Smart Contracts on Bitcoin Blockchain

BitFury Group

Sep 04, 2015 (Version 1.1)

Abstract

Smart contracts are one of the more promising directions for Bitcoin. Now, with the advent of so called Bitcoin 2.0 technologies such as Ethereum, the problem of smart contracts within the Bitcoin ecosystem gains new importance. This report studies possible smart contract applications for Bitcoin and outlines directions for further research.

Version History

Version	Date	Change description
1.0	Aug 13, 2015	Initial version
1.1	Sep 04, 2015	Added a copyright notice and a watermark

Smart contracts are one of the more promising directions for cryptocurrencies and Bitcoin in particular. A smart contract is understood as a computer protocol used to facilitate and automate financial contracts [1]. The term smart contract was introduced by Nick Szabo in 1990s [2] and became more relevant than ever before in 2010s after digital currencies gained popularity.

One of the advantages in using Bitcoin as a medium for smart contracts is the inherent low trust approach. Built-in Bitcoin mechanisms let users minimize counterparty risks by utilizing mathematical and algorithmic tools, not by relying on a mediator's authority, as it is often the case in the traditional approach.

Bitcoin is criticized because of the insufficient expressive means for smart contracts. The Ethereum project [3] launched on July 30th, 2015 was developed as a replacement of the Bitcoin protocol specifically targeted for smart contract users. Because of the critique, the following questions arise:

- What possibilities and perspectives does the Bitcoin protocol have as a medium for smart contracts?
- What are the main drawbacks of Bitcoin compared to the alternatives (e.g., Ethereum)? Can these drawbacks be eliminated or alleviated while staying within the protocol?

1 Theory

1.1 Scripts

All information in the Bitcoin protocol is stored in the form of a decentralized ledger – the blockchain. The building block of the blockchain is a transaction containing on or more inputs and outputs [4]. Each transaction input is an unspent output (UTXO) of one of previous transactions recorded on the blockchain[1]. Each transaction output is associated with the value – an integer denoting the spendable amount of currency in satoshi (10^{-8} BTC). The sum of all transaction input values (i.e., the values of UTXOs associated with these inputs) equals to the sum of all transaction output values, plus the transaction fee paid to a miner for including the transaction into a block. Therefore, we can view a transaction as redistribution of wealth by "destroying" existing UTXOs and creating new ones.

Each UTXO has to specify a person (or several persons) eligible for spending wealth associated with it. To accomplish this, the Bitcoin protocol introduces a scripting language [5]. The language describes the execution of a certain program on a stack machine [6], therefore resembling Forth progamming language [7]. The capabilities of the scripting language are severely limited compared to general purpose PLs; for example, it lacks cycles, I/O, and persistent states. These restrictions were introduced to protect the Bitcoin network from DoS-attacks and get acceptable timings for transaction and block verification.

Each transaction output contains a script which locks, or encumbers, the money associated with the UTXO. This script is commonly referred to as **scriptPubKey**. To spend this money, a user of the

[1]Except for coinbase transaction inputs which we won't consider in the report as they are not so interesting from the point of view of smart contracts.

Bitcoin network must demonstrate the proof of ownership in the form of an unlocking script **scriptSig**. Hence, a script is a part of each input and each output of a transaction.

Verification of the pair (**scriptPubKey**, **scriptSig**) is performed in several steps:

1. Execute the unlocking script **scriptSig**; remember the stack state after the execution.
2. Not changing the stack, execute the locking script **scriptPubKey**.
3. If after executing both scripts, the stack isn't empty and the top value does not evaluate to false (i.e., it contains one or more 1-bits), the verification is successful.

Several scripting language instructions may determine the verification state immediately upon their execution (e.g., **RETURN** and **VERIFY**).

Example. For the most common transaction type (pay-to-pubKeyHash), the unlocking script **scriptSig** consists of pushing two numbers into the stack:

- the digital signature of the spending transaction
- the public key of the user.

The locking script **scriptPubKey** performs the following steps:

1. Check that the public key upon hashing is equal to the address [8] of the UTXO. The address in the binary form is a part of the locking script.
2. Check that the public key corresponds to the digital signature and the transaction.

If both conditions hold, a specific transaction input is correct. If all transaction inputs are correct, the transaction is correct in whole and can be propagated or included in a block.

Example. Arbitrary data can be stored in the Bitcoin blockchain using the following form of the locking script:

1. **RETURN** instruction
2. data

RETURN makes the validation process fail immediately. This means all money for UTXOs encumbered with the specified form of the locking script is lost forever; the only useful function of the UTXO is storing the data in the script.

1.2 Transaction Versions

Two Bitcoin features helping implement smart contracts are *transaction lock time* and *input versions* [4]. Both these fields are used to determine whether the transaction in question is *final*, i.e. whether or not it can be included in blocks. Currently, non-final transactions are not propagated across the network; nevertheless, they are useful for creating and updating smart contracts.

Transaction lock time is an unsigned 4-byte integer that specifies either the minimal timestamp or the minimal block height when the transaction is considered final. “Ordinary” transactions that just move the value around have this field set to zero; for these transactions the value of both lock time and input versions are ignored. If the lock time is a non-zero value not exceeding 500 million, it is treated as block height; otherwise, it is considered as a UNIX timestamp.

Example. A transaction with lock time set to 400,000 cannot be included in any block with the height less than 400,000. Note that Bitcoin clients currently forbid the propagation of such a transaction across the network until the 400,000th block is actually mined; however, it can be safely propagated after the lock time condition is met.

Each transaction input also has a 4-byte unsigned integer value associated with it, the input version. The versions are useful in constructing transactions from several sources, as they allow signing inputs independently (see the explanation of signatures below). The input versions are active only when the containing transaction specifies a non-zero lock time. In theory, the transaction with the higher version of inputs should replace the transaction with the lower version in the transaction memory pool. However, the corresponding functionality was removed from the reference Bitcoin client implementation, and as of now, version fields are mostly useless.

1.3 Signatures

Bitcoin protocol allows several kinds of user signatures. The core functionality of a signature is as follows:

Problem. Prove that the user possessing the private key corresponding to the given public key **pubKey**, has authorized the transaction **tx** to spend their money.

To solve this problem, the Bitcoin protocol uses elliptic curve digital signatures [9]. Each signature is computed based on the hash of the transaction **tx**. This makes it impossible for third parties to alter the transaction after it entered the Bitcoin network while maintaining the validity of the signature, e.g. in a MitM attack [10].

The parts of the transaction used to produce the hash used in digital signatures depend on the type of the signature. This type is specified as an additional byte at the end of the signature [11]. There are three types of signatures:

- **SIGHASH_ALL** (default type). All of the transaction content is used to get the hash, except for unlocking scripts (if these scripts were included, signature embedded in one of the inputs would invalidate signatures in all other unlocking scripts), but including all input versions, references to UTXOs, and specifications of all outputs. This type of signature can be expressed as: “The user agrees to put their money in, as long as the others put their money in the specified way, and the money is spent as agreed”.

- **SIGHASH_NONE.** The tx hash is based on all transaction inputs, and ignores input versions and outputs. This type of signature can be expressed as: “The user agrees to put their money in, as long as the others put their money in the specified way; the user does not care how the money is spent”.
- **SIGHASH_SINGLE.** The tx hash uses a single output, the index of which is the same as the index of the current input. This type of signature can be expressed as: “The user agrees to put their money in, as long as the others put their money in the specified way; the user only cares about their part of the money”.

Additionally, the **ANYONECANPAY** modifier can be used together with any of these three methods. The modifier means that the tx hash ignores all transaction inputs except the current one. It can be expressed as, “The user does not care where the money came from; he or she only cares about all / none / one of the outputs” (depending on the base signature type).

Example. **ANYONECANPAY** together with **SIGHASH_ALL** can be used to implement charity or crowdfunding. A user that wants to participate creates a UTXO signed with the amount of bitcoin he is willing to donate (e.g., 1 BTC). Then, he creates a transaction with a **SIGHASH_ALL** + **ANYONECANPAY** signature with the output set to the address of the crowdfunding / charity service and the amount of payment equal to the goal of the campaign (e.g., 1,000 BTC). As the output of the transaction is higher than its input, it cannot be propagated across the Bitcoin network; the user sends it directly to the charity / crowdfunding service. The service aggregates all transactions from the funders into a single transaction with a large number of inputs (with sum of funds equal to 1,000 BTC) and a single output. Because of the **ANYONECANPAY** flag, all signatures in the unlocking scripts remain valid in the aggregated transaction. Therefore, the transaction is valid and can be propagated through the network and included in the blockchain.

2 Applications

This section details some possible applications of smart contracts built on top of the Bitcoin blockchain. Many of these example applications are taken from the Bitcoin wiki page authored by Mike Hearn [12].

One important type of smart contracts included into Bitcoin protocol is M-of-N multisignature transactions; they are implemented as a separate scripting language instruction, **CHECKMULTISIG**. These transactions require a certain amount (M) of digital signatures in the unlocking script. Each of the signatures must correspond to one of N, $N \geqslant M$ public keys embedded into the locking script. There are several ways this type of signatures can be used in smart contracts, some of which are discussed below.

2.1 Escrow

Problem. Alice wants to buy certain goods from Bob using Bitcoin, however, she doesn't trust him enough to simply send money using pay-to-pubKeyHash transaction before she gets the goods; similarly, Bob doesn't trust Alice to send the goods without a guarantee of payment.

Solution

1. Alice and Bob agree on a mediator service (which, for example, may be associated with the online market they use to trade).
2. Alice creates and broadcasts a transaction **Tx1** with 2-of-3 multisignature locking script, with the public keys belonging to her, to Bob, and to the mediator service.
3. After Bob sees Alice's transaction on the blockchain, he is sure that Alice cannot double-spend the money (as it would require her to conspire with the mediator). He therefore can safely ship goods to Alice.

The locked money can be spent by signing a transaction **Tx2** which spends **Tx1**, with the two signatures:

- by Alice and Bob (a successful trade, or Bob agrees to reimburse Alice)
- by Alice and the mediator (a charge-back)
- by Bob and the mediator (Bob gets money despite the dispute).

■

2.2 Multisig Wallets

There exist numerous Bitcoin wallet services that utilize 2-of-2 or 2-of-3 payment schemes. With the 2-of-2 multisignature scheme, each user's UTXO must be signed with both his own signature, and the service's signature. With the 2-of-3 scheme, user's UTXO are normally signed by him and by the service, too. However, in this scheme the wallet user also has a reserve cold-storage private key that can be used to sign transactions instead of the service in case the service disappears or becomes temporarily unavailable.

The signature of a wallet service is somewhat a seal of approval: at least several services utilize machine learning algorithms similar to credit scoring to detect suspicious transactions. It means the transactions signed by the service are likely harmless and are accepted by merchants faster than usual transactions.

2.3 Deposit

Problem. Alice wants to prove her trustworthiness with the operator of a certain web site (for example, a forum or a wiki), Bob. The site offers a solution in form of a temporary Bitcoin deposit. Alice wants to be sure that the site won't steal her money.

Solution

1. Alice obtains website's public key (for safety reasons, Bob may want to generate a new key pair for each potential contract) and creates a transaction **Tx1** locked with a 2-of-2 multisignature locking script that requires both her and website signatures. The amount of money locked in the transaction is the amount Bob is content with (e.g., 0.1 BTC).
2. Alice sends **Tx1** to Bob. For maximum security, Alice can send only the hash of the transaction if there is a considerable risk that Bob publishes **Tx1** and refuses to cooperate further (in this case, Alice has no means to return her money). The hash of **Tx1** is enough to construct the contract transaction.
3. Bob creates a *contract transaction* **Tx2** which spends the UTXO of **Tx1** by sending it back to Alice. The lock time of **Tx2** is set to some date the future (e.g., $+6$ months). Bob then signs the input of **Tx2** with his signature and sends this incomplete transaction back to Alice.
4. Alice checks the contract, completes it by signing it with her signature, and broadcasts **Tx1**. **Tx2** can be broadcast when the lock time is expired.

If the parties agree to terminate the contract prematurely or prolong it, they only have to change the lock time parameter of **Tx2** and sign it again.

■

2.4 Micropayment Channels

Problem. Alice wants to pay Bob's web-based service with Bitcoin micropayments (say, Bob owns an Internet radio or TV where the use of the service is paid for each second). Alice knows that Bitcoin is a good medium for micropayments as it is a highly divisible currency; however, the high volume of micropayment transactions is bad for Bitcoin ecosystem, so Alice would prefer a solution where individual transactions are off-chain (i.e., not stored on the blockchain).

Solution

1. Alice creates a new public key and requests a public key from Bob.
2. Alice creates a new transaction **Tx1** with an amount of money enough for a typical user session (e.g., 0.1 BTC) and sets the locking script to the 2-of-2 multisignature verification. She uses both her and service's public keys to lock the money.
3. Alice creates a refund transaction **Tx2** that spends money from UTXO of **Tx1** by sending it all back to herself. The lock time of **Tx2** is set to the mutually agreed time in the future (e.g., $+1$ day). Alice signs **Tx2** and sends this incomplete transaction to Bob.
4. Bob signs **Tx2** making it a complete transaction and sends it back to Alice.
5. Alice broadcasts **Tx1** and starts using the service.

6. From time to time, Alice creates a micropayment transaction **Tx3** which spends UTXO of **Tx1** by sending a part of money to Bob and the rest to herself. She signs this transaction with her signature and sends it to Bob. Bob checks if Alice's signature is valid and the amount of money paid for the service is right. If it is so, Bob continues to provide the service to Alice. Note that **Tx3** is broadcast by neither Alice (she has an incomplete version that lacks Bob's signature) nor Bob (he can make **Tx3** complete, but has no reason to do that as updated versions of the transaction pay him more money).
7. When Alice finishes using the service, Bob signs the latest version of **Tx3** and broadcasts it through the Bitcoin network.

■

Note that in most cases updates of the micropayment transaction are performed automatically by a client application that provides the service.

2.5 Oracles

By design, Bitcoin scripts cannot access data from outside sources. This problem can be overcome using oracle services.

Problem. Alice and Bob want to have a bet: Alice is willing to wage 10 BTC on the fact 1 Bitcoin is worth $500 by the end of 2015; Bob is more pessimistic. The two want to use an independent service (an oracle) they both trust in order not to forget about the bet and conduct it in an orderly manner.

Solution

1. The oracle service creates a new public key and obtains the public keys from Alice and Bob. Using these keys, the oracle creates a 2-of-3 multisig address which is sent to the parties.
2. Alice creates a new transaction **Tx1** that sends the desired amount of money to the multisig address and broadcasts it across the Bitcoin network.
3. The oracle verifies **Tx1** and creates three transactions spending the UTXO of **Tx1**:
   - **TxA** sends the money back to Alice and corresponds to her winning the bet
   - **TxB** sends the money to Bob and corresponds to him winning the bet
   - **TxR** is a refund transaction that sends money back to Alice; unlike **TxA**, it has a non-zero lock time corresponding to a certain moment in the future.
4. The service sends **TxA**, **TxB** and **TxR** to Alice and Bob. **TxA** and **TxB** are sent unsigned; one of these transactions can be executed without the help of the service if both Alice and Bob agree on it. **TxR** is signed with the oracle's signature which means it can become valid as soon as the lock time expires.

5. Upon receiving transactions from the oracle, Alice signs **TxA** and **TxB** with her signatures and returns them to the oracle. Now, the oracle can sign and broadcast either **TxA** or **TxB** (but not both) as soon as the state of the bet becomes definite.

■

The described process is highly flexible:

- The condition may be checked once on the specified time (December 31, 2015 23:59:59 in the example above) or periodically (for example, each day or each minute).
- The condition may express any information about the outside world accessible to the oracle. For example, the condition may specify a bet, or an assurance contract, or even a will.
- The contract may involve more than one oracle, or more than two participants.
- To monetize the service, the oracle can include a small payment to itself into the contract transactions (**TxA** and **TxB** in the above example).

2.6 Smart Property

(This example is taken from the Bitcoin Wiki page, authored by Mike Hearn [13]. It is a bit hypothetical given that no cars or other goods currently have embedded Bitcoin chips but the technology itself is within the realm of possibility.)

Problem. Alice wants to buy a car from Bob using smart contracts on the Bitcoin blockchain.

Solution

This example assumes that Bob's car has an associated ownership key – an ordinary elliptic cryptography key pair used in the Bitcoin protocol. A small amount of money (say, 0.001 BTC) is deposited on the address determined by the public part of the ownership key.

1. Bob creates a key pair and tells Alice the public key and the car price.
2. Alice generates a new ownership key for the car.
3. Alice creates a contract transaction with two inputs and two outputs. The inputs are her payment for the car, and the input spending the UTXO associated with Bob's ownership key. The outputs go to the address given to Alice by Bob, and to the output encumbered by Alice's ownership key. As Alice doesn't have the private part of Bob's ownership key, she only signs the first input of the transaction and sends it to Bob.
4. Bob checks the contract transaction, signs the second input and broadcasts the complete transaction.
5. Alice and Bob wait for several confirmations.

6. Alice presents the car the contract transaction. The embedded software installed in the car is able to verify that the transaction is valid, has the required number of confirmations, and has the special form that allows to treat as a reassignment of ownership.

■

There are several other smart property applications:

- Instead of an ordinary key pair, ownership can be denoted with a multisignature key; for example, Alice can allow her husband Chad to drive her car.
- A slightly more complicated contract can be used to lease property or to use it as collateral in money loans.

3 Proposed Upgrades to Bitcoin Protocol

There are several proposed upgrades to the Bitcoin protocol aimed to ease the implementation of smart contracts.

3.1 Removing Transaction Malleability

Currently, Bitcoin transactions are malleable: signatures do not cover all the data in a transaction that is used to create the transaction hash [14]. One of the uncovered fragments of the transaction is unlocking scripts. Malleability means it is possible for a third party to change a transaction sent over the Bitcoin network in a way that its hash is invalidated. While the outputs of the transaction are safe from the attack, it is unsafe to accept a chain of unconfirmed transactions because the later transactions depend on the hashes of previous transactions, and those can be changed until the transaction is included in a block. This behavior complicates the organization of payment channels (Section 2.4).

One of the proposed improvements to deal with transaction malleability is BIP 62 [15]. Note that the complete removal of malleability is not feasible (obviously, a transaction signature cannot cover the signature itself); the imposed constraints on unlocking scripts may be harmful for development of smart contracts.

3.2 CHECKLOCKTIMEVERIFY

CHECKLOCKTIMEVERIFY is a proposed additional instruction for the Bitcoin scripting language [16]. The goal of the instruction is to prove algorithmically that a certain UTXO cannot be spent until some time in the future. **CHECKLOCKTIMEVERIFY** operates as follows:

1. Take the argument of the operation from the top of the stack and treat it as an unsigned integer.
2. Ensure that the argument and the lock time of the transaction containing the unlocking script are comparable, that is, they both describe either block heights or timestamps. If this is not true, fail immediately.

3. Compare the argument to the lock time of the transaction. If the argument is greater than the lock time, terminate transaction verification with the fail status.

As transactions with the lock time set in the future cannot be included in a block, **CHECKLOCKTIMEVERIFY** can be used to verify indirectly that the desired block height or block time has been reached; until this is the case, the UTXO encumbered with a script that includes the **CHECKLOCKTIMEVERIFY** operation remains unspendable.

3.2.1 Example: Escrow

Problem. The same as in Section 2.1. Additionally, the mediator would like a more safe access to the funds to discourage third parties from attempting to get mediator's secret key, and to assure Alice and Bob that the mediator won't conspire with either of them.

Solution

Instead of using a standard 2-of-3 multisig locking script, Alice (the customer) creates a transaction with a locking script which includes two branches of execution using a conditional operation (**IF**):

- The money can be spent with a transaction signed by the mediator service and either Alice or Bob (the merchant), but only after some time passes. This clause corresponds to a charge-back or a forced payment.
- The money can be spent with a transaction signed by both Alice and Bob.

Note that according to the contract, mediator's signature only becomes useful after the specified moment in the future. Thus, there is no risk that the mediator conspires with either of the parties to steal the funds.

■

3.2.2 Example: Micropayment Channels

Problem. The same as in Section 2.4.

Solution

CHECKLOCKTIMEVERIFY instruction makes it possible to embed the refund clause directly into the contract transaction. Just like with the escrow, the customer of the service creates transaction with a locking script which includes two branches of execution. One of these branches returns all money back to the customer and is time-locked; another is a 2-of-2 multisignature verification.

■

3.2.3 Example: Freezing Funds

Problem. Alice wants to freeze her funds for a certain period of time. She could use existing solutions such as a cold storage, but wants to ensure that the money are stored on the blockchain.

Solution

Alice creates a transaction with the locking script that consists of the two parts:

1. **CHECKLOCKTIMEVERIFY** instruction that prevents the UTXO from being spent before the specified time
2. ordinary set of instructions to prove ownership (for example, pay-to-pubKeyHash instructions).

■

3.3 Transaction Replacement Mediated via Sequence Numbers

As it was mentioned in Section 1.2, sequence numbers of transaction inputs remain mostly useless as of now. The original intention was to replace transactions in the memory pool based on their sequence numbers but this behavior is hard to enforce, especially if a transaction with lower sequence number is more profitable for a miner than the newer transaction. As of now, transactions with non-default sequence numbers (differing from $2^{32} - 1$, the maximal possible value) are ignored by the reference Bitcoin client: not only are they not included in blocks, these transactions are not propagated as well.

Therefore, an improvement BIP 68 [17] has been proposed to change the semantics of input sequence numbers. The modification requires a soft fork and goes as follows:

- A transaction with default sequence numbers can be included into any block.
- A transaction with a non-default sequence number n can be included $(2^{32}-1-n)$ blocks after the transaction containing a UTXO the corresponding input is spending. For example, a transaction with a single input having sequence number $2^{32}-2$ (one less than the maximum) can be included into any block after the block with transaction **TxR** referenced by the input, but not into the same block as **TxR**.
- Alternatively, if a sequence number of a certain transaction is less than $2^{32} - 500 \cdot 10^6$, it denotes a minimal required time span between the block containing the transaction, and the block containing the transaction referenced by its input.

In essence, the proposed change is similar to the lock time field of the transaction: it acts as a per-input relative lock time.

3.3.1 Example: Bidirectional Payment Channel

Problem. Alice and Bob want to have an off-chain payment channel with money going in both directions.

Solution

1. Alice creates a funding transaction **Tx1** that sends the estimated maximal amount of money to a UTXO encumbered with a signature that allows to spend funds by either the consent of both

parties (2-of-2 multisig) or Alice herself after a long timeout, e.g. $T = 30$ days. This can be accomplished using a **CHECKLOCKTIMEVERIFY** instruction as described in Section 3.2; alternatively, Alice may use a separate refund transaction.

2. If Alice wants to send money to Bob, she constructs a transaction with two outputs (one to Bob, other to herself) spending **Tx1**. She sets the relative lock time of the transaction to a period lesser than T (e.g., 29 days), signs the transaction and sends it to Bob.
3. Alice can increase the amount of money she sends to Bob by sending him new incomplete transactions.
4. If Bob then wants to send money back to Alice, he constructs a transaction with two outputs (one to Alice, other to himself) spending **Tx1**. The sequence number of Bob's transaction must be set to the time lesser than the relative lock time of Alice's transactions (e.g., 28 days). This is because Bob has in his possession a more favorable transaction sent to him before by Alice; with the lesser lock time, Alice has a full day to broadcast the updated transaction before the next transaction matures.
5. The lock time is decreased each time the payment channel switches direction until the relative lock time becomes zero, or both parties agree to close the channel. The last transaction has a sequence number set to the default value $2^{32} - 1$ and is signed by both parties.

■

4 Existing Smart Contract Solutions

4.1 Wallet Services

As Vitalik Buterin put it in the article [18], in the next few years all Bitcoin wallet services will transition to 2-of-3 multisignature scheme described in Section 2. Compared to pay-to-pubKeyHash, this scheme offers additional security both for users of a wallet and merchants accepting their payments. Indeed, the multisig scheme becomes more and more popular and is currently used by the following services:

- **Armory Wallet** (bitcoinarmory.com)
- **BitGo Wallet** (bitgo.com)
- **Copay** (copay.io). Allows creating arbitrary multisignature configurations
- **Cubits Wallet** (cubits.com). The multisignature scheme is also used to transfer money from user's cold wallet to their hot wallet
- **Gem Wallet** (gem.co)
- **MyMoneyEx** (mymoneyex.com). Also supports 2-of-2 multisignature scheme
- **Ninki** (ninkip2p.com)
- **GreenAddress** (greenaddress.it). Uses 2-of-2 multisignature wallets

4.2 Fraud Detection

Related to multisignature wallets are fraud detection services. These services sign user's transactions with a 2-of-3 multisignature locking script. Two of the keys belong to the user (one is an ordinary wallet key, and the other is a physical recovery key). The third key belongs to the service and is used to sign transactions depending on their risk score:

- transactions with **low risk** are signed immediately
- **medium risk** transactions are delayed and require a confirmation by e-mail or phone
- **high risk** transactions such as an attempt to empty the wallet require several kinds of confirmation and are substantially delayed or not signed at all.

Some of the services implementing this kind of fraud prevention are:

- **CryptoCorp** (cryptocorp.co)
- **Bitrated** (bitrated.com), coupled with the web of trust.

4.3 Oracle Services

Oracle services implement reading data from off-chain sources as described in Section 2.5. Two commonly used types of data read by oracles are

- **Boolean switch** with the value depending on whether a certain web page specified by the parties contains a specific text. For example, this type of data can be used to organize bets.
- **Floating-point value** equal to the exchange rate of Bitcoin or other currency, usually read from the API of some Bitcoin exchange. This type of data is useful for organizing option and hedging contracts.

Bitcoin oracle services include:

- **Hedgy** (hedgy.co), an oracle service enabling hedging contracts stored on the Bitcoin blockchain.
- **Early Temple** (earlytemple.com), an oracle service using web page-based contracts.
- **Orisi** (orisi.org), a network for distributed oracles that communicate using BitMessage protocol. Orisi system supports multiple oracles for the same contract for increased safety.
- **BTCOracle** (btcoracle.com), an oracle service for Bitcoin options.
- **RealityKeys** (realitykeys.com), an oracle service using exchange rates and other APIs to get data. Unlike other services, RealityKeys does not create transactions on the Bitcoin blockchain; instead, the service creates two pairs of keys, one for each possible outcome of the contract. Public keys are available immediately and as such can be used to draft contract transactions. The private key corresponding to the real outcome of the contract is released on a date specified by the parties and can be used to sign the contract transaction. The other private key is discarded.

4.4 Off-Chain Payment Channels

There are several existing prototypes or working solutions of off-chain payment channels:

- Unidirectional payment channels are included into **bitcoinj** (bitcoinj.github.io), a library for working with the Bitcoin protocol implemented in Java [19].
- **Lightning Network** (lightning.network), a concept of bidirectional payment channels [20]. Using some soft forks to the Bitcoin protocol (see Section 3), the network may be deployed in the observable future [21].
- **Impulse** (impulse.is), a concept of instant payment channels secured by the Bitcoin blockchain.
- Payment channels is one of possible applications of **Sidechain Elements** by Blockstream (blockstream.com).

4.5 Generic Smart Contracts

- **SmartContract** (smartcontract.com), a service offering several types of Bitcoin-based smart contracts:
  - conditional escrow released once the contract's underlying data proves actual performance
  - stabilized payments binding together USD and BTC payment options
  - verified payments for performance
  - smart property contracts (not standardized, created manually)
  - location-based contracts using GPS tracking (not standardized, created manually)
- **Mirror** (mirror.co), a peer-to-peer platform for Bitcoin smart contract trading
- **Counterparty** (counterparty.io), a platform that introduces custom tokens on top of Bitcoin blockchain. Tokens can be used for smart contracts with the Counterparty protocol itself acting as a mediator.

5 Competitors

Despite the rise of popularity of digital currencies in 2010s, smart contracts remain a relatively unexplored area of currency application. There is only one major competitor to smart contracts based on the Bitcoin blockchain, **Ethereum** (ethereum.org). **Counterparty** developers have ported Ethereum functionality to their platform to enable smart contracts based on the Bitcoin blockchain; as of July 2015, the project is in beta. Another promising project with similar goals, **Codius** (codius.org), was discontinued recently [22].

5.1 Ethereum

Ethereum is a protocol for hosting decentralized applications. The service was designed with smart contracts in mind and addresses several drawbacks of the Bitcoin protocol:

- Similarly to Bitcoin, data in Ethereum is stored in a distributed blockchain secured with cryptographic proof-of-work algorithm. Unlike Bitcoin, proof-of-work used in Ethereum is tuned to have lesser block generation time (approximately 10 seconds), so transactions are confirmed much faster.
- Like Bitcoin, Ethereum utilizes a scripting language to implement smart contracts. Unlike Bitcoin, the Ethereum scripting language is Turing-complete (i.e., it can perform any computations that can be performed by a general purpose PL).
- Smart contracts in Ethereum are independent actors with a certain address. Each smart contract has associated scripts that allow it to process incoming transactions. Thus, transactions in Ethereum have no predefined semantics compared with Bitcoin where all transactions transfer value; the semantics of a transaction is defined by its destination.
- The Ethereum blockchain stores not only transactions, but also system states. The Ethereum scripting language has special instructions to read and write data from / to the blockchain.

These features of the Ethereum protocol can potentially make it more appealing for implementing smart contracts than Bitcoin. However, Ethereum currently suffers from several drawbacks:

- The project is still at its early stages (it was only launched on July 30, 2015), which means it likely has undiscovered vulnerabilities and design errors. Ethereum is equally likely to undergo serious design changes, judging by the pre-launch development phase.
- Comparatively little hashing power dedicated to securing the Ethereum blockchain makes it vulnerable to attacks. In the aspect of security, the Bitcoin blockchain is more appealing and will retain this status in the observable future.
- Versatility of the Ethereum protocol makes it complicated compared to Bitcoin. For example, while data can be stored in the Ethereum blockchain, operations writing big chunks of data are (quite probably, prohibitively) expensive, so the oracle-type contracts are likely to store data off-chain anyway.
- As a research paper [23] suggests, the Ethereum protocol may be flawed by design: it discourages miners from both including computationally heavy transactions in blocks and performing a complete verification of all incoming blocks.
- Ethereum suffers from suboptimal management. For example, details of the proposed transition from proof of work to a proof-of-stake block discovery algorithm are not known as of yet. The date of this transition and several other key events in the Ethereum lifecycle are not defined rigorously, but rather manually determined by the project team.

5.2 Counterparty

Since 2014, Counterparty includes a smart contract environment based on Ethereum virtual machine and providing virtually the same capabilities [24]; currently, it is only available for testing purposes. The environment makes it possible to run Ethereum smart contracts on top of the Bitcoin blockchain while enjoying benefits of Counterparty platform like management of user-defined digital assets. Smart contracts stored on Counterparty will be more secure and will use a more application-rich environment compared to Ethereum. On the other hand, it is unclear how well smart contracts (especially those that act upon information from off-chain sources) will deal with long confirmation times on the Bitcoin blockchain. There may be more efficient ways of integrating Bitcoin and Ethereum, e.g. using Ethereum as a pegged sidechain [25].

5.3 Codius

Codius is a platform for hosting decentralized applications including smart contracts. The platform was developed by RippleLabs for nearly one year before it was closed in June 2015 because of the lack of demand. Similary to Ripple Network, Codius wasn't really decentralized: it would use several trusted hosting providers chosen by RippleLabs to run decentralized applications. This would make the platform cheaper in terms of both computing power and electricity consumption.

Smart contracts is one of the aspects of Codius. Unlike Bitcoin or Ethereum, contracts are implemented using general purpose programming languages with no special requirements on the execution environment; contracts can hold and manipulate assets in one or more digital currencies such as Bitcoin or Ripple. On the one hand, this makes smart contracts in Codius more flexible; on the other hand, unlike Bitcoin contracts, they are not inherently enforcible. For example, the refund clause in a Bitcoin contract is transparent to all parties. It follows from the properties of the Bitcoin protocol itself that

- the refund is executable
- the refund cannot be executed earlier than specified
- the refund can be executed instead of the main contract and not together with it.

This is not the case with Codius or any other system relying on a general purpose computing environment.

6 Conclusion

Smart contracts are a promising area of development in the Bitcoin ecosystem. The scripting language included into the Bitcoin protocol is a powerful tool for constructing various types of smart contracts, such as escrow and payment channels. The main advantages of the Bitcoin blockchain as a medium for smart contracts are

- low trust being factored into the system, and
- security coming from mathematical laws and not mediator's authority.

Despite their advantages, smart contracts remain a relatively unexplored Bitcoin domain. However, the interest in smart contract applications steadily rises since 2014 due to the appearance of Bitcoin-like technologies, such as Ethereum, deigned specifically to host such applications. The other motivation for developing smart contracts is the realization that the Bitcoin blockchain cannot host all communication between parties; instead, it should be used to secure accomplished deals.

The most promising categories of smart contracts on the Bitcoin blockchain are

- **Multisignature verification.** The technology has a multitude of applications including securing user's deposits within the wallet service, providing fraud detection, and two-party escrow for peer-to-peer markets.
- **(Micro)payment channels.** There are several concepts of uni- and bidirectional off-chain payment channels which can be utilized in a variety of web services, such as Internet radio / TV.
- **Financial tools.** Smart contracts can be used for efficient trading: hedging, options, etc.
- **Smart property.** Embedded Bitcoin chips can be used to implement property with ownership rights secured by the blockchain. Such property will support advanced contracts such as shared ownership, leasing, acting as collateral in loans, etc.

See Table 1 for the summary. Other types of smart contracts (e.g., GPS-tracking), while less explored, can be useful for creating next-generation decentralized applications.

Compared to the main competitor, Ethereum, Bitcoin contracts have the following advantages:

- better security both because of dedicated computer power and more resources spent on fixing bugs and verifying the system
- more stability
- more tools, educational resources, and developers familiar with the technology
- better awareness of the general audience.

On the other hand, Ethereum offers several advantages compared to Bitcoin:

- arbitrary semantics of transactions making them more appropriate for implementing complex contracts
- flexibility of the scripting language allowing for various types of contracts
- reduced transaction confirmation time.

To harness the full power of blockchain-backed contracts, several soft forks should be introduced to the Bitcoin protocol (namely, absolute and relative lock time verification and fixing transaction malleability). Unlike block size increase, these changes are almost universally agreed on, so implementing the changes is unlikely to cause substantial problems.

Table 1: Categories of smart contracts and their potential

Category	Stage	Market properties	Companies / Products
Mutisignature wallets	Fully implemented	Large market size; relatively easy to implement; many competitors	Armory, BitGo, Copay, Cubits, Gem, MyMoneyEx, Ninki, GreenAddress
Multisignature fraud prevention	Fully implemented	Medium market size; hard to implement (requires good knowledge of machine learning and the preexisting database); few competitors	CryptoCorp, Bitrated
Payment channels	Working prototypes	Medium to large market size; optimal implementation requires (non-controversial) changes to Bitcoin protocol	Lightning Network, Impulse, Blockstream
Financial tools	Partially implemented or prototypes	Small to medium market size; medium implementation difficulty; few powerful competitors	Hedgy, Mirror, SmartContract
Oracles	Partially implemented	Small market size; distributed implementation requires multiple trustful oracle services (non-existent for now)	Early Temple, Orisi
Smart property	Concept	Potentially huge market size; hard to implement (requires development of Bitcoin chips and corresponding embedded software and cooperation with product manufacturers)	(none)

References

[1] Smart contract. In: English Wikipedia
URL: `https://en.wikipedia.org/wiki/Smart_contract`

[2] *Nick Szabo* (1998). Formalizing and securing relationships on public networks
URL: `http://szabo.best.vwh.net/formalize.html`

[3] Ethereum Frontier
URL: `https://www.ethereum.org/`

[4] Transaction. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/Transaction`

[5] Script. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/Script`

[6] Stack machine. In: English Wikipedia
URL: `https://en.wikipedia.org/wiki/Stack_machine`

[7] Forth (programming language). In: English Wikipedia
URL: `https://en.wikipedia.org/wiki/Forth_(programming_language)`

[8] Address. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/Address`

[9] ECDSA. In: English Wikipedia
URL: `https://en.wikipedia.org/wiki/ECDSA`

[10] Man-in-the-middle attack. In: English Wikipedia
URL: `https://en.wikipedia.org/wiki/Man-in-the-middle_attack`

[11] OP_CHECKSIG. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/OP_CHECKSIG`

[12] Contract. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/Contract`

[13] Smart property. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/Smart_Property`

[14] Transaction malleability. In: Bitcoin Wiki
URL: `https://en.bitcoin.it/wiki/Transaction_Malleability`

[15] *Pieter Wuille* (2014). Dealing with malleability (BIP 62)
URL: `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`

[16] *Peter Todd* (2014). OP_CHECKLOCKTIMEVERIFY (BIP 65)
URL: `https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki`

[17] *Mark Friedenbach* (2015). Consensus-enforced transaction replacement signalled via sequence numbers (BIP 68)
URL: `https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki`

[18] *Vitalik Buterin* (2014). Bitcoin multisig wallet: the future of Bitcoin. In: Bitcoin Magazine
URL: `https://bitcoinmagazine.com/11108/multisig-future-bitcoin/`

[19] Working with micropayment channels. In: Bitcoinj documentation
URL: `https://bitcoinj.github.io/working-with-micropayments`

[20] *Joseph Poon, Thaddeus Dryja* (2015). The Bitcoin Lighting Network: scalable off-chain instant payments
URL: `http://lightning.network/lightning-network-paper.pdf`

[21] *Alyssa Hertig* (2015). Lightning Network could strike sooner than expected. In: Coin Telegraph
URL: `http://cointelegraph.com/news/114945/lightning-network-could-strike-sooner-than-expected`

[22] *Stefan Thomas* (2015). Codius - one year later. In: Codius Blog
URL: `https://codius.org/blog/codius-one-year-later/`

[23] *Loi Luu, Jason Teutsch, Raghav Kulkarni, Prateek Saxena* (2015). Demystifying incentives in the consensus computer
URL: `http://eprint.iacr.org/2015/702.pdf`

[24] Counterparty recreates Ethereum's smart contract platform on Bitcoin. In: Counterparty News
URL: `http://counterparty.io/news/counterparty-recreates-ethereums-smart-contract-platform-on-bitcoin/`

[25] *Adam Back, Matt Corallo, Luke Dashjr et al.* (2014). Enabling blockchain innovations with pegged sidechains
URL: `https://www.blockstream.com/sidechains.pdf`