

INICIO / INFORMAR

Última actualización de la página : 9 de febrero de 2021

En esta página



# Informe técnico de Ethereum

*Este documento introductorio fue publicado originalmente en 2013 por Vitalik Buterin, el fundador de [Ethereum](#), antes del lanzamiento del proyecto en 2015. Vale la pena señalar que Ethereum, como muchos proyectos de software de código abierto impulsados por la comunidad, ha evolucionado desde su inicio inicial.*

*Aunque tiene varios años, mantenemos este documento porque continúa sirviendo como una referencia útil y una representación precisa de Ethereum y su visión. Para conocer los últimos desarrollos de Ethereum y cómo se realizan los cambios en el protocolo, recomendamos [esta guía](#).*

## Un contrato inteligente de próxima generación y una plataforma de aplicaciones descentralizada

El desarrollo de Bitcoin de Satoshi Nakamoto en 2009 a menudo ha sido aclamado como un desarrollo radical en dinero y moneda, siendo el primer ejemplo de un activo digital que simultáneamente no tiene respaldo o [valor intrínseco](#) y no tiene un emisor o controlador centralizado. Sin embargo, otra parte, posiblemente más importante, del experimento de Bitcoin es la tecnología blockchain subyacente como una herramienta de consenso distribuido, y la atención está comenzando a desplazarse rápidamente a este otro aspecto de Bitcoin. Las aplicaciones alternativas comúnmente citadas de la tecnología blockchain incluyen el uso de activos digitales en blockchain para representar monedas e instrumentos financieros personalizados ( [monedas de colores](#) ), la propiedad de un dispositivo físico subyacente (

[propiedad inteligente](#)). [↗](#) ), activos no fungibles como nombres de dominio ( [Namecoin](#) [↗](#) ), así como aplicaciones más complejas que implican tener activos digitales controlados directamente por un fragmento de código que implementa reglas arbitrarias ( [contratos inteligentes](#) [↗](#) ) o incluso [organizaciones autónomas descentralizadas](#) [↗](#) (DAO) basadas en blockchain . . Lo que Ethereum pretende proporcionar es una cadena de bloques con un lenguaje de programación Turing completo integrado que se puede usar para crear "contratos" que se pueden usar para codificar funciones de transición de estado arbitrarias, lo que permite a los usuarios crear cualquiera de los sistemas descritos anteriormente. , así como muchos otros que aún no nos hemos imaginado, simplemente escribiendo la lógica en unas pocas líneas de código.

## Introducción a Bitcoin y conceptos existentes

### Historia

El concepto de moneda digital descentralizada, así como aplicaciones alternativas como los registros de propiedad, ha existido durante décadas. Los protocolos anónimos de efectivo electrónico de las décadas de 1980 y 1990, en su mayoría dependientes de una primitiva criptográfica conocida como cegamiento chaumiano, proporcionaron una moneda con un alto grado de privacidad, pero los protocolos en gran medida no lograron ganar tracción debido a su dependencia de un intermediario centralizado. . En 1998, el [b-money](#) de [Wei Dai](#) se convirtió en la primera propuesta para introducir la idea de crear dinero mediante la resolución de acertijos computacionales y un consenso descentralizado, pero la propuesta era escasa en detalles sobre cómo se podría implementar el consenso descentralizado. En 2005, Hal Finney introdujo un concepto de [pruebas de trabajo reutilizables](#). [↗](#) , un sistema que utiliza ideas de b-money junto con los difíciles rompecabezas Hashcash computacionalmente difíciles de Adam Back para crear un concepto para una criptomoneda, pero una vez más no alcanzó el ideal al confiar en la computación confiable como backend. En 2009, Satoshi Nakamoto implementó por primera vez en la práctica una moneda descentralizada, combinando primitivas establecidas para administrar la propiedad a través de la criptografía de clave pública con un algoritmo de consenso para realizar un seguimiento de quién posee las monedas, conocido como "prueba de trabajo".

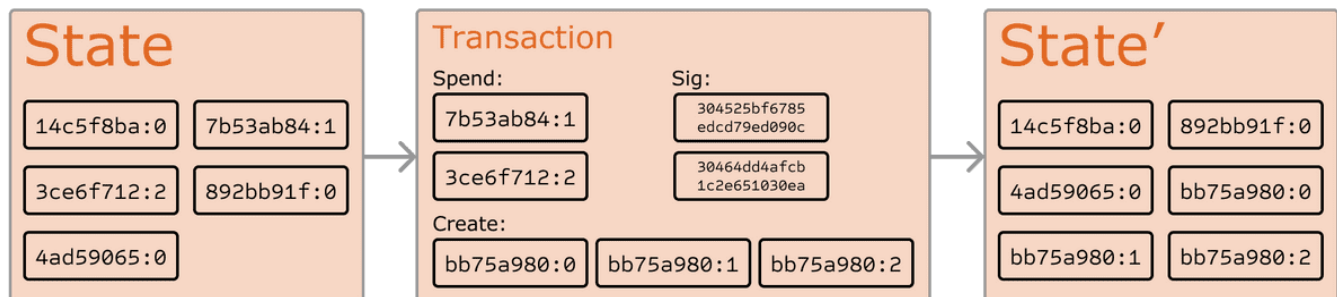
El mecanismo detrás de la prueba de trabajo fue un gran avance en el espacio porque resolvió simultáneamente dos problemas. Primero, proporcionó un algoritmo de consenso simple y moderadamente efectivo, permitiendo que los nodos de la red acuerden colectivamente un conjunto de actualizaciones canónicas del estado del libro mayor de Bitcoin. En segundo lugar

conjunto de actualizaciones canónicas del estado del libro mayor de Bitcoin. En segundo lugar, proporcionó un mecanismo para permitir la entrada libre en el proceso de consenso, resolviendo el problema político de decidir quién puede influir en el consenso, al mismo tiempo

que previene los ataques de las sibila. Lo hace sustituyendo una barrera formal a la participación, como el requisito de estar registrado como una entidad única en una lista en particular, con una barrera económica: el peso de un solo nodo en el proceso de votación por consenso es directamente proporcional al poder de cómputo que trae el nodo. Desde entonces, *prueba de participación*, calculando el peso de un nodo proporcional a sus tenencias de moneda y no recursos computacionales; la discusión de los méritos relativos de los dos enfoques está más allá del alcance de este documento, pero debe tenerse en cuenta que ambos enfoques pueden usarse para servir como la columna vertebral de una criptomoneda.

Aquí hay una publicación de blog de Vitalik Buterin, el fundador de Ethereum, sobre la [prehistoria de Ethereum](#) . [Aquí](#) hay otra publicación de blog con más historia.

## Bitcoin como sistema de transición estatal



Desde un punto de vista técnico, el libro mayor de una criptomoneda como Bitcoin se puede considerar como un sistema de transición de estado, donde hay un "estado" que consiste en el estado de propiedad de todos los bitcoins existentes y una "función de transición de estado" que toma un estado y una transacción y genera un nuevo estado que es el resultado. En un sistema bancario estándar, por ejemplo, el estado es un balance general, una transacción es una solicitud para mover \$ X de A a B, y la función de transición de estado reduce el valor en la cuenta de A en \$ X y aumenta el valor en B cuenta por \$ X. Si la cuenta de A tiene menos de \$ X en primer lugar, la función de transición de estado devuelve un error. Por tanto, se puede definir formalmente:

```
APPLY(S,TX) -> S' or ERROR
```

En el sistema bancario definido anteriormente:

```
APPLY({ Alice: $50, Bob: $50 }, "send $20 from Alice to Bob") = { Alice: $30, Bob: $70 }
```

Pero:

```
APPLY({ Alice: $50, Bob: $50 }, "send $70 from Alice to Bob") = ERROR
```

El "estado" en Bitcoin es la colección de todas las monedas (técnicamente, "salidas de transacciones no gastadas" o UTXO) que se han extraído y aún no se han gastado, y cada UTXO tiene una denominación y un propietario (definido por una dirección de 20 bytes que es esencialmente una clave pública criptográfica [fn.1](#)). Una transacción contiene una o más entradas, y cada entrada contiene una referencia a una UTXO existente y una firma criptográfica producida por la clave privada asociada con la dirección del propietario, y una o más salidas, y cada salida contiene una nueva UTXO que se agregará a el estado.

La función de transición de estado  $APPLY(S, TX) \rightarrow S'$  se puede definir aproximadamente de la siguiente manera:

1. Para cada entrada en  $TX$ :

- Si el UTXO referenciado no está  $S$ , devuelve un error.
- Si la firma proporcionada no coincide con el propietario del UTXO, devuelve un error.

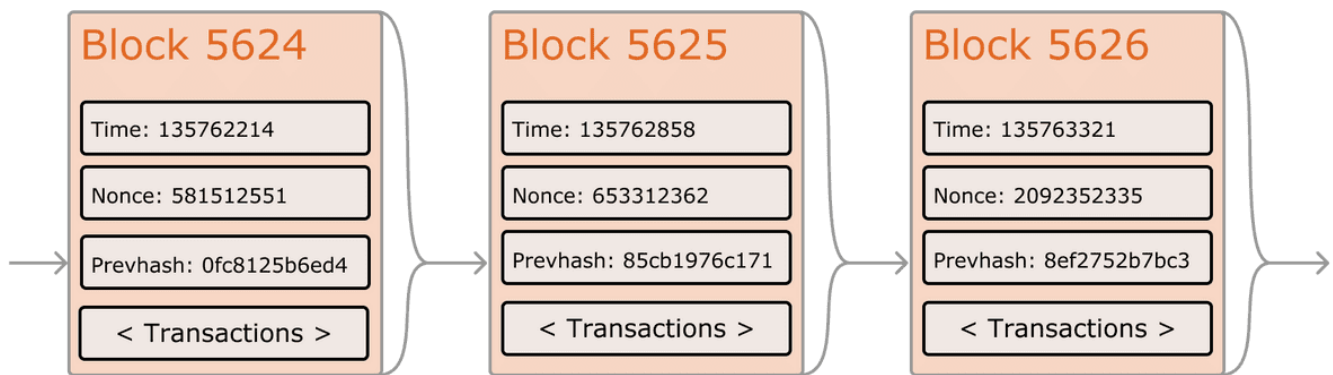
2. Si la suma de las denominaciones de todas las entradas UTXO es menor que la suma de las denominaciones de todas las salidas UTXO, devuelve un error.

3. Regrese  $S'$  con todas las entradas UTXO eliminadas y todas las salidas UTXO agregadas.

La primera mitad del primer paso evita que los remitentes de transacciones gasten monedas que no existen, la segunda mitad del primer paso evita que los remitentes de transacciones gasten las monedas de otras personas y el segundo paso refuerza la conservación del valor.

Para usar esto para el pago, el protocolo es el siguiente. Suponga que Alice quiere enviar 11.7 BTC a Bob. En primer lugar, Alice buscará un conjunto de UTXO disponibles que posea y que asciendan al menos a 11,7 BTC. Siendo realistas, Alice no podrá obtener exactamente 11,7 BTC; digamos que lo más pequeño que puede obtener es  $6 + 4 + 2 = 12$ . Luego crea una transacción con esas tres entradas y dos salidas. La primera salida será 11.7 BTC con la dirección de Bob como su propietario, y la segunda salida será el "cambio" restante de 0.3 BTC, siendo la propietaria la propia Alice.

## Minería



duro de un servidor centralizado para realizar un seguimiento del estado. Sin embargo, con Bitcoin estamos tratando de construir un sistema monetario descentralizado, por lo que necesitaremos combinar el sistema de transición estatal con un sistema de consenso para asegurarnos de que todos estén de acuerdo con el orden de las transacciones. El proceso de consenso descentralizado de Bitcoin requiere que los nodos de la red intenten continuamente producir paquetes de transacciones llamados "bloques". La red está destinada a producir aproximadamente un bloque cada diez minutos, y cada bloque contiene una marca de tiempo, un nonce, una referencia a (es decir. hash de) el bloque anterior y una lista de todas las transacciones que han tenido lugar desde el bloque anterior. Con el tiempo, esto crea una "cadena de bloques" persistente y en constante crecimiento que se actualiza constantemente para representar el estado más reciente del libro mayor de Bitcoin.

El algoritmo para comprobar si un bloque es válido, expresado en este paradigma, es el siguiente:

1. Compruebe si el bloque anterior al que hace referencia el bloque existe y es válido.

2. Compruebe que la marca de tiempo del bloque sea mayor que la del bloque anterior [fn. 2](#) v

2. Compruebe que la marca de tiempo del bloque sea mayor que la del bloque anterior y menos de 2 horas en el futuro.
3. Compruebe que la prueba de trabajo en el bloque sea válida.
4. Sea  $S[0]$  el estado al final del bloque anterior.
5. Supongamos que  $TX$  es la lista de transacciones del bloque con  $n$  transacciones. Para todo  $i$  incluido  $0 \dots n-1$ , establezca  $S[i+1] = \text{APPLY}(S[i], TX[i])$ . Si alguna aplicación devuelve un error, salga y devuelva falso.
6. Devuelve verdadero y registra  $S[n]$  como el estado al final de este bloque.

Esencialmente, cada transacción en el bloque debe proporcionar una transición de estado válida desde lo que era el estado canónico antes de que se ejecutara la transacción a un nuevo estado. Tenga en cuenta que el estado no está codificado en el bloque de ninguna manera; es puramente una abstracción que debe recordar el nodo de validación y solo se puede calcular (de forma segura) para cualquier bloque comenzando desde el estado de génesis y aplicando secuencialmente cada transacción en cada bloque. Además, tenga en cuenta que el orden en el que el minero incluye transacciones en el bloque es importante; si hay dos transacciones A y B en un bloque de modo que B gasta un UTXO creado por A, entonces el bloque será válido si A viene antes que B pero no de otra manera.

La única condición de validez presente en la lista anterior que no se encuentra en otros sistemas es el requisito de "prueba de trabajo". La condición precisa es que el hash SHA256 doble de cada bloque, tratado como un número de 256 bits, debe ser menor que un objetivo ajustado dinámicamente, que en el momento de escribir este artículo es aproximadamente  $2^{187}$ . El propósito de esto es hacer que la creación de bloques sea computacionalmente "difícil", evitando así que los atacantes sybil rehagan toda la cadena de bloques a su favor. Debido a que SHA256 está diseñado para ser una función pseudoaleatoria completamente impredecible, la única forma de crear un bloque válido es simplemente prueba y error, incrementando repetidamente el nonce y viendo si el nuevo hash coincide.

En el objetivo actual de  $\sim 2^{187}$ , la red debe realizar un promedio de  $\sim 2^{69}$  intentos antes de encontrar un bloque válido; en general, el objetivo es recalibrado por la red cada 2016 bloques, de modo que en promedio algún nodo de la red produce un nuevo bloque cada diez minutos. Para compensar a los mineros por este trabajo computacional, el minero de cada bloque tiene derecho a incluir una transacción que se otorgue a sí mismos 12.5 BTC de la nada. Además, si alguna transacción tiene una denominación total más alta en sus entradas que en sus salidas, la diferencia también va al minero como una "tarifa de transacción". Por cierto, este es también el único mecanismo mediante el cual se emiten BTC; el estado de génesis no contenía monedas

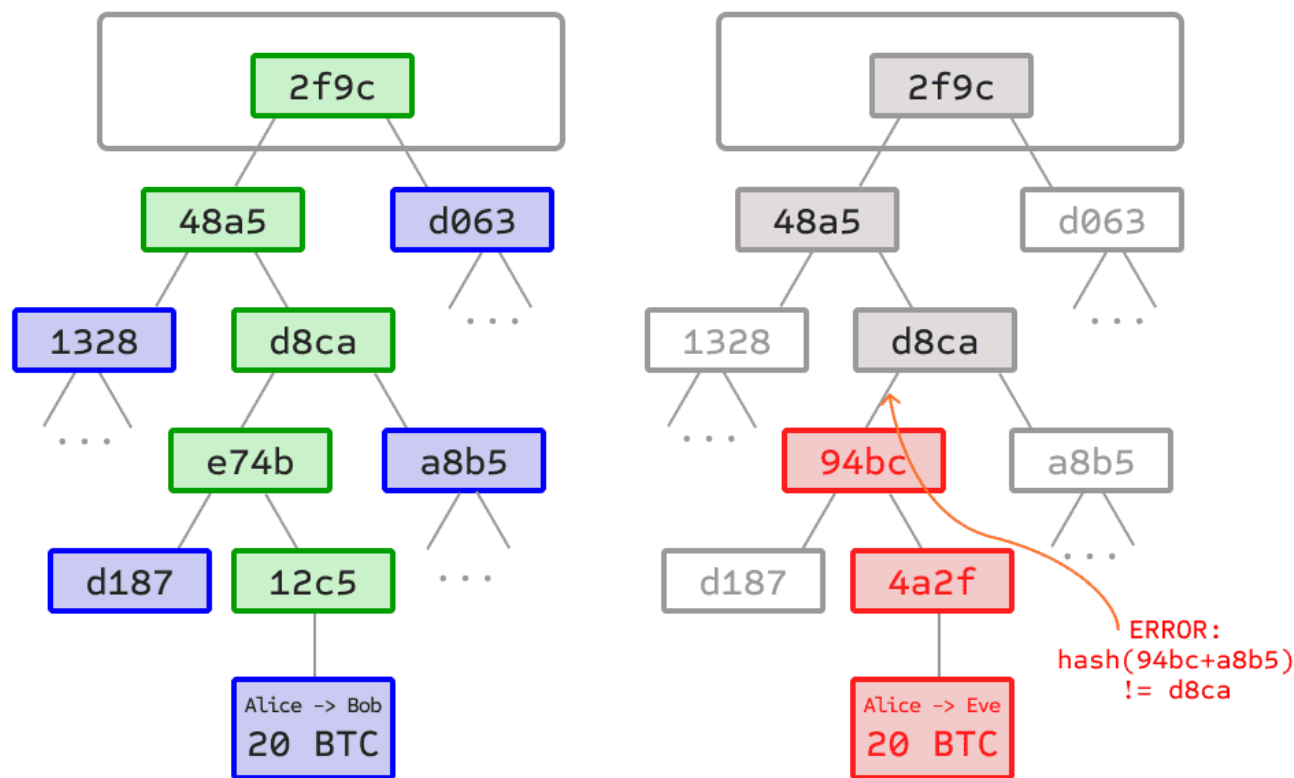
en absoluto.

Para comprender mejor el propósito de la minería, examinemos qué sucede en caso de un atacante malintencionado. Dado que se sabe que la criptografía subyacente de Bitcoin es segura, el atacante apuntará a la única parte del sistema Bitcoin que no está protegida directamente por la criptografía: el orden de las transacciones. La estrategia del atacante es simple:

1. Envíe 100 BTC a un comerciante a cambio de algún producto (preferiblemente un bien digital de entrega rápida)
2. Espere la entrega del producto
3. Producir otra transacción enviándose los mismos 100 BTC a sí mismo
4. Trate de convencer a la red de que su transacción para sí mismo fue la primera.

Una vez que se ha realizado el paso (1), después de unos minutos, algún minero incluirá la transacción en un bloque, digamos el bloque número 270. Después de aproximadamente una hora, se habrán agregado cinco bloques más a la cadena después de ese bloque, con cada uno de los esos bloques apuntan indirectamente a la transacción y así la "confirman". En este punto, el comerciante aceptará el pago como finalizado y entregará el producto; dado que asumimos que se trata de un bien digital, la entrega es instantánea. Ahora, el atacante crea otra transacción enviándose los 100 BTC a sí mismo. Si el atacante simplemente lo libera, la transacción no se procesará; los mineros intentarán correr `APPLY(S, TX)` y notar que TX consume un UTXO que ya no está en el estado. Entonces, en cambio, el atacante crea una "bifurcación" de la cadena de bloques, comenzando por extraer otra versión del bloque 270 que apunta al mismo bloque 269 como padre, pero con la nueva transacción en lugar de la anterior. Debido a que los datos del bloque son diferentes, esto requiere rehacer la prueba de trabajo. Además, la nueva versión del bloque 270 del atacante tiene un hash diferente, por lo que los bloques originales 271 a 275 no "apuntan" a él; por lo tanto, la cadena original y la nueva cadena del atacante están completamente separadas. La regla es que en una bifurcación se considera que la cadena de bloques más larga es la verdad, por lo que los mineros legítimos trabajarán en la cadena 275 mientras que el atacante solo está trabajando en la cadena 270. Para que el atacante haga que su cadena de bloques sea la más larga,

## Árboles Merkle



*Izquierda: basta con presentar solo un pequeño número de nodos en un árbol de Merkle para dar una prueba de la validez de una rama.*

*Derecha: cualquier intento de cambiar cualquier parte del árbol Merkle eventualmente conducirá a una inconsistencia en algún lugar de la cadena.*

An important scalability feature of Bitcoin is that the block is stored in a multi-level data structure. The "hash" of a block is actually only the hash of the block header, a roughly 200-byte piece of data that contains the timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle tree storing all transactions in the block. A Merkle tree is a type of binary tree, composed of a set of nodes with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, also formed from the hash of its two children, representing the "top" of the tree. The purpose of the Merkle tree is to allow the data in a block to be delivered piecemeal: a node can download only the header of a block from one source, the small part of the tree relevant to them from another source, and still be assured that all of the data is correct. The reason why this works is that hashes propagate upward: if a malicious user attempts to swap in a fake transaction into the bottom of a Merkle tree, this change will cause a change in the node above, and then a change in the node above that, finally changing the root of the tree and therefore the hash of the block, causing the protocol to register it as a completely different block (almost certainly with an invalid proof of work).



The Merkle tree protocol is arguably essential to long-term sustainability. A "full node" in the Bitcoin network, one that stores and processes the entirety of every block, takes up about 15 GB of disk space in the Bitcoin network as of April 2014, and is growing by over a gigabyte per month. Currently, this is viable for some desktop computers and not phones, and later on in the future only businesses and hobbyists will be able to participate. A protocol known as "simplified payment verification" (SPV) allows for another class of nodes to exist, called "light nodes", which download the block headers, verify the proof of work on the block headers, and then download only the "branches" associated with transactions that are relevant to them. This allows light nodes to determine with a strong guarantee of security what the status of any Bitcoin transaction, and their current balance, is while downloading only a very small portion of the entire blockchain.

## Alternative Blockchain Applications

The idea of taking the underlying blockchain idea and applying it to other concepts also has a long history. In 1998, Nick Szabo came out with the concept of [secure property titles with owner authority](#) <sup>2</sup>, a document describing how "new advances in replicated database technology" will allow for a blockchain-based system for storing a registry of who owns what land, creating an elaborate framework including concepts such as homesteading, adverse possession and Georgian land tax. However, there was unfortunately no effective replicated database system available at the time, and so the protocol was never implemented in practice. After 2009, however, once Bitcoin's decentralized consensus was developed a number of alternative applications rapidly began to emerge.

- **Namecoin** - created in 2010, [Namecoin](#) <sup>3</sup> is best described as a decentralized name registration database. In decentralized protocols like Tor, Bitcoin and BitMessage, there needs to be some way of identifying accounts so that other people can interact with them, but in all existing solutions the only kind of identifier available is a pseudorandom hash like 1LW79wp5ZBqaHW1jL5TCiBCrhQYtHagUWy . Ideally, one would like to be able to have an account with a name like "george". However, the problem is that if one person can create an account named "george" then someone else can use the same process to register "george" for themselves as well and impersonate them. The only solution is a first-to-file paradigm, where the first registerer succeeds and the second fails - a problem perfectly suited for the Bitcoin consensus protocol. Namecoin is the oldest, and most successful, implementation of a name registration system using such an idea.
- **Colored coins** - the purpose of [colored coins](#) <sup>4</sup> is to serve as a protocol to allow people to create their own digital currencies - or, in the important trivial case of a currency with one

unit, digital tokens, on the Bitcoin blockchain. In the colored coins protocol, one "issues" a new currency by publicly assigning a color to a specific Bitcoin UTXO, and the protocol recursively defines the color of other UTXO to be the same as the color of the inputs that the transaction creating them spent (some special rules apply in the case of mixed-color inputs). This allows users to maintain wallets containing only UTXO of a specific color and send them around much like regular bitcoins, backtracking through the blockchain to determine the color of any UTXO that they receive.

- **Metacoins** - the idea behind a metacoin is to have a protocol that lives on top of Bitcoin, using Bitcoin transactions to store metacoin transactions but having a different state transition function,  $\text{APPLY}'$ . Because the metacoin protocol cannot prevent invalid metacoin transactions from appearing in the Bitcoin blockchain, a rule is added that if  $\text{APPLY}'(S, TX)$  returns an error, the protocol defaults to  $\text{APPLY}'(S, TX) = S$ . This provides an easy mechanism for creating an arbitrary cryptocurrency protocol, potentially with advanced features that cannot be implemented inside of Bitcoin itself, but with a very low development cost since the complexities of mining and networking are already handled by the Bitcoin protocol. Metacoins have been used to implement some classes of financial contracts, name registration and decentralized exchange.

Thus, in general, there are two approaches toward building a consensus protocol: building an independent network, and building a protocol on top of Bitcoin. The former approach, while reasonably successful in the case of applications like Namecoin, is difficult to implement; each individual implementation needs to bootstrap an independent blockchain, as well as building and testing all of the necessary state transition and networking code. Additionally, we predict that the set of applications for decentralized consensus technology will follow a power law distribution where the vast majority of applications would be too small to warrant their own blockchain, and we note that there exist large classes of decentralized applications, particularly decentralized autonomous organizations, that need to interact with each other.

The Bitcoin-based approach, on the other hand, has the flaw that it does not inherit the simplified payment verification features of Bitcoin. SPV works for Bitcoin because it can use blockchain depth as a proxy for validity; at some point, once the ancestors of a transaction go far enough back, it is safe to say that they were legitimately part of the state. Blockchain-based meta-protocols, on the other hand, cannot force the blockchain not to include transactions that are not valid within the context of their own protocols. Hence, a fully secure SPV meta-protocol implementation would need to backward scan all the way to the beginning of the Bitcoin blockchain to determine whether or not certain transactions are valid. Currently, all "light" implementations of Bitcoin-based meta-protocols rely on a trusted server to provide the data, arguably a highly suboptimal result especially when one of the primary purposes of a

cryptocurrency is to eliminate the need for trust.

## Scripting

Even without any extensions, the Bitcoin protocol actually does facilitate a weak version of a concept of "smart contracts". UTXO in Bitcoin can be owned not just by a public key, but also by a more complicated script expressed in a simple stack-based programming language. In this paradigm, a transaction spending that UTXO must provide data that satisfies the script. Indeed, even the basic public key ownership mechanism is implemented via a script: the script takes an elliptic curve signature as input, verifies it against the transaction and the address that owns the UTXO, and returns 1 if the verification is successful and 0 otherwise. Other, more complicated, scripts exist for various additional use cases. For example, one can construct a script that requires signatures from two out of a given three private keys to validate ("multisig"), a setup useful for corporate accounts, secure savings accounts and some merchant escrow situations. Scripts can also be used to pay bounties for solutions to computational problems, and one can even construct a script that says something like "this Bitcoin UTXO is yours if you can provide an SPV proof that you sent a Dogecoin transaction of this denomination to me", essentially allowing decentralized cross-cryptocurrency exchange.

However, the scripting language as implemented in Bitcoin has several important limitations:

- **Lack of Turing-completeness** - that is to say, while there is a large subset of computation that the Bitcoin scripting language supports, it does not nearly support everything. The main category that is missing is loops. This is done to avoid infinite loops during transaction verification; theoretically it is a surmountable obstacle for script programmers, since any loop can be simulated by simply repeating the underlying code many times with an if statement, but it does lead to scripts that are very space-inefficient. For example, implementing an alternative elliptic curve signature algorithm would likely require 256 repeated multiplication rounds all individually included in the code.
- **Value-blindness** - there is no way for a UTXO script to provide fine-grained control over the amount that can be withdrawn. For example, one powerful use case of an oracle contract would be a hedging contract, where A and B put in \$1000 worth of BTC and after 30 days the script sends \$1000 worth of BTC to A and the rest to B. This would require an oracle to determine the value of 1 BTC in USD, but even then it is a massive improvement in terms of trust and infrastructure requirement over the fully centralized solutions that are available now. However, because UTXO are all-or-nothing, the only way to achieve this is through the very inefficient hack of having many UTXO of varying denominations (eg. one UTXO of  $2^k$  for

every  $k$  up to  $30$ ) and having  $O$  pick which UTXO to send to  $A$  and which to  $B$ .

- **Lack of state** - a [UTXO can either be spent or unspent](#) ; there is no opportunity for multi-stage contracts or scripts which keep any other internal state beyond that. This makes it hard to make multi-stage options contracts, decentralized exchange offers or two-stage cryptographic commitment protocols (necessary for secure computational bounties). It also means that UTXO can only be used to build simple, one-off contracts and not more complex "stateful" contracts such as decentralized organizations, and makes meta-protocols difficult to implement. Binary state combined with value-blindness also mean that another important application, withdrawal limits, is impossible.
- **Blockchain-blindness** - UTXO are blind to blockchain data such as the nonce, the timestamp and previous block hash. This severely limits applications in gambling, and several other categories, by depriving the scripting language of a potentially valuable source of randomness.

Thus, we see three approaches to building advanced applications on top of cryptocurrency: building a new blockchain, using scripting on top of Bitcoin, and building a meta-protocol on top of Bitcoin. Building a new blockchain allows for unlimited freedom in building a feature set, but at the cost of development time, bootstrapping effort and security. Using scripting is easy to implement and standardize, but is very limited in its capabilities, and meta-protocols, while easy, suffer from faults in scalability. With Ethereum, we intend to build an alternative framework that provides even larger gains in ease of development as well as even stronger light client properties, while at the same time allowing applications to share an economic environment and blockchain security.

## Ethereum

The intent of Ethereum is to create an alternative protocol for building decentralized applications, providing a different set of tradeoffs that we believe will be very useful for a large class of decentralized applications, with particular emphasis on situations where rapid development time, security for small and rarely used applications, and the ability of different applications to very efficiently interact, are important. Ethereum does this by building what is essentially the ultimate abstract foundational layer: a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. A bare-bones version of Namecoin can be written in two lines of

code, and other protocols like currencies and reputation systems can be built in under twenty. Smart contracts, cryptographic "boxes" that contain value and only unlock it if certain

conditions are met, can also be built on top of the platform, with vastly more power than that offered by Bitcoin scripting because of the added powers of Turing-completeness, value-awareness, blockchain-awareness and state.

## Philosophy

The design behind Ethereum is intended to follow the following principles:

1. **Simplicity:** the Ethereum protocol should be as simple as possible, even at the cost of some data storage or time inefficiency.<sup>[fn. 3](#)</sup> An average programmer should ideally be able to follow and implement the entire specification,<sup>[fn. 4](#)</sup> so as to fully realize the unprecedented democratizing potential that cryptocurrency brings and further the vision of Ethereum as a protocol that is open to all. Any optimization which adds complexity should not be included unless that optimization provides very substantial benefit.
2. **Universality:** a fundamental part of Ethereum's design philosophy is that Ethereum does not have "features".<sup>[fn. 5](#)</sup> Instead, Ethereum provides an internal Turing-complete scripting language, which a programmer can use to construct any smart contract or transaction type that can be mathematically defined. Want to invent your own financial derivative? With Ethereum, you can. Want to make your own currency? Set it up as an Ethereum contract. Want to set up a full-scale Daemon or Skynet? You may need to have a few thousand interlocking contracts, and be sure to feed them generously, to do that, but nothing is stopping you with Ethereum at your fingertips.
3. **Modularity:** the parts of the Ethereum protocol should be designed to be as modular and separable as possible. Over the course of development, our goal is to create a program where if one was to make a small protocol modification in one place, the application stack would continue to function without any further modification. Innovations such as Ethash (see the [Yellow Paper Appendix ↗](#) or [wiki article ↗](#)), modified Patricia trees ([Yellow Paper ↗](#), [wiki ↗](#)) and RLP ([YP ↗](#), [wiki ↗](#)) should be, and are, implemented as separate, feature-complete libraries. This is so that even though they are used in Ethereum, even if Ethereum does not require certain features, such features are still usable in other protocols as well. Ethereum development should be maximally done so as to benefit the entire cryptocurrency ecosystem, not just itself.
4. **Agility:** details of the Ethereum protocol are not set in stone. Although we will be extremely judicious about making modifications to high-level constructs, for instance with the [sharding](#)

[roadmap ↗](#), abstracting execution, with only data availability enshrined in consensus.

Computational tests later on in the development process may lead us to discover that certain modifications, e.g. to the protocol architecture or to the Ethereum Virtual Machine (EVM), will substantially improve scalability or security. If any such opportunities are found, we will exploit them.

5. **Non-discrimination** and **non-censorship**: the protocol should not attempt to actively restrict or prevent specific categories of usage. All regulatory mechanisms in the protocol should be designed to directly regulate the harm and not attempt to oppose specific undesirable applications. A programmer can even run an infinite loop script on top of Ethereum for as long as they are willing to keep paying the per-computational-step transaction fee.

## Ethereum Accounts

In Ethereum, the state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. An Ethereum account contains four fields:

- The **nonce**, a counter used to make sure each transaction can only be processed once
- The account's current **ether balance**
- The account's **contract code**, if present
- The account's **storage** (empty by default)

"Ether" is the main internal crypto-fuel of Ethereum, and is used to pay transaction fees. In general, there are two types of accounts: **externally owned accounts**, controlled by private keys, and **contract accounts**, controlled by their contract code. An externally owned account has no code, and one can send messages from an externally owned account by creating and signing a transaction; in a contract account, every time the contract account receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.

Note that "contracts" in Ethereum should not be seen as something that should be "fulfilled" or "complied with"; rather, they are more like "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own ether balance and their own key/value store to keep track of persistent variables.

## Messages and Transactions

The term "transaction" is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain:

- The recipient of the message
- A signature identifying the sender
- The amount of ether to transfer from the sender to the recipient
- An optional data field
- A `STARTGAS` value, representing the maximum number of computational steps the transaction execution is allowed to take
- A `GASPRICE` value, representing the fee the sender pays per computational step

The first three are standard fields expected in any cryptocurrency. The data field has no function by default, but the virtual machine has an opcode which a contract can use to access the data; as an example use case, if a contract is functioning as an on-blockchain domain registration service, then it may wish to interpret the data being passed to it as containing two "fields", the first field being a domain to register and the second field being the IP address to register it to. The contract would read these values from the message data and appropriately place them in storage.

The `STARTGAS` and `GASPRICE` fields are crucial for Ethereum's anti-denial of service model. In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas"; usually, a computational step costs 1 gas, but some operations cost higher amounts of gas because they are more computationally expensive, or increase the amount of data that must be stored as part of the state. There is also a fee of 5 gas for every byte in the transaction data. The intent of the fee system is to require an attacker to pay proportionately for every resource that they consume, including computation, bandwidth and storage; hence, any transaction that leads to the network consuming a greater amount of any of these resources must have a gas fee roughly proportional to the increment.

## Messages

Contracts have the ability to send "messages" to other contracts. Messages are virtual objects

Contracts have the ability to send messages to other contracts. Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment. A message contains:

- The sender of the message (implicit)
- The recipient of the message
- The amount of ether to transfer alongside the message
- An optional data field
- A `STARTGAS` value

Essentially, a message is like a transaction, except it is produced by a contract and not an external actor. A message is produced when a contract currently executing code executes the `CALL` opcode, which produces and executes a message. Like a transaction, a message leads to the recipient account running its code. Thus, contracts can have relationships with other contracts in exactly the same way that external actors can.

Note that the gas allowance assigned by a transaction or contract applies to the total gas consumed by that transaction and all sub-executions. For example, if an external actor A sends a transaction to B with 1000 gas, and B consumes 600 gas before sending a message to C, and the internal execution of C consumes 300 gas before returning, then B can spend another 100 gas before running out of gas.

## Ethereum State Transition Function



The Ethereum state transition function,  $\text{APPLY}(S, \text{TX}) \rightarrow S'$  can be defined as follows:

1. Check if the transaction is well-formed (ie. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.
2. Calculate the transaction fee as  $\text{STARTGAS} * \text{GASPRICE}$ , and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.
3. Initialize  $\text{GAS} = \text{STARTGAS}$ , and take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

For example, suppose that the contract's code is:

```
if !self.storage[calldataload(0)]:
    self.storage[calldataload(0)] = calldataload(32)
```

Note that in reality the contract code is written in the low-level EVM code; this example is written in Serpent, one of our high-level languages, for clarity, and can be compiled down to EVM code. Suppose that the contract's storage starts off empty, and a transaction is sent with 10 ether value, 2000 gas, 0.001 ether gasprice, and 64 bytes of data, with bytes 0-31 representing the number 2 and bytes 32-63 representing the string CHARLIE.<sup>[fn. 6](#)</sup> The process for the state transition function in this case is as follows:

1. Check that the transaction is valid and well formed.
2. Check that the transaction sender has at least  $2000 * 0.001 = 2$  ether. If it is, then subtract 2 ether from the sender's account.

3. Initialize  $\text{gas} = 2000$ : assuming the transaction is 170 bytes long and the byte fee is 5

3. Initialize `gas = 2000`, assuming the transaction is 170 bytes long and the byte-fee is 5, subtract 850 so that there is 1150 gas left.
4. Subtract 10 more ether from the sender's account, and add it to the contract's account.
5. Run the code. In this case, this is simple: it checks if the contract's storage at index 2 is used, notices that it is not, and so it sets the storage at index 2 to the value `CHARLIE`. Suppose this takes 187 gas, so the remaining amount of gas is  $1150 - 187 = 963$
6. Add  $963 * 0.001 = 0.963$  ether back to the sender's account, and return the resulting state.

If there was no contract at the receiving end of the transaction, then the total transaction fee would simply be equal to the provided `GASPRICE` multiplied by the length of the transaction in bytes, and the data sent alongside the transaction would be irrelevant.

Note that messages work equivalently to transactions in terms of reverts: if a message execution runs out of gas, then that message's execution, and all other executions triggered by that execution, revert, but parent executions do not need to revert. This means that it is "safe" for a contract to call another contract, as if A calls B with G gas then A's execution is guaranteed to lose at most G gas. Finally, note that there is an opcode, `CREATE`, that creates a contract; its execution mechanics are generally similar to `CALL`, with the exception that the output of the execution determines the code of a newly created contract.

## Code Execution

The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation. In general, code execution is an infinite loop that consists of repeatedly carrying out the operation at the current program counter (which begins at zero) and then incrementing the program counter by one, until the end of the code is reached or an error or `STOP` or `RETURN` instruction is detected. The operations have access to three types of space in which to store data:

- The **stack**, a last-in-first-out container to which values can be pushed and popped
- **Memory**, an infinitely expandable byte array
- The contract's long-term **storage**, a key/value store. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

The code can also access the value, sender and data of the incoming message, as well as block

header data, and the code can also return a byte array of data as an output.

The formal execution model of EVM code is surprisingly simple. While the Ethereum virtual machine is running, its full computational state can be defined by the tuple `(block_state, transaction, message, code, memory, stack, pc, gas)`, where `block_state` is the global state containing all accounts and includes balances and storage. At the start of every round of execution, the current instruction is found by taking the `pc`-th byte of `code` (or 0 if `pc >= len(code)`), and each instruction has its own definition in terms of how it affects the tuple. For example, `ADD` pops two items off the stack and pushes their sum, reduces `gas` by 1 and increments `pc` by 1, and `SSTORE` pops the top two items off the stack and inserts the second item into the contract's storage at the index specified by the first item. Although there are many ways to optimize Ethereum virtual machine execution via just-in-time compilation, a basic implementation of Ethereum can be done in a few hundred lines of code.

## Blockchain and Mining

The Ethereum blockchain is in many ways similar to the Bitcoin blockchain, although it does have some differences. The main difference between Ethereum and Bitcoin with regard to the blockchain architecture is that, unlike Bitcoin (which only contains a copy of the transaction list), Ethereum blocks contain a copy of both the transaction list and the most recent state. Aside from that, two other values, the block number and the difficulty, are also stored in the block. The basic block validation algorithm in Ethereum is as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Ethereum-specific concepts) are valid.
4. Check that the proof of work on the block is valid.

5. Let `S[0]` be the state at the end of the previous block

5. Let  $S[0]$  be the state at the end of the previous block.

6. Let  $TX$  be the block's transaction list, with  $n$  transactions. For all  $i$  in  $0 \dots n-1$ , set  $S[i+1] = \text{APPLY}(S[i], TX[i])$ . If any application returns an error, or if the total gas consumed in the block up until this point exceeds the  $GASLIMIT$ , return an error.

7. Let  $S\_FINAL$  be  $S[n]$ , but adding the block reward paid to the miner.

8. Check if the Merkle tree root of the state  $S\_FINAL$  is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.

The approach may seem highly inefficient at first glance, because it needs to store the entire state with each block, but in reality efficiency should be comparable to that of Bitcoin. The reason is that the state is stored in the tree structure, and after every block only a small part of the tree needs to be changed. Thus, in general, between two adjacent blocks the vast majority of the tree should be the same, and therefore the data can be stored once and referenced twice using pointers (ie. hashes of subtrees). A special kind of tree known as a "Patricia tree" is used to accomplish this, including a modification to the Merkle tree concept that allows for nodes to be inserted and deleted, and not just changed, efficiently. Additionally, because all of the state information is part of the last block, there is no need to store the entire blockchain history - a strategy which, if it could be applied to Bitcoin, can be calculated to provide 5-20x savings in space.

A commonly asked question is "where" contract code is executed, in terms of physical hardware. This has a simple answer: the process of executing contract code is part of the definition of the state transition function, which is part of the block validation algorithm, so if a transaction is added into block  $B$  the code execution spawned by that transaction will be executed by all nodes, now and in the future, that download and validate block  $B$ .

## Applications

In general, there are three types of applications on top of Ethereum. The first category is financial applications, providing users with more powerful ways of managing and entering into contracts using their money. This includes sub-currencies, financial derivatives, hedging contracts, savings wallets, wills, and ultimately even some classes of full-scale employment contracts. The second category is semi-financial applications, where money is involved but there is also a heavy non-monetary side to what is being done; a perfect example is self-enforcing bounties for solutions to computational problems. Finally, there are applications such as online voting and decentralized governance that are not financial at all.

## Token Systems

On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization. Token systems are surprisingly easy to implement in Ethereum. The key point to understand is that a currency, or token system, fundamentally is a database with one operation: subtract X units from A and give X units to B, with the provision that (1) A had at least X units before the transaction and (2) the transaction is approved by A. All that it takes to implement a token system is to implement this logic into a contract.

The basic code for implementing a token system in Serpent looks as follows:

```
def send(to, value):
    if self.storage[msg.sender] >= value:
        self.storage[msg.sender] = self.storage[msg.sender] - value
        self.storage[to] = self.storage[to] + value
```

This is essentially a literal implementation of the "banking system" state transition function described further above in this document. A few extra lines of code need to be added to provide for the initial step of distributing the currency units in the first place and a few other edge cases, and ideally a function would be added to let other contracts query for the balance of an address. But that's all there is to it. Theoretically, Ethereum-based token systems acting as sub-currencies can potentially include another important feature that on-chain Bitcoin-based meta-currencies lack: the ability to pay transaction fees directly in that currency. The way this would be implemented is that the contract would maintain an ether balance with which it would refund ether used to pay fees to the sender, and it would refill this balance by collecting the internal currency units that it takes in fees and reselling them in a constant running auction. Users would thus need to "activate" their accounts with ether, but once the ether is there it would be reusable because the contract would refund it each time.

## Financial derivatives and Stable-Value Currencies

Financial derivatives are the most common application of a "smart contract", and one of the simplest to implement in code. The main challenge in implementing financial contracts is that the majority of them require reference to an external price ticker; for example, a very desirable

application is a smart contract that hedges against the volatility of ether (or another cryptocurrency) with respect to the US dollar, but doing this requires the contract to know what the value of ETH/USD is. The simplest way to do this is through a "data feed" contract maintained by a specific party (eg. NASDAQ) designed so that that party has the ability to update the contract as needed, and providing an interface that allows other contracts to send a message to that contract and get back a response that provides the price.

Given that critical ingredient, the hedging contract would look as follows:

1. Wait for party A to input 1000 ether.
2. Wait for party B to input 1000 ether.
3. Record the USD value of 1000 ether, calculated by querying the data feed contract, in storage, say this is \$x.
4. After 30 days, allow A or B to "reactivate" the contract in order to send \$x worth of ether (calculated by querying the data feed contract again to get the new price) to A and the rest to B.

Such a contract would have significant potential in crypto-commerce. One of the main problems cited about cryptocurrency is the fact that it's volatile; although many users and merchants may want the security and convenience of dealing with cryptographic assets, they may not wish to face that prospect of losing 23% of the value of their funds in a single day. Up until now, the most commonly proposed solution has been issuer-backed assets; the idea is that an issuer creates a sub-currency in which they have the right to issue and revoke units, and provide one unit of the currency to anyone who provides them (offline) with one unit of a specified underlying asset (eg. gold, USD). The issuer then promises to provide one unit of the underlying asset to anyone who sends back one unit of the crypto-asset. This mechanism allows any non-cryptographic asset to be "uplifted" into a cryptographic asset, provided that the issuer can be trusted.

In practice, however, issuers are not always trustworthy, and in some cases the banking infrastructure is too weak, or too hostile, for such services to exist. Financial derivatives provide an alternative. Here, instead of a single issuer providing the funds to back up an asset, a decentralized market of speculators, betting that the price of a cryptographic reference asset (eg. ETH) will go up, plays that role. Unlike issuers, speculators have no option to default on their side of the bargain because the hedging contract holds their funds in escrow. Note that this approach is not fully decentralized, because a trusted source is still needed to provide the

price ticker, although arguably even still this is a massive improvement in terms of reducing infrastructure requirements (unlike being an issuer, issuing a price feed requires no licenses and can likely be categorized as free speech) and reducing the potential for fraud.

## Identity and Reputation Systems

The earliest alternative cryptocurrency of all, [Namecoin](#), attempted to use a Bitcoin-like blockchain to provide a name registration system, where users can register their names in a public database alongside other data. The major cited use case is for a [DNS](#) system, mapping domain names like "bitcoin.org" (or, in Namecoin's case, "bitcoin.bit") to an IP address. Other use cases include email authentication and potentially more advanced reputation systems. Here is the basic contract to provide a Namecoin-like name registration system on Ethereum:

```
def register(name, value):  
    if !self.storage[name]:  
        self.storage[name] = value
```

The contract is very simple; all it is a database inside the Ethereum network that can be added to, but not modified or removed from. Anyone can register a name with some value, and that registration then sticks forever. A more sophisticated name registration contract will also have a "function clause" allowing other contracts to query it, as well as a mechanism for the "owner" (ie. the first registerer) of a name to change the data or transfer ownership. One can even add reputation and web-of-trust functionality on top.

## Decentralized File Storage

Over the past few years, there have emerged a number of popular online file storage startups, the most prominent being Dropbox, seeking to allow users to upload a backup of their hard drive and have the service store the backup and allow the user to access it in exchange for a monthly fee. However, at this point the file storage market is at times relatively inefficient; a cursory look at various [existing solutions](#) shows that, particularly at the "uncanny valley" 20-200 GB level at which neither free quotas nor enterprise-level discounts kick in, monthly prices for mainstream file storage costs are such that you are paying for more than the cost of the entire hard drive in a single month. Ethereum contracts can allow for the development of a decentralized file storage ecosystem, where individual users can earn small quantities of money by renting out their own hard drives and unused space can be used to further drive down the

costs of file storage.

The key underpinning piece of such a device would be what we have termed the "decentralized Dropbox contract". This contract works as follows. First, one splits the desired data up into blocks, encrypting each block for privacy, and builds a Merkle tree out of it. One then makes a contract with the rule that, every N blocks, the contract would pick a random index in the Merkle tree (using the previous block hash, accessible from contract code, as a source of randomness), and give X ether to the first entity to supply a transaction with a simplified payment verification-like proof of ownership of the block at that particular index in the tree. When a user wants to re-download their file, they can use a micropayment channel protocol (eg. pay 1 szabo per 32 kilobytes) to recover the file; the most fee-efficient approach is for the payer not to publish the transaction until the end, instead replacing the transaction with a slightly more lucrative one with the same nonce after every 32 kilobytes.

An important feature of the protocol is that, although it may seem like one is trusting many random nodes not to decide to forget the file, one can reduce that risk down to near-zero by splitting the file into many pieces via secret sharing, and watching the contracts to see each piece is still in some node's possession. If a contract is still paying out money, that provides a cryptographic proof that someone out there is still storing the file.

## Decentralized Autonomous Organizations

The general concept of a "decentralized autonomous organization" is that of a virtual entity that has a certain set of members or shareholders which, perhaps with a 67% majority, have the right to spend the entity's funds and modify its code. The members would collectively decide on how the organization should allocate its funds. Methods for allocating a DAO's funds could range from bounties, salaries to even more exotic mechanisms such as an internal currency to reward work. This essentially replicates the legal trappings of a traditional company or nonprofit but using only cryptographic blockchain technology for enforcement. So far much of the talk around DAOs has been around the "capitalist" model of a "decentralized autonomous corporation" (DAC) with dividend-receiving shareholders and tradable shares; an alternative, perhaps described as a "decentralized autonomous community", would have all members have an equal share in the decision making and require 67% of existing members to agree to add or remove a member. The requirement that one person can only have one membership would then need to be enforced collectively by the group.



A general outline for how to code a DAO is as follows. The simplest design is simply a piece of self-modifying code that changes if two thirds of members agree on a change. Although code is theoretically immutable, one can easily get around this and have de-facto mutability by

having chunks of the code in separate contracts, and having the address of which contracts to call stored in the modifiable storage. In a simple implementation of such a DAO contract, there would be three transaction types, distinguished by the data provided in the transaction:

- $[0, i, K, V]$  to register a proposal with index  $i$  to change the address at storage index  $K$  to value  $V$
- $[1, i]$  to register a vote in favor of proposal  $i$
- $[2, i]$  to finalize proposal  $i$  if enough votes have been made

The contract would then have clauses for each of these. It would maintain a record of all open storage changes, along with a list of who voted for them. It would also have a list of all members. When any storage change gets to two thirds of members voting for it, a finalizing transaction could execute the change. A more sophisticated skeleton would also have built-in voting ability for features like sending a transaction, adding members and removing members, and may even provide for [Liquid Democracy](#)-style vote delegation (ie. anyone can assign someone to vote for them, and assignment is transitive so if A assigns B and B assigns C then C determines A's vote). This design would allow the DAO to grow organically as a decentralized community, allowing people to eventually delegate the task of filtering out who is a member to specialists, although unlike in the "current system" specialists can easily pop in and out of existence over time as individual community members change their alignments.

An alternative model is for a decentralized corporation, where any account can have zero or more shares, and two thirds of the shares are required to make a decision. A complete skeleton would involve asset management functionality, the ability to make an offer to buy or sell shares, and the ability to accept offers (preferably with an order-matching mechanism inside the contract). Delegation would also exist Liquid Democracy-style, generalizing the concept of a "board of directors".

## Further Applications

**1. Savings wallets.** Suppose that Alice wants to keep her funds safe, but is worried that she will lose or someone will hack her private key. She puts ether into a contract with Bob, a bank, as follows:

- Alice alone can withdraw a maximum of 1% of the funds per day.
- Bob alone can withdraw a maximum of 1% of the funds per day, but Alice has the ability to make a transaction with her key shutting off this ability.
- Alice and Bob together can withdraw anything.

Normally, 1% per day is enough for Alice, and if Alice wants to withdraw more she can contact Bob for help. If Alice's key gets hacked, she runs to Bob to move the funds to a new contract. If she loses her key, Bob will get the funds out eventually. If Bob turns out to be malicious, then she can turn off his ability to withdraw.

**2. Crop insurance.** One can easily make a financial derivatives contract by using a data feed of the weather instead of any price index. If a farmer in Iowa purchases a derivative that pays out inversely based on the precipitation in Iowa, then if there is a drought, the farmer will automatically receive money and if there is enough rain the farmer will be happy because their crops would do well. This can be expanded to natural disaster insurance generally.

**3. A decentralized data feed.** For financial contracts for difference, it may actually be possible to decentralize the data feed via a protocol called [SchellingCoin](#). SchellingCoin basically works as follows: N parties all put into the system the value of a given datum (eg. the ETH/USD price), the values are sorted, and everyone between the 25th and 75th percentile gets one token as a reward. Everyone has the incentive to provide the answer that everyone else will provide, and the only value that a large number of players can realistically agree on is the obvious default: the truth. This creates a decentralized protocol that can theoretically provide any number of values, including the ETH/USD price, the temperature in Berlin or even the result of a particular hard computation.

**4. Smart multisignature escrow.** Bitcoin allows multisignature transaction contracts where, for example, three out of a given five keys can spend the funds. Ethereum allows for more granularity; for example, four out of five can spend everything, three out of five can spend up to 10% per day, and two out of five can spend up to 0.5% per day. Additionally, Ethereum multisig is asynchronous - two parties can register their signatures on the blockchain at different times and the last signature will automatically send the transaction.

**5. Cloud computing.** The EVM technology can also be used to create a verifiable computing environment, allowing users to ask others to carry out computations and then optionally ask for

proofs that computations at certain randomly selected checkpoints were done correctly. This allows for the creation of a cloud computing market where any user can participate with their desktop, laptop or specialized server, and spot-checking together with security deposits can be used to ensure that the system is trustworthy (ie. nodes cannot profitably cheat). Although such a system may not be suitable for all tasks; tasks that require a high level of inter-process communication, for example, cannot easily be done on a large cloud of nodes. Other tasks, however, are much easier to parallelize; projects like SETI@home, folding@home and genetic algorithms can easily be implemented on top of such a platform.

**6. Peer-to-peer gambling.** Any number of peer-to-peer gambling protocols, such as Frank Stajano and Richard Clayton's [Cyberdice](#) , can be implemented on the Ethereum blockchain. The simplest gambling protocol is actually simply a contract for difference on the next block hash, and more advanced protocols can be built up from there, creating gambling services with near-zero fees that have no ability to cheat.

**7. Prediction markets.** Provided an oracle or SchellingCoin, prediction markets are also easy to implement, and prediction markets together with SchellingCoin may prove to be the first mainstream application of [futarchy](#) as a governance protocol for decentralized organizations.

**8. On-chain decentralized marketplaces,** using the identity and reputation system as a base.

## Miscellanea And Concerns

### Modified GHOST Implementation

The "Greedy Heaviest Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in [December 2013](#) . The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing a stale block 70% of the time (since the other 30% of the time A produced the last block and so will get mining data immediately) whereas B will have a risk of producing a stale block 90% of the time. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by

virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

As described by Sompolinsky and Zohar, GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale descendants of the block's ancestor (in Ethereum jargon, "uncles") are added to the calculation of which block has the largest total proof of work backing it. To solve the second issue of centralization bias, we go beyond the protocol described by Sompolinsky and Zohar, and also provide block rewards to stales: a stale block receives 87.5% of its base reward, and the nephew that includes the stale block receives the remaining 12.5%. Transaction fees, however, are not awarded to uncles.

Ethereum implements a simplified version of GHOST which only goes down seven levels. Specifically, it is defined as follows:

- A block must specify a parent, and it must specify 0 or more uncles
- An uncle included in block  $B$  must have the following properties:
- It must be a direct child of the  $k$ -th generation ancestor of  $B$ , where  $2 \leq k \leq 7$ .
- It cannot be an ancestor of  $B$
- An uncle must be a valid block header, but does not need to be a previously verified or even valid block
- An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block (non-double-inclusion)
- For every uncle  $U$  in block  $B$ , the miner of  $B$  gets an additional 3.125% added to its coinbase reward and the miner of  $U$  gets 93.75% of a standard coinbase reward.

This limited version of GHOST, with uncles includable only up to 7 generations, was used for two reasons. First, unlimited GHOST would include too many complications into the calculation of which uncles for a given block are valid. Second, unlimited GHOST with compensation as used in Ethereum removes the incentive for a miner to mine on the main chain and not the chain of a public attacker.

## Fees

Because every transaction published into the blockchain imposes on the network the cost of

needing to download and verify it, there is a need for some regulatory mechanism, typically involving transaction fees, to prevent abuse. The default approach, used in Bitcoin, is to have purely voluntary fees, relying on miners to act as the gatekeepers and set dynamic minimums. This approach has been received very favorably in the Bitcoin community particularly because it is "market-based", allowing supply and demand between miners and transaction senders determine the price. The problem with this line of reasoning is, however, that transaction processing is not a market; although it is intuitively attractive to construe transaction processing as a service that the miner is offering to the sender, in reality every transaction that a miner includes will need to be processed by every node in the network, so the vast majority of the cost of transaction processing is borne by third parties and not the miner that is making the decision of whether or not to include it. Hence, tragedy-of-the-commons problems are very likely to occur.

However, as it turns out this flaw in the market-based mechanism, when given a particular inaccurate simplifying assumption, magically cancels itself out. The argument is as follows. Suppose that:

1. A transaction leads to  $k$  operations, offering the reward  $kR$  to any miner that includes it where  $R$  is set by the sender and  $k$  and  $R$  are (roughly) visible to the miner beforehand.
2. An operation has a processing cost of  $C$  to any node (ie. all nodes have equal efficiency)
3. There are  $N$  mining nodes, each with exactly equal processing power (ie.  $1/N$  of total)
4. No non-mining full nodes exist.

A miner would be willing to process a transaction if the expected reward is greater than the cost. Thus, the expected reward is  $kR/N$  since the miner has a  $1/N$  chance of processing the next block, and the processing cost for the miner is simply  $kC$ . Hence, miners will include transactions where  $kR/N > kC$ , or  $R > NC$ . Note that  $R$  is the per-operation fee provided by the sender, and is thus a lower bound on the benefit that the sender derives from the transaction, and  $NC$  is the cost to the entire network together of processing an operation. Hence, miners have the incentive to include only those transactions for which the total utilitarian benefit exceeds the cost.

However, there are several important deviations from those assumptions in reality:

1. The miner does pay a higher cost to process the transaction than the other verifying nodes, since the extra verification time delays block propagation and thus increases the chance the block will become a stale.

2. There do exist non-mining full nodes.
3. The mining power distribution may end up radically inegalitarian in practice.
4. Speculators, political enemies and crazies whose utility function includes causing harm to the network do exist, and they can cleverly set up contracts where their cost is much lower than the cost paid by other verifying nodes.

(1) provides a tendency for the miner to include fewer transactions, and (2) increases  $NC$ ; hence, these two effects at least partially cancel each other out. [How? ↗](#) (3) and (4) are the major issue; to solve them we simply institute a floating cap: no block can have more operations than  $BLK\_LIMIT\_FACTOR$  times the long-term exponential moving average. Specifically:

```
blk.oplimit = floor((blk.parent.oplimit \* (EMAFACTOR - 1) +
floor(parent.opcount \* BLK\_LIMIT\_FACTOR)) / EMA\_FACTOR)
```

$BLK\_LIMIT\_FACTOR$  and  $EMA\_FACTOR$  are constants that will be set to 65536 and 1.5 for the time being, but will likely be changed after further analysis.

There is another factor disincentivizing large block sizes in Bitcoin: blocks that are large will take longer to propagate, and thus have a higher probability of becoming stales. In Ethereum, highly gas-consuming blocks can also take longer to propagate both because they are physically larger and because they take longer to process the transaction state transitions to validate. This delay disincentive is a significant consideration in Bitcoin, but less so in Ethereum because of the GHOST protocol; hence, relying on regulated block limits provides a more stable baseline.

## Computation And Turing-Completeness

An important note is that the Ethereum virtual machine is Turing-complete; this means that EVM code can encode any computation that can be conceivably carried out, including infinite loops. EVM code allows looping in two ways. First, there is a `JUMP` instruction that allows the program to jump back to a previous spot in the code, and a `JUMPI` instruction to do conditional jumping, allowing for statements like `while x < 27: x = x * 2`. Second, contracts can call other contracts, potentially allowing for looping through recursion. This naturally leads to a problem: can malicious users essentially shut miners and full nodes down by forcing them to enter into an infinite loop? The issue arises because of a problem in computer science known as the halting problem: there is no way to tell, in the general case, whether or

not a given program will ever halt.

As described in the state transition section, our solution works by requiring a transaction to set a maximum number of computational steps that it is allowed to take, and if execution takes longer computation is reverted but fees are still paid. Messages work in the same way. To show the motivation behind our solution, consider the following examples:

- An attacker creates a contract which runs an infinite loop, and then sends a transaction activating that loop to the miner. The miner will process the transaction, running the infinite loop, and wait for it to run out of gas. Even though the execution runs out of gas and stops halfway through, the transaction is still valid and the miner still claims the fee from the attacker for each computational step.
- An attacker creates a very long infinite loop with the intent of forcing the miner to keep computing for such a long time that by the time computation finishes a few more blocks will have come out and it will not be possible for the miner to include the transaction to claim the fee. However, the attacker will be required to submit a value for `STARTGAS` limiting the number of computational steps that execution can take, so the miner will know ahead of time that the computation will take an excessively large number of steps.
- An attacker sees a contract with code of some form like `send(A, contract.storage[A]); contract.storage[A] = 0`, and sends a transaction with just enough gas to run the first step but not the second (ie. making a withdrawal but not letting the balance go down). The contract author does not need to worry about protecting against such attacks, because if execution stops halfway through the changes they get reverted.
- A financial contract works by taking the median of nine proprietary data feeds in order to minimize risk. An attacker takes over one of the data feeds, which is designed to be modifiable via the variable-address-call mechanism described in the section on DAOs, and converts it to run an infinite loop, thereby attempting to force any attempts to claim funds from the financial contract to run out of gas. However, the financial contract can set a gas limit on the message to prevent this problem.

The alternative to Turing-completeness is Turing-incompleteness, where `JUMP` and `JUMPI` do not exist and only one copy of each contract is allowed to exist in the call stack at any given time. With this system, the fee system described and the uncertainties around the effectiveness of our solution might not be necessary, as the cost of executing a contract would be bounded above by its size. Additionally, Turing-incompleteness is not even that big a limitation; out of all the contract examples we have conceived internally, so far only one required a loop, and even that loop could be removed by making 26 repetitions of a one-line piece of code. Given the

serious implications of Turing-completeness, and the limited benefit, why not simply have a

Turing-incomplete language? In reality, however, Turing-incompleteness is far from a neat solution to the problem. To see why, consider the following contracts:

```
C0: call(C1); call(C1);  
C1: call(C2); call(C2);  
C2: call(C3); call(C3);  
...  
C49: call(C50); call(C50);  
C50: (run one step of a program and record the change in storage)
```

Now, send a transaction to A. Thus, in 51 transactions, we have a contract that takes up  $2^{50}$  computational steps. Miners could try to detect such logic bombs ahead of time by maintaining a value alongside each contract specifying the maximum number of computational steps that it can take, and calculating this for contracts calling other contracts recursively, but that would require miners to forbid contracts that create other contracts (since the creation and execution of all 26 contracts above could easily be rolled into a single contract). Another problematic point is that the address field of a message is a variable, so in general it may not even be possible to tell which other contracts a given contract will call ahead of time. Hence, all in all, we have a surprising conclusion: Turing-completeness is surprisingly easy to manage, and the lack of Turing-completeness is equally surprisingly difficult to manage unless the exact same controls are in place - but in that case why not just let the protocol be Turing-complete?

## Currency And Issuance

The Ethereum network includes its own built-in currency, ether, which serves the dual purpose of providing a primary liquidity layer to allow for efficient exchange between various types of digital assets and, more importantly, of providing a mechanism for paying transaction fees. For convenience and to avoid future argument (see the current mBTC/uBTC/satoshi debate in Bitcoin), the denominations will be pre-labelled:

- 1: wei
- $10^{12}$ : szabo
- $10^{15}$ : finney
- $10^{18}$ : ether



This should be taken as an expanded version of the concept of "dollars" and "cents" or "BTC" and "satoshi". In the near future, we expect "ether" to be used for ordinary transactions, "finney" for microtransactions and "szabo" and "wei" for technical discussions around fees and protocol implementation; the remaining denominations may become useful later and should not be included in clients at this point.

The issuance model will be as follows:

- Ether will be released in a currency sale at the price of 1000-2000 ether per BTC, a mechanism intended to fund the Ethereum organization and pay for development that has been used with success by other platforms such as Mastercoin and NXT. Earlier buyers will benefit from larger discounts. The BTC received from the sale will be used entirely to pay salaries and bounties to developers and invested into various for-profit and non-profit projects in the Ethereum and cryptocurrency ecosystem.
- 0.099x the total amount sold (60102216 ETH) will be allocated to the organization to compensate early contributors and pay ETH-denominated expenses before the genesis block.
- 0.099x the total amount sold will be maintained as a long-term reserve.
- 0.26x the total amount sold will be allocated to miners per year forever after that point.

Group At launch After 1 year After 5 years

Currency units	1.198X	1.458X	2.498X	Purchasers	83.5%	68.6%	40.0%	Reserve spent pre-sale
	8.26%	6.79%	3.96%	Reserve used post-sale	8.26%	6.79%	3.96%	Miners
					0%	17.8%	52.0%	

### Long-Term Supply Growth Rate (percent)

*Despite the linear currency issuance, just like with Bitcoin over time the supply growth rate nevertheless tends to zero*

The two main choices in the above model are (1) the existence and size of an endowment pool, and (2) the existence of a permanently growing linear supply, as opposed to a capped supply as in Bitcoin. The justification of the endowment pool is as follows. If the endowment pool did not exist, and the linear issuance reduced to 0.217x to provide the same inflation rate, then the total quantity of ether would be 16.5% less and so each unit would be 19.8% more valuable. Hence, in the equilibrium 19.8% more ether would be purchased in the sale, so each unit would once again be exactly as valuable as before. The organization would also then have 1.198x as much BTC, which can be considered to be split into two slices: the original BTC, and the additional 0.198x. Hence, this situation is *exactly equivalent* to the endowment, but with one important difference: the organization holds purely BTC, and so is not incentivized to support the value of the ether unit.

The permanent linear supply growth model reduces the risk of what some see as excessive wealth concentration in Bitcoin, and gives individuals living in present and future eras a fair chance to acquire currency units, while at the same time retaining a strong incentive to obtain and hold ether because the "supply growth rate" as a percentage still tends to zero over time. We also theorize that because coins are always lost over time due to carelessness, death, etc, and coin loss can be modeled as a percentage of the total supply per year, that the total currency supply in circulation will in fact eventually stabilize at a value equal to the annual issuance divided by the loss rate (eg. at a loss rate of 1%, once the supply reaches 26X then 0.26X will be mined and 0.26X lost every year, creating an equilibrium).

Note that in the future, it is likely that Ethereum will switch to a proof-of-stake model for security, reducing the issuance requirement to somewhere between zero and 0.05X per year. In

the event that the Ethereum organization loses funding or for any other reason disappears, we leave open a "social contract": anyone has the right to create a future candidate version of

Ethereum, with the only condition being that the quantity of ether must be at most equal to  $60102216 * (1.198 + 0.26 * n)$  where  $n$  is the number of years after the genesis block. Creators are free to crowd-sell or otherwise assign some or all of the difference between the PoS-driven supply expansion and the maximum allowable supply expansion to pay for development. Candidate upgrades that do not comply with the social contract may justifiably be forked into compliant versions.

## Mining Centralization


The Bitcoin mining algorithm works by having miners compute SHA256 on slightly modified versions of the block header millions of times over and over again, until eventually one node comes up with a version whose hash is less than the target (currently around  $2^{192}$ ). However, this mining algorithm is vulnerable to two forms of centralization. First, the mining ecosystem has come to be dominated by ASICs (application-specific integrated circuits), computer chips designed for, and therefore thousands of times more efficient at, the specific task of Bitcoin mining. This means that Bitcoin mining is no longer a highly decentralized and egalitarian pursuit, requiring millions of dollars of capital to effectively participate in. Second, most Bitcoin miners do not actually perform block validation locally; instead, they rely on a centralized mining pool to provide the block headers. This problem is arguably worse: as of the time of this writing, the top three mining pools indirectly control roughly 50% of processing power in the Bitcoin network, although this is mitigated by the fact that miners can switch to other mining pools if a pool or coalition attempts a 51% attack.

The current intent at Ethereum is to use a mining algorithm where miners are required to fetch random data from the state, compute some randomly selected transactions from the last  $N$  blocks in the blockchain, and return the hash of the result. This has two important benefits. First, Ethereum contracts can include any kind of computation, so an Ethereum ASIC would essentially be an ASIC for general computation - ie. a better CPU. Second, mining requires access to the entire blockchain, forcing miners to store the entire blockchain and at least be capable of verifying every transaction. This removes the need for centralized mining pools; although mining pools can still serve the legitimate role of evening out the randomness of reward distribution, this function can be served equally well by peer-to-peer pools with no central control.

This model is untested, and there may be difficulties along the way in avoiding certain clever optimizations when using contract execution as a mining algorithm. However, one notably interesting feature of this algorithm is that it allows anyone to "poison the well", by introducing a large number of contracts into the blockchain specifically designed to stymie certain ASICs. The economic incentives exist for ASIC manufacturers to use such a trick to attack each other. Thus, the solution that we are developing is ultimately an adaptive economic human solution rather than purely a technical one.

## Scalability

One common concern about Ethereum is the issue of scalability. Like Bitcoin, Ethereum suffers from the flaw that every transaction needs to be processed by every node in the network. With Bitcoin, the size of the current blockchain rests at about 15 GB, growing by about 1 MB per hour. If the Bitcoin network were to process Visa's 2000 transactions per second, it would grow by 1 MB per three seconds (1 GB per hour, 8 TB per year). Ethereum is likely to suffer a similar growth pattern, worsened by the fact that there will be many applications on top of the Ethereum blockchain instead of just a currency as is the case with Bitcoin, but ameliorated by the fact that Ethereum full nodes need to store just the state instead of the entire blockchain history.

The problem with such a large blockchain size is centralization risk. If the blockchain size increases to, say, 100 TB, then the likely scenario would be that only a very small number of large businesses would run full nodes, with all regular users using light SPV nodes. In such a situation, there arises the potential concern that the full nodes could band together and all agree to cheat in some profitable fashion (eg. change the block reward, give themselves BTC). Light nodes would have no way of detecting this immediately. Of course, at least one honest full node would likely exist, and after a few hours information about the fraud would trickle out through channels like Reddit, but at that point it would be too late: it would be up to the ordinary users to organize an effort to blacklist the given blocks, a massive and likely infeasible coordination problem on a similar scale as that of pulling off a successful 51% attack. In the case of Bitcoin, this is currently a problem, but there exists a blockchain modification [suggested by Peter Todd](#)  which will alleviate this issue.

In the near term, Ethereum will use two additional strategies to cope with this problem. First, because of the blockchain-based mining algorithms, at least every miner will be forced to be a full node, creating a lower bound on the number of full nodes. Second and more importantly, however, we will include an intermediate state tree root in the blockchain after processing each transaction. Even if block validation is centralized, as long as one honest verifying node exists

transaction. Even if block validation is centralized, as long as one honest verifying node exists, the centralization problem can be circumvented via a verification protocol. If a miner publishes an invalid block, that block must either be badly formatted, or the state  $S[n]$  is incorrect.

Since  $S[0]$  is known to be correct, there must be some first state  $S[i]$  that is incorrect where  $S[i-1]$  is correct. The verifying node would provide the index  $i$ , along with a "proof of invalidity" consisting of the subset of Patricia tree nodes needing to process  $\text{APPLY}(S[i-1], \text{TX}[i]) \rightarrow S[i]$ . Nodes would be able to use those Patricia nodes to run that part of the computation, and see that the  $S[i]$  generated does not match the  $S[i]$  provided.

Another, more sophisticated, attack would involve the malicious miners publishing incomplete blocks, so the full information does not even exist to determine whether or not blocks are valid. The solution to this is a challenge-response protocol: verification nodes issue "challenges" in the form of target transaction indices, and upon receiving a node a light node treats the block as untrusted until another node, whether the miner or another verifier, provides a subset of Patricia nodes as a proof of validity.

## Conclusion

The Ethereum protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits, financial contracts, gambling markets and the like via a highly generalized programming language. The Ethereum protocol would not "support" any of the applications directly, but the existence of a Turing-complete programming language means that arbitrary contracts can theoretically be created for any transaction type or application. What is more interesting about Ethereum, however, is that the Ethereum protocol moves far beyond just currency. Protocols around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer. Finally, there is also a substantial array of applications that have nothing to do with money at all.

The concept of an arbitrary state transition function as implemented by the Ethereum protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, Ethereum is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come.

## Notes and Further Reading

### Notes

1. A sophisticated reader may notice that in fact a Bitcoin address is the hash of the elliptic curve public key, and not the public key itself. However, it is in fact perfectly legitimate cryptographic terminology to refer to the pubkey hash as a public key itself. This is because Bitcoin's cryptography can be considered to be a custom digital signature algorithm, where the public key consists of the hash of the ECC pubkey, the signature consists of the ECC pubkey concatenated with the ECC signature, and the verification algorithm involves checking the ECC pubkey in the signature against the ECC pubkey hash provided as a public key and then verifying the ECC signature against the ECC pubkey.
2. Technically, the median of the 11 previous blocks.
3. The Ethereum protocol should be as simple as practical, but it may be necessary to have quite a high level of complexity, for instance to scale, to internalize costs of storage, bandwidth and I/O, for security, privacy, transparency, etc. Where complexity is necessary, documentation should be as clear, concise and up-to-date as possible, so that someone completely unschooled in Ethereum can learn it and become an expert.
4. See the [Yellow Paper](#) for the Ethereum Virtual Machine (which is useful as a specification and as a reference for building an Ethereum client from scratch), while also there are many topics in the [Ethereum wiki](#), such as sharding development, core development, dapp development, research, Casper R&D, and networking protocols. For research and possible future implementation there is [ethresear.ch](#).
5. Another way of expressing this is abstraction. The [latest roadmap](#) is planning to abstract execution, allowing execution engines to not necessarily have to follow one canonical specification, but for instance it could be tailored for a specific application, as well as a shard. (This heterogeneity of execution engines is not explicitly stated in the roadmap. There is also heterogeneous sharding, which Vlad Zamfir conceptualized.)
6. Internally, 2 and "CHARLIE" are both numbers, with the latter being in big-endian base 256 representation. Numbers can be at least 0 and at most  $2^{256}-1$ .

### Further Reading

1. [Intrinsic value ↗](#)
2. [Smart property ↗](#)
3. [Smart contracts ↗](#)
4. [B-money ↗](#)
5. [Reusable proofs of work ↗](#)
6. [Secure property titles with owner authority ↗](#)
7. [Bitcoin whitepaper ↗](#)
8. [Namecoin ↗](#)
9. [Zooko's triangle ↗](#)
10. [Libro blanco de monedas de colores ↗](#)
11. [Libro blanco de Mastercoin ↗](#)
12. [Corporaciones autónomas descentralizadas, Revista Bitcoin ↗](#)
13. [Verificación de pago simplificada ↗](#)
14. [Árboles Merkle ↗](#)
15. [Patricia árboles ↗](#)
16. [FANTASMA ↗](#)
17. [StorJ y agentes autónomos, Jeff Garzik ↗](#)
18. [Mike Hearn sobre propiedad inteligente en el Festival de Turing ↗](#)
19. [Ethereum RLP ↗](#)
20. [Árboles Ethereum Merkle Patricia ↗](#)
21. [Peter Todd sobre árboles de suma Merkle ↗](#)

Para conocer el historial del libro blanco, consulte <https://github.com/ethereum/wiki/blob/old-before-deleting-all-files-go-to-wiki-wiki-instead/old-whitepaper-for-historical-reference.md> ↗

*Ethereum, como muchos proyectos de software de código abierto impulsados por la comunidad, ha evolucionado desde sus inicios. Para conocer los últimos desarrollos de Ethereum y cómo se realizan los cambios en el protocolo, recomendamos [esta guía](#) .*

Última actualización del sitio web : 18 de marzo de 2021



## Usar Ethereum

Carteras de Ethereum

Conseguir ETH

Aplicaciones descentralizadas (dapps)

Monedas estables

Apostar ETH

## Aprender

¿Qué es Ethereum?

¿Qué es el ether (ETH)?

Guías y recursos

Historia de Ethereum

[Informe de Ethereum](#)

Ethereum 2.0

Glosario de Ethereum

Propuestas de mejora de Ethereum

## Desarrolladores

Comenzar

Documentación

Tutoriales

Aprender mediante codificación

Configurar entorno local

Recursos para desarrolladores

## Ecosistema

Comunidad Ethereum

Fundación Ethereum

Blog de la Fundación Ethereum ↗

Programa de soporte del ecosistema ↗

Programas de subvenciones para ecosistemas

Activos de marca de Ethereum

Devcon ↗

## Empresa

<https://ethereum.org/es/whitepaper/>

## Acerca de ethereum.org



[Empreses](#)

[Red principal de Ethereum](#)

[Red privada de Ethereum](#)

[Empresa](#)

[Redes de Ethereum.org](#)

[Sobre nosotros](#)

[Trabajos](#)

[Contribuciones](#)

[Idiomas admitidos](#)

[Política de privacidad](#)

[Términos de uso](#)

[Política de cookies](#)

[Contacto ↗](#)