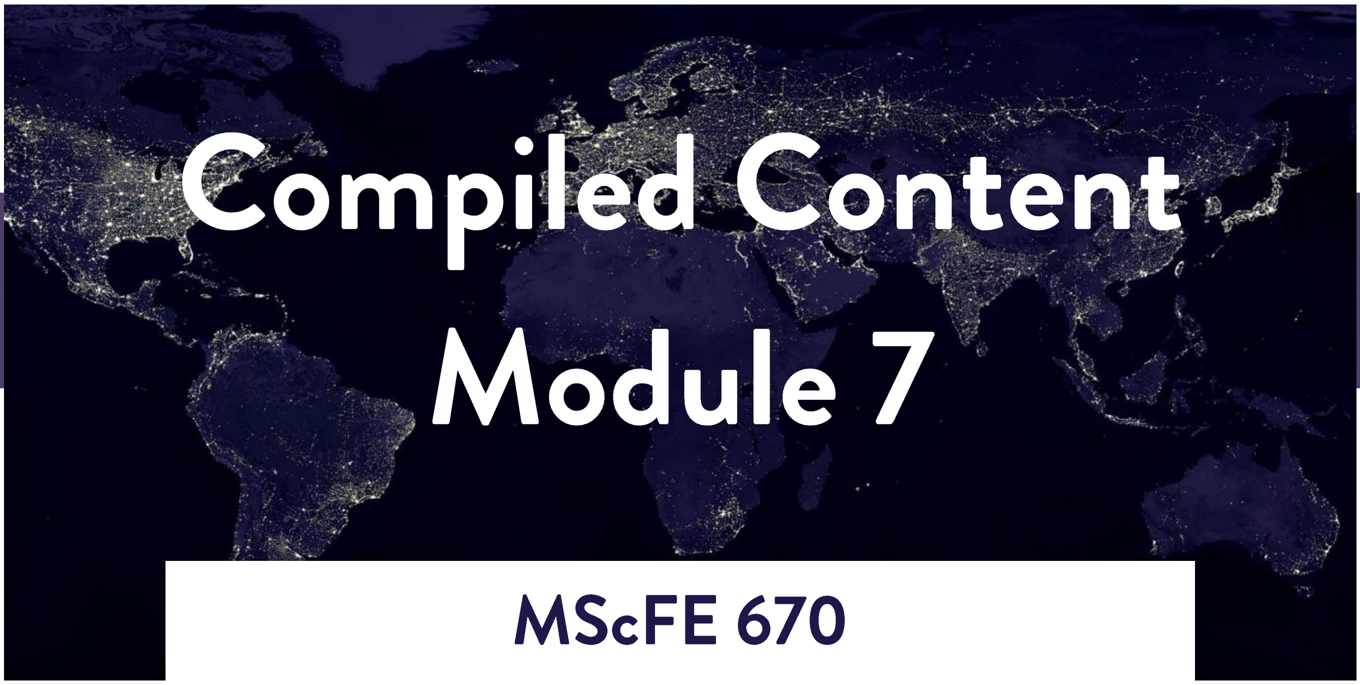
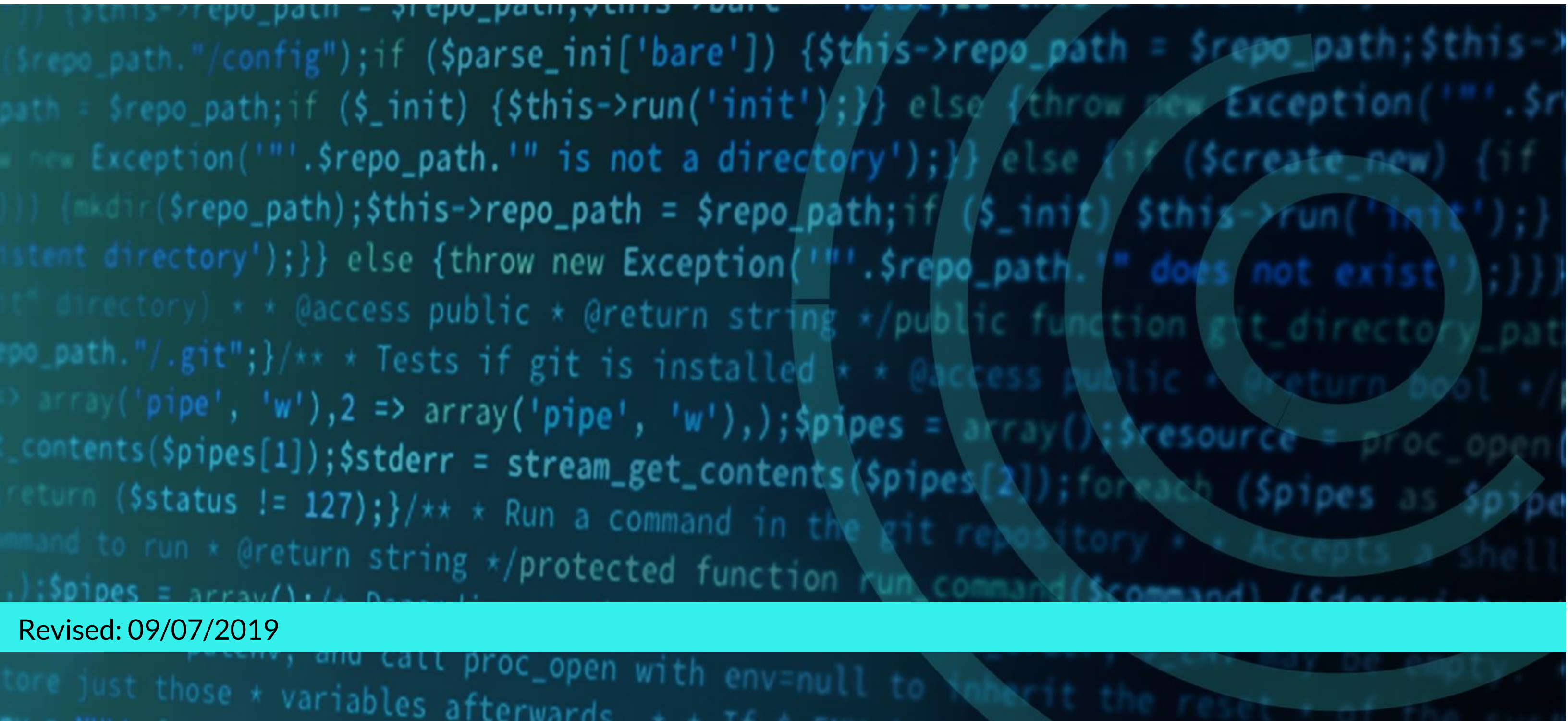
WORLDQUANT UNIVERSITY



Compiled Content Module 7

MScFE 670

Data Feeds and Technology



Revised: 09/07/2019

Table of Contents

Module 7: Smart Contract Development for Blockchains: Part 2

Module 7 begins by describing gas and efficiency as major considerations when developing smart contracts due to memory and processing power constraints in the Ethereum network. The module continues by examining the internal structure of the Ethereum network and explaining how security is implemented by using practical examples. The module concludes by illustrating key concepts in smart contracts including governance, languages, and standards. Considerations to be taken into account while developing upgradeable smart contracts are discussed as well.

Unit 1: Gas and Efficiency

Gas

In Ethereum, every computational operation has an associated gas usage. For example, a single keccak-256 hash uses 30 gas, and it takes 2000 gas to add a word to storage (at the time of writing). These values can change with time when software upgrades are made to the network, and they need to be finely balanced to ensure the network operates well. (You can look up the changes in the Constantinople Hard Fork for an example of this[1].)

But what is gas? Gas usage for computations can be thought of similarly to gas usage when driving a car – i.e. more computational resources used corresponds to greater distance driven in this metaphor. The cost of this gas goes to the miner (a full computational node in the network) who succeeds in adding the next block to the chain. This is an incentive for the miner to include the transactions that pay them the most, since from a game theoretical point of view the goal of miners is to maximize profit.

But let us make that metaphor more concrete: we can compare **Ether** – the native currency of the Ethereum Network – to the US Dollar, and gas to petrol. Then what determines the price of gas? Supply and demand. This is not too dissimilar to how the price of petrol is determined: where there is a limited supply of petrol, there is a limited amount of gas available to be used by people who want to process transactions. However, this gas takes no physical form. The price of gas is usually denoted in Gwei (giga-wei) where 1 Ether = 1 000 000 000 (i.e. 10^9) Gwei.

This serves two purposes. Firstly, it means that any computation is guaranteed to halt, since it cannot continue once all its designated gas is used. The maximum amount of gas a transaction can consume is defined by the network as the Block Gas Limit. The Block Gas Limit is the maximum gas all transactions in that block together can use, whether it is 100 small transactions, such as sending ERC20 tokens, or a single big transaction, such as deploying a new smart contract.

Secondly, gas is used as an anti-spam mechanism for the network. Gas is calculated relative to the computational load that the miners are required to use to process the computational operations. This means that the network doesn't suffer due to badly written user code as to the network each block is only capable of processing a certain amount of gas. In fact, in October of 2016, an

[1] The Constantinople Hard Fork: What You Need to Know. (2019). Consensys Media. Available here. [Accessed 28 February 2019]

imbalance in gas usage for certain computations was the root of an attack on the Ethereum network which resulted in an adjustment of these fees via a hard fork.

A **hard fork** means that the updated code running on these machines is incompatible with the previous version, as opposed to a **soft fork** where software upgrades can run concurrently with the previous versions without major consensus disagreements.

The gas limit of the block is not fixed. It can be adjusted by the miner by some factor, and so it changes over time according to this kind of decentralized process. The gas limit is crucial to keeping the network decentralized. The higher the gas limit is, the longer it takes for blocks to be propagated to all the nodes and processed. This naturally benefits larger miners with more powerful equipment and thus the Ethereum network would become less decentralized. You can see the gas limits of the different Ethereum networks at the following links: https://etherscan.io/blocks and https://rinkeby.etherscan.io/blocks.

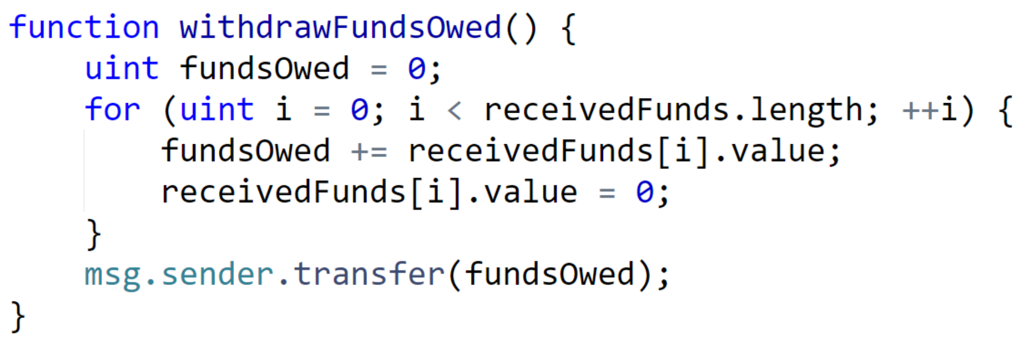
You may notice that the different networks have different gas limits (due to the process by which they are determined), and this has caused problems with some main net deployments. In the previous module, the contract deployment to the demo net was successful since the contract was small enough but deploying the contract to the main network failed since it cost too much gas to deploy to the network.

Although gas use for a given computation is fixed, users of the network choose the price to pay per unit of gas in Ether. Ether is the native currency that secures the Ethereum network, and whose price is driven by scarcity and speculation. So, the machines that run the computations of the network, the miners, select transactions that have the highest gas price available to maximize their profit. Thus, the gas price increases with congestion.

Efficiency

If a function is half as efficient as it could have been had it been written more efficiently (in terms of storage usage and computational steps) it also costs twice as much. If this this function is executed frequently by the network it can cause a lot of wastage, costing people money and congesting the network unnecessarily. This is not a new concept, of course. A big company like Facebook will try everything they can to reduce the amount of data they send to you over the internet. If they send half as much data, they have half as much load on their servers – over millions of users, this adds up!

As a general rule, you should avoid writing functions that have anything more than a **constant time** running time in your smart contracts – i.e. you should avoid code that runs for longer or shorter depending on the input. As a concrete example, any function that has a for loop in it is prone to a denial of service attack, as seen below. All an attacker needs to do to paralyze your smart contract is to make the `'receivedFunds'` array long enough that any attempt to loop through it will require computational resources exceeding the gas limit of the network at that time.



```
function withdrawFundsOwed() {
    uint fundsOwed = 0;
    for (uint i = 0; i < receivedFunds.length; ++i) {
        fundsOwed += receivedFunds[i].value;
        receivedFunds[i].value = 0;
    }
    msg.sender.transfer(fundsOwed);
}
```

Figure 1: DoS-prone withdrawal function

A fix to this could be to remove the for loop and make the user specify the index of '`receivedFunds`' to receive funds from and execute a separate transaction, or even a finite and limited range.

Writing code for blockchains today is similar to trying to write code for old computers. There are heavy constraints on memory and processing power. Modern personal computers have become powerful enough today that this isn't much of an issue for most common applications. In blockchains today, even the simplest applications require lots of careful planning and thought. In traditional computing, it has been the general rule observed in **Moore's Law** – i.e. that the number of transistors on a computer chip doubled every year – that brought the convenience of personal computing to the masses. It is still not clear if, or how, such big steps will be taken with blockchains. Many are hopeful about endeavors such as Ethereum 2.0, which is a complete rewrite of the Ethereum protocol to scale with the size of the network via sharding and other techniques. Time will tell.

Unit 2: Ethereum Internal Structure Deep Dive

Internally, all data contained in a specific smart contract is stored in a Merkle tree which is a hashed tree data structure. In fact, all data stored in any given block in the blockchain is stored in a Merkle tree, with sub-Merkle trees for each contract in the system. A Merkle tree is a hashing data structure in which the leaves represent the data you want to store. Each leaf node is then hashed, and those hashes are combined recursively until a single root hash is reached. This allows the formation of "proofs" (known as Merkle proofs) that the data is unchanged, by including the hashes of the route to the root hash, as well as any hashes adjacent to this path. Additionally, this means that the insert, delete, and lookup time for all data is relatively quick, since to find any data you only need to traverse down the tree once.

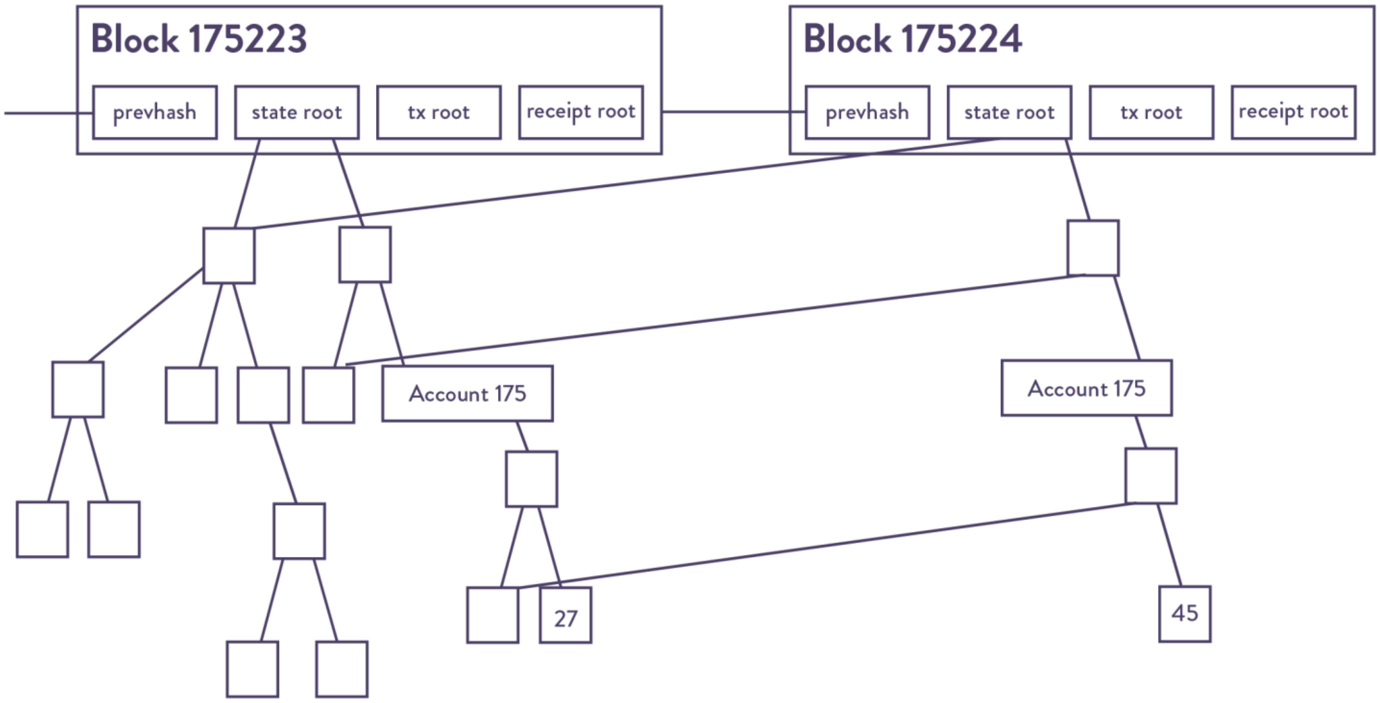


Figure 3: Ethereum internal structure deep dive

The finer details of this are not too important to the DApp developer, but two things are important. The first is that you can have light nodes that can answer the questions below fully and verifiably because the state of the Ethereum network is concisely stored in a single Merkle tree:

1. Has this transaction been included in a particular block?
2. Tell me all instances of an event of type X – e.g. a crowdfunding contract reaching its goal – emitted by this address in the past 30 days.

3. What is the current balance of my account?
4. Does this account exist?
5. Simulate running this transaction on this contract. What would the output be?

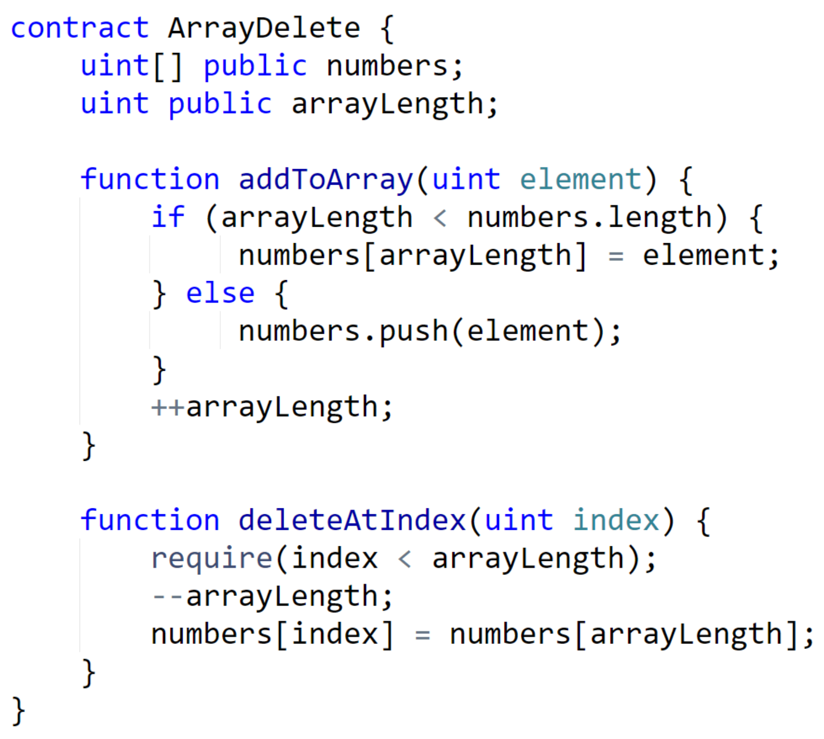
Note: this is not possible in Bitcoin, where the state of the system is only stored as the outcome of all changes rather than the result itself.

One such application of this concept is the Status-im chat application, which aims to be "the WeChat of Ethereum". With just 36MB of memory used on the device, the device is quickly able to verify the validity of any desired information in the above five points and will continue to be able to do so as long as the Keccak-256 hash (the primary hashing algorithm used in Ethereum) remains unbroken and collision free. Users of apps with these built in "light nodes" can be sure that the data they are receiving about the blockchain is accurate and not provided by a malicious entity, thus hardening apps running on these kinds of platforms to hackers[2].

The second important reason a developer needs to know about the underlying structure of data in an Ethereum application is that it dictates how you structure data within your code for these applications. The programming languages used to write code for the EVM (Ethereum Virtual Machine) are low-level and don't attempt to unnecessarily abstract away this underlying structure.

When dealing with arrays, it is often best to let them be append-only, since for loops should be avoided. Rather let end user clients do any required looping off-chain and allow clients to access individual elements by index. If, however, it is important that you delete elements from an array, then the below method is usually preferred. Since moving items in storage is expensive and for loops should be avoided, the strategy is to replace the element that you want to delete with the last element of the array.

[2] Status-im is a very interesting project, both technically and from a UX perspective. If you are interested in reading more about it, please see the link included in the additional resources section.



```
contract ArrayDelete {
    uint[] public numbers;
    uint public arrayLength;

    function addToArray(uint element) {
        if (arrayLength < numbers.length) {
            numbers[arrayLength] = element;
        } else {
            numbers.push(element);
        }
        ++arrayLength;
    }

    function deleteAtIndex(uint index) {
        require(index < arrayLength);
        --arrayLength;
        numbers[index] = numbers[arrayLength];
    }
}
```

Figure 4: Non-looping delete function

The main point is that there is no database or query language in Ethereum applications – the structure of the data in your app is left up to you as the developer. CRUD (create, retrieve, update, delete) is a common concept in databases, where create, retrieve, update, and delete are the four operations that you need to have on your data. A common idea is to create your own kind of CRUD pattern to make your smart contracts easier to work with.

Types of memory

There are three places that variables can be stored in Ethereum: "**call data**", which is the data included in a transaction that defines which function should be called and the parameters of that function; "**memory**", which is the internal memory used by computations that only lasts the duration of the transaction; and "**storage**", which is persistent and represents the state of the blockchain. The difference between them is important and needs to be kept in mind in order to make sure an application is secure from attacks from hackers.

Storage is very important to the functioning of a DApp. I would like to go back and continue explaining how data is laid out inside the storage of a given smart contract because I think this is one of the most fascinating parts of Ethereum from a computer science perspective.

For this section you need to recall the hashed structure of Merkle tree. (The technical name here should be a Merkle Patricia tree, however members of the Ethereum developer community still call it a Merkle tree.) In Ethereum, every memory slot of the "storage" is set to zero by default. In the tree that is constructed over this address space, any branch that only contains zero-values is truncated. In other words, there is no need to record the parts of the tree and use up storage space if they are default values. All other branches of the Merkle Patricia tree that contain non-zero values are calculated and stored in storage. This has the major advantage that all the address space in a smart contract can be retrieved via functions/opcodes, whether or not the data has been defined. As a result, a very common pattern to use in Solidity is to use a "bool exists" tag to prevent overwriting any data in a mapping by mistake.

```
struct ComplexObject {
    bool exists;
    //other important data
}

mapping(bytes32 => ComplexObject) objects;

function addObject(bytes32 id, /*other parameters*/) {
    if (!objects[id].exists) {
    // create new object
    }
}
```

Figure 5: Preventing data overwriting

Another common pattern is to use an index in an array to check if an object exists. This is more useful than a Boolean in applications that use arrays, but this requires extra – and careful – testing if your smart contract allows the index pointer and the actual object in the array to get out of sync.

Another interesting fact, that becomes very important when attempting to construct highly gas-efficient code, is that a lower gas fee is charged for changing an existing storage field in a smart contract than setting a brand-new value. This is logical since the creation of new storage has the overhead of taking up additional memory in the machines of miners in the form of these branches. Whereas if the branches are there, they just have to update their hashes. On a similar note, there is a negative gas fee associated with deletion of storage space, but only up to the total cost of the transaction (so you still cannot be paid to execute a transaction). With that, it important to note that the operation-codes '`SSTORE`' and '`SLOAD`' that operate on storage are far more expensive in gas than their "memory" equivalents. Operation code, normally abbreviated to **op-code**, is the low-level operations that the EVM understands.

Going back to the structure of this storage tree, each storage slot at the leaves is 256 bits (32 bytes) long. And there are 2^{256} such slots, which as you may remember from the last module is a very big number. Static sized data is sequentially packed from position zero, one after the other. This specific sequence is determined at compilation time. Dynamically sized data, such as an array which can grow any length, is stored elsewhere, but storage used is consecutive from there, with only the pointer to the start of this dynamically sized data (arrays) being stored at the beginning. For a **mapping** (effectively the dictionary lookup of Ethereum), the value at a given key is stored at a pseudo-random location. **Pseudo-random** means that the data is stored in a "slot" that is deterministically chosen, but that choice appears random from the outside. As you can see, the locations of the data in storage is rather haphazard, but nevertheless deterministic.

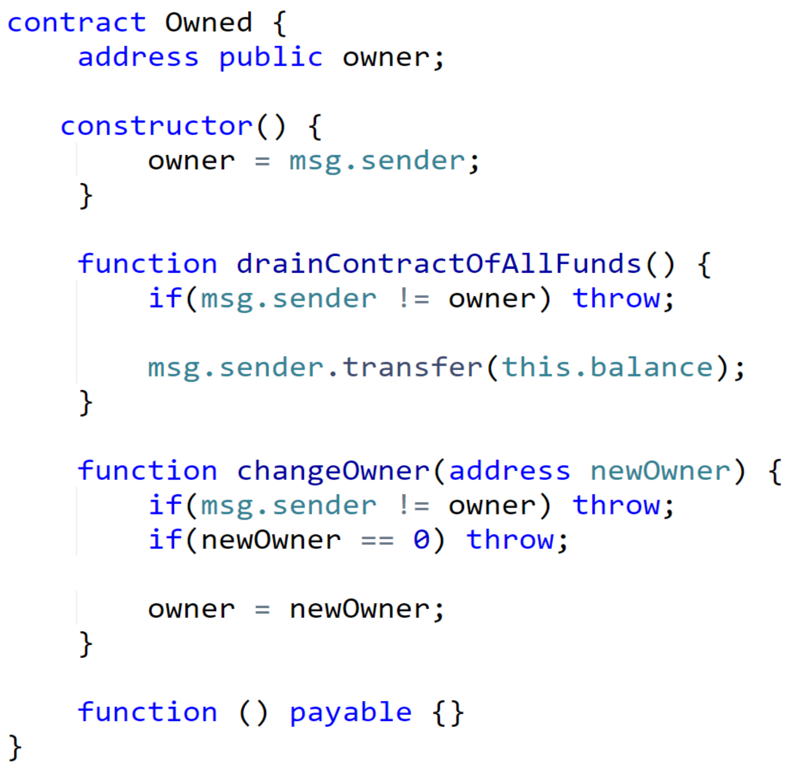
There is a chance this information is raising internal alarm bells. You might be wondering what mechanism is preventing data from being allocated to the same location, and the answer is that nothing is. This just illustrates how important it is to have a collision-free hash function, and how this is one of the most conceptually beautiful aspects of cryptocurrencies. If one of the 7.5 billion people in the world were to generate the same Ethereum address as me, they *are* me as far as the network is concerned. However, 2^{256} is a massive number and each smart contract has a total storage capacity of $32 * 2^{256}$ bytes = $3 * 10^{66}$ Terabytes. So, the chances of two dynamically sized data items will collide or need to write to the initial few bytes is very small. It is, of course, still possible for a smart contract with a giant storage to have such a collision. Maybe the correct question we need to be asking ourselves is, "In which year, if ever, in the future will we decide that we need a bigger hash function?"

Unit 3: Security in Ethereum

It is easy to see that security is of utmost importance in DApp development, as Ethereum funds can only be stolen from a smart contract if the logic of the smart contract allows it. However, there is still a lot of room for human error.

In traditional software systems, ownership and access control are usually thought of in a centralized manner, where a server or central authority of sorts grants access permissions to various users of the system. In Ethereum, the need for central control and access control are separated. The smart contract author can restrict access to various functionality with the simplest of programming constructs, the “if statement”.

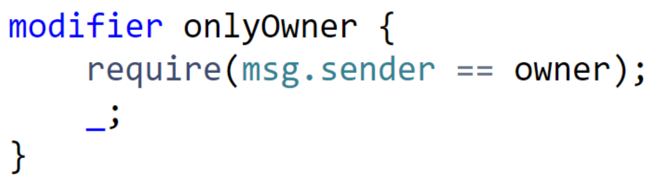
In Solidity, variables cannot be manipulated from outside the smart contract unless a setter function of sorts is created. Even making variables “public” creates an externally accessible getter function, but not a setter function (as is expected in other programming languages). This means that access control can be used on functions that modify important variables that are used for access control.



```
contract Owned {
    address public owner;

   constructor() {
        owner = msg.sender;
    }

    function drainContractOfAllFunds() {
        if(msg.sender != owner) throw;

        msg.sender.transfer(this.balance);
    }

    function changeOwner(address newOwner) {
        if(msg.sender != owner) throw;
        if(newOwner == 0) throw;

        owner = newOwner;
    }

    function () payable {}
}
```

Figure 7: Access control in Ethereum

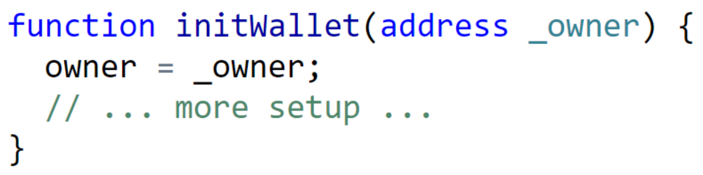
The above example shows a basic mechanism for access control in Ethereum. Here only the "owner" can withdraw the funds. As can be imagined, the `if(msg.sender != owner)throw;` could possibly be duplicated all over the code, so Solidity has some nice "syntactic sugar", called modifiers, that can be added to the header of any function (as seen below). **Syntactic sugar** is a term used in programming for a more concise way to write some syntax. A **modifier** is simply a piece of code that executes at the beginning of a function.



```
modifier onlyOwner {
    require(msg.sender == owner);
    _;
}
```

Figure 8: onlyOwner modifier

As a case study, we will examine a case where simply leaving out a modifier from a sensitive function allowed about 32 million dollars' worth of Ether to be stolen, with around 60 cents' worth of transaction costs. The contract containing the funds was a multi-sig wallet, which was meant to require permission from multiple parties to withdraw funds. The smart contract was structured as a number of subordinate smart contracts that acted as libraries for additional functionality. The following snippet was meant only to be called on construction, as the Wallet contract delegates the initialization of the contract's state to the `WalletLibrary` contract. However, the `onlyOwner` modifier was left off, making things extremely easy for the attacker. The most challenging thing the attacker needed to do was to do was scan the blockchain to find out the address of deployed versions of this code.



```
function initWallet(address _owner) {
  owner = _owner;
  // ... more setup ...
}
```

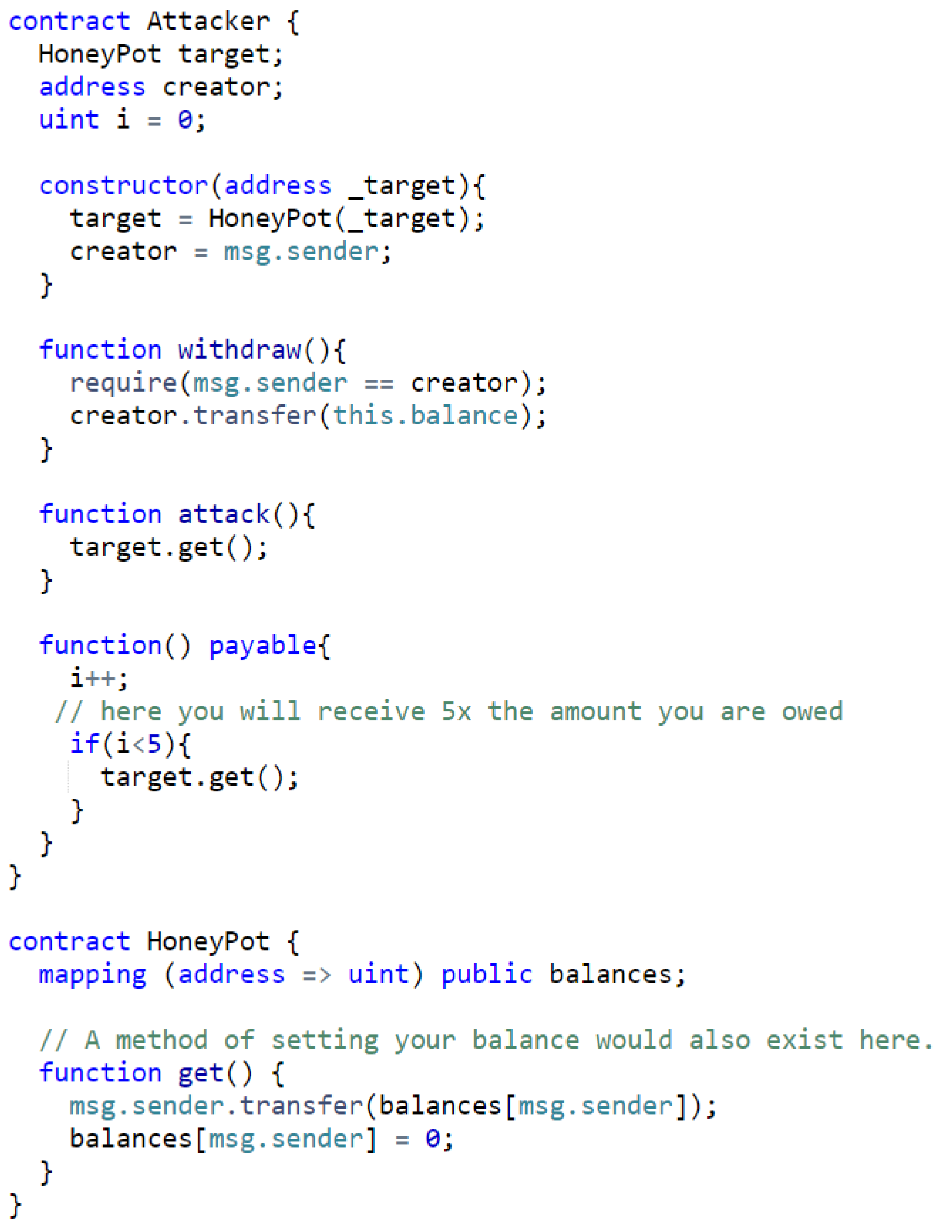
Figure 9: initWallet function

There are two other ideas that are critically important for the security of smart contracts that will be explained in this section. The first is to fail early and fail hard

and the second is to do all accounting optimistically.

To fail early and fail hard is a very simple concept that comes from Rob Hitchens in his courses at B9Lab and his various interactions online – for example: https://ethereum.stackexchange.com/a/15147/4642. The idea is simple: the goal of the contract developer is to maintain the integrity of the state of the contract at all costs. When a require statement fails, it reverts the state of the transaction as if it never happened. The smart contract developer should always aim to write code that checks everything is as it should be early on in the code and should be unforgiving in reverting the transaction if there are any problems.

To understand the importance of optimistic accounting, our second important principle, we look at the re-entrance bug. In Ethereum, when a smart contract calls a function in another smart contract, it gives up all control to that other smart contract. That other smart contract is then free to recursively call the same function again. This is illustrated in the below code:



```
contract Attacker {
  HoneyPot target;
  address creator;
  uint i = 0;

  constructor(address _target){
    target = HoneyPot(_target);
    creator = msg.sender;
  }

  function withdraw(){
    require(msg.sender == creator);
    creator.transfer(this.balance);
  }

  function attack(){
    target.get();
  }

  function() payable{
    i++;
   // here you will receive 5x the amount you are owed
    if(i<5){
      target.get();
    }
  }
}

contract HoneyPot {
  mapping (address => uint) public balances;

  // A method of setting your balance would also exist here.
  function get() {
    msg.sender.transfer(balances[msg.sender]);
    balances[msg.sender] = 0;
  }
}
```

Figure 10: Example of re-entrance bug

The simplest fix is to do optimistic accounting, so that on re-entry the balance has been set to zero. Additionally, the function should fail early and fail hard if anything is amiss. In smart contracts, unlike conventional software products, you don't want to "error" gracefully, since anything less that completely correct could be an attack on the system.

```
function get() {
  require(balances[msg.sender] != 0);
  uint toSend = balances[msg.sender];
  balances[msg.sender] = 0;
  msg.sender.transfer(toSend);
}
```

Figure 11: Example of optimistic accounting

This kind of re-entrancy bug is what allowed the famous "The DAO" hack to take place in 2016. In the hack, 3.6 million Ether was stolen out of a total of 12.7 million Ether that was held by the smart contract. This was enough for members of the Ethereum community to initiate a very controversial hard fork (which left us with Ethereum and Ethereum Classic) as a way to reverse the hack on Ethereum.

It must be emphasized that security must be taken very seriously when developing smart contracts. If a smart contract is going to deal with a non-trivial amount of value, then it must be audited by a third-party expert. There are new tools being created to make it easier to write secure smart contracts by helping developers discover vulnerabilities. We are only human though, and it is all too easy for a human to overlook a mistake, as can be seen by some of the examples given above where the smart contract wasn't secured sufficiently.

Testing every aspect of your smart contract with an automated test is extremely important, but there is a lot of research taking place on improving how code is tested. One such tool which is popular for Ethereum is called Mythril, which performs various types of analysis on your code to find errors. Echidna is another tool that is able to fuzz input values to your functions in an attempt to find inputs to your functions that are incorrect. Another approach which is gaining popularity is called "formal verification" of code. The idea of formal verification is that you can analyze the code at a mathematical level to prove whether it correctly conforms to a specification or not. It is a sure way to make certain that your code doesn't have any bugs.

Unit 4: A Few More Important Concepts

In the context of DApps and the networks they run on, **governance** describes the degree of influence users have over various aspects of the applications and network. These aspects may pertain to user protection, user participation, and may change management processes, both in terms of the goals to be achieved and approach being followed.

However, as these decentralized systems are trustless, governance differs significantly from traditional systems that are overseen and maintained by a central executive authority. Instead, DApps rely on their authors to design rules for the system that allows decentralized representation amongst its userbase.

Governance

There are two things to consider in terms of governance in the realm of decentralized applications. One is the governance over the network as a whole, its maintenance, and upgrades. The other is within the applications themselves. Although there are lots of lots of difficulties with the former, we will primarily focus on the latter, as it relates to developers creating DApps.

Although these decentralized systems are trustless, it doesn't mean that the rules in the system cannot be coercive. When designing the rules of a system, the developer should be careful to keep the system so that it is easy to both opt in and opt out.

Languages and tools

Due to the nature of the Ethereum network, the maturing ecosystem, high costs of computation, and efficiency considerations mentioned in the previous section, it is extremely important that developers understand the basics of how the Ethereum Virtual Machine (EVM) works. It is a stack-based machine where each element on the stack is 32 bytes. It is sometimes useful to go through the debugger and watch the stack when there are issues with the code. Such a debugger is present in the RemixIDE that was presented in the previous module. The other time you may need to use op-codes directly is for advanced usage such as upgradeable contracts.

Ethereum is a protocol-based system. This has numerous benefits. One is that various Ethereum node software implementations exist in different languages, many of which are actively maintained by the Ethereum Foundation. This means that irregularities are much less likely to get into the protocol since this parallel development style catches these irregularities.

The other benefit of this protocol-based approach is from the user-client perspective. Although there are well established and popular frameworks to work in specific languages, any language that can perform a few simple operations can interact with the Ethereum ecosystem. The language needs to be able to pack the data required for the transaction correctly, sign it, and communicate this transaction to the rest of the network.

```
$> curl -X POST --data '{"jsonrpc":"2.0","method":"eth_coinbase","params":[],"id":1}' localhost:8545
```

Figure 16: Call to an Ethereum node running on localhost

The command is an example call to an Ethereum node running on localhost. There are a variety of different ways of communicating with the network, and it is best practice to allow the user to choose the type of connection depending on what device they are on. In JavaScript, for example, the `web3` library allows you to place a "provider" object into the library, which gives the developer leeway to choose which type of provider to use.

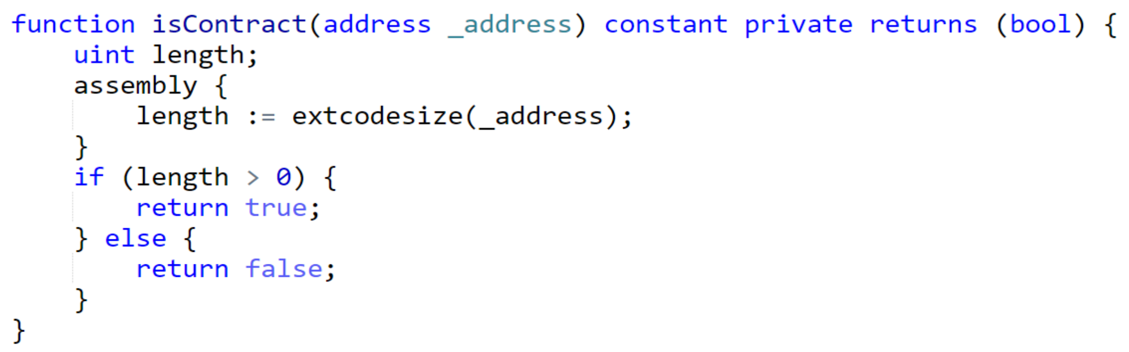
```
// Checking if Web3 has been injected by the browser
if (typeof web3 !== 'undefined') {
  // Use Mist/MetaMask's provider.
  provider = web3.currentProvider
} else {
  // Fallback to localhost if no web3 injection.
  const {host, port} = Config.networks[process.env.NODE_ENV]
  provider = new Web3.providers.HttpProvider('http://' + host + ':' + port)
}
```

Figure 17: Example of provider selection

Standards

A number of standards are being formalized to allow different smart contracts to communicate with each other in an effective manner. The most commonly used standard is the ERC20 (Ethereum Request for Comments) standard, which aims to come up with a common interface for all Ethereum-based tokens. This has allowed smart contracts to deal with, store, send, and receive ERC20 tokens in a similar way to how they would with Ether. This unlocks lots of potential to tokenize representations of anything from concert tickets to loyalty points to new forms of money to be exchanged and traded within any DApp designed to do so.

Of course, due to the rigidity of Ethereum, any standard needs to be extremely well thought through to allow for all future use cases and (as much as possible) to prevent malicious or accidental mishaps from happening. One such consideration with regards to the ERC20 token is to prevent it from being sent to contracts that don't know how to handle receiving tokens. To make it clear why this is an issue, consider that over 100 thousand USD worth of tokens for BAT and Golem (which are tokens powering different projects) have been sent to their own respective smart contracts by mistake. These contracts themselves have no way to transfer these tokens so they are effectively stuck there forever. One clever way in which a smart contract can check if it is dealing with another smart contract or a user account is via the `EXTCODESIZE` operation which returns the size of the data stored at that address. If it is zero, it is a user address, otherwise it is a contract. With this information further logic can be executed.



```
function isContract(address _address) constant private returns (bool) {
    uint length;
    assembly {
        length := extcodesize(_address);
    }
    if (length > 0) {
        return true;
    } else {
        return false;
    }
}
```

Figure 18: Example of validating data stored at an address

Upgradeable smart contracts

Since code deployed to the blockchain cannot be changed, consideration for future upgradability should be made. An approach to allow for upgradability like this is to separate the data and the logic of smart contracts. Then the logic of your system can be swapped out or improved without the data that the system runs on.

One way to do this is via a controller contract that is used as a proxy/interface (using the EVM op-code `DELEGATECALL`) to the other two components, namely the logic and the data. To upgrade the logic of a smart contract, one only needs to update the reference in the controller contract to the new logic. Then anytime a user calls a function in the controller it internally directs the call to the new logic.

The more difficult question is, "Who should be able to upgrade the smart contract?" If it is just the owner of the smart contract that can upgrade it, then the app becomes very centralized, by definition. If your DApp handles significant value, or you intend for it to handle significant value, it may be best to not include an upgrade mechanism. This is because, firstly, it introduces a massive security risk if someone else gets hold of your keys (as the owner of the smart contracts). Secondly, if there is a real risk that you may decide to change the logic of your code at random times, it may be difficult for users of your DApp to trust you enough to use it, greatly putting their funds at risk.

One method of managing upgrades is via something called a DAO (decentralized autonomous organization). You may remember "the DAO" being mentioned earlier in the re-entrancy bug section. "The DAO" was an early and extremely ambitious project with this kind of idea, which, as you might know, failed rather spectacularly. The basic idea of a DAO is that the members/users of the DAO vote (or delegate votes to others) on decisions that need to be made, and the code ensures that what is decided is what happens. For example, the decision could be, "Should we upgrade this contract?" Naturally, this decision can be much more decentralized than giving the power to a single person. However, a lot of thought needs to be given to the structure of the DAO to be sure that its decision-making capabilities are distributed over enough people.

There hasn't been an attempt quite as big or ambitious since, but there are a couple of projects that are doing research in this space. DAOstack have been working on creating implementations and open standards around different kinds of DAOs, and recently formed a partnership with Gnosis – a prominent Ethereum company that primarily focuses on creating a prediction market. They plan to launch a project called dxDAO that uses an interesting system of acquired reputation as voting power in the DAO. The dxDAO is intended to manage the DutchX exchange that Gnosis have built.

Other projects, such as MakerDAO, have a slightly simpler form of governance: for them, it is one token one vote. However, they do not include upgrading existing deployed smart contracts in this voting power; instead, they use this voting power to determine parameters in their system, such as how much collateral is required. The MakerDAO is an interesting decentralized finance project, and they have successfully created a decentralized token on Ethereum that has stayed at 1 USD value for over a year.

Last note

I hope you have enjoyed this introduction to Ethereum. You have learnt some of the core principles of Ethereum and some important concepts. There are many projects, lots of research, and many more resources out there to learn from. The Ethereum community is growing all around the world, so it is worth looking at websites like meetup.com to see if there is an Ethereum or Blockchain meetup in your city or town. It is great connecting with like-minded people who share an interest in Blockchain technology, and it is a great way to continue your learning. We have included some great links to more material in the additional resources section.

Bibliography

CoinDesk. (n.d.). *Understanding The DAO Attack*. [Online] Available at: https://www.coindesk.com/understanding-dao-hack-journalists.

Etherscan.io. (n.d.). *Ethereum Blocks*. [Online] Available at: https://etherscan.io/blocks.

Ethereum Stack Exchange. (2017). *Would it be better to use `throw` instead of `return false`?* [Online] Available at: https://ethereum.stackexchange.com/a/15147/4642.

Rinkeby.etherscan.io. (n.d.). *Rinkeby Blocks*. [Online] Available at: https://rinkeby.etherscan.io/blocks.

General resources

Delegatecall.com. (n.d.). *Blockchain based Q&A*. [Online] Available at: https://delegatecall.com.

Ethereum Research.com. (n.d.). *Ethereum Research*. [Online] Available at: https://ethresear.ch/.

Fellowship of Ethereum Magicians. (n.d.). *Fellowship of Ethereum Magicians*. [Online] Available at: https://ethereum-magicians.org.

GitHub. (n.d.). *Ethereum Developer Tools List*. [Online] Available at: https://github.com/ConsenSys/ethereum-developer-tools-list.

The Status Network: A strategy towards mass adoption of Ethereum. (2017). [PDF] Available at: https://status.im/whitepaper.pdf.

Interactive tutorials

Chainshot.com. (n.d.). *ChainShot*. [Online] Available at: https://www.chainshot.com/.

Cryptozombies.io. (n.d.). *CryptoZombies - Learn to code games on Ethereum*. [Online] Available at: https://cryptozombies.io/.

Cryptoeconomics.study. (n.d.). *Cryptoeconomics: An Introduction*. [Online] Available at: https://cryptoeconomics.study/.

Docs.learnchannels.org. (n.d.). *LearnChannels*. [Online] Available at: https://docs.learnchannels.org/.

The Ethernaut. (n.d.). [Online] Available at: https://ethernaut.zeppelin.solutions/.

Kauri.io. (n.d.). *Welcome to Kauri.* [Online] Available at: https://kauri.io.

Learnplasma.org. (n.d.). *Learn Plasma.* [Online] Available at: https://www.learnplasma.org/en/.

Subscriptions

EatTheBlocks. (2018). *Learn To Build Ethereum Dapps With Videos Tutorials.* [Online] Available at: https://eattheblocks.com/.

Weekinethereum.com. (n.d.). *Week in Ethereum News.* [Online] Available at: http://www.weekinethereum.com.

Online tools

Lab.superblocks.com. (n.d.). *Superblocks Lab.* [Online] Available at: https://lab.superblocks.com/.

Remix.ethereum.org. (n.d.). *Remix – Solidity IDE.* [Online] Available at: https://remix.ethereum.org/.

Book

Antonopoulos, A. and Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps.* Sebastopol: O'Reilly Media.

Podcasts

Epicenter. (n.d.). *Epicenter – Weekly Podcast on Blockchain, Ethereum, Bitcoin and Distributed Technologies.* [Online] Available at: https://epicenter.tv/.

Enigma. (n.d.). *Decentralize This!* [Online] Available at: https://blog.enigma.co/podcast/home.

Falls, A. (2019). *Arthur Falls.* [Online] SoundCloud. Available at: https://soundcloud.com/arthurfalls.

Podcast.ethhub.io. (2019). *Into the Ether.* [Online] Available at: https://podcast.ethhub.io/.

Rose, A. and Harrysson, F. (2019). *Zero Knowledge*. [Online] Zero Knowledge. Available at: https://www.zeroknowledge.fm/.

Shin, L. (2019). *Unchained Podcast*. [Online] Unchainedpodcast.co. Available at: http://unchainedpodcast.co/.

Tong, J. (2019). *The Smartest Contract.* [Online] The Smartest Contract. Available at: http://www.thesmartestcontract.com/.

Unconfirmed.libsyn.com. (2019). *Unconfirmed: Insights and Analysis From the Top Minds in Crypto*. [Online] Available at: http://unconfirmed.libsyn.com/.