Ethereum: Platform Review

Opportunities and Challenges for Private and Consortium Blockchains

Vitalik Buterin

The Ethereum platform was originally conceived in November 2013 with the goal of creating a more generalized blockchain platform, combining together the notion of ***public economic consensus*** via proof of work (or eventually proof of stake) with the abstraction power of a ***stateful Turing-complete virtual machine*** in order to allow application developers to much more easily create applications that benefit from the decentralization and security properties of blockchains, and particularly avoid the need to create a new blockchain for each new application. Whereas previous blockchain protocols could be viewed as single-function tools, like pocket calculators, or at best multi-function tools like Swiss army knives, Ethereum is the smartphone of blockchains: a universal platform where, whatever you want to build, you can just build it as an "app", and Ethereum users will be able to benefit from it immediately without downloading any new special software.

Although the project originally appeared as a proposed feature upgrade to Mastercoin offering support for wider arrays of financial contracts, interest quickly expanded to a much larger set of applications including financial contracts, bets, digital token issuance, decentralized file storage incentivization, voting, "decentralized autonomous organizations" and much more. Development of the Ethereum platform was funded via a public "crowdsale" event in August 2014, and the public Ethereum blockchain was launched in summer 2015. Since then, it has seen the emergence of over 100 applications ranging from financial clearing and settlement to insurance, digital asset issuance, and even non-financial applications in areas including voting and the Internet of things.

Basic Design

All blockchains have a notion of a **history** - the set of all previous transactions and blocks and the order in which they took place - and the **state** - "currently relevant" data that determines whether or not a given transaction is valid and what the state after processing a transaction will be. Blockchain protocols also have a notion of a **state transition rule**: given what the state was before, and given a particular transaction, (i) is the transaction valid, and (ii) what will the state be after the transaction?

We can give an example using Bitcoin[1]. In Bitcoin, the state is the set of account balances (eg. address 39BaMQCphFXyYAvcoGpeKtnptLJ9v6cdFY has 522.11790015 bitcoins, address 375zAYokrLtBVv6bY47bf2YdJH1EYsgyNR has 375 bitcoins....). The state transition function takes a transaction containing a sender address, a destination address and a value and asks: (i) is the transaction correctly cryptographically signed by the sender, and (ii) does the sender account contain enough bitcoins to send? If either answer is negative, the transaction is invalid and **cannot** be included in a block, ie. if a block contains a

[1] As explained in a later section, Bitcoin actually uses a concept of UTXOs and not balances; however, the description of the protocol using balances is simpler and suffices to illustrate the concept.

transaction which is invalid under the current state, then that block is ignored by the network[2]. If both answers are positive, then the transaction value is subtracted from the sender's balance and added to that of the recipient.

In Ethereum, the design is somewhat more complicated. The state can be described as the set of all **accounts**, where each account is either an **externally owned account** (EOA) or a **contract**. If the account is an EOA, the state simply stores the account's balance in ether (Ethereum's internal crypto-token, similar to bitcoin or XRP in function) and a sequence number used to prevent transaction replay attacks. If the account is a contract, the state stores the contract's **code**, as well as the contract's **storage**, a key-value database.

A **transaction** in Ethereum specifies (alongside other information that will later be described as required) a destination address, a quantity of ether to transact and a "data" field which theoretically can contain any information (and also a sender address, although this is implicit in the signature and thereforeis not specified explicitly). If a transaction is sent to an EOA, or a not-yet-existent account, then it simply acts as a transfer of ether, and serves no other purpose. If a transaction is sent to a contract, however, the contract's code runs. This code has the ability to:

- Read the transaction data.
- Read the quantity of ether sent in the transaction
- Read and write to the contract's own storage.
- Read environment variables (eg. timestamp, block difficulty, previous block hashes)
- Send an "internal transaction" to another contract.

Essentially, one can think of a contract as being a kind of "virtual object" stored in the Ethereum state, but one which can maintain its own internal persistent memory, and which has the right to perform the same kinds of actions and have the same kinds of relationships with other contracts that external users can. An internal transaction is a transaction created by a contract; like a regular "external" transaction, it also has an implicit sender, a destination, a quantity of ether, and message data, and if an internal transaction is sent to a contract then *that contract's* code runs. Upon exiting execution, the contract's code has the ability to return zero or more bytes of data, allowing internal transactions to also be used to "ask" other contracts for specific

[2] More precisely, miners are *supposed* to reject blocks that contain invalid transactions. In practice, miners may earn higher profits by not bothering to validate blocks, instead free-riding off of the presumed validation of others. If enough miners do this, such behavior can lead to dangerously long forks of invalid blocks, as took place in June 2015. The "Verifier's Dilemma" paper from the National University of Singapore discusses the economic incentives behind non-validating miners in an Ethereum context in much more detail; the general conclusion is that if block verification is sufficiently computationally intensive then the system could flip to the non-validating equilibrium, and this gives yet another reason why public blockchains have constraints on their throughput that place their maximum safe capacity *far below* the theoretical processing power of a single CPU.

information. A new contract can be created either by a transaction, by placing the contract's code in the transaction data and not specifying a destination address, or from inside of contract code itself via the CREATE opcode.

In simple terms, rather than enforcing one specific set of rules targeted toward one particular application, Ethereum allows users to write programs specifying whichever rules they want, upload the programs to the blockchain, and the blockchain will interpret the rules for them.

On the public Ethereum blockchain, this contract mechanism has been used in many ways:

- As databases to keep track of issuer-backed assets
- As "smart contracts" (see below for a more in-depth explanation of the concept) that control other assets (including issuer-backed assets and ether) and send them to specific parties depending on particular conditions; this generally further decomposes into several subcategories including (i) financial contracts (eg. CFDs, binary options, derivatives), (ii) escrow (implementing "trust-free atomic swaps of digital assets"), (iii) multi-party protocols such as auctions, assurance contracts, economic games that incentivize revealing specific information, etc.
- As registries for an on-blockchain domain name system
- As accounts that represent a user and organization but have complex access permissions, eg. multisig
- As "software libraries", allowing code to be written and published to the blockchain once and then used by anyone else

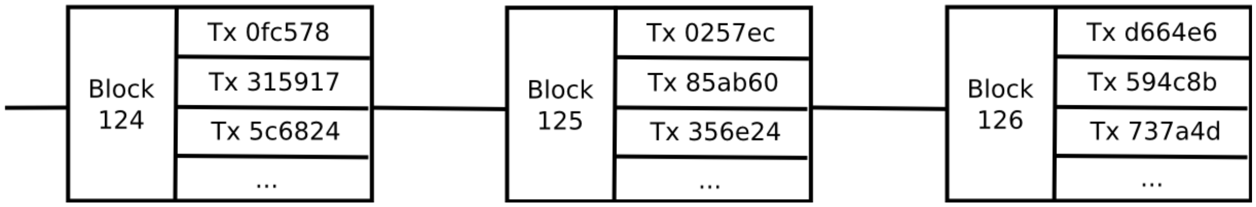
The notion of a "**smart contract**", defined most simply as "a computer program that directly controls digital assets”[3], is particularly important. Contracts have their own addresses, and so can serve as owners of digital assets in the same way that users can; if a contract does "own" digital assets, that means that (i) only the contract's code executing can send the asset to another party, and (ii) every party that sees and can verify the blockchain is aware that the asset is under this program's control.

For example, one can implement a trust-free trade of asset A for asset B by having the owner of asset A send the asset into a program whose code is roughly "if I receive asset B within 24 hours, I will send asset A to the sender and send asset B to my creator, otherwise I will return asset A to my creator". The owner of asset B can see that asset A is under the control of the contract, and so knows that if they send asset B into the

[3] The question of who actually legally owns assets while they are under the control of a smart contract is complex; common law naturally did not evolve with the understanding that the *underlying owners* of assets could be computer programs executed on a consensus ledger. See Robert Sams’s footnote 32 from Consensus-as-a-service for more details.

contract as well, the contract will execute the trade fairly and correctly. Contracts do not have "owners"; once the original owner of asset A sends the asset into the contract, they no longer have any way to manipulate the contract to get it back, they can only wait for either the trade to succeed and for them to receive asset B or for the trade not to succeed within 24 hours at which point they will automatically get asset A back.

Blockchain

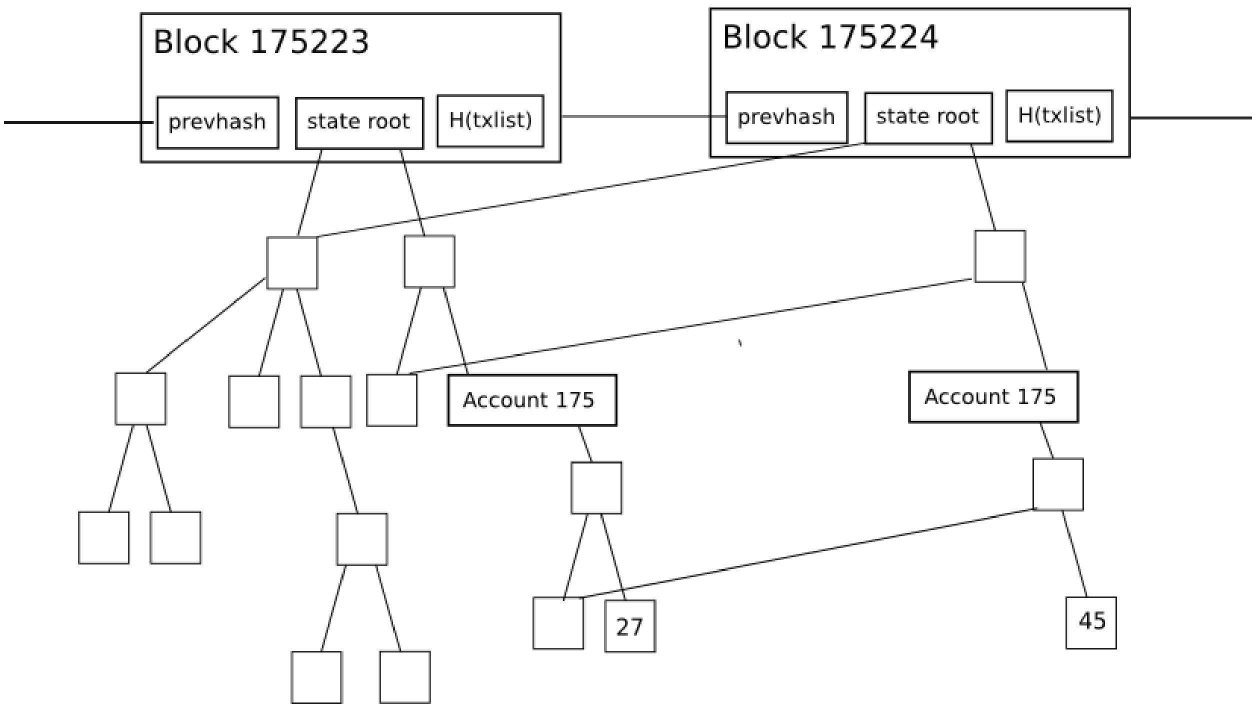


The Ethereum blockchain is, at a high level, very similar to Bitcoin. Primarily and most importantly, the blockchain consists of a series of blocks with each block containing a pointer to the previous block and an ordered list of transactions. The blocks are secured by proof of work, with the "longest chain" (as defined by total difficulty) defining the set of confirmed transactions and the order in which they must be processed.

In order to arrive at the "current state" of the Ethereum blockchain, a node can start from the "genesis state" (a commonly agreed initial state which is included with every Ethereum client) and process every transaction, applying any resulting balance/sequence number/code/storage changes from transaction processing and code execution sequentially. The same process happens in Bitcoin; although Ethereum's emphasis on the "state transition" *model* of transaction execution is unique (whereas, for example, in Bitcoin, a transaction is often philosophically viewed as spending "outputs" of a *previous transaction in the history*, not objects *in the state*), the way that the code works in Ethereum, Bitcoin and other protocols such as Ripple, Dogecoin, etc., is essentially the same.

There are, however, notable differences, including Ethereum's use of a memory hard hash function, the uncle incentivization mechanism, the difficulty adjustment algorithm, gas limit adjustment algorithm, etc., but these are optimizations and arguably not central to the essence of Ethereum.

One other notable feature in Ethereum that is lacking in Bitcoin is the more extensive use of Merkle trees, allowing for the existence of "light clients" that download and verify only block headers and but can nevertheless determine and securely verify any specific part of the blockchain state if need be. Whereas in Bitcoin a block header contains only a root hash of a Merkle tree containing transactions, in Ethereum every block header contains a "state root", essentially a root hash of a cryptographic hash tree containing the entire current state, including account balances, sequence numbers, code and storage.



A light client will thus normally download block headers (~500 bytes per 17 seconds) only, and if a light client wants to learn some specific value in the state (eg. "what is the balance of account 0x124b6f72?" or "what is the storage of account 0x38c9f1e5 at key 178233?") it can ask any "full node" in the network to supply a "branch", a series of hashes from the block of data in the tree that specifies that particular information up to the root, and the light client can itself verify the integrity of the branch. If the branch is correct, it accepts the answer. The "light client" model is useful particularly for smartphone and IoT/embedded users of Ethereum, as well as for users with low-quality computers and slow or poor internet connections.

Public and Private Ethereum

It is important to note that while the original Ethereum blockchain is a public blockchain, the *Ethereum state transition rules* (i.e. the part of the protocol that deals with processing transactions, executing contract code, etc.) can be separated from the *Ethereum public blockchain consensus algorithm* (i.e. proof of work), and it is entirely possible to create private (run by one node controlled by one company) or consortium (run by a prespecified set of nodes) blockchains that run the Ethereum code[4]. Ethereum technology itself is thus arguably agnostic between whether it's applied in a public, consortium or private model, and it is our goal to maximally aim for interoperability between various instantiations of Ethereum - i.e. one should be able to take contracts and applications that are written for public chain Ethereum and port them to private chain Ethereum and vice versa.

While there are versions of Ethereum that are currently being developed for a private-chain context, eg. HydraChain, significant work still remains to be done before they can become viable and achieve the scalability improvements that private chain Ethereum implementations should theoretically be able to attain.

Applications in Finance

Although Ethereum is a highly generic platform and can theoretically be used for a very wide variety of use cases, a substantial majority of Ethereum applications are at least partially financial in nature. These applications further split into "purely financial", dealing solely with the management of valuable digital financial assets, creating financial contracts, escrows, etc., and the "semi-financial", which combine a financial or payments use case with the provision of services that are not themselves financial in nature.

Examples of the latter category are diverse, ranging from "decentralized Uber" platforms to institution-focused trade finance projects such as the CargoChain that won the Shanghai blockchain hackathon in January 2016; these projects take advantage of the Ethereum blockchain both for its digital asset management and smart contract functionality and for its utility in non-financial capacities, serving as a database for shipping tracking records, reputation and rating information, and proof-of-existence for signed legal contracts.

On the "purely financial" side, we see:

- Blockchain-based processing of financial contracts and derivatives (eg. Clearmatics)
- Other financial instruments placed on the blockchain (eg. UBS's "smart bond" platform)
- Digitization of real-world assets, ranging from gold (eg. Digix Global) to "settlement coins" denominated in fiat currencies both for purposes of financial transactions and simple mainstream payments
- Using blockchain-based contracts for difference (CFDs), enforced using smart contracts and collateralized with and settled in counterparty-risk-free cryptocurrencies, to create "mirror assets"

[4] For a more in-depth explanation of private and public blockchains, see https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

that simulate assets in the real world, offering equal returns to the underlying asset provided a sufficiently liquid arbitrage market

- Blockchain or smart contract-based collateral management

There are also *non-financial blockchain applications that are themselves useful in (blockchain-based or non-blockchain based) financial platforms*; perhaps the best example of this is identity verification.

Evaluating the Suitability of Private or Public Chain Ethereum for your Application

Evaluating the suitability of these applications for Ethereum involves two major steps. The first is determining whether a blockchain is indeed the right tool for the job. The basic technical advantages of blockchains, including reliability, security, auditability and decentralization, are now well-understood, but what is less understood is the specific areas where these advantages shine. Do mainstream payments systems need to be cryptographically auditable and decentralized using a consortium or public database? What about securities trading? What about markets for shipping or airline tickets, or file storage or computation? What about merchant loyalty points? On the non-financial side, what about tracking ownership of various real world assets, or digital assets such as domain names? What about email and social networks?

At this point, there then comes the choice of whether a public or private blockchain is preferable. The benefits of Ethereum are arguably different depending on whether public chains or consortium chains are used. On a public chain, aside from simply having the ability to easily implement programmatic escrow and financial contracts, a key benefit of Ethereum, as recently described by an Ethereum application developer, is ***synergy***. Contracts in Ethereum can serve different functions, and every application built on Ethereum can theoretically leverage any other application.

For example, if you want to use a blockchain to manage ownership of company shares, that may be useful by itself from the point of view of secure and verifiable record-keeping, but one secondary benefit is that it makes it much easier to implement equity crowdfunding: one can send one's company shares into a smart contract that automatically sends X shares to whoever sends Y units of cryptocurrency Z into the contract, and then immediately forward the coins into an account from which money can be withdrawn only with the permission of three of the five directors. If local regulations require some form of KYC or have restrictions on who can invest, then as soon as someone creates a blockchain-based KYC and accreditation platform, both that equity crowdfunding platform and any other equity crowdfunding platform (or financial application in general) can immediately interface with that system, building into the contract the restriction that only incoming funds from authorized individuals constitute valid purchases.[5]

In particular, note that synergy is arguably one of the key distinguishing factors between Ethereum and so-called "two-layer" attempts at blockchain-based smart contracts (eg. the now-discontinued Codius), that try to treat the blockchain as purely a layer for keeping track of asset ownership (or even an even "dumber" pure data layer) and ask each individual application to separately process smart contracts either through multi-signature "notaries" or through users individually processing the blockchain and "interpreting" the result; a design goal of synergy requires applications to agree on a common source of correctness for the result of smart contracts that everyone agrees is secure, and so it simply makes economic sense to introduce such a mechanism at protocol level and incentivize it as part of the consensus algorithm as a public good.[6]

In private chains, the arguments tend to be different. Private chains are generally being explored as tools for creating settlement platforms for specific industries, allowing inter-institutional transactions to be processed as efficiently as intra-institutional transactions by virtue of being on a single common database. The ideal “user” of a private chain is in fact an industry which is *already* moderately decentralized, with no single company having more than low-double-digit market share, where unfortunately with current technology the decentralization causes substantial efficiency losses: if a customer of one company wishes to perform some interaction (sending a payment, making a trade, etc.) with a customer of another company, a complex and cumbersome inter-corporate reconciliation and settlement process is required, often at substantial cost and delay. With blockchains, the existing semi-decentralized "political structure" of the industry can be preserved - there is no need to convince every participant to agree to merge or become clients of a super-company, and no need to convince the antitrust regulators to allow them to do so - while gaining the benefits of a large network effect and a high degree of interoperability through a simple switch in technology.

Another possible benefit is improving efficiencies in the financial industry through a "separation of concerns": banks do not need to innovate in every category of financial applications themselves, leaving that effort to more nimble financial firms that evolve into something more similar to software providers that help users send cryptographically signed transactions executing trades and entering into contracts involving asset-

[5] See miiCard and Tradle for examples of companies that are trying to create blockchain-based KYC platforms. Note that a current major problem with such schemes is that companies are often required to all conduct KYC checks themselves, and cannot simply piggyback off of external services; hence, achieving the full gains of blockchain-based KYC will require regulatory support.

[6] For an alternate view specific to a private blockchain context, see Gideon Greenspan’s blog posts and IBTimes article.

backed tokens on the blockchain without actually custodying any assets themselves (and thereby reducing their own regulatory burdens), while the banks keep their core business of taking deposits and making loans.

Synergy within private chains tends to be less important, as each private chain is likely designed for one particular application. However, there are routes that allow synergy in an inter-private-chain or combined public-and-private-chain context to still exist. There has already been work in the specific use case of cross-chain asset trades, including TierNolan’s cross-chain swap protocol and more recent work by Interledger; Ethereum’s strategy, however, is more ambitious, seeking to create more general “bridge” mechanisms that are capable of reading one blockchain from inside another blockchain (eg. btcrelay between Bitcoin and Ethereum), and a long-term goal would be to implement a general-purpose asynchronous programming language that allows applications to span across multiple chains. If this is desired, then much of the planned Ethereum 2.0 technology could be repurposed to achieve these goals; if it is not too important, then the benefits of Ethereum will be confined to the smart contracting and future-proofness aspect, and the tradeoffs between the “dumb blockchain” and “smart blockchain” approach will be confined to the question of exactly what kinds of inefficiency and complexity users find most important.

Once a company decides that a blockchain is the way to go, they then have the choice of which platform to use. Ethereum's benefits come in the form of programmability, flexibility, synergy, modularity, and a philosophy of humility, reflecting that we can never know exactly what requirements every developer is going to have for each application, both now and even more so five years in the future. If legal research determines that KYC verifications, accreditation restrictions or other rules are required for a specific application, then an identity system can be built as a separate layer, and contracts can be written that directly plug into it. If your particular privately traded company, whose shares you want to have recorded on a blockchain, wants to have a restriction that new shareholders must be approved by 51% of existing shareholders, then you can do that without any change to either the base layer or any other part of your system.

Even for simpler applications such as payments, initially constructing the applications on top of Ethereum allows a more rapid "on-ramp" to more advanced applications such as financial contracts, collateralization, trust-free atomic swaps, etc. on top of the basic asset layer, allowing that functionality to be very easily added at any future time when it is desired. Future versions of Ethereum will extend this "future-compatibility" to even the level of cryptography, allowing users to choose what cryptographic algorithms they use to protect their accounts. For example, if you personally are particularly paranoid about quantum computers and want to upgrade to Lamport signatures quickly, you can upgrade to Lamport, no need to wait for the entire blockchain protocol to move forward with you.

The natural counterargument is the usual "don't use a complex tool to do the job when a simpler tool will suffice", alongside discussions about specific efficiency disadvantages that Ethereum incurs either because of its generality itself or because of its particular way of implementing it. The former argument should be weighed by each project against Ethereum's merits, and the latter arguments will be discussed in great detail in the later sections on Ethereum's roadmap.

The Ethereum Development Roadmap Summary

The Ethereum project's currently expected future milestones, which will be referred to extensively further down in this section, are as follows:

- **Metropolis:** release of the Mist browser, expected summer/fall 2016
- **Serenity ("Ethereum 1.5"):** release of the proof of stake (Casper) version of the blockchain, also including Ethereum Improvement Proposals (EIPs) 101 and 105. Expected early 2017.
- **WebAssembly release ("Ethereum 1.75"):** faster virtual machine. Expected 2017.
- **Ethereum 2.0 (yet unnamed):** initial scalability release. Expected late 2017.
- **Ethereum 3.0 (yet unnamed):** "unlimited" scalability release. Expected late 2018.

These names will be used extensively in later sections.

Security

The notion of a massively replicated Turing-complete state machine, to which anyone in the world with the ability to buy 0.3 cents worth of ether can upload code that every participant in the network will be forced to run on their local machine, is certainly not without its significant security concerns. After all, many other platforms that offer similar functionality, including Flash and Javascript, have had frequent run-ins with heap and buffer overflow attacks, sandbox escapes and a whole host of other vulnerabilities that often allow the attacker to do anything up to and including taking control over your entire machine. Aside from that, there is always the threat of a denial of service attack, carried out by executing code that forces the virtual machine to go into an infinite loop.

The Ethereum project has employed a substantial array of strategies in order to mitigate these challenges. First and foremost, the Ethereum virtual machine is a highly simplified and limited construction, and the design choice of making a new VM from scratch instead of reusing an existing one such a Java was in part informed by the goal of being able to specifically target the VM toward high security. Opcodes for accessing system resources, directly reading memory or interacting with the file system are not found in the EVM specification; rather, the only "system" or "environment" opcodes that do exist interact with structures that are themselves defined in the Ethereum state: VM memory, stack, storage, code and blockchain environment info such as the timestamp.

Most EVM implementations are interpreted, though one JIT compiled VM has been developed. The EVM (along with the rest of the Ethereum protocol) now has six implementations, with over 50000 unit tests ensuring perfect compatibility and deterministic execution[7]. The Go, C++ and Python implementations have had extensive security audits with the help of our security auditing firm, Deja Vu. In the few months after Ethereum's launch, there were several security vulnerabilities found that needed to be patched, but the frequency of them has been greatly decreasing: in the two months prior to the time of this paper, the Ethereum network has been running without any significant security issues being found[8].

The infinite loop attack is the one for which the solution is most intricate. In general, any Turing-complete programming language is theoretically vulnerable to the Halting problem, which states that, in general, it is not possible to determine ahead of time whether or not the process of executing a given program with a given input will *ever* halt. An example is the Collatz conjecture. Consider the following program:

```
def collatz(n):
    while n > 1:
        if n % 2:
            n = n * 3 + 1
        else:
            n = n / 2
```

It is conjectured that if you run the program with any input, that the program will always eventually stop, but we cannot prove that that is the case; we could prove that it *isn't* the case if we found a counterexample, but so far no counterexample has been found for inputs up to over 1 trillion. Hence, the fear is that an attacker could create a contract which contains obfuscated infinite-looping code, and then send a transaction to the contract once to crash the system.

The solution in Ethereum is the concept of **gas**: the transaction sender specifies a maximum number of computational steps that they authorize the code execution to take, and pay a transaction fee in ether that is

[7] Note that here lies a major philosophical difference between Ethereum and many other protocols, where there is no standard independently defined protocol specification, and in fact a policy that *the protocol is the implementation* is often explicitly adopted. If there is a bug in the implementation then that bug will often simply become part of the protocol. Perhaps the best known example is OP_CHECKMULTISIG, which was accidentally implemented in a way that consumes the top value on the stack with no effect before doing anything else; this is now solidified as a protocol rule.

[8] A few security bugs have been found in the months immediately after launch, see these three blog posts alerting users to update; however, the frequency has been rapidly decreasing, and of course Bitcoin itself has not been without problems.

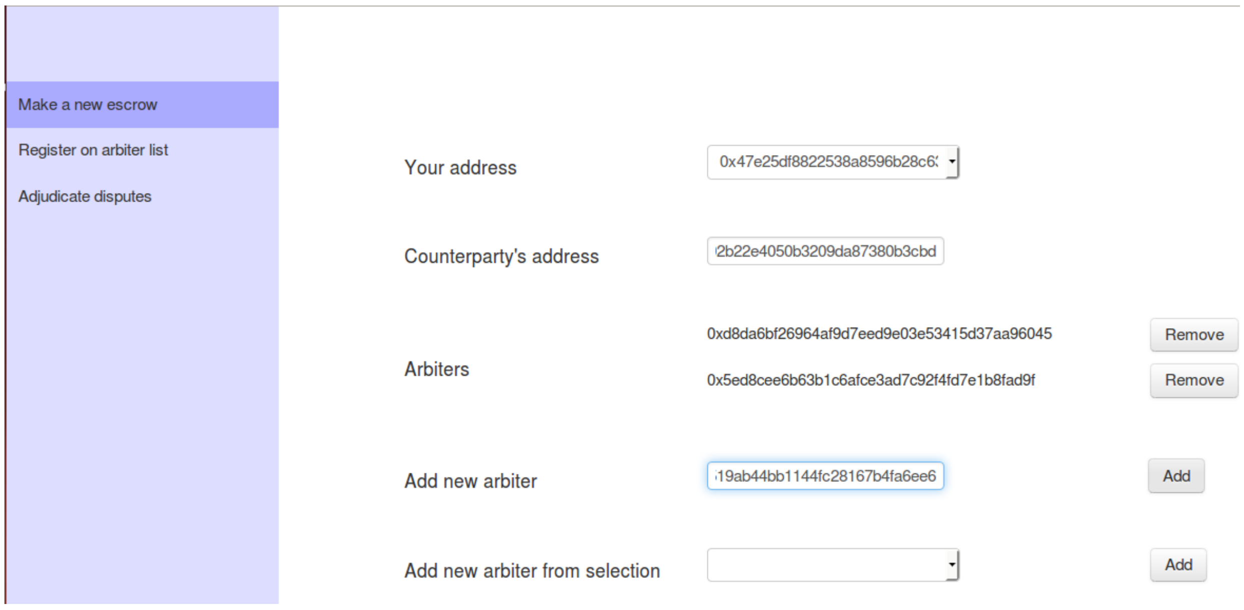
proportional to the number of computational steps spent. In practice, different operations have different gas costs, and these costs reflect concerns other than just the computation time of each operation (eg. full node storage and memory consumption are also taken into account), so gas is not *just* a counter of computational steps, but as an initial approximation the idea that "gas = computational steps" is sufficient.

If the process of transaction execution "runs out of gas", ie. exceeds its maximum allowed budget of computational steps, the transaction execution is reverted but the transaction is still valid (just ineffective) and the sender must still pay the fee; hence, the transaction sender must weigh the tradeoff between setting a higher gas limit, and thereby potentially accidentally paying a large transaction fee, and setting a lower limit, and thereby potentially accidentally creating a transaction whose execution is reverted, requiring it to be resent with a higher limit. Messages themselves also set gas limits, so it is possible for a contract to interact with another contract without trusting it with its full "budget" of gas. This mechanism has been reviewed by cryptocurrency researcher Andrew Miller and others, and the conclusion has universally been that it *does* succeed at accomplishing its objective of bypassing the halting problem and economically rationing computational power, though edge cases where incentivization is not optimal exist.[9]

A third security problem arises at the *high level*: if I see someone offering a smart contract who *says* that it's a binary option that costs $10 and will pay $20 only if the price of gold on March 30 is more than $1100, then how do I know that's *actually* the case? If it's a contract managing an entire marketplace of binary options, allowing users to offer a bid, claim a bid, and then closing, how do I know that the contract does not have a backdoor that simply lets the author take all of the money and run? Better still, how does the *author*, as an imperfect software developer who makes mistakes, know that they haven't accidentally put a flaw in the code that prevents *anyone* from taking the money out, ever?

The two problems described above - programmer malice and programmer error - are separate problems, and the solutions do somewhat diverge, but there are commonalities. As with other challenges in Ethereum, there are two classes of solutions, one lower-tech and one higher-tech, where the higher-tech solution theoretically offers much more promise, but at the cost of being harder to implement and reliant on much more complex tools. The lower-tech solution for malice is two-fold. First, in Ethereum, we make a distinction between the **core** of an application, which is purely the set of smart contracts, and the **interface**, the HTML and Javascript code that talks to the core by reading the blockchain and sending transactions.

[9] Outside of the Least Authority analysis, the "Demystifying Incentives in the Consensus Computer" paper from the National University of Singapore provides additional insights regarding the limitations of incentive-compatibility in Ethereum



Your address 0x47e25df8822538a8596b28c6:

Counterparty's address 2b22e4050b3209da87380b3cbd

Arbiters 0xd8da6bf26964af9d7eed9e03e53415d37aa96045
0x5ed8cee6b63b1c6afce3ad7c92f4fd7e1b8fad9f

Add new arbiter 19ab44bb1144fc28167b4fa6ee6

Add new arbiter from selection

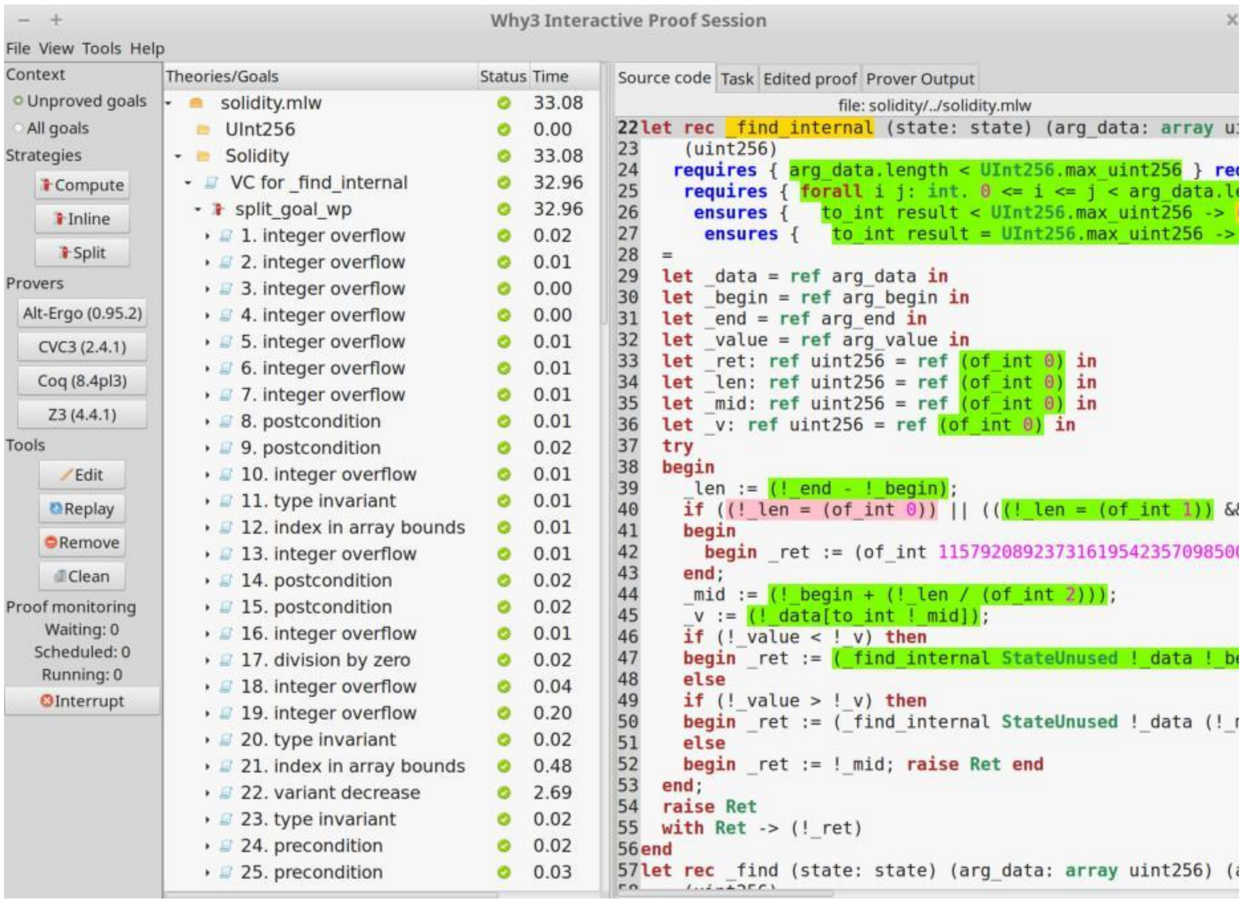
A simple and pretty user interface for an escrow dapp. But how do you know hitting "Remove" doesn't actually send the developer all of your money?

The goal is for the core to be trusted, and to do so the core should be as small as possible and heavily audited and reviewed, whereas the interface, which may contain very large amounts of code, can be less trusted; a medium-term design objective is to make the Ethereum user environment *protect the user from bad interfaces* as much as possible, via mandatory "are you sure you want to send a transaction with this data to this contract?" dialogs and other such devices. We hope for there to emerge a market of professional firms, similar to law firms creating standard-form legal contracts today, that create standard-form contracts for a variety of use cases, and for these contracts to receive a high level of scrutiny from multiple independent auditors each.

Auditing and standardization can also protect against coder error, though in the specific case of error there has also for several decades been active research into very strongly typed programming languages that specifically try to make errors easier to avoid; such languages allow you to much more richly specify the *meaning* of each type of data, and automatically prevent data from being combined together in obviously incorrect ways (eg. a timestamp plus a currency value, or an address divided by a block hash).

The more advanced set of solutions relies on a technology known as **formal verification**. Formal verification is, in simple terms, the science of using computer programs to automatically mathematically prove statements about other computer programs. A simple example is mathematically proving correctness of a sorting algorithm: given a piece of code that takes an array as input and provides an array as output, one can feed in a "theorem", like `for x ∈ I: x ∈ O; j > i => O[j] >= O[i]` (in English, "prove that for every value in the input, that value is also in the output, so the output must be some permutation of the input, and also prove that the output is in increasing order"). The formal prover will then process the code, and either construct a mathematical proof that the theorem is true, construct a mathematical proof that the theorem is false, or give up (as it might on the Collatz conjecture). Advanced formal provers such as Aesthetic Integration's Imandra even automatically find counterexamples for incorrect theorems.

Currently, Christian Reitwiessner, the lead developer of the Ethereum high-level programming language Solidity (which compiles down to EVM bytecode), is actively working on integrating the formal proving engine why3 into Solidity, allowing users to insert proofs of mathematical claims into Solidity programs and having them be verified at compile time; there is also a project for integrating Imandra into EVM code.



Theories/Goals	Status	Time
solidity.mlw	✓	33.08
UInt256	✓	0.00
Solidity	✓	33.08
VC for _find_internal	✓	32.96
split_goal_wp	✓	32.96
1. integer overflow	✓	0.02
2. integer overflow	✓	0.01
3. integer overflow	✓	0.00
4. integer overflow	✓	0.00
5. integer overflow	✓	0.01
6. integer overflow	✓	0.01
7. integer overflow	✓	0.01
8. postcondition	✓	0.01
9. postcondition	✓	0.02
10. integer overflow	✓	0.01
11. type invariant	✓	0.01
12. index in array bounds	✓	0.01
13. integer overflow	✓	0.01
14. postcondition	✓	0.02
15. postcondition	✓	0.02
16. integer overflow	✓	0.01
17. division by zero	✓	0.02
18. integer overflow	✓	0.04
19. integer overflow	✓	0.20
20. type invariant	✓	0.02
21. index in array bounds	✓	0.48
22. variant decrease	✓	2.69
23. type invariant	✓	0.02
24. precondition	✓	0.02
25. precondition	✓	0.03

file: solidity/../solidity.mlw

```
let rec _find_internal (state: state) (arg_data: array u
    (uint256)
   requires { arg_data.length < UInt256.max_uint256 } rec
    requires { forall i j: int. 0 <= i <= j < arg_data.le
     ensures {   to_int result < UInt256.max_uint256 ->
      ensures {   to_int result = UInt256.max_uint256 ->
  =
  let _data = ref arg_data in
  let _begin = ref arg_begin in
  let _end = ref arg_end in
  let _value = ref arg_value in
  let _ret: ref uint256 = ref (of_int 0) in
  let _len: ref uint256 = ref (of_int 0) in
  let _mid: ref uint256 = ref (of_int 0) in
  let _v: ref uint256 = ref (of_int 0) in
  try
  begin
    _len := (!_end - !_begin);
    if ((!_len = (of_int 0)) || (((!_len = (of_int 1)) &&
    begin
      begin _ret := (of_int 115792089237316195423570985008
    end;
    _mid := (!_begin + (!_len / (of_int 2)));
    _v := (!_data[to_int !_mid]);
    if (!_value < !_v) then
    begin _ret := (_find_internal StateUnused !_data !_be
    else
    if (!_value > !_v) then
    begin _ret := (_find_internal StateUnused !_data (!_r
    else
    begin _ret := !_mid; raise Ret end
  end;
  raise Ret
  with Ret -> (!_ret)
end
let rec _find (state: state) (arg_data: array uint256) (a
```

Formal verification of Solidity via why3

Formal verification is a very powerful technology; however, there is one problem that it cannot solve: how do we know what we want to prove in the first place? Human notions of fairness or correctness are often very complex, and the complexities are often hard to detect. Market order books are often subject to requirements including basic correctness (you either get what you want at the price you want, if not better, or you get your money back), order-invariance (so as to protect against front-running), ensuring that each buyer and seller is matched against the most favorable possible counter-offer, etc; and unfortunately, we can't easily list all of these conditions and be absolutely sure that we haven't missed anything. Unfortunately, for the time being it seems like choosing what to verify will continue to be an art, although formal verification techniques will certainly reduce the "attack space" that auditors need to check.

Key Recommendations

Financial institutions should consider using either public or private chain Ethereum or an EVM-based system as the virtual machine has been designed and tested with the goals of perfect determinism and security in mind - design goals ideal for the distributed ledger use case; particularly, no mainstream virtual machine currently has a concept of gas counting built it, whereas something like a gas counting mechanism is necessary in order to both (i) avoid infinite loop attacks, and (ii) avoid non-deterministic behavior in edge cases. In a private chain instantiation, paying for gas with ether (or another cryptographic token) is not strictly necessary; one can instead allocate users' "gas units" directly, much like CPU hours are allocated and tracked on AWS, or one can simply require all transactions to have a given pre-set maximum (eg. 1 million gas).

However, the security of the virtual machine is not the whole story: users interacting with a smart contract on Ethereum need to be sure that the code that they are interacting with actually does what they think it does. Blockchains are a convenient platform to cryptographically achieve guarantees, as the code stored on the blockchain can be viewed by all participants to a contract and users are sure that that is the code that will actually be executed, but removing any room for malicious or accidental bugs (see the Underhanded C contest for some surprisingly subtle examples) remains a challenge; users are recommended to explore formal verification technology as a route for automatically proving various assurances about the code that they are interacting with, particularly in financial use cases where the stakes can be very high.

Efficiency

One of the main concerns about Ethereum is the concern about efficiency, and more specifically the idea that by being so general the Ethereum platform acquires a quality of being "jack of all trades, master of none". One category of concerns, most vocally brought up by the Bitshares team but also by others, is the fact that code executed inside of a virtual machine is inevitably going to be slower than native code, and so a platform that supports a much larger array of "pre-compiled" operations is inevitably going to be more efficient. There is also the argument from Gideon Greenspan about concurrency, but parallelization and computational efficiency are distinct concepts and so this will be addressed in the scalability section instead of here. A third concern is that some of the operations done in the Ethereum blockchain, particularly Merkle hashing, are needlessly slow, and could be removed to increase efficiency (and are arguably less needed in a private blockchain context).

The EVM

The virtual machine issue is arguably by far the most complex. The original blockchain scripting language, Bitcoin Script, was a stack-based language modeled on FORTH, and features basic arithmetic with unlimited-sized integers, string operations, as well as a number of advanced cryptographic primitives. The execution engine remains quite inefficient, and so is certainly unsuitable for creating any kind of complex applications at the script level. However, because of the inherent statelessness of the pure UTXO model, creating advanced applications on top of Bitcoin has always been impossible, and so there was never even an opportunity for the efficiency of executing Bitcoin Script programs to even become an issue (until recently: in a recent blog post, the Bitcoin Core development team argued against increasing block size from 1MB to 2MB on the grounds that a maximum-sized transaction could take an entire ten minutes to verify).

The original Ethereum language was intended to be similar to Bitcoin Script, except with a stateful rather than stateless execution environment. The 256-bit integer restriction was added in order to prevent integer multiplication (still disabled in Bitcoin Script) from being an denial-of-service attack vector; the notion of dynamic gas costs based on integer size was considered, but ultimately rejected for efficiency reasons. 256 bits was deemed an optimal maximum size because it is still reasonably fast to calculate and execute, but just large enough to handle both elliptic curve cryptography (which uses 256-bit numbers) and hashing (most major hash algorithms offer a 256-bit output). Explicit opcodes for hashing and elliptic curve operations were included.

The intention at the time was to support simple financial scripts, if-then clauses and other applications where the inefficiencies of a 256-bit VM, while present, would be negligible compared to the cost of verifying the elliptic curve signature on the transaction - much like is the case with Bitcoin Script. And in most cases, this is indeed the case: verifying a signature requires processing over a thousand modular multiplication operations, whereas running the actual VM code requires simply processing a few conditional clauses and variable changes. Hence, the claim that there is somehow a giant leap in overhead between “just” verifying signatures and running “entire” programs is largely false: regardless of the VM’s performance, the time bottleneck is going to be cryptographic verification, not processing a few if-then clauses.

The Limits of Good Enough

However, there are a few places where very substantial overhead does exist. The first is in operations that carry out a large number of state changes. Currently, each state change needs to be represented as a modification to the Merkle tree, triggering 5 to 20 deserialize, hash and reserialize operations and database updates. As described earlier, Merkle trees are a very important part of the protocol from the point of view of light client friendliness, and become even more important for cross-chain receipts; however, the inefficiencies of Merkling as it currently stands are somewhat excessive - benchmark tests show that in some contracts, Merkle tree updates can take as much time, or even more time, than verifying the signature.

There are two routes to mitigating this problem. The first solution, preferred by the Bitshares team, is to remove the Merkle tree entirely, and simply store the state in leveldb directly. This may increase the efficiency of state updates by up to 5-20x, but at the cost of light client friendliness. For many private chain applications, this may well be the way to go. The second approach, and one that Ethereum will likely be following in its 1.1 Serenity release, is a mitigation strategy: instead of just allowing 32-byte key/value pairs in the Merkle tree, unlimited size values will be allowed, allowing application developers to store data structures that are frequently updated together in one place and thereby greatly reducing the number of actual tree updates that need to be made.[10]

A second point of weakness is the use of advanced mathematical and cryptographic algorithms inside of the EVM. So far, candidate applications for development inside the EVM have included ring signatures, zk-SNARK protocols, RSA-style public key cryptography, singular value decomposition and even memory-hard hash functions such as scrypt. And there, the performance of the EVM has indeed proven too low for practicality:

- An implementation of elliptic curve signature verification in native Python takes 0.017 seconds; in the Python EVM it takes 0.57 seconds
- An implementation of ring signature verification in native Python takes 0.119 seconds with 5 keys; in the Python EVM it takes 3.68 seconds
- An implementation of the memory-hard hash function scrypt in Python takes less than one second; in the Python EVM it takes so long that the process of executing the transaction needs to be split across 118 blocks

There are two primary reasons for this. First, existing established VMs have had over a decade worth of effort gone into creating optimized just-in-time implementations for them; Ethereum has only a fairly simple JIT.

[10] A third mitigation route is to parallelize at least the hashing portion of the work in Merkle tree updates (theoretically not too difficult to do particularly if tree split/merge operations are implemented), and a fourth is to look for processors with hardware-accelerated Keccak implementations (or switch to a hash function that does have hardware acceleration in the processors that you intend to use)

Getting up to par with established VMs will take years of effort. Second, the 256-bit integer requirement slows down the virtual machine greatly, as the underlying machine uses a 64-bit architecture, and much of the code for the algorithms only deals with integers that are much smaller than 64 bits and so do not need the overhead of an entire 256 bit integer arithmetic machine being applied for every addition and multiplication operation.

In the short term, our solution to this problem has been a notion of "precompiled contracts". Essentially, if there is very high demand for a given specific cryptographic feature (eg. SHA3, SHA256, elliptic curve signature verification), then we assign each function an address (we use the low addresses for this currently, ie. 1, 2, 3, etc), and add a protocol rule that a transaction (or internal transaction) to this address consumes a standard low amount of gas and outputs the function applied to the input. The function is not implemented in EVM code; rather, it is implemented in every Ethereum client in native code, allowing it to be executed cheaply and at native speed. This is essentially a "bandaid fix" that solves the problem for specific use cases, though it is clearly not optimal: it essentially means that the "permissionless innovation" aspect of being able to write anything as a smart contract without changing the protocol applies much more weakly to smart contracts that use weird and computationally intensive cryptography.

WebAssembly as "EVM 2.0"

For this reason, one of our researchers, Martin Becze, is currently in the process of exploring WebAssembly as a route for a faster virtual machine to more adequately solve the problem for the long term. The project is in two steps. The first step consists of writing a just-in-time compiler of EVM code to WebAssembly code, allowing for a fast EVM implementation. The second step consists of using WebAssembly to create an "EVM 2.0", where the only addition to WebAssembly would be a code translator that would (i) insert gas counting operations, and (ii) disallow opcodes deemed too ambiguous or non-deterministic or accessing information that the EVM should not have access to (eg. threads, floating point operations).

WebAssembly has been identified as an ideal candidate for several reasons, all having to do with similarities between WebAssembly's design goals and those of Ethereum:

- WebAssembly is intended to have very small code sizes for small applications. More traditional stacks (eg. bytecode generated with C++) with default tools tend to produce very large code sizes (eg. 4kb+) even for tiny applications (eg. see this developer's attempt to generate a small Linux executable), which is acceptable for today's extremely large hard drives but much less so for blockchain environments that may contain code uploaded by millions of users.
- WebAssembly is designed to run untrusted code supplied by potentially anonymous actors.
- WebAssembly already has multiple (JIT) implementations, with a goal of making them all compatible.

- WebAssembly's main weaknesses for our purposes, the lack of gas counting and possible non-total determinism, is addressed via the code translator. Ideally, users could write code using the same high-level language and development tools that would be used to write WebAssembly code, and the code would execute on the blockchain in exactly the same way that it would execute in a WebAssembly environment, except that the code translation layer would keep track of computational steps and ban non-deterministic operations. Preliminary tests suggest that gas counting may only add 5% overhead. WebAssembly itself appears from initial tests appears to be, depending on context, 25-90% as fast as native C++ - arguably sufficiently close to closing the gap between interpreted and native execution for our purposes.

The WebAssembly implementation is currently in its initial stages; a proof of concept for transforming WebAssembly code to add gas counting is already running, although a substantial amount of work needs to be done to flesh it out into a final product. Remaining work includes testing multiple implementations of WebAssembly, perhaps adding a Python implementation for completeness, and integrating them into the clients; if the WebAssembly plan continues, it is likely that WebAssembly will be added in a release between the Serenity release and the scalable public-chain Ethereum 2.0 that will be discussed in a later section, i.e. some time during 2017. Also, in general, it is quite likely that all improvements to the Ethereum protocol will be made available on private chain implementations first, as security concerns are lower than on the public chain where correcting errors is much harder.

Note that all of these solutions aim to fix only raw efficiency. The other major technical argument levied against unrestricted state-changing computation, parallelizability, is an important one; although at low levels of throughput it does not particularly matter if computation is done on one computer or spread across several, at high levels of throughput, as we will see, parallelizability, even more than efficiency, is key to surviving the challenges of scale.

Key Recommendations

The Ethereum Virtual Machine is currently substantially less efficient than many other mainstream VMs primarily due to design decisions made early on that emphasized convenience over high performance. For many simple applications, this is not an issue, as even with the EVM’s overhead the computational cost of processing a few additions, multiplications and if-then clauses remains negligible compared to the cost of verifying the cryptographic signature. However, it is currently not very practical to build very complex computations (such as custom cryptography) into the EVM directly.

In the medium term, a future release of Ethereum will likely use a combination of a code pre-processing step and WebAssembly as a way of creating a faster EVM that executes at close to the speed of native code.

Before this, the Serenity release will include features to explicitly allow running some additional cryptographic operations at native speed In the short term, we recommend that application developers looking to use computationally intensive algorithms (see the later section on privacy for many practical examples) who are not willing to wait that long start off by using an Ethereum private chain extend Ethereum's "precompiled contract" mechanism, adding additional precompiled contracts that execute these advanced operations directly in native code. Users that need a higher degree of scalability that are using a private chain should also consider (i) immediately implementing some of the modifications that are planned for the Serenity release, and (ii) removing the Merkle tree entirely if light-client friendliness is not needed.

Scalability

One of the fundamental challenges of every blockchain technology is that of scalability. Traditionally, decentralized protocols have been prized in part because of their *greater* scalability, allowing individuals and businesses to very cheaply and efficiently distribute files and allowing applications to manage communications between millions of users, and so it might seem to a non-technical user at first glance that blockchains, the tool of "peer-to-peer money" (and now "peer-to-peer smart contracts"), should have the same benefits.

Alas, that is not the case. Blockchains are unique in that the specific *kind* of decentralization that they provide is one of a different character - one that is not *distributed*, in the sense of "split up between different parties", but *replicated*: every single node on the network processes every transaction and maintains the entire state. The result of this is that while blockchains gain some of the benefits of decentralization, particularly fault tolerance and political neutrality, and the ability of each individual user to personally verify every transaction on the chain provides some extremely strong authenticity properties, their scalability is *much worse* than traditional systems - in fact, no matter how many nodes a blockchain has, its transaction processing capacity can never exceed that of a single node. Furthermore, unlike BitTorrent-like networks which get *stronger* the more nodes are added to the network, the blockchain can never be more powerful than the processing power of a single node, and even gets *weaker* the more nodes are added due to logarithmically increasing inter-node latency.

The troubles for *public* blockchains do not end there. Not only is it the case that every node on a public blockchain (and given that such systems seek to be democratic and allow everyone to participate, that essentially means a regular consumer laptop) must process the capacity of the entire community, but we also can't even expect each node to be spending anywhere close to 100% of its CPU capacity to do so. This is for a few reasons. The first is that the CPU is running other applications at the same time and is not always online. The second is that on top of the physical constraints, public blockchains have to deal with *economic* constraints: if the cost of maintaining a node is a substantial part of a node's total costs, then there is too large of an incentive for the network to centralize, as two nodes can make a significant gain by having one trust the other and thereby only run one node[11]. Alternatively, nodes could simply stop validating entirely and try to free-ride off the validation of others - an obvious tragedy of the commons and a large systemic risk if too many miners attempt to do this at the same time. Centralization and non-validating equilibria are failure modes that must be prevented, and with great safety margins, as if a system falls into an equilibrium that is undesirable it may be hard to change the rules or force it to go back.

For this reason, even though a C++ ethereum node's maximum theoretical transaction processing capacity is over 1000 tx/sec, the Ethereum network's block gas limit imposes a soft cap of 10 tx/sec (assuming simple transactions; for more complex transactions it's closer to 1-5 tx/sec, and given current statistics on livenet it's ~6.8 tx/sec). Bitcoin, despite a theoretical CPU-based limit of 4000 tx/sec, now has a de-facto hard cap of ~7 tx/sec assuming small transactions and ~3 tx/sec live[12]. Private blockchains do not have these concerns, as they can require every node to have a high-quality computer and internet connection and make sure offline that each participant is running a real node; hence, private blockchains based on Bitcoin or Ethereum can, without too much difficulty, achieve the full capacity of slightly over 1000 tx/sec. As a result, in the short term, private blockchains seem, to many enterprise users, to be a far superior and more practical option to achieve sufficient scalability for non-trivial use cases, although at large scale even they are not sufficient: the DTCC performs 100 million operations per day (~1200 per second), the Shanghai Stock Exchange's trading system has a peak order capacity of over 80,000 transactions per second, and one can expect the micro-transaction applications that many blockchain advocates see as holding the greatest promise to require even more.

Solutions

We are exploring two categories of solutions for the scalability challenge: **sharding** and **state channels**. Sharding refers to techniques similar to database sharding, where different parts of the state are stored by different nodes and transactions are directed to different nodes depending on which shards they affect so that they can be processed in parallel, and state channels are a generalization of Bitcoin-style "lightning networks", which seek to conduct most transactions off the blockchain between parties directly, using the

[11] There are also other forms of quasi-centralization; for example, most miners now pass blocks to each other using the Bitcoin Relay Network instead of the P2P network used by regular nodes. Increasing block sizes also increases incentives for these kinds of behaviors to emerge.

[12] There do exist various proposals to increase the de-facto Bitcoin block size, including the Bitcoin Classic 2MB hard fork, and the Segregated Witness proposal, which essentially (alongside some transaction malleability fixes) includes an accounting change such that bytes from transaction signatures are now only partially counted toward the block size, thereby increasing the de-facto block size to 1.5-3 MB depending on the types of transactions that are sent. However, these only increase the maximum throughput by a factor of two, several orders of magnitude below what is required to achieve the throughput necessary for mainstream applications.

blockchain only as a kind of final arbiter in case of disputes. (Real question: does Dominic William's proposal with dfinity/pebble make use of this type of sharding?)

In the case of the former approach, there are different considerations in the case of private blockchain and public blockchain sharding. In the case of private blockchain sharding, the goal is actually quite modest: make Ethereum maximally parallelizable. The property is retained that every node processes every transaction, the only difference being that when computation is parallelized the scalability of the network can be arbitrarily expanded by adding more CPU cores and more memory to each node.

In the public blockchain case, the task is more ambitious: create a blockchain protocol where the network can survive with exactly zero full nodes - that is, create a network where every node only processes a small portion of all transactions, using "light client" techniques to access the rest of the blockchain, while still maintaining security. Transactions would be processed not just on *different CPU cores*, but in fact on *different computers*, all of which are assumed, as is standard in public cryptoeconomic analysis[13], not to trust each other. The long term goal for Ethereum 2.0 and 3.0 is for the protocol to quite literally be able to maintain a blockchain capable of processing VISA-scale transaction levels, or even several orders of magnitude higher, using a network consisting of nothing but a sufficiently large set of users running nodes on consumer laptops.[14]

Our general approach to this problem is to start off by implementing a parallelizability scheme through Ethereum Improvement Proposal (EIP) 105 (scheduled for Serenity), which will provide the desired scalability properties to private chain implementations plus modest improvements to the public chain, and in a later Ethereum 2.0 release build in the cryptoeconomic incentivization schemes to securely achieve scalability through sharding for the public chain as well. EIP 105 has already been implemented in a Python proof of concept alongside the other changes planned for Serenity; a possible release date is late 2016 for a testnet and early 2017 for a live network, although delays are always possible and we have a philosophy of releasing when it's ready, not to fit a schedule. For the first half of this roadmap, the primary challenge is development effort, though later stages involve research challenges around economic incentives and peer-to-peer network structure as well.

[13] See Vlad Zamfir's DEVCon 1 presentation and other writings for a more detailed description of what public-blockchain "cryptoeconomics" entails.

[14] The research on how this can actually be accomplished has been carried out by several independent groups that have converged on roughly similar solutions; see Chain Fibers Redux and Dominic Williams' Dfinity project; the work by Maidsafe with its concept of "close groups" is similar. No system has been implemented in practice yet, and so of course many technical challenges in implementation remain.

Parallelization in Private Blockchains

In general, if computation can be fully parallelized, then private blockchain scalability is unlimited: you can always just add more CPU cores to each node. If that is not the case, however, then the maximum theoretical speedup is governed by Andahl's law: the absolute maximum by which you can speed up the system is limited to the reciprocal of the portion which cannot be parallelized. If you can parallelize 99%, you can speed up 100x, but if you can only parallelize 95% then you can only speed up 20x. We can make the characterization even more precise: assuming a fixed CPU clock speed, the minimum theoretical amount of time needed to process a set of transactions is proportional to the *length of the longest chain in the dependency graph* of transaction execution; that is, the longest chain of transactions which could theoretically interfere with each other, and so must be processed in sequence.

This is perhaps the greatest strength of the multisig-oracle model of smart contract execution, where each contract gets its own set of “notaries” that vote on the result of executing that contract independently of every other contract: *a lack of synergy also means a lack of cross-dependency*, and so figuring out how to parallelize the underlying asset management layer or data layer becomes considerably simpler[15]. With the recursive and synergistic model used in Ethereum, the challenge is this: *there are no restrictions whatsoever on what data from the state a contract execution process could theoretically depend on*, and so running computation entirely in series is the best that we can do.[16]

Hence, it becomes obvious that if we want to achieve scalability through sharding, we will need to make compromises to the Ethereum execution model that force parallelizability, and here insights from traditional parallel computing theory become useful. Rather than modeling the entire state as a single machine, we model the state as a distributed memory machine, where transactions are limited to processing one part of the state, and so they can be processed in parallel. To maintain cross-shard communication capabilities we also add a message-passing mechanism; the primary challenge is simply to make sure that it is deterministic. The approach that we are taking, one that can be effectively translated into both a private and public blockchain context, is a “receipt” paradigm: transaction execution can change the state of the local shard in which it is execution, but it can also generate “receipts”, which are stored in a kind of shared memory that can later be viewed (but not modified!) by transaction execution processes in other shards.

[15] In general, “the multisig-oracle model of smart contract execution” consists of a design where N parties put their assets into a contract where a majority of a given set of M “notaries” have full control over the assets; at that point, the M notaries execute the code, maintain any internal state and move the funds as required. See Codius and CIYAM Automated Transactions for examples of attempts to implement this in practice.

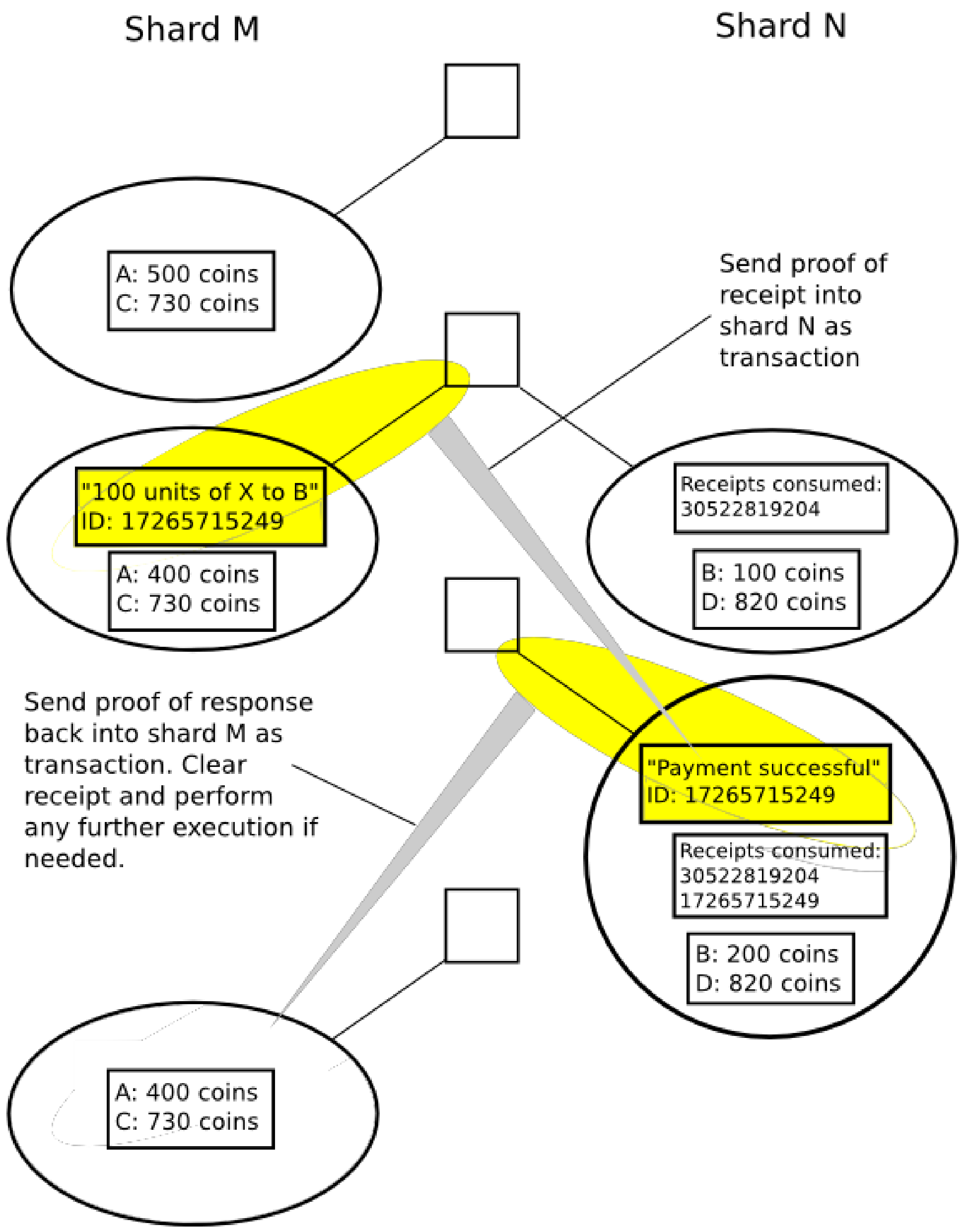
[16] Note that the challenge of coming up with worst-case examples for contract executions that are unparallelizable is exactly equivalent to the problem of creating sequential memory-hard hash functions; perhaps somewhat ironically, the cache generation process in Ethereum’s mining algorithm Ethash is one example

To see how this could be used in practice, consider at a high level a digital token example. User A owns 500 units of token X; this information is stored in shard M. User B owns 100 units of token X; this information is stored in shard N. Suppose that user A wants to send 100 units of token X from shard M to shard N. In a synchronous model where all shards are stored by all nodes, this would be simple; as the Ethereum code examples frequently found on slide presentations and on the back of our developer T-shirts frequently point out, the code would look something like this:

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

In an asynchronous model, however, we will need to decompose the process into stages:

1. On shard M, reduce A's balance by 100, and generate a receipt stating "B should receive 100 units of X. This receipt has ID 17265715249."
2. A proof that the receipt has been created (a simple pointer to a shared memory module in a private blockchain context, and a Merkle proof in a public blockchain) is introduced as a transaction on shard N. The contract verifies that (i) the proof is correct, and (ii) a receipt with that ID has not yet been consumed. The contract increases B's balance by 100, and marks in storage that the receipt with that ID has been consumed.
3. If, in a more advanced application, shard M is running a smart contract that needs to do something else after the payment completes, then the record in shard N can itself be used as a receipt to prove to shard M that the payment was made, at which point the execution on shard M can continue if needed.



The problem of ever-growing spent receipt records can be solved by giving receipts a timeout (say, of a few weeks) and flushing the spent receipt list of receipts that are too old.

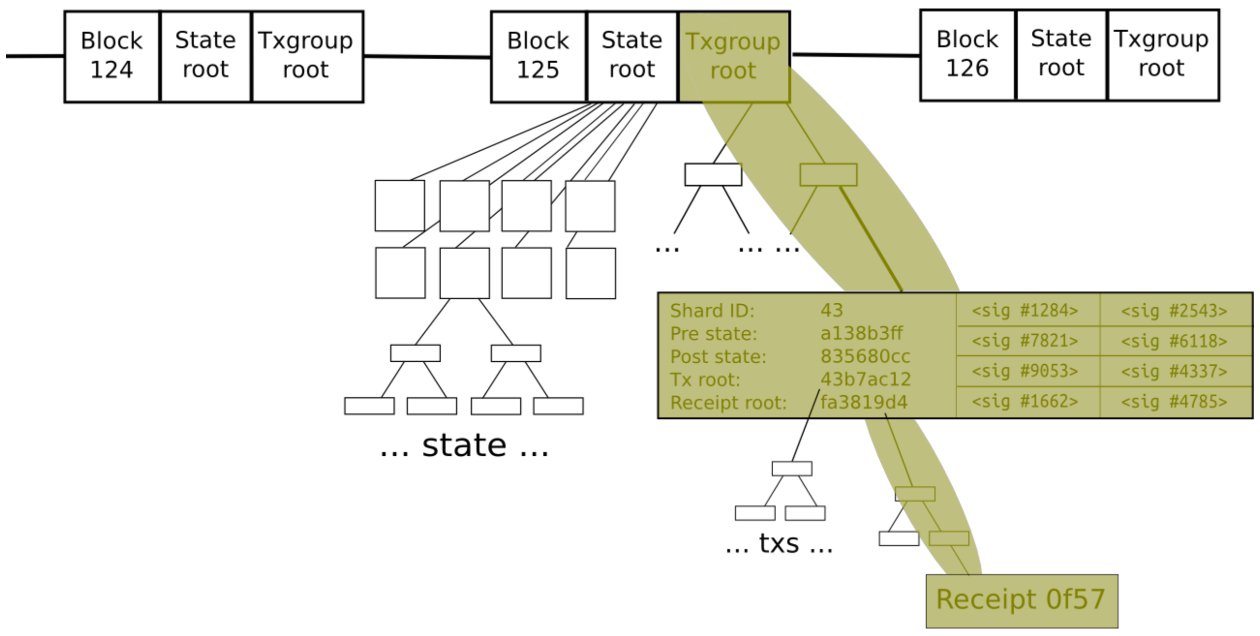
The receipt model is convenient because it neatly corresponds to two programming concepts from separate fields that are already well-understood by developers: **asynchronous programming** and **UTXOs** (the "unspent transaction outputs" concept in Bitcoin). In Bitcoin, the way that a transaction actually works is that it *consumes* a collection of UTXOs created by previous transactions, and then *produces* one or more new UTXOs, which can then be consumed by future transactions. Each UTXO can be thought of as being like a "coin": it has a denomination and an owner, and the primary two rules which a transaction must satisfy to be valid are that (i) the transaction must contain a valid signature for the owner of each UTXO that it consumes, and (ii) the total denomination of the UTXOs consumed must be equal or greater than the total denomination of the UTXOs that it produces. Although the UTXO model is unwieldy in some ways, requiring complex wallet code to determine the order in which a user consumes their UTXOs to optimize transaction fees, and exposing wallets that are not implemented carefully to denial-of-service vulnerabilities, it does have its benefits in somewhat improving user privacy (as not all user funds are immediately linked to each other) and parallelizability, as well as being more extensible to some of the cryptographic techniques that will be described in the "privacy" section. Similarly to UTXOs, receipts are also objects that are produced by transaction execution and can be "consumed" by execution of a further transaction; hence, they can be thought of as the natural Turing-complete smart contract programmatic generalization of UTXOs.

The connection to asynchronous programming is clear. If function A in shard M asynchronously calls function B in shard N, then this can be implemented by code executing function A generating a receipt containing the arguments for function B, and a subsequent transaction executing function B can then consume this receipt. If a callback is desired, then when calling function B, function A would also leave an object in the state specifying that it is awaiting a reply from function B, which could later be consumed by a receipt produced by function B specifying a response.

Note that in a private blockchain context, receipts could be processed across all shards immediately after transactions are processed, and the process could continue until all chains of receipt generation and execution have been completed. Due to the receipts' asynchrony, the process would still be highly parallelized: transactions in every shard would be processed in parallel, then receipts going into every shard would be processed in parallel, and so forth; hence, as long as there are not too many transactions taking place within a single shard, and as long as receipt-callback chains are prevented from being too long, parallelization problems would be completely resolved.

From Private to Public

In a public blockchain context, we are faced with two additional problems. First, the nodes processing transaction execution within different shards would be geographically separated from each other, running on different computers; this has the consequence that processing receipts is not instant, and a complex asynchronous contract execution could potentially take over a minute to fully execute. Second, these computers do not trust each other. Hence, it is not enough for a node processing transactions on shard M to simply *say* to the nodes processing transactions on shard N that a receipt was created; rather, it would need to *prove* it to them. Fortunately, Merkle proofs are an ideal solution.



In a public chain Ethereum 2.0, the intention is to devise a scheme where the bulk of the state is, in fact, distributed between the different nodes in the network, with each node only holding a small number of "shards" of the state. All nodes would keep track of a "header chain", containing Merkle tree root hashes of the state. Hence, a receipt scheme can easily be implemented: once a receipt is created in shard M, a Merkle branch (ie. a chain of hashes proving that the receipt exists) could be passed as a transaction into shard N, and then, if a callback is desired, another receipt could be created and another Merkle branch could be passed back.

The next problem is determining who validates each shard. Of course, the whole point of sharding is to move away from the "everyone processes everything" paradigm, and split up the validation responsibility among many nodes. With naive proof of work, doing that securely is difficult: proof of work as implemented in Bitcoin is a completely anonymous (not even pseudonymous)[17] consensus algorithm, and so if any one shard is secured by only a small portion of the total hashpower, an attacker can direct all of their hashpower

[17] In general, cryptographers view the concept of "identity" as encompassing any solution to problems of the form "prove that action A and action B were made by the same entity". "Anonymous" comes from the Greek "no name"; essentially, no actions can be correlated with each other. "Pseudonymous" ("false name") means that there is an identifier, but it is not connected to the identifier that that entity uses in other contexts (eg. a legal name). Mining by default is anonymous since blocks do not come with any information; the Puzzle Towers scheme requires miners to specify an address, and in order to earn substantial profits miners need to mine multiple blocks with the same address, hence the address is an identifier, albeit a "pseudonymous" one.

toward attacking that shard, thereby disrupting the entire blockchain potentially with less than 1% of the hashpower of the entire network.

This changes, however, with proof of stake, advanced proof of work schemes such as puzzle towers, and the Byzantine fault-tolerant consensus algorithms used in private blockchains[18]: the participants in the consensus process do have some kind of identity, even if it's just the pseudonymous cryptographic identity of an address, and so we can solve the “targeted attack” problem with random sampling schemes, randomly selecting some set of nodes to process any given set of transactions from the entire pool of validators, making it impossible for attackers to specifically target any particular transaction or any particular shard. Furthermore, malfeasance is easier to disincentivize and police; as Vlad Zamfir has said, “[proof of stake schemes] would be like if the ASIC farm burned down when it mined an invalid block”[19].

Hence, the general approach in a public blockchain context is to first collate transactions into “transaction groups” whose effect is disjoint from each other, and then for each transaction group to select a random set of validators to certify its validity. The header chain, rather than verifying the entire transaction group itself, simply verifies that the certificate of validity provided alongside a transaction group contains signatures from a sufficient majority of the selected set of validators.

Applying Sharding to Blockchain Applications

Now, there remain two questions. First, how should the state be sharded? In a public blockchain context, this decomposes further into *inter-application sharding*, where the goal is to shard the state in a way that allows multiple mostly-independent applications to be processed in parallel, and *intra-application sharding*, where individual applications are split up across multiple shards. The first problem is arguably an easier problem, as interactions between applications are often infrequent and the simple solution is to shard by address space. In a private blockchain context, the blockchain will often be used only for one application, and so intra-application sharding is the dominant concern in any case. Second, how do we design high-level programming

[18] This generally includes algorithms such as PBFT; a more detailed overview can be found on the wikipedia page on Byzantine fault tolerance.

[19] Proof of stake is a broad area of research; it’s also worth noting that there are generally two categories of proof of stake: “first generation” proof of stake algorithms, which try to maximally mimic proof of work, and “second generation” proof of stake algorithms which try to add more rigorous cryptoeconomic incentivization (and disincentivization) measures to either first-generation-style algorithms, traditional Byzantine fault tolerant consensus algorithms, or combinations or variations of the two categories. First-generation algorithms have stake grinding and nothing-at-stake vulnerabilities; for this reason they are generally viewed with suspicion by the academic community, even though newer iterations have removed stake grinding concerns an argument can be made that stake grinding has not been an issue to first-generation PoS blockchains in practice. Second-generation algorithms, including Tendermint and Ethereum’s Casper, are robust against these concerns, but have not yet been tested “in the wild”.

languages to take advantage of the blockchain's intra-application sharding capabilities and make the benefits maximally accessible to developers?

From a sharding perspective, the challenge is this: how do we create a sharding scheme that provides inter-application sharding and intra-application sharding at the same time, without introducing too much complexity? Currently, our leading proposal for this is Serenity EIP 105. The basic principles of EIP 105 are as follows. First, the address space is broken up into up to 65536 shards, and synchronous transaction execution can only happen within each shard; everything else would need to be asynchronous. Second, a contract with a given piece of code can exist on many shards at the same address, making it easy for individual applications to exist across many shards at the same time. Contracts may want to use any of the same partitioning techniques that are used in traditional parallel computing contexts (eg. SQL queries) to determine how to store their data: partition by some specific property of some specific key, randomly assign new objects to new partitions, some more clever load-balancing algorithm, etc. In Solidity, a possible route is to define a `shardedMap` data structure, defining at compile time a scheme for mapping keys to a shard ID. We expect high-level language developers to explore a combination of SQL-style sharding schemes, asynchronous programming concepts such as callbacks and promises, and other techniques to make it as easy for developers to navigate a scalable blockchain environment as possible.

As far as specific blockchain applications go, perhaps the easiest to shard is proof-of-existence (using blockchains to show that particular pieces of data existed at a particular point in time). This use-case does not have double-spending concerns, and so has been implemented by Factom and other projects in scalable form already. Proof-of-inexistence - the ability to prove that a proof-of-existence message has *not* been published to a blockchain (eg. certificate revocation) is slightly harder as one cannot so easily use Factom's "only publish the Merkle tree root" trick to solve it, but it is nevertheless massively parallelizable[20].

Digital asset use cases are harder, but doable with asynchronous programming, as was illustrated above; the only deficiency is a slight lag between when a payment is sent and when it becomes usable by the recipient. Two-party (or N-party) smart contracts are not particularly harder than digital assets, and face similar limitations. Some applications, such as identity in finance, may involve interoperability between digital asset transfer systems and identity systems; here too asynchrony is not a particularly large problem. Atomic trade implementations will need to deal with the special case where two parties attempt to participate in a trade at the same time; locking mechanisms and contracts generating refund transactions are perhaps the most natural strategies for dealing with this problem. Blockchain-based markets, processing order books in real-

[20] okTurtles' DNSChain uses blockchains in part to achieve easy certificate revocation; see okTurtles' FAQ for a more detailed description. Also, a more specific certificate revocation system using the bitcoin blockchain has been built by Christopher Allen. However, neither system has yet seen substantial usage.

time, seem to be the hardest; the Bitshares development team has concluded that it can’t be done without introducing multiple markets and the associated arbitrage spreads, although non-instant markets such as frequent batch auctions may be more feasible using parallel sorting algorithms. (if this is the case, what implication does it have for projects like T0 / Medici from Overstock?)

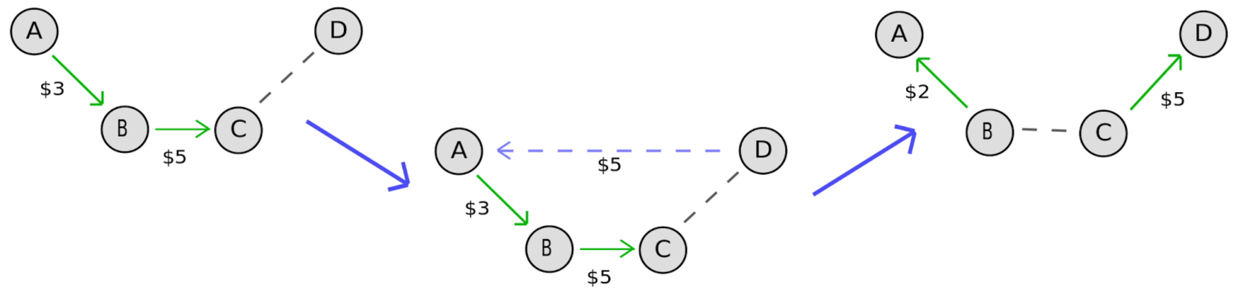
State Channels

State channels are a strategy that aims to solve the scalability challenge by keeping the underlying blockchain protocol the same, instead changing how the protocol is used: rather than using the blockchain as the primary processing layer for every kind of transaction, the blockchain is instead used purely as a settlement layer, processing only the final transaction of a series of interactions, and executing complex computations only in the event of a dispute.

To understand the concept in more detail, let us first consider the base case of a state channel used for payments. Suppose that party A has 100 coins, and wants to make frequent micropayments to party B. Party A starts off by sending 100 coins into a smart contract. When party A wants to make his first payment (say, of 0.1 coins), he creates a digitally signed "ticket" (NOT a valid blockchain transaction) stating (99.9, 0.1, 0), where 99.9 represents how much party A should get out of the contract, 0.1 represents how much party B should get and 0 is a sequence number. Party B counter-signs. If party A wants to make a second payment of 0.2 coins, he signs a new ticket (99.7, 0.3, 1), and party B counter-signs. Now, if party B wants to send 0.05 coins back, she signs a ticket (99.75, 0.25, 2), and party A countersigns.

At any point during the process, either party can send a transaction into the contract to close the channel and start a settlement procedure. This starts a time limit within which party A or B can submit tickets, and the ticket with the highest sequence number is processed. If party A maliciously submits a ticket with an earlier sequence number, trying to pretend that later payments did not happen, party B can submit the ticket with the latest sequence number themselves. The blockchain is viewed as a kind of “cryptographic court”: it serves as an expensive arbitration procedure that is used as a last resort to provide security, and thereby disincentivizes cheating by preventing cheaters from getting away with fraud, but if the process works well the blockchain should only be used very rarely for final settlement.

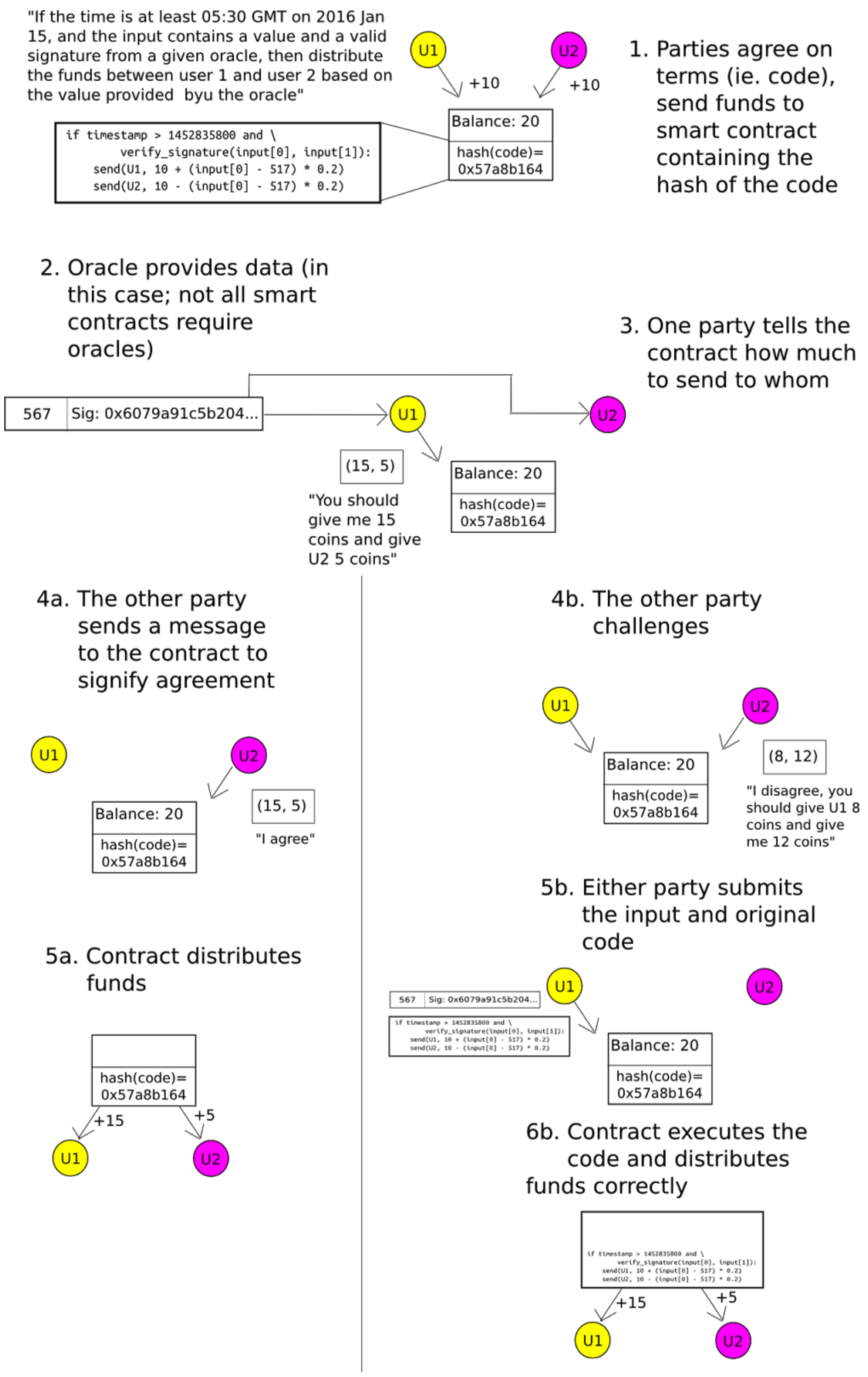
This simple payment channel mechanism lends itself to two generalizations. First, if you want to support a very large array of payments from anyone to anyone, and not just A to B, without creating N^2 channels from everyone to everyone, then you can "compose" channels: if there is a channel from A to B, a channel from B to C, and a channel from C to D, then you can convert a transfer from A to D into three channel updates, one along each channel.[21]



Second, one can add in smart contracts. If A and B want to enter into a financial contract which makes some transfer based on a formula, then they can both sign a ticket `(H(C), k)` where `H(C)` is the hash of a piece of code and k is the most recent sequence number; this essentially tells the channel settlement contract "evaluate the code that hashes to C and let the result tell you how much to send to A and B". C may, for example, be a financial contract such as a contract for difference; if the state channel is in control of multiple assets it could be one of a large class of different types of trades, leveraged trading collateral management agreements, options, etc.

When it comes time to withdraw the assets from the channel, and A and B can run C and thus both know determine much they're supposed to get, the blockchain does not need to evaluate the contract; if C would give A 75 coins and B 25 coins, they would both simply sign a new ticket `(75, 25, k+1)` for convenience (if either party refuses, then the other can simply submit `(H(C), k)` and C themselves, and thereby have the blockchain run the computation and lead to the same result).

[21] See the Lightning Network for an implementation on Bitcoin, and Raiden for an implementation on Ethereum.


"If the time is at least 05:30 GMT on 2016 Jan 15, and the input contains a value and a valid signature from a given oracle, then distribute the funds between user 1 and user 2 based on the value provided byu the oracle"

```
if timestamp > 1452835800 and \
        verify_signature(input[0], input[1]):
    send(U1, 10 + (input[0] - 517) * 0.2)
    send(U2, 10 - (input[0] - 517) * 0.2)
```

1. Parties agree on terms (ie. code), send funds to smart contract containing the hash of the code

2. Oracle provides data (in this case; not all smart contracts require oracles)

3. One party tells the contract how much to send to whom

"You should give me 15 coins and give U2 5 coins"

4a. The other party sends a message to the contract to signify agreement

"I agree"

5a. Contract distributes funds

4b. The other party challenges

"I disagree, you should give U1 8 coins and give me 12 coins"

5b. Either party submits the input and original code

6b. Contract executes the code and distributes funds correctly

State channels are not a perfect solution; particularly, it is less clear how they extend to massively-multi-user applications, and they offer no scalability improvements over the original blockchain in terms of its ability to store a large state size - they only increase de-facto transaction throughput. However, they have a number of benefits, perhaps the most important of which is that on top of being a scalability solution they are also a *privacy solution*, as the blockchain does not see *any* of the intermediate payments or contracts except for the final settlement and any disputes, and a *latency solution*, as state channel updates between two parties are instant - much faster than any direct on-blockchain solution, private or public, possibly could be, and potentially even faster than centralized approaches as channel updates from A to B can be secure without going through a centralized server.[22]

State channels theoretically do not require any updates to the Ethereum protocol in order to implement, and groups such as Raiden are implementing state channel networks on Ethereum already; Jeff Coleman and others are in the process of researching the best ways to implement state channels on Ethereum right now. Hence, they are arguably the safest short-term solution for scalability (and, as will be seen in the next section, privacy).

Key Recommendations

In the short term, institutions concerned about scalability have two primary routes: (i) explore the option of building as much of the application logic as possible inside of state channels, whether on a public or private chain, and (ii) implement EIP 105 in a private-chain Ethereum version, and use asynchronous programming with the sharding mechanism to achieve parallelizability. Before going this route, however, users should first determine if their application can run on the current single-threaded Ethereum implementation; applications that require less than ~2000 tx/sec should not require the parallelization features and could just be implemented now on a private-chain version of Ethereum as is. Without state channels, public chains are currently not sufficiently scalable for a large portion of financial applications, as the Ethereum public chain can only safely support ~10-20 tx/sec.

[22] Another idea that can be philosophically viewed as being in some sense similar to state channels is the notion of "fidelity-bonded banking", first conceived by Peter Todd in 2013. Fidelity-bonded banking, as a general category, is a combination of two ideas. First, we have known for a long time that it is possible to design financial services in such a way that every operation carried out by the operator is cryptographically provable, allowing users to audit the service and make sure that it is acting honestly; doing so was a key design goal of the OpenTransactions project, as well as Greg Maxwell's proof-of-solvency scheme. Second, once these proofs exist, we can create an Ethereum smart contract such that if a valid "proof of fraud" is submitted, the contract takes money out of the bank's deposit and sends the user the amount that they are entitled to. Fidelity-bonded banking hence relies on similar notions to state channels, and it may have applications in democratizing the provision of financial services by allowing service providers to build services that are mathematically provably trust-free, thereby removing the need for the providers to be trustworthy themselves.

In the medium term, the mainline Ethereum implementations themselves will add EIP 105 and other scalability features to its software; these features are likely to be available for use in private chains first as less testing is required in that area, and in the long term the public Ethereum chain will also be built out in a way that allows much higher degrees of scalability.

Privacy

The next major challenge in Ethereum applications, and blockchain applications is that of privacy. The problem is simple: even though having transactions be processed on a system maintained and audited by many parties is great for authenticity, it poses high costs in terms of privacy. There is in many cases a serious gap in blockchain users understanding this particular point: many companies first start looking at blockchain technology because they hear that blockchains are an excellent information security technology, but it often takes some time for them to understand that the kind of information security that blockchains provide has to do with *authenticity* not *privacy*, whereas privacy may in fact be their primary information security concern.

First of all, it is important to note that **private blockchains are NOT a solution to privacy** (unless, to some extent, the blockchain has only one node, in which case it's debatable about whether or not it's even a blockchain). A blockchain with N nodes can definitely be fault-tolerant up to N/3 or even, depending on your choice of network model, N/2 faults from the point of view of *authenticity*, but it cannot tolerate even one fault from the point of view of *privacy*. The reason is simple: any node that has the data can publish the data. If your primary information security concern is that you want privacy, and you are not interested in getting into advanced crypto or at least moderately complex cryptoeconomic mechanisms such as state channels, then you likely want a server and not a blockchain.

Truly effective solutions to blockchain privacy generally come in two flavours: low-tech solutions, relying simply on clever combinations of cryptographic hashes, signatures and cryptoeconomics to minimize the amount of revealing information or minimize revealing links between information that is on chain, and advanced cryptographic solutions that use cryptographic techniques to try to provide very strong and mathematically provable guarantees of privacy, allowing the blockchain to process transactions and contract in plain sight but obfuscating the content in such a way that its *meaning* cannot be deciphered. Both kinds of strategies have their own limitations: low-tech approaches tend to be simpler to implement, but usually offer privacy gains that are only statistical in nature, whereas high-tech approaches have greater promise but are also more challenging to develop and may rely on less well-tested security assumptions.

Understanding the Goals

Before starting to develop a cryptographic privacy solution for a (private or public) blockchain, one must first start by understanding what the requirements are. In a public blockchain, a common goal is simply to try to hide as much as possible from everyone. A theoretical ideal would be a “black box” blockchain, using cryptographic techniques to obfuscate the entire state and state transition rules, so that users can send encrypted transactions in and read data that is explicitly associated to them, but be able to decipher nothing else. The technology to actually accomplish this, cryptographic obfuscation, theoretically exists, but is currently so inefficient that it may as well be impossible. A recent paper estimates that "executing [a 2-bit multiplication] circuit on the same CPU would take 1.3 * 10^8 years". However, there are weaker forms of cryptography that can much more practically provide guarantees of complete privacy for specific kinds of data or metadata, and it is these mechanisms that will be discussed in most detail below.

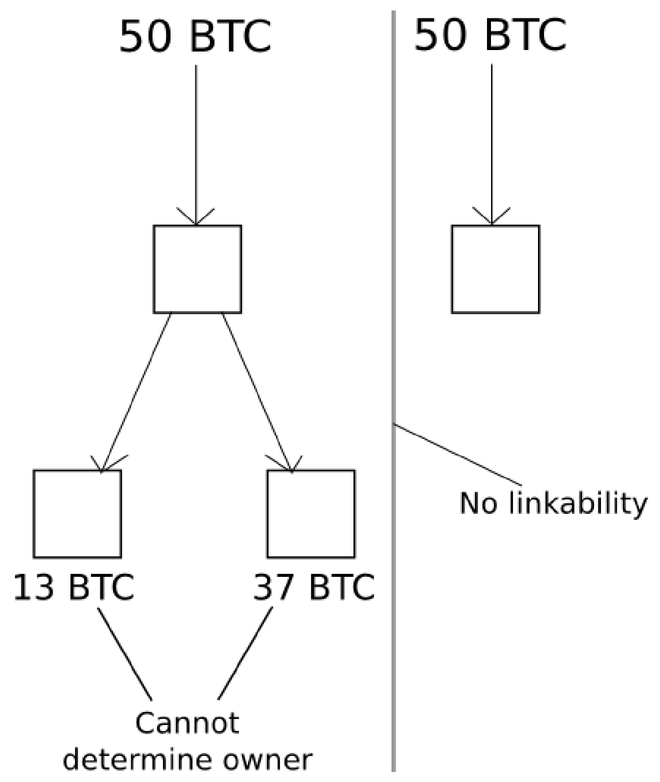
In private blockchains, as well as many kinds of financial applications on public blockchains, the needs are more specific; particularly, there are often regulatory requirements that require information to be divulged in specific cases. There are also requirements for financial institutions to either know some personal information about their users, or at least to have assurances that that personal information has certain properties (eg. one possible requirement may simply be for institutions to have some kind of assurance that their users are not US residents or citizens, as excluding such individuals is a common means for banks and financial startups alike to reduce their regulatory risks and compliance burdens). There may also be requirements based on value amounts, eg. a common requirement is for transactions or related sets of transactions exceeding $10,000 to be reported to a relevant financial regulatory authority.

The "base case" for handling these requirements is to simply publish as much identifying information about each person's account on the blockchain as possible, allowing applications to examine the information associated with each account before interacting with it. However, this arguably reveals far too much; a recent DTCC paper cautions that "the data associated with identity would not be appropriate to have stored on a decentralized ledger until the technology has matured and proven its ability to survive an attack". Hence, the challenge is to see if there are ways to satisfy regulatory and other informational requirements but at the same time reveal only the minimum possible amount of information that is not necessary toward satisfying these requirements.

Low(er)-Tech Solutions

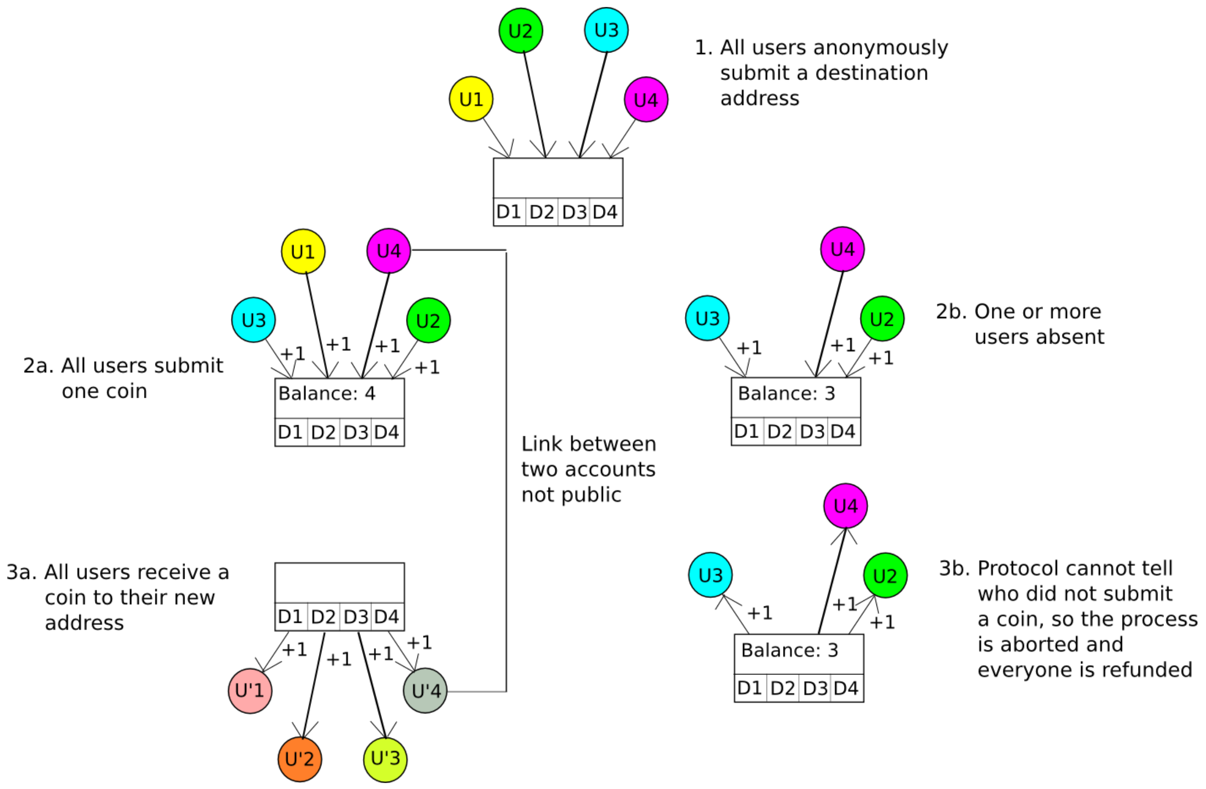
From the low-tech side, there has been an impressive array of solutions that have been designed over the past half decade specifically to improve privacy for the public blockchain payments (and more generally asset transfer) use case. The simplest strategies have to do with making maximum use of separate one-time accounts for each operation, so that a single individual's activities cannot be linked to each other, and then employing so-called **merge avoidance** techniques to deliberately keep one's activities maximally separate from each other so as to minimize the extent to which those activities can be linked or correlated.[23]

[23] See https://crypto.stanford.edu/seclab/sem-14-15/pustogarov.html and http://www.forbes.com/sites/andygreenberg/2013/09/05/follow-the-bitcoins-how-we-got-busted-buying-drugs-on-silk-roads-



A more advanced strategy, **CoinJoin**, creates an on-blockchain protocol where N parties merge and re-split their assets on-chain in such a way that an outside actor viewing the chain cannot tell which input corresponds to which output:

black-market/#64dea53189a8 for examples of bitcoin account de-anonymization using computer-science-theoretic techniques being employed in practice. Sometimes, such techniques are not even required; see the fate of Carl Mark Force after attempting to steal hundreds of thousands of dollars in bitcoin from Silk Road: http://motherboard.vice.com/read/how-a-two-timing-dea-agent-got-busted-for-making-money-off-the-silk-road


1. All users anonymously submit a destination address
U1
U2
U3
U4
D1 D2 D3 D4
2a. All users submit one coin
+1
Balance: 4
2b. One or more users absent
Balance: 3
Link between two accounts not public
3a. All users receive a coin to their new address
U'1
U'2
U'3
U'4
3b. Protocol cannot tell who did not submit a coin, so the process is aborted and everyone is refunded

For the smart contracts use case (more precisely, two-party or N-party financial contracts), state channels, as described in the scalability section above, are by far the best low-tech solution to privacy: unless there is a dispute, or one of the parties to a contract disappears, no one aside from the participants need ever find out the terms of a given contract, or even that the contract ever took place.

If there are requirements for specific institutions to be able to identify which parties own which accounts, there are several ways to satisfy this. Perhaps the simplest is to require each account to be authorized by a specific “KYC authority”; this authority would then know the real-world identity behind each account, and be able to trace funds through merge-avoided transaction chains and CoinJoin instances, but everyone else would simply see semi-anonymized chains of transaction outputs. Identifying information could be published onto the blockchain through a Merkle tree: a KYC authority could sign the root of the tree, and if a user wants to prove specific information to an application (eg. name, age, ID number, citizenship status), then they can provide branches of the tree to the application without revealing anything else.

The largest challenges to implementing such a scheme are likely legal, not technical; as mentioned in a previous section, financial applications are often required to do their own KYC, and cannot simply rely on processes carried out by others. However, the challenges are likely to be highly specific to each application, and many applications may well be able to avoid having to deal with such concerns directly by simply being software providers, leaving custodianship of assets to a smaller set of entities that may be required to perform a larger set of checks.

High-tech Solutions

From the high-tech side, the three best technologies that can be adapted to be used on Ethereum are **ring signatures**, **ZK-SNARKs** and **secret sharing**.

A ring signature is a special type of cryptographic signature that proves that the signer has a private key corresponding to one of a specific set of public keys, without revealing which one. A more advanced version, the linkable ring signature, adds an additional property: if you sign twice with the same private key, that fact can be detected – but no other information is revealed. Ring signatures in general are useful for revealing membership in a set: for example, given a set of authorized users of a particular service, an individual can reveal that they are one of those users without revealing which one. Linkable ring signatures are particularly useful in a blockchain context because linkability adds the key property of double-spend-resistance: even though the ring signature retains its full privacy properties, a single participant cannot make two of them without getting caught.

One natural application of this is simpler privacy-preserving schemes for blockchain-based asset management. For example, one can make a simplified CoinJoin implementation where users simply send one unit of a given asset into a smart contract which then entitles them to withdraw one unit to another account if they can provide a linkable ring signature proving that (i) they are one of the depositors, and (ii) that they have not already provided a signature and made a withdrawal.[24]

Another application, if layered on top of an existing identity platform, is as a secure privacy-preserving one-per-person identity scheme. For example, suppose that regulations allow financial platforms to offer deposits and transfers for small amounts (eg. under $200) without any identifying information, but require identifying information to be revealed for anything larger. Currently, phone numbers are often used as a one-per-person token in such cases, but they have the flaws that (i) they are not privacy-preserving, and (ii) they are not completely one-per-person. One possible solution (assuming the presence of some kind of cryptographic digital identity scheme[25]) is to allow users to supply a linkable ring signature proving that (i) they are an authorized user, and (ii) they have not yet opened an account, at which point they are allowed to open an account whose balance is capped to $199. Users would be allowed to upgrade to accounts with higher limits

[24] Monero is an example of a public-blockchain cryptocurrency that provides such functionality built in.

[25] The question of how to actually put cryptographic digital identity schemes into people's hands is a complicated one, but governments are moving in that direction; perhaps the most successful example so far is Estonia's e-Residency program, which allows anyone in the world to obtain a smart card that can be used to cryptographically sign documents in a way that can be verified

only if they reveal their identity. The privacy properties do not need to be quite so absolute; any time accounts or monetary transfers under a given threshold require a lower level of information to be divulged and accounts or transfers above that threshold require a higher information, linkable ring signatures can be used to preserve the privacy of the higher tier of information.

A natural companion to linkable ring signatures is additively homomorphic encryption. The concept behind additively homomorphic encryption is this: one can encrypt a value in such a way that if you take two values, `x` and `y`, and encrypt them to produce `e(x)` and `e(y)`, a third party can calculate `e(x + y)` without knowing `x` or `y` or any other private information themselves. This technique, together with a cryptographic concept called "range proofs", has been expanded by Greg Maxwell and others into Confidential Transactions, a scheme that allows for all transaction values and account balances on a blockchain to be encrypted, with users only seeing the balance of their own account and the values on transactions coming in to them, while at the same time preserving correctness (i.e. that transactions are on net zero-sum and do not create new money out of thin air and, importantly, that the balance of every account is non-negative; surprisingly, from a mathematical standpoint this latter requirement is the source of the bulk of the difficulty in developing such a scheme)[26].

Such a confidentiality scheme could theoretically be combined with state channels to create an extremely strong privacy-preserving smart contract platform: users would enter into state channels, engage in interactions unseen by other users, and at the end agree on a set of new balances, with both the old and new balances completely encrypted and the only publicly verifiable guarantees being that (i) the change in balances is zero-sum, and (ii) all of the new balances are non-negative. Linkable ring signatures could be used to protect identity, additively homomorphic encryption and range proofs ensuring confidentiality of balances, and state channels ensuring the safety and privacy of every interaction that makes a change to that information. As usual, if there is a requirement for some central party to have greater insight into what is going on, this can be built into the set of smart contracts that users are allowed to use.[27]

[26] And of course, note that the math that ensures balances are non-negative in CT schemes can also be flipped around and repurposed to enforce *maximum* balances; this technique could also be used to enforce account balance or transfer limits while preserving privacy

[27] If it is desired to restrict the kinds of actions that users can take, then one must think carefully about the set of smart contracts that users are allowed to create and participate in; sending assets into a smart contract that says "anyone who can provide a signature matching public key X can withdraw the assets" is essentially an asset transfer to an arbitrary recipient. Another case worth keeping in mind is that a smart contract that says "anyone who can provide a signature matching public key X can withdraw the assets or change the value of X" - essentially, a tradeable contract that holds some assets - could be used to circumvent non-transferability requirements. Whitelisting the set of smart contract templates that users are allowed to enter into at least initially seems like the safest choice.

ZK-SNARKs are an extremely powerful cryptographic primitive whose definition can roughly be described as follows: a ZK-snark is a cryptographic proof that there exists a private input `I`, known only to the prover such that, for some publicly known program P and publicly known value O, `P(I) = O`, without revealing any other information about what I is. To see the power of this primitive, consider the specific example of identity. With ZK-SNARKs, you can provide a cryptographic proof of a claim such as "the owner of this cryptographic key has a digital ID issued by authority X, and is at least 19 years old, a citizen of country Y, and has a driver's license" - all without revealing anything else about who you are.[28]

ZK-SNARKs can be used to create privacy-preservation schemes for asset transfers, essentially providing total unlinkability between transactions, and it can also be used to create strong privacy for smart contracts. A project called **Hawk** introduces a notion of "private contracts" which *trust zero parties for authenticity but one party for privacy*. There is a "manager" that is tasked with centrally updating the state of an encrypted smart contract and providing proofs along the way that the updates are valid.his manager does see all of the data, but even if the manager misbehaves, other parties can continue the interaction on their own at some cost to the privacy of that particular interaction. The manager could simply be one of the participants in the interaction, or it could be a central party; the central party case could be ideal in a regulated financial industry setting where the application developers are required to know what is going on anyway.[29]

A separate question is this: if KYC information must be stored for each account, who stores it? A private blockchain by itself is arguably not sufficient if privacy is the goal, for the reasons outlined at the start of this section. However, there is another kind of cryptography that can get you further: secret sharing. In general, the concept of secret sharing is that it is the equivalent of multi-signature access for data. A piece of data can be shared among N parties in such a way that any M of them can work together to recover the data, but M-1 parties working together would see nothing but encrypted randomness (the values of M and N can be set to any amount, eg. 2-of-3, 6-of-12, 9-of-9, etc. are all possible). Theoretically, one can combine this technology with ZK-SNARKs to create a conditional financial privacy scheme similar to David Chaum's encrypted messaging proposal: have transactions be generally fully privacy-preserving and anonymous, but require them to include further information about the transaction (identity of the sender and recipient, value, etc.) secret-shared between some set of N master keyholders. ZK-SNARKs would be used to ensure that the data is

[28] As usual, this requires either a government authority or an acceptable substitute entity to be willing to create cryptographically signed digital documents that can be interpreted by the zero-knowledge proof scheme

[29] An alternative to these schemes is Enigma, which uses secure multiparty computation to achieve "private contracts" with an M-of-N trust model. Secure multiparty computation makes serious efficiency sacrifices (eg. most protocols require a set of network messages to be exchanged for each multiplication, although if the "circuit" can be processed in parallel then the effect of this in practice is mitigated), but the resulting trust model is quite strong particularly in a private blockchain context. In a public blockchain context, there is no way to prove that the consensus participants did not collude to leak information, hence honesty cannot be incentivized, whereas in a private blockchain context if such collusion is discovered the response of "just suing the bad actors" is always an option.

valid, and transactions that do not carry such valid metadata or proofs would not be accepted into the blockchain.

ZK-SNARKs are an extremely powerful piece of technology, but come with some untested security assumptions, as well as efficiency concerns: currently, creating such a cryptographic proof takes over 90 seconds on an average computer. Hence, for many applications, weaker but more efficient forms of cryptography such as ring signatures and additively homomorphic encryption may be superior.

The Ethereum team's role in implementing all of these schemes is twofold. First of all, many of these techniques can be practically built on Ethereum already; projects such as Raiden have been at work building state channel networks on Ethereum for months, and linkable ring signatures have also been built in an early-alpha form. We intend to work with cryptographers and other developers to implement some of the schemes described above inside of Ethereum to the maximum practical extent. Second, the major bottleneck to implementing these techniques quickly is virtual machine efficiency, and this will continue to be one of the project's major focuses for future development. Hence, the timeline for bringing the full extent of these techniques into practice on the live Ethereum public chain is likely to track closely the development of the Serenity release and the possible WebAssembly rollout. Private chain implementations can of course achieve these gains by integrating the features much sooner, as well as simply adding a few more precompiled contracts.

Key Recommendations

To a much greater degree than is the case for scalability, there is no magic bullet for privacy on a blockchain. Private blockchains are decidedly not a solution; while private blockchains do have 33% or 50% fault tolerance in terms of *authenticity*, they have far less security than a centralized server model in terms of privacy: Instead of there being one node to go after, there are now many nodes, each one of which, if compromised, would reveal *all* of the information.

Instead, there are a number of application-specific tricks that allow you to get high degrees of privacy for certain use cases,particularly, digital asset transfers, identity verification, N-party smart contracts and data storage. The solutions here entail the use of advanced cryptography such as ring signatures, additively homomorphic encryption, zero-knowledge proofs and secret sharing. Privacy solutions can be highly customized, for example divulging specific information to specific parties (the participants to a transaction, financial institutions, regulators, etc.) only under specific circumstances. The extensive use of such systems presents an exciting promise of applications being able to satisfy regulatory and other informational requirements while simultaneously preserving a very high degree of privacy.

It is on the Ethereum project's roadmap to implement these schemes and make it as easy as possible to develop privacy-preserving applications on Ethereum using these technologies, and in the case of state channels there are projects implementing them already. However, private-chain Ethereum users are free to race ahead and implement whatever schemes from the above list are desired as precompiled contracts on their own schedule.

Purity

Perhaps the most important point to keep in mind when discussing how these techniques can be applied to Ethereum is this: the guiding philosophy of the platform is one of maximal generality and one of maximal developer-friendliness. Unlike platforms such as Monero, which try to build in linkable ring signatures as a mandatory part of the coin management scheme on the bottom level, Ethereum views itself as a computing platform, agnostic to what is built on top. Hence, it is not the goal of Ethereum to "become an anonymous coin", or, to take the opposite extreme, build a real-world identity management system into the protocol layer for either the public or private chain instantiations of the protocol.

Rather, Ethereum's guiding philosophy is to build the technology in layers. "Layer 1" is the Ethereum platform itself, including the Ethereum Virtual Machine, the execution environment and the blockchain consensus algorithm. On top of this, one can build all of these privacy preserving schemes as a "Layer 2". Currently, the Ethereum protocol is arguably not completely "pure": the protocol explicitly builds in a notion of accounts with balances and sequence numbers, protected by the secp256k1 digital signature algorithm, as the primary "entry point" for transaction execution. With Serenity EIP 101, however, we plan to change this, abstracting even these features out: all transactions will be valid as long as they are correctly formatted, and all transactions will appear to "come from" the entry point account `0xffffffff…` User accounts will become "pass-through contracts", smart contracts on Ethereum that accept messages from the entry point and pass them along only if they contain valid cryptographic signatures and sequence numbers.

A key consequence of this is that Ethereum's current model of accounts with sequence numbers will no longer be privileged; instead, account management schemes involving UTXOs, linkable ring signatures plus confidential transactions, or any other desired scheme could be implemented and will feel just as "native" as accounts and sequence numbers do today. While the EVM has not yet upgraded to WebAssembly (or some other alternative if we end up going that route), there is still room for some cryptographic operations to be explicitly included so that they avoid virtual machine overhead and be processed with reasonable efficiency, but these operations are still maximally encapsulated, and the eventual plan is to move everything to being done inside WebAssembly code. At that point, a private or public blockchain instantiation of Ethereum in practice may well come with a complex set of second-level tools already included, allowing users to create cryptographically secured accounts, manage their identities, selectively reveal or prove information about their identities to specific parties, or carry out any other such operations. The underlying base layer will be maximally simple and efficient, so as to give developers maximum freedom to build the tools that suit the requirements of various industries and applications on top.

Conclusion

Ethereum offers a highly generalized platform that allows users to make applications for a very wide variety of use cases with much less effort than it would take to create their own blockchain. The platform’s vision is that of “the world computer”: to create a system which looks and feels to users as much as possible like a computer, while gaining the security, auditability and decentralization benefits of blockchain technology. Future developments in the roadmap aim to maximally solve the problems inherent in blockchain technology, including scalability and privacy, in a way that is maximally friendly to as many categories of applications as possible and that maximally preserves, and improves upon, the current “developer user experience”.

Areas where Ethereum does not make sense largely consist of those where (i) the benefits of creating a specialized platform, including possibly the efficiencies of achieving scaling in one particular way that targets one specific application, exceed the additional development costs of creating a custom solution, or (ii) blockchains do not make sense at all. Areas where Ethereum makes the most sense are those where either (i) a high degree of future-proofness and ability to quickly add in new functionality, possibly even without the cooperation of the blockchain node operators, is desired, (ii) synergy between large clusters of applications is desired, or (iii) the ability for users to be able to enter into arbitrary programmatic “smart contracts” is useful.

The choice between using the Ethereum public and a private or consortium chain depends on the developer and the application. Developers without institutional support have so far overwhelmingly gone with the public chain, as doing so adds fewer barriers to entry or hurdles in convincing others to adopt their application. Financial institutions which already have millions of users do not have this concern, and so such institutions should evaluate both options. One “safe” strategy in the near term is to try consortium chains with 5-25 nodes (one or several per institution) first, particularly because (i) their scalability is superior in the near term, (ii) it is easier to convince legal departments and regulators about the low-risk nature of a “controlled” system, and (iii) they will have access to new features such as EIP 101 and 105 before the public chain; in the longer term, the choice of private vs. consortium vs. public will depend on the specific application, and particularly the tradeoff between performance and interoperability.

Users that are interested in privacy should not look to private chains as a panacea; any multi-node blockchain will inherently be less secure than a single-server solution at the protocol level, and so cryptographic schemes such as state channels, ZK-SNARKs, ring signatures, etc. should be explored as they allow financial data to be stored and computed on, using blockchains in a way that does preserve high, though not infinite, degrees of user privacy. It is in Ethereum’s roadmap to be as friendly as possible to implementing such privacy-preserving schemes on top of the protocol.