Smart oracles provide a simple, flexible way to implement "smart contracts", which encode business logic, laws, and other agreed-upon rules. Smart oracles build on the idea of oracles, or entities that provide smart contracts with information about the state of the outside world, and combine information gathering with contract code execution. In such a system, rules can be written in any programming language and contracts can interact with any service that accepts cryptographically signed commands. This includes, but is not limited to, cryptocurrency networks. We introduce an implementation of smart oracles, called Codius (based on the Latin “ius” meaning “law”), which uses Google’s Native Client for code sandboxing.

In this paper, we begin with some definitions and background on the concept of smart contracts. From there, we move into our proposal for smart oracles and describe some of the technical implementation details and the security threat model for the system. In the final sections, we describe some of the financial and non-financial applications and the greater potential for this approach to smart contracts as a whole.

Definitions

Smart Contract

The Cornell Legal Information Institute defines a contract as:

An agreement creating obligations enforceable by law. The basic elements of a contract are mutual assent, consideration, capacity, and legality […] Possible remedies for breach of contract include general damages, consequential damages, reliance damages, and specific performance.

Smart Contracts are programs that formally encode certain conditions and outcomes. The code is agreed upon by the contracting parties in advance and must be faithfully executed by a disinterested, neutral system.

The three key steps in developing and utilizing a smart contract are:

Translating the terms of the contract into code. Since digital systems are deterministic, all possible outcomes of a contract, including penalties for breach of contract and referral to a (non-deterministic) arbitrator, are specified explicitly.

Agreeing on the precise code that will be run. In practice, parties would usually build their contract from widely used configurable contract modules. Once the contract is agreed upon, it is very important to ensure that the same code actually ends up being executed. See sections Deterministic Compilation and Unique Secret and Key Pair.

Executing the code in a trustworthy manner. The code must be run by an impartial third party or by a group of independent entities that are highly unlikely to collude. Smart contracts can also be used without ever actually executing the code; see the section on Offline Contracts for more details on this use case.

Ultimately the benefits of using smart contracts instead of traditional contracts come from the increased speed, efficiency, and trust that the contract will be executed exactly as agreed.

Oracle

Some smart contracts systems, including the one built into Bitcoin, are strictly deterministic. In order to interact with the real world, these systems rely on cryptographic signatures submitted by outside systems called "oracles."

Oracles are trusted entities which sign claims about the state of the world. Since the verification of signatures can be done deterministically, it allows deterministic smart contracts to react to the (non-deterministic) outside world.

Smart Oracle or Contract Host

Smart oracles are explained in the section following the definitions, but we should note that we use the terms "smart oracle" and “contract host” interchangeably. In this proposal, the hosts that execute the contract code are the same as the “oracles” that in other systems might only be set up to provide contracts running outside of their systems with information about the outside world.

Contracting Parties

The contracting parties are the people or businesses who agree to use a smart contract to carry out an arrangement. Other relevant, but possibly distinct entities, are the contract author and the contract owner. The contract author is the one who wrote the code, though they may not be involved in the particular arrangement at all. For example, the author could be a developer or group of developers that have published an open source auction contract. The contract owner is the entity that sets it up to be executed by the smart oracle(s).

Note that the contract host should not be one of the contracting parties or affiliated with any of them.

Public/Private Key Cryptography

Public/private key cryptography enables messages to be encrypted, or translated into a seemingly random set of characters. When a message is encrypted with the public key, only the holder of the corresponding private key can decrypt or decipher it.

Public/private key cryptography also enables the holder of the private key to cryptographically "sign" messages. Anyone can definitively verify that the signature could only have been created by the holder of that private key.

Public/private key cryptography underpins some of the common use cases for smart oracles, so it may be useful to have a basic understanding of how asymmetric encryption and cryptographic signatures work. For more background, we recommend the Wikipedia articles on public-key cryptography and digital signatures, and this simplified explanation of public key cryptography.

Distributed Networks and Consensus Databases

Although this smart oracles proposal is independent of all existing distributed networks, consensus databases, and cryptocurrencies, it has been heavily inspired by the concepts underpinning Bitcoin and Ripple. For an introduction to Bitcoin, the peer-to-peer network and

digital currency, see bitcoin.org. For more information on Ripple, the distributed protocol for any type of value transfer, see ripple.com.

From Oracles to Smart Oracles

The concepts of smart contracts and oracles have existed for some time. Several earlier designs (including Bitcoin) have relied on executing the contracts within consensus networks, leading to the requirement that their execution be deterministic. In this paper we aim to show that placing contract execution in the hands of smart oracles generalizes and simplifies the system significantly.

The concept of smart contracts is widely attributed to Nick Szabo who, in the late 1990s, argued that formalizing relationships and encoding them in software and hardware would simplify and secure business logic and functionality. He wrote of embedding contractual clauses, such as bonds and property rights. Szabo used the example of the vending machine as a "primitive ancestor" of smart contracts because its hardware and software enforce a simple contractual agreement. Anyone who inserts money will receive a snack in return, even though no explicit contract was ever made with the machine's owner. Wei Dei also wrote about digital contracts in his B-money proposal of the late 1990s, describing self-enforcing, cryptography-based contracts not too dissimilar to Szabo's ideas.

Recently, the advent and explosion of interest in cryptocurrencies has spurred a resurgence of interest in smart contracts. Math-based currency networks provide an important building block for smart contracts: valued digital assets that can be transferred with a cryptographic signature. Assets in protocols such as Bitcoin and Ripple are owned by accounts identified by public/private key pairs. Payments are executed when the transaction carries a cryptographic signature that could only have been produced by the holder of the account's private key. Smart contracts can trivially create such cryptographic signatures and, thus, be designated the partial or sole owner of any type of digital asset.

Unfortunately, cryptocurrency developers have found it challenging to design a system that encompasses both a powerful smart contracts language and a robust consensus system. Bitcoin scripts allow for simple logic to be encoded and executed on the Bitcoin network. However, encoding advanced logic and executing untrusted code have proven more complicated to integrate.

Consensus networks must be conservative about their feature set. Since everyone in the network must agree on each change the technology is relatively difficult to modify or upgrade. The Bitcoin wiki explicitly mentions this concern, saying "Some of the more complicated opcodes [script commands] are disabled out of concern that the client might have a bug in their implementation; if a transaction using such an opcode were to be included in the chain any fix would risk forking the chain." Forking the blockchain, or distributed ledger, means creating multiple competing states of the network, which is a highly undesirable outcome for a consensus-based system.

We argue that it is possible to implement powerful smart contracts in a secure and trustworthy manner without increasing the complexity of existing consensus networks such as Bitcoin or Ripple.

The execution of untrusted code should be decoupled from the consensus databases and other services that track and transfer asset ownership. The separate contract system can handle untrusted code execution and interact with the consensus databases through cryptographic signatures. These signatures are already native to consensus protocols so no modifications are necessary. Decoupling contracts from consensus networks gives the added benefit that contracts can interact with multiple networks at once as well as virtually any type of online service. This means that a single smart contract could interact with Bitcoin and Ripple, web-based services like PayPal, Google, Ebay, etc. or even other Internet protocols, such as SSH, LDAP, SMTP and XMPP.

If the contract execution is decoupled from existing systems, where should the code be run? This is where smart oracles come in.

Most proposals for smart contracts, even those that are internal to consensus networks like Bitcoin, depend on independent entities to inform contracts about the state of the outside world. Bitcoin contracts rely on "oracles" to attest to facts from the outside world by introducing signatures into the network if and only if specific conditions are met. Smart oracles takes this concept a step further to place the untrusted code execution in the oracles' hands. The smart oracles are trusted or semi-trusted entities that can both provide information about the outside world and execute the code to which the contracting parties agreed.

Implementing Smart Oracles

Smart oracle implementations could take many different forms. In the following sections we outline some of the elements we see as essential for most, if not all, smart oracles. Namely, the key components are: securely identifying code, sandboxing code, oracle APIs, contract hosting and billing models, and contract clients.

We should note that the following sections dive into the more technical details of smart oracles. Readers who are more interested in the big picture and less interested in those details may want to skip to the last three sections: Offline Contracts, Financial Applications and Beyond and the Conclusion.

Securely Identifying Code

Once contracting parties have agreed on the terms of their arrangement they must translate the rules into code. It is crucial that the parties inspect the proposed code and ensure that it represents the business logic to which they agreed to be bound. It is equally important that they can easily verify that the code uploaded to the smart oracle(s) is exactly that which they already inspected. This is where deterministic code compilation, hashing, and code reuse with modules come in.

Deterministic Compilation

All parties to a contract have a large stake in ensuring that the final, machine-executable code represents the logic they agreed to. For compiled languages, this means that the source code must be shared along with a reproducible process to compile it to machine code, such as with Gitian. For interpreted languages it is sufficient to share the source code. Either way it is critical that participants agree upon the final instructions that will be executed by the smart oracle.

Hashing

Cryptographically secure hashes are a convenient way to identify agreed-upon binaries or source code files. Hashing functions take arbitrary amounts of data as inputs and produce a short, fixed-length string of characters that. For practical purposes, this "hash" can be used to uniquely identify any text or data.

Although it might not be strictly necessary, we recommend using collision-resistant hash functions. This means that it would be impractical to attempt to find two inputs with the same output hash. It would be exceptionally difficult to produce two pieces of working code with the same hash, even using a hash function that is only second preimage resistant. However, it would cause serious problems if someone could create two distinct contracts with the same hash. Therefore we recommend hash functions that are second preimage and collision resistant.

Modules

Traditional contracts often share common "boilerplate" elements and smart contracts are no different. Any smart oracle system is likely to offer some form of code reuse. This adds convenience as well as security.

Many contracts will have relatively simple and easy to understand logic built on top of well-known and widely used modules. Modules could encompass basic functionality, such as mechanisms to connect to Bitcoin or Ripple. They could also include more advanced features such as a standard auction, escrow, or bond implementation. The logic would likely be widely used and verified by many independent parties.

Codius uses modules identified by hashes that can be shared and imported by multiple contracts.

Code Sandboxing
The heart of the smart oracles concept is the ability for users to agree on the code of a contract and then to upload it to a trusted third party or parties for them to execute it. Smart oracles must be able to safely execute the user code, which is untrusted and may actually be malicious. Oracles must protect their own systems and the integrity of the other contracts they are running.

The four most commonly cited methods for sandboxing, or constraining the functionality of, untrusted code are described below. Different smart oracle systems may choose to use distinct subsets of these options, but they can be layered together for increased security. At the end of this section we will discuss the four methods together and explain our selection of Google's Native Client for Codius

1. Virtual Machines (VMs)

Virtual Machines (VMs) are environments that emulate separate computers within a single physical machine. A server or computer can run multiple VMs and each will have its own complete operating system. In most modern implementations, VM security relies on the computer processor's virtual instruction set. Communications between the VM and the outside world or the host machine are strictly managed by a Virtual Machine Monitor, also known as a hypervisor.

VM technology dates back to the 1960s and is now widely used for sandboxing code. Most cloud computing providers use VMs to run multiple users' code per server. VMs may be one of the more secure methods for sandboxing code but the downside is that they are relatively costly in terms of computer resources. Since each instance contains a full operating system, the time and energy to start one each time a contract is run is impractical for many contracts. Nevertheless, hosts may choose to offer VMs as an option for high value contracts where the owners will be willing to pay more for a safer execution environment.

2. Operating System Protection Domains

Protection domains, or "rings", are built into many processor architectures, notably x86. They are used typically by the operating system and allow it to isolate individual processes from each other and from access to the underlying hardware. Security techniques that ultimately rely on protection domains include process-based isolation, FreeBSD jails, linux containers (LXC), SELinux, AppArmor and many more.

Container systems, such as Docker, are quickly gaining popularity over traditional VMs for software deployment because they are lighter and faster to start. However, these containers are not secure enough to be used as a sandboxing technology for untrusted code.

Under this model, all of the security relies on the host operating system's ability to enforce the privilege layers. This means that bugs in the kernel may lead to sandbox exploits. Furthermore, most popular operating system kernels provide a very large attack surface making it fundamentally harder to guarantee their security compared to a sandbox that relies on a smaller trusted code base. That said, operating system-based features such as protection rings and process-based isolation can be layered with other mechanisms for additional security.

3. Software Fault Isolation

Software Fault Isolation (SFI) relies on compiling software to a reduced instruction set, or constrained set of possible commands at the machine code level. The sandbox can enforce its rules by statically verifying the binary to ensure that it does not include any operations outside of the allowed set.

SFI is an attractive form of sandboxing, because the verifier is the only trusted component and can be implemented using very little code. This results in a minimal trusted code base, which makes the system easier to verify and therefore more likely to be secure.

4. Capability-based security

Capability-based security is the design principle that programs should not even be able to reference functionality or resources that they are not meant to use. It is most simply understood as akin to "Newspeak" from Orwell's 1984, which seeks to "eliminate personal thought by restricting the expressiveness of the English language." You cannot break the law if you do not even have the words to express an illegal thought. The principle for programs is similar.

A smart contract system could sandbox the untrusted code by requiring contracts to be written in a specific capability-based language. The programming language E was specifically designed

to require that all resources be accessed using unforgeable capability tokens. Interpreted languages such as JavaScript fundamentally use a similar principle: the language only exposes certain classes and functionality – such as web APIs – in the browser. A system could also implement a separate custom language with these same properties. Unfortunately, if capability-based security is the only sandboxing layer used, every contract author will be forced to use a single language that is not widely used outside of this context.

Codius and Google Native Client

Google's Native Client is a sandbox for running untrusted x86 code, the low-level commands used by most computer processors. Native Client was developed to run compiled binary code on the web, as opposed to the HTML/CSS/Javascript that websites are normally limited to. Native Client makes a number of improvements on top of software fault isolation (described above) to provide a constrained execution environment that protects users from potentially malicious code.

Native Client can be used to run any programming language and currently supports C, C++, Python, V8 JavaScript, Ruby, Go, Mono, and Lua. Recent versions of NaCl support x86-32 and x86-64 architectures, as well as ARM and MIPS. Google uses Native Client for computationally intensive web apps, such as Hangouts Video and QuickOffice, among others, as well as ChromeOS apps and datacenter hosting of untrusted code. The latest benchmarks have shown that Portable Native Client modules run only 10-25% slower than LLVM-compiled native code, so Native Client is not only efficient to start up but it also provides performant execution.

Codius uses Native Client because it provides a unique combination of security, performance, and flexibility.

Native Client is lighter weight than a VM and provides a much smaller attack surface than an entire operating system managing containers. VMs and containers may develop a better balance of performance and security in the future, but they do not meet our requirements at present.

We argue that requiring contracts to be written in a specific capability-based programming language would needlessly hamper the adoption of the system. Rather, if some authors or hosts prefer contracts to be written in a custom or capability-based language they can also do so within the Native Client sandbox and it will provide yet another security layer.

Software fault isolation and Native Client rely on a minimal trusted code base while being flexible enough to support all programming languages and allow for the reuse of already-developed and widely-used modules.

Contract APIs

Even though the smart contract code is sandboxed and the functionality is constrained, smart oracles will want to expose specific APIs (Application Programming Interfaces) to the contracts they run. In order to give more granular control over contract functionality, Codius contract authors explicitly specify what APIs they should be able to access.

Although contract hosts can develop and expose any APIs they choose, we would in general argue for an approach that emphasizes code inside the sandbox over external APIs. APIs are

the building blocks for complex functionality, but they must become part of the trusted code base of the smart oracle. Modules, on the other hand, can be easily developed and included for specific contracts and are not integrated into host.

Below we describe some of the core APIs we expect smart oracles to offer, and those that Codius will provide.

Unique Secret and Key Pair

One of the key properties of a contract is that it has a cryptographic identity. Specifically, a running instance of a contract has to be able to prove to the contracting parties that it is indeed a specific, identified code base running on a specific contract host.

In a previous section we addressed how parties can agree on the hash of the code. To associate this hash with a given instance running on a specific host, this host will generate a unique key pair for each contract and sign the public key.

Entropy (Randomness)

Many contract use cases require cryptography. While cryptographic primitives can be implemented inside of the sandbox (assuming the sandbox is efficient enough), many cryptographic protocols require a good source of entropy, or random values. Therefore smart oracles should provide an API for obtaining entropy that the host deems secure enough for cryptographic applications.

Internet

The classic use of oracles is to interact with the real world and provide information about its state to smart contracts. Generally this means interacting with services via the Internet. Even though HTTP services are probably the most interesting by far, we see no reason to restrict API capabilities higher than the transport layer (OSI layer 4, e.g. TCP/UDP). This means that any application layer protocols, such as HTTP, SMTP, even Bitcoin or Ripple can be used from within the sandbox simply by making direct transport layer calls to the outside.

Codius provides APIs for TCP and UDP. Currently, only outgoing sockets are permitted, due to the fact that we assume contracts will be short-running programs. However, in the future there may be functionality for listening and handling events as well. (Contracts may be suspended until a request arrives etc.)

Filesystem

As contracts grow in complexity so too will the need to manage this complexity. It may be possible to compile each contract into a single binary or source code file. However, it makes more sense to implement a virtual filesystem that contracts can access. This provides a way to bundle contracts with static data. Codius contracts must include the hashes of any static files they link to so the smart oracle can enforce access controls.

The filesystem API is slightly different than the functionality to include modules, although the exact implementation may be similar. Including modules allows contracts to avoid duplicating

code that others have written and perfected. Having a virtual filesystem allows contract authors to structure their projects in logical ways. Furthermore, many normal code projects and utilities are written using filesystem commands so emulating that structure makes it significantly easier to port existing programs into the sandbox.

We also anticipate that contracts will want to make use of some local storage facility. Providing filesystem write access seems unwise but an equivalent to the web browser's local storage may prove useful.

Time

Having access to accurate time information is a useful ability for any computing platform. For example, some cryptographic algorithms, such as time-based one-time password (TOTP), require an accurate time reference.

Highly accurate clocks are also used in distributed databases, such as Google's Spanner, in order to make these more efficient, fair and accurate. Google operates GPS and atomic clocks, because these devices are cheap to purchase and operate while providing very accurate timing data, especially when used together. We recommend that contracts hosts provide a timing API using the most accurate clock they can muster.

Additional APIs

Although we expect the aforementioned APIs to be made available by most smart oracles, this is not an exhaustive list of possibilities. Smart oracles can define the sets of APIs they will offer or even develop custom ones. However, as was previously mentioned, we argue that APIs should be thought of as building blocks while reusable modules should be the main mechanism for harnessing more complex functionality.

We should highlight the fact that although this system has been inspired by Bitcoin and Ripple, it is independent of any particular distributed network and is not even limited to interacting with these types of networks.

Contract Hosting
Single (Trusted) Host Model

The most basic implementation of smart oracles involves only one oracle. The oracle is trusted to execute code properly, and the participants must have faith that it will not disappear with any assets the contract controls or collude with any of the contracting parties.

Many companies today offer to run code for other businesses and individuals, and are trusted to do so. The single trusted host model is similar to internet hosting services and software as a service (SaaS) providers. We anticipate that for many use cases the security of using a single host will be sufficient, and the simplicity of such a setup will make it an attractive option.

In order for smart contracts to publish their outcomes in a verifiable way, smart oracles will likely supply each contract with a unique public/private key. The oracles can cryptographically sign a token for each smart contract to publicly assert that they generated that public/private key pair specifically for the instance of that particular contract running on their system. The smart oracles

would have well-known or easily accessible public keys so that contracting parties could verify these signatures. In the single host model, contracts could also be given shared private values, such as the API keys for other centralized web services. A contract could thus report its results or initiate some kind of transaction using the API key instead of using a cryptographic signature.

Multiple (Untrusted) Host Model

Although the single host model may be deemed adequate for a large number of use cases, scenarios that involve high value or low trust will be served best by the multiple host model. In this model the contract code is distributed to some number of independent smart oracles, for example 10. A threshold is set such that some number of those oracles must agree on the results in order for the contract outcomes to be realized. For example, one could use a 7-of-10 scheme that allows for up to three oracles to behave maliciously, be offline, or even be hacked without affecting the execution of the contract. This would be more complicated and costly to setup than the single host model, but it would provide better security guarantees because there would be no single point of potential failure. In practice, the multiple host model would likely be implemented using cryptographic multi-signatures or threshold signatures, both described below.

Multi-Signatures

Multi-signature schemes involve multiple predetermined entities signing a single bit of information such that the result is only considered valid if a specific number of the original entities' signatures are present. Smart contracts that are set up to use multi-signatures would be run on independent smart oracles, and each instance would be given a unique public/private key pair. Similar to the single host model, each smart oracle would publicly attest to the fact that the key pair is unique to the specific contract running on its system. The contract instances would each send their signatures to some central entity or publish them somewhere publicly.

Currently, Bitcoin scripts enable multi-signature controlled accounts and Ripple multi-signature support is under development. A contract Bitcoin or Ripple account could be set up to be jointly controlled by the key pairs of the contract instances running on a number of smart oracles so that the contract would have exclusive control over those assets. The downside of multi-signatures is that verifying signatures is a relatively costly operation for the Bitcoin and Ripple transaction validators, so adding more signatures for a single transaction will lead to higher transaction fees. Nevertheless, the advantage of this model is that all of the contracts can produce their results independent of one another, and the signatures can be trivially collected by another entity, such as the contracting party that ultimately "wins" control of the funds.

Unfortunately, the more hosts are participating in the scheme, the more signatures have to be submitted. This matters especially for distributed consensus networks, where every validator has to verify every signature. Because of this, most consensus networks impose limits on the number of signatures or the number of signers. For example, at the time of writing, Bitcoin allows up to 15 signers in standard transactions.

To address this issue, threshold signatures schemes are an attractive option as they result in a single valid signature, no matter how many parties have participated in the signing. We'll discuss them in the next section.

Threshold Signatures

Threshold signatures are signatures computed by mathematically combining multiple distinct signatures. Different threshold signature schemes allow for different levels of customizability for the precise threshold used, but some support arbitrary thresholds. It is important to note that the joint private key is never recreated but rather the independent pieces of the signature are computed separately and then merged into a single signature for a specific piece of data.

For (EC)DSA Signatures

The Threshold Elliptic Curve Digital Signature Algorithm is a method described by Ibrahim, M.H. et al (2003) and Goldfeder, et al (2014) to jointly generate an ECDSA signature, the type used by Bitcoin and Ripple amongst others. The algorithm is designed so that none of the parties learns anything about the secret values of the others. The signature produced is indistinguishable from a normal ECDSA signature so it would work immediately with Bitcoin, Ripple, and other systems without any modifications. The downside is that at present the best version of the algorithm requires that the total number of parties, n, be at least 2t+1, where t is the security threshold. This would make arbitrary threshold schemes impossible. Furthermore, this algorithm uses multi-party computation, which means that the contract instances must know about and communicate with one another to generate the signature.

For (EC)Schnorr Signatures

The Schnorr signature algorithm is an example of a scheme that allows for the composition of multiple independently generated signatures into one. Each signing entity, a contract instance running on one of the smart oracles, would be able to produce and publish their signature. Another entity could verify each of the shares of the signature and, if there are a sufficient number, compose them. OpenSSH recently added support for a particular type of Schnorr signature called Ed25519, and Ripple is currently considering adding support for this signature algorithm. Unfortunately, Bitcoin does not support Ed25519 so this particular threshold scheme would not currently be compatible with it. Nevertheless, Ed25519 represents an efficient algorithm for composable, flexible threshold signatures so we believe it may be a good scheme to be used in concert with the multiple host smart oracle model.

Billing

One of the most flexible pieces of smart oracles is the billing system that allows contracting parties to pay the smart oracles for the contract execution. The billing system is entirely decoupled from the core system design so smart oracles can accept any payment methods they choose, from credit cards to Bitcoin. The decision to require costs to be prepaid or billed after the fact is also left entirely to the oracles' operators.

Fault Tolerance

There are several guarantees that a smart contract could aim to provide, including:

Validity - The code is faithfully executed as written.

Availability - It is possible to interact with the contract at any time.

Confidentiality - Values that are not explicitly meant to be disclosed, will not be disclosed.

Our proposal for smart oracles does not mandate a specific algorithm to be used for fault tolerance, meaning the level of fault tolerance depends entirely on the contract.

For our analysis we assume an example contract using a threshold signature scheme where t is the threshold and n is the total number of contract hosts. The threshold t *must be greater than one and less than *n. To create a signature t+1 signers are required. We assume that all requests to the contract hosts are initiated by some client. We assume that each client wants its own request to succeed.

Such a contract would provide validity guarantees for up to t faults, availability guarantees for up to (n - t) - 1 faults and no fault tolerance for confidentiality guarantees.

Confidentiality

If stronger confidentiality guarantees are required, the parties have several options. They may choose to compartmentalize parts of the contract and use different sets of contract hosts for each part. In this case it would be possible to use blind signatures to hide the association between the different parts of the contract.

Another option is to use cryptographic techniques such as homomorphic encryption and zero-knowledge proofs to hide the actual data that the contract hosts are operating on from them.

Finally, in many cases, parties may choose to use offline contracts, which means they do not have to reveal any information to any contract host unless a dispute arises. See section Offline Contracts.

Contract Clients
In order to manage their participation in smart contracts, the parties need software which implements their side of the interaction with the contracts host.

For many contracts modules we expect there to be corresponding client software. Depending on the use case, this software may have very different characteristics, from elegant graphical user interfaces to barebones command-line tools. For the purposes of this white paper, we assume that the authors of contracts will also provide suitable clients or instructions for third-party developers to let their existing clients interact with the contract.

Offline Contracts
In the current legal system, most contracts complete "offline", i.e. without invoking the legal system. It is the threat of invoking the legal system which incentivizes the parties to adhere to the agreed-upon terms.

In the realm of smart contracts, offline contracts are also possible and useful. Parties would follow a protocol along these lines:

Create a contract with penalties for each party if they violate the agreed-upon terms.

Add a random value to the end of the contract code. This will make it intractable to gain any knowledge about the contract details from the hash of the code.

Request the public key or keys that the contract would receive from the contract host or hosts, but do not upload the actual contract.

Give the public key(s) access to escrowed funds or some other form of power over the parties' assets.

The parties can now proceed to interact with each other offline.

If any party cheats any other, the victim can upload the contract and run it to impose penalties for breach of contract.

As long as none of the parties cheat, the contract will never get uploaded and executed. Some notes:

Parties will have to make credible threats that they will actually upload the contract and impose penalties if the other side cheats. Again, this is analogous to the current legal system, where parties have to have a convincing threat that they will sue. One difference is that with smart contracts, the "suit" may be cheaper (nearly free), faster (nearly instant) and have a predictable outcome.

As mentioned, this type of contract avoids the costs of running the contracts except in the event of a dispute. This may raise the question how contract hosts make money. We can see two ways:

Contract hosts make money from disputes

Contract hosts make money from issuing the public keys (see step 3 above)

It is possible to avoid the call to the contract host in step 3 using identity-based cryptography or by allowing all contracts on the same host to sign messages with a common key (but certain constraints on the message, e.g. forcing the message to adhere to a specific format which includes the contract hash.) It would be the contract host's choice whether or not to offer such functionality.

Financial Applications and Beyond

Smart contracts in general can be used to model any type of agreement or relationship that consists of clear conditions and outcomes. Smart oracles makes the implementation simple, flexible, and powerful.

The following are some of the applications we can anticipate now, loosely ordered from the simplest to the most complex. Since the system is so extensible, we expect the functionality to expand greatly from here as the ecosystem develops and contract authors build on the ever growing base of modules and existing contracts.

Bridges between value networks. Distributed networks like Bitcoin and Ripple maintain separate ledgers or blockchains that track accounts and balances. Traditional financial systems have

their own ledgers as well. Contracts built on smart oracles can create automatic and fully trustworthy bridges between disparate systems. Such a bridge could accept payments in one system and immediately issue a balance or initiate a payment in another.

Escrow. Smart contracts can easily be set up as escrow accounts that monitor an exchange between two people. The buyer of some goods, property, or services would transfer the payment to the contract account. The contract would monitor external services, such as WHOIS registries for domain names or public home ownership records for real estate purchases. When the ownership has been transferred from the seller to the buyer, the contract would automatically release the funds to the seller.

Cryptocurrency wallet controls. Currently Bitcoin and Ripple have no good mechanism for enabling pull payments, where the seller can initiate a payment on behalf of the buyer in the way that credit and debit cards do. Wallets controlled by contracts could include many different types of complex controls, from daily withdrawal limits to granting and revoking access for specific entities. This would enable subscription and conditional payments, and granular controls over wallet access without disclosing the private key.

Auctions for digital assets. A smart contract can trivially carry out the rules for an auction if it is given ownership over a digital asset or title. It can either be setup to accept payments on a network like Bitcoin or Ripple as bids, and return all but the winning person's bid, or signed transactions can be stored by the contract code directly and only the winner's submitted.

Derivatives. Contracts that monitor the performance of digital or non-digital assets can also be used as futures, forwards, swaps, options.

Debt and equity. Other securities based on payments and rights that are carried out according to predefined rules can also be written as smart contracts.

Smart property. The classic example of smart property is a car that knows who its owner is based on a transferable but non-forgeable digital token. Contracts can be set up to govern the transfer of ownership and accompanying rules. This includes temporary delegation and potentially use of the property as collateral in other agreements.

Voting. In the future smart contracts can be used to enforce democratic, bureaucratic, and other types of control structures over assets or even organizations. As with all of the other applications, the contracts enforce predefined rules, even including rules for modifying the contract's own code. Many non-financial applications require more complicated infrastructure and a more developed ecosystem so we expect that it will take some time for this to be built.

Conclusion

Smart contracts are an exciting new frontier for technology, business, and law. We hope that this paper and our implementation is a contribution towards bringing these concepts to life.

Smart oracles combine the idea of an oracle, which provides information about the real world, with a sandboxed code execution environment. It is independent of existing distributed networks such as Bitcoin and Ripple and can interact with any Internet-based service, including all distributed consensus databases. Separating the untrusted code execution from distributed networks reduces the complexity and thus increases the security of both systems.

The Codius implementation uses Google's Native Client to sandbox untrusted code, which enables developers to write contracts in any programming language. It uses deterministic compilation, hashes and signed keys to securely identify contracts and modules. We suggest methods for multi-party signatures to distribute computing for lower trust or higher value scenarios.

Codius and smart oracles in general open up new possibilities for developers, entrepreneurs, and enterprising legal and financial professionals. Agreements that previously required lengthy legal contracts can be translated into code and run automatically by smart oracles. Smart contracts hold the potential to empower people to build a fairer, more affordable and more efficient legal system and smart oracles are one of the simplest ways to realize that dream.

Want to get involved? Join the community! Visit www.codius.org