Scripting smart contracts
for distributed ledger technology

Pablo Lamela Seijas, Simon Thompson
{pl240,S.J.Thompson}@kent.ac.uk
University of Kent, UK

Darryl McAdams
darryl.mcadams@iohk.io
IOHK

10th February 2017

Abstract

We give an overview of the scripting languages used in existing cryptocurrencies, and in particular we review in some detail the scripting languages of Bitcoin, Nxt and Ethereum, in the context of a high-level overview of Distributed Ledger Technology and cryptocurrencies. We survey different approaches, and give an overview of critiques of existing languages. We also cover technologies that might be used to underpin extensions and innovations in scripting and contracts, including technologies for verification, such as zero knowledge proofs, proof-carrying code and static analysis, as well as approaches to making systems more efficient, e.g. Merkelized Abstract Syntax Trees.

1 Introduction

Distributed Ledger Technology (DLT) has recently attracted attention from individuals and start-ups, as well as commercial organisations and national and international institutions [18]. DLT offers the possibility of securely recording information in a non-centralised, distributed way, and building on this it is possible to construct not only distributed databases, but also to record the results of transactions that have financial value, most notably cryptocurrencies. Various kinds of *scripting* languages have been introduced that allow users to write *smart contracts* that describe these transactions.

In this paper, we give an overview of the scripting languages used in existing cryptocurrencies, and in particular we review the scripting languages of Bitcoin, Nxt and Ethereum, in the context of a high-level overview of Distributed Ledger Technology and cryptocurrencies. We also cover technologies that might be used to underpin extensions and innovations in scripting and contracts, including technologies for verification, such as zero knowledge proofs, proof-carrying code and static analysis, as well as approaches to making systems more efficient, e.g. Merkelized Abstract Syntax Trees. A survey of research issues for Bitcoin and cryptocurrencies in general at the end of 2015 is provided by [7].

In surveying a cross-section of the work in the field, it is striking to see a wide variety of different approaches to scripting.

- Some scripting languages are Turing-incomplete: Bitcoin script has no facilities for looping or recursion, for example; whilst others, such as Ethereum script, are Turing-complete, at least in principle, supporting looping constructs. On the other hand, they lose completeness in practice because of run-time bounds placed on the possible execution time, stack size and so on.

- Bitcoin and Ethereum provide a low-level computation model by means of a virtual machine, into which it is possible to compile a higher-level language, such as Ethereum's Solidity; other systems, including Nxt, provide a higher-level API in a general purpose language (here JavaScript).

- Another approach sees DLT transactions as fitting into a wider business process model, and would propose deriving them by transformation of a language such as BPMN, or finite state machines.

- Finally, it is possible to use virtualization technology to isolate computations, as is done in Hyperledger with Docker containers.

While we have talked about scripting languages as if they were general purpose programming languages, they can also go beyond this in providing particular facilities including randomness, name registration, anonymity, incentive alignment, control of transactionality and so on. Because of this, we see interactions between the language and the general DLT infrastructure on which it is built:
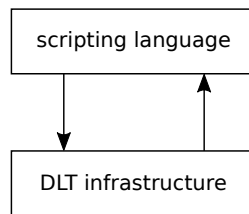
Obviously, the DLT infrastructure pretty much shapes the scripting language, but there can be interactions in the reverse direction too. In particular, exploring the way in which randomness is provided in Ethereum, shows that the typical use of timestamps as a source of randomness can provide a potential attack mechanism. We give an overview of this and other potential security vulnerabilities towards the end of the paper.

In order to give some context, we provide brief introductions and references with information about the different cryptocurrencies and decentralised systems, but the main aim of this work is to overview their scripting languages and related technologies.

We begin by giving introductions to Bitcoin scripting in context in Section 2 and do the same for Ethereum and Nxt in Sections 3 and 4. Other DLT systems are surveyed in Section 5, and other approaches to scripting in Section 6. Section 7 presents some critiques of existing systems, and Section 8 presents a number of technologies that might contribute to the next generation of blockchain scripting. Section 9 concludes the paper.

2 Bitcoin

Bitcoin is the first widely-used implementation of a decentralised cryptocurrency and introduced the decentralised consensus mechanism of a *blockchain*.

Bitcoin's blockchain is a back-linked list of blocks of transactions [2]. Each block within the blockchain is identified by a hash, generated using the SHA256 cryptographic hash algorithm on the header of the block. Each block also references a previous block, known as the parent block, through the "previous block hash" field in the block header. In other words, each block contains the hash of its parent inside its own header. The sequence of hashes linking each block to its parent creates a chain going back all the way to the first block ever created, known as the genesis block.

In essence, it can be considered like a public log that, in the case of Bitcoin, is used to keep track of transactions. Assuming that the miners that represent more than 50% of the computing power that contributes to the creation of new blocks for the blockchain are "honest", and that the hash functions used are irreversible and unpredictable, information in the blockchain becomes statistically harder to revoke as new blocks are added on top of it.

Changing a block implies recreating all the blocks on top of it, since every block (except the genesis block) contains the hash of the previous block. Creating a block is costly because it requires a proof-of-work, which only allows miners to create blocks proportionally to their computing power (as compared to that of the rest of miners); and only the longest chain is considered valid, so it is out of reach for a single agent to replace the main chain, as long as no single agent controls more than 50% of the total computing power.

2.1 Proof-of-work

Proof-of-work is a mechanism that is used to ensure that a distributed consensus is achieved without either the presence of a central authority or the requirement that a set of participating users are identified in some way.

Proof-of-work is implemented as a requirement for each block in the blockchain to have a hash smaller than a given target number. This target number is determined through a calculation that is carried out every 2016 blocks (roughly two weeks), and aims to ensure that, independently of the global amount of computing power, a new block will be generated roughly every 10 minutes in average.

2.2 Guarantees

The design of the Bitcoin system means that it is able to provide a range of guarantees, under an explicitly-articulated set of assumptions. In this section we explain the guarantees, together with the cryptographic mechanisms used to support them.

The algorithm used for hashing the blocks in the blockchain is double SHA256, which consists in applying SHA256 twice to the header of the block. Since the hash function is considered irreversible under reasonable assumptions, the most efficient known way of obtaining a hash that is smaller than the target is by brute force, that is, by repeatedly trying different inputs until one that complies is found. In order to

alter the resulting hash, the input of the hashing function is to be modified by changing a reserved field called a *nonce* (or alternatively other parts of the header). Users that perform the work to create new blocks are called *miners*.

2.2.1 Decentralised agreement

In the case of Bitcoin, agreement on the creation of blocks is achieved thanks to the difficulty of block creation: the computing power necessary to "hijack" the creation of a new, alternative, consensus is highly unlikely to be achieved by a single miner or cartel of miners.

2.2.2 Verifiability

Another guarantee provided by Bitcoin is the possibility for users to unilaterally verify, without trusting anyone else, that a block, and hence the whole blockchain, is valid (that it conforms with the rules). This latter property is possible since every block includes a hash of the previous block, and the first block (or genesis block) is hard-coded in the source code of Bitcoin clients.

2.2.3 Probable irrevocability

On the other hand, the irrevocability of transactions is only guaranteed eventually by the assumption that at least 50% of the Bitcoin mining power comes from honest users. Honest users will accept the longest blockchain as the true one and build upon it, thus, assuming most mining power is honest, it becomes increasingly unlikely that a dishonest agent will be able to create a fork from an old block so that the new fork is longer than the true one. If this were not the case, then a malicious user would be able to revert transactions by creating a fork, but even in this case it would still be impossible to get invalid blocks to be accepted by honest users, thanks to verifiability.

Because the difficulty of block creation is variable, when we talk about the longest chain or blockchain we actually mean the one with the highest combined difficulty, not the one with the highest number of blocks. Otherwise, it would be possible for a malicious user to create an isolated chain to which only him can contribute and thus, the difficulty of this chain would be lower (since it adapts to the total processing power of the miners of the chain). By having a chain with lower difficulty, an attacker would be able to create blocks very fast and potentially obtain a chain with more blocks than the globally accepted chain.

2.3 Mining

There are two incentives for miners. On the one hand, each block gives an amount of fresh bitcoins to the miner that created it. On the other hand, each transaction typically contains a transaction fee (which is specified implicitly as the bitcoins unspent in the transaction), the taxes of all transactions in the block also go to the miner.

In the case of Bitcoin, the prize for creating a block, other than the fees, is reduced by half every 210,000 blocks (roughly every four years), it was originally 50 bitcoins and at the time of writing is 12.5 bitcoins.

2.4 Merkle tree

In order for it to be possible to prove that a transaction exists in the blockchain without transferring the whole blockchain, the transactions are stored in a structure called *Merkle tree*. Such a tree stores transactions at leaf nodes, and inner nodes contain the combined hashes of their immediate subtrees. This makes it possible to prove that a transaction has been hashed without having to rehash all the transactions of a block, only the sequence of hashes called Merkle path, obtained by traversing the tree from the root to the relevant leaf node.

2.5 Transactions

Blocks in the blockchain collect transactions. These transactions transfer bitcoins or *unspent transaction outputs* (UTXOs) between users. But the effective owner of a particular amount of bitcoins is not specified in the transaction; rather, the transaction declares, for each unspent output, a program written in Script (see Section 2.6); whoever wants to spend that output must provide an input to the program that makes it succeed (that is, return zero).

Typically, this program provides a cryptographic challenge that only the owner can solve, for example, providing a signature made with the private key of the owner, but it could be any program. For example, a program can check that several people have signed a transaction, that a subset of a set of people have signed a transaction, or even that a transaction provides the solution to a puzzle.

2.6 Script

The language used for creating scripts in Bitcoin is called "Script". Script is a Forth-like bytecode stack-based language but, unlike Forth, Script is designed purposely so that its execution is guaranteed to terminate. Scripts consist of a sequence of instructions, and these are executed linearly, with no jumps backwards; hence, execution time is bounded above by execution time is bounded above by the length of the script after the instruction pointer. This limitation prevents denial of service attacks on the nodes validating the blocks.

Script provides a set of more than a hundred primitives that allow [46]:

- The addition of constants to the stack.

- Some basic conditional flow control, that is non-looping, but lazy (not requiring the evaluation of both alternatives).

- Stack manipulation (including basic alternative stack access).

- String manipulation (mostly disabled in the standard client implementation [17]).

- Bitwise manipulation (mostly disabled).

- Some basic 32-bit arithmetic with overflow (multiplication and division disabled).

- Some basic cryptographic primitives for hashing and signature verification.

- Two primitives for delaying and expiring uncommitted transactions, with respect to time or the current height of the blockchain.

Some characteristics that are available in Forth but are not available in Script include variables, arrays, functions and loops.

Usually, stack based systems only allow access to the few top items in the stack, which forces programs to recalculate values that may have already been evaluated. In Forth, it is possible to declare variables and arrays, so these allow random access, but these functionalities are not available in Script. Nevertheless, Script provides a pair of instructions that allow random read access to the stack:

- `OP_PICK`: copy an item from arbitrarily deep on the stack to the top.

- `OP_ROLL`: move an item from arbitrarily deep on the stack to the top.

These primitives allow random read access to values that have been calculated previously in the script, but Script does not provide any primitives that allow random write access to either of the stacks.

By using these random access instructions and the alternative stack, it is possible to write a script in Script that efficiently emulates a particular Turing machine performing a finite number of steps bounded by the length of the script. However, it seems it would not be possible to efficiently emulate a RASP machine, and this emulation does not imply that the language is Turing complete: to execute a looping program it is necessary to unroll it the number of times that it loops, and this can only be done for a single specific input, rather than an arbitrary input configuration on the tape.

2.6.1 Limitations

In practice, many limitations have been imposed that avoid the construction of scripts that diverge from the ones considered standard, some of these limitations are:

- A limitation on the size of the blocks (which is set to roughly 1MB at the time of writing [6])

- A limitation on the length of the scripts (which is set to 512 bytes for each element and 10.000 bytes per script at the time of writing [29])

- A limitation on the number of opcodes (which is set to 201 for most op-codes at the time of writing [32])

- Many disabled opcodes [46]

Nevertheless, there does not seem to be any explicit limitation to the size of the stack, which is formally unnecessary because it is restricted implicitly by the limitations to the length of the script.

2.6.2 Pay to script hash (P2SH)

There is an extension to Bitcoin that allows the payment to scripts by providing its hash as destination address. This makes it easier for users to use non-standard scripts since they can transfer funds to them like a normal address, and scripts can be created by a different user. It could potentially be used to split scripts into several transactions and allow more complicated types of contracts but, in practice, there is a limitation that forbids the recursive use of P2SH (it does not work to point a P2SH script to another P2SH script).

3 Ethereum

Ethereum is a blockchain based cryptocurrency system that aims to provide a decentralised general purpose computer. Its underlying currency is called ether. The programs that run on this decentralised computer are usually referred to as smart-contracts and are automatically enforced through the blockchain validation process that is carried out by all full nodes independently.

Full nodes are those that download and validate the whole blockchain, these nodes do not need to trust any other node, since they can validate the whole transaction history. In contrast, because the size of the blockchain is considerable (roughly 89 GB at the time of writing), portable devices often use so called lightweight clients, which only store part of the blockchain and rely on full nodes to validate transactions.

3.1 General structure

Unlike Bitcoin, the Ethereum scripting mechanism allows for looping behaviour through the use of both jumps and recursive calls. If this were the only difference with Bitcoin, a malicious attacker could perform a successful DoS (Denial of Service) attack by sending a transaction that loops forever, since smart-contracts are validated by every node. In order to avoid this problem, Ethereum also introduces a limit to the execution time of each transaction that is called gas.

3.1.1 Gas

Gas is an amount of ether, paid in advance when a transaction is issued, that covers the cost of executing the transaction. If a transaction runs out of gas while it is being executed, the transaction will be rolled-back but the gas consumed will not be not returned.

Because creating transactions requires their creators to specify and allocate the maximum amount of gas that they are willing to pay, the miners have the opportunity of detecting transactions that will take too long to validate without actually having to compute their result.

3.1.2 Contracts

In addition to carrying out computations and transferring ether, it is also possible for transactions to create standalone contracts that are saved in the blockchain and can store ether, data, executable code, can communicate with other contracts and even create new ones in turn.

Basically contracts act as users, with the difference that they cannot initiate transactions (they are reactive). This limitation was imposed in order to avoid DoS attacks against existing contracts. With this design, the gas required to execute the code triggered by a transaction must be initially paid by the user that issued the transaction in the first place. But contracts may programmatically choose to refund legitimate users so, effectively, it is possible to have contracts that users can use for free.

3.1.3 Blockchain

The structure and workings of the blockchain in Ethereum are very similar to those of Bitcoin, but there are two fundamental differences: state information and uncle incentivization.

State information   In Bitcoin, all transaction history is stored in the blockchain, and in order to find whether an amount of bitcoins are unspent, it may be necessary to consult blocks that are located deep in the blockchain. Thus, in order to know whether a transaction is invalid, it is necessary to have a complete copy of the blockchain or to ask someone that does.

In Ethereum, every block contains a snapshot with information about all of the unspent ether, active contracts, and so on. For this reason, it is possible to prune old nodes to store only their headers, this allows for important savings and reduces the need for lightweight nodes.

Uncle incentivization   Validating blocks potentially gives miners a competitive disadvantage since they take longer to start hashing. This, combined with the reduced block generation time of Ethereum (currently about 15 seconds) can lead some miners to skip the validation of some or all transactions.

In order to palliate this problem, Ethereum offers a reward to uncles, that is, blocks that do not make it into the blockchain but are, nevertheless, valid.

3.2   Scripting

The code of Ethereum's smart-contracts is written in bytecode and executed into a virtual machine called EVM. EVM has a fixed word size of 32 bits and is untyped for simplicity.

3.2.1   Storage

Unlike Bitcoin, Ethereum's EVM provides a single stack limited to 1024 elements, but it provides two additional types of storage:

- Temporary storage (memory), which is a byte array and isdeleted at the end of the execution of each transaction.

- Permanent storage, which is a word-indexed key-value dictionary, and is preserved on the blockchain between executions, but which can be deallocated explicitly.

3.2.2 Jump operations

Ethereum's EVM language provides both conditional and unconditional jump operations. In order to allow for easier and efficient implementations of JIT-compilers [16], these jump operations can only target parts of code marked as jump destinations.

3.2.3 Contract operations

Contracts are virtual entities that reside in the blockchain and can store ether and bytecode, can send and receive messages and ether, and create other contracts. Unlike external accounts managed by users, contracts cannot initiate transactions: they are reactive.

Creation New contracts can be directly created by users or by other contracts through the `CREATE` bytecode operation.

Calls Contracts can send messages to other contracts or accounts through the use of the call operations. These allow to send ether, to execute the code of another contract, to stipulate a maximum amount of gas for the code executed by the call (which may be smaller than the gas available at the time of the call), and can pass and receive information.

Contracts in Ethereum have a single block of bytecode that is executed by calls, but high level languages like Solidity automatically define a function selector at the beginning of the block that redirects calls to the appropriate part of the bytecode.

Suicide In order to save storage space, Ethereum allows contracts to delete themselves when they are no longer necessary. In order to promote this, part of the cost of creating contracts is refunded when the suicide operation is called.

3.2.4 Inspection operations

Ethereum's EVM allows contracts to access several kinds of meta information about the blockchain, about the contracts themselves, and even about the code of other contracts, which can be copied to memory.

3.2.5 Logging operations

Another functionality provided by the EVM is logging. There is a set of bytecodes that allow smart-contracts to log values. These logs are returned as the "receipt" that results from processing the transaction, but they are not explicitly stored in the blockchain [51].

3.2.6 High-level languages

In addition to EVM bytecode, several high level languages that are compiled to it exist. The most popular ones are probably:

- Solidity – Solidity is a contract-oriented, high-level language whose syntax is similar to that of JavaScript. Solidity is statically typed, supports inheritance, libraries and complex user-defines types among other features [48].

- Serpent – Serpent, as suggested by its name, is designed to be very similar to Python; it is intended to be maximally clean and simple, combining many of the efficiency benefits of a low-level language with ease-of-use in programming style, and at the same time adding special domain-specific features for contract programming [47].

4 Nxt

Nxt is designed to be a general foundation for DLT-based economic transactions. It is inspired by the success of Bitcoin, but aims to provide more performance and scalability through being based on proof of stake rather than proof of work.

The NRS (Nxt Reference Software) uses a client-server architecture. The NRS server is a Java application with two interfaces: one for communicating with other servers through the Internet (forming a network of nodes), and one for responding to requests from clients through its API (Application Program Interface). The client component of the NRS is a browser-based, user-friendly interface to the NRS server (via the API), often referred to as the Nxt Wallet.

As with other blockchain systems, anyone is able to install and run the software. The system is open source, and, as with other projects, the core developers act as gatekeepers for system changes. Individual users can modify instances of the software, but as with other blockchain systems, changes on a collection of installed instances with more than 50% of the stake would be required to mount on the integrity of the whole system.

It is possible to write JavaScript plugins for the client, but these do not extend the core functionality at all, and facilitate e.g. visualisation or block exploration.

There are proposals for a successor to Nxt, Ardor, and this is expected to be released in the first half of 2017. This will separate the notions of tokens for "forging" and for (user) "transactions"; this is intended to support more efficient operation of the blockchain. The current situation with a single blockchain will become one in which there is both a single "infrastructural" blockchain which takes care of forging tokens as well as the overall consistency and progress of the chain, together with many child chains, which can perform operations in their own currencies.

4.1 Programmability

Crucially, the programmability of the system is provided through a "fat" high level API, which is accessible from Nxt clients through a REST interface. The API provides functionality supporting various kinds of transactions, and the transactions are categorised

into types and subtypes "for modular growth and development of the Nxt protocol". Each type dictates required and optional parameters, as well as the "processing method" of the operation.

The whitepaper explicitly says that "the core software doesn't support any form of scripting language"; rather, users are expected to work with the built in transaction types and transactions that support some 250 primitive operations in a number of areas, including basic payments; an alias system (for strings that can be stored on the block-chain, representing e.g. URIs); messaging (messages can be sent between accounts, but also represent structured data by means of JSON objects); exchanging assets; buying and selling through a digital goods store; infrastructure; phasing operations, etc.[1] Scriptability of the API is given by <script> at the client side, but this does not allow any access to any more fundamental levels of the implementation.[2]

In conclusion, while there is no scripting functionality in the core, it is possible to build functionality by putting together API calls in JavaScript. This will produce a sequence of transactions in the API. In contrast to systems with a lower-level VM, an attack would have to be by means of these API functions, rather than a general VM-code program. The security of the system therefore rests on (open source) implementations of this API: is each of the operations secure, and is there no way of putting together an attack through a sequence of API calls? While the VM allows a broader range of potential scripts, there is a conceptual cost in understanding the scope of possibilities presented by a large and more complex high-level API.

5 Other DLT systems

Since the development of Bitcoin, numerous different alternative cryptocurrencies or alt-coins have been created. These can be classified by their relationship to Bitcoin [50]:

- They may be implemented within Bitcoin, and be called *meta-coins*.

- They may have their own blockchain but be linked to the blockchain of Bitcoin (or other cryptocurrency). These include *side-chains* and blockchains that rely on merged mining.

- They may be inspired by Bitcoin but implemented in a completely independent way.

5.1 Meta-coins

Meta-coins are mechanisms that are implemented on top of Bitcoin, and can be seen as specific ways of using and interpreting Bitcoin. They benefit from the stability of its blockchain, which guarantees their irrevocability, and they will generally have additional rules that interpret the meaning of Bitcoin transactions and may ignore those

---

[1]The API is fully documented here: `http://nxtwiki.org/wiki/The_Nxt_API`

[2]Some details of coding are found here: `https://nxt.org/category/coding/` which gives some examples of <script>s.

that do not make sense (invalid transactions from the point of view of the meta-coin), but they are transparent to other Bitcoin users.

They may store the information as part of Bitcoin transactions (this can be done in several ways [14]), or they may just store a hash of their data and store the actual data somewhere else.

In this section, we review three examples of meta-coins.

5.1.1 Coloured coins

Coloured coins [9] assign extra meaning to bitcoins, even though the idea can be applied to other cryptocurrencies and even to physical currencies. They may be used, for example, as a means to getting a bitcoin UTXO to represent the ownership of a real-world item (e.g: a piece of land, a share of an enterprise, or a ticket to a concert). Of course, coloured coins do not, by themselves, enforce this "extra meaning", it is required that some authoritative organisation or convention recognises them.

5.1.2 Type-coin

Type-coin [13] is a mechanism for general-purpose affine-commitment. It is related to coloured coins (see Section 5.1.1) in that it can be used for representing affine resources similarly to how coloured coins represent assets, but it is more expressive because it can express predicates using affine logic. Type-coin is implemented on top of Bitcoin and its state is interpreted by applying a set of rules to Bitcoin's blockchain.

5.1.3 Counterparty

Counterparty [11, 12] provides the functionality of Ethereum as a meta-coin. The internal currency (XCP) is allocated through "proof of burn" of bitcoins, that is, in order to obtain XCP units users send bitcoin to a special address that is unspendable. Transactions are encoded as normal bitcoin transactions and they are validated by using the rules specified in the source-code of Counterparty's client.

5.2   Sidechains and merged mining

Some systems have their own blockchain but it is linked to Bitcoin's through techniques such as 2way-pegging or merged mining. These techniques aim to enhance Bitcoin by providing new functionalities while reusing some of its strong properties:

2way-pegging allows the direct conversion of bitcoin to an alternative currency and back, by temporarily locking the funds through a script. This works in the same way as "proof of burn" but it can be reversed and, thus, allow the exchange of currency in both directions. This mechanism has not been widely adopted; it has been notably implemented in RootStock and there exist some other experimental projects.

Merged mining aims to reuse the mining power of Bitcoin to add security to a blockchain that is less popular. It works by adding the hash of some or all of the blocks in the blockchain of the alternative cryptocurrency to those of Bitcoin's.

5.2.1 RootStock

Rootstock [42] works as a sidechain of Bitcoin with 2way-pegging, bitcoins can be transferred to Rootstock blockchain and they become "rootcoins" (RTC), and rootcoins can be transferred back into Bitcoin blockchain. Rootstock provides similar functionality to that of Ethereum. In the paper [43], the authors claim that the Roostock VM (RVM) is op-code level compatible with EVM (the Ethereum VM).

The native currency of Rootstock (RTC) is used to pay miners of Rootstock blockchain for executing the contracts. Rootstock also supports merge mining Bitcoin and Rootstock simultaneously, and offers some protection mechanisms, like checkpoints and increased maturity time for mined coins (period during which recently mined coins cannot be used), against DoS by Bitcoin miners.

5.2.2 Namecoin

Namecoin [35] started as a fork of the Bitcoin software and aims at providing a decentralised system to register names. It can be used as alternative for DNS and currently can be used for resolving the `.bit` domains. It charges a small fee for registering names and it requires owners to update them roughly once every 250 days, otherwise they expire. Namecoin has its own blockchain but it is linked to Bitcoin in that it provides the possibility of merged mining with it.

5.3 Tezos

Tezos [22, 21] is a cryptocurrency that, at the time of writing, is under development, but authors provide a detailed specification of its scripting language and an explanation of the system.

Tezos claims to be the first cryptocurrency that is democratically-amendable. It provides an explicit mechanism for deciding on future modifications of its own protocol, and initially it considers a voting mechanism and a trial period.

The scripting language of Tezos is stack-based but includes high-level primitives like lambdas, sets, maps, and some for context specific tasks; and it provides a full specification to aid formal verification of contracts. It also provides functionalities aimed at increasing readability of bytecode like the labelling of elements in the stack and the nesting of primitive expressions. To solve the problem of script-based DoS, Tezos defines a fixed limit for the number of computation steps per program.

Tezos also allows the creation of contracts stored in the blockchain that can store an amount of currency and data (up to 16KB), and are controlled by a user or "manager". Contracts without currency are automatically destroyed.

Tezos' Contract Script Language is the most conservative of the new blockchain languages. It's a stack language, much like Bitcoin Script. However, unlike Bitcoin Script, Tezos' language is statically typed. Additionally, it has a detailed specification, including a formal operational semantics for the stack machine. This makes it easy to construct formal verifications in Coq and related tools. The main drawback of Tezos' language is that, like Bitcoin script and Forth, it's hard to program in. Stack languages are notoriously difficult to use, making it undesirable as user-facing language.

5.4 Hawk

Hawk [30] is a decentralised smart contract system that anonymises transactions, and provides mechanisms that simplify the task of hiding the inputs and participants in smart contracts (see Section 8.3), through the use of cryptographic commitments, zero-knowledge proofs (see Section 8.4), and the use of a third-party or manager.[3]

The work in [30] can be seen to provide a foundation for smart contracts *à la* Ethereum, but with the possibility of making the participants and values in the transactions private, in the style of Zerocash, with *mint* and *pour* operations. It presents an abstraction of what the blockchain can be seen to provide (in Section II-A), and how computation proceeds in it (Section II-C), including *immediate* and *delayed* computations, which are represented via "ticks" on the chain.

The description of contracts is done by using an abstraction that allows for the declaration of private and public parts, and can be compiled to a lower level implementation that uses zero-knowledge proofs or SNARKs for hiding the inputs of the private parts. Transactions are programmed in three phases: *freeze*, *compute* and *finalize*.

The system provides two levels of description for computations: the more abstract *programs* and the more concrete *functionality*, with a wrapper going from the former to the latter. This wrapper can be seen as encapsulating some of the general context, including timers, pseudonyms and ledgers, and these "ideal" programs can be seen in some way as a specification of the lower level ones.

The protocols described in the paper [30] are formally proven secure under the Universal Composability framework, against the abstraction of the blockchain provided.

At the time of writing there is no publicly available implementation of Hawk.

6 Other ideas for scripting

Other approaches to scripting are possible: in particular, it is possible to generate scripts from other artefacts, including business process modelling (BMP) languages, and finite-state machines; we look at these here.

6.1 Compilation from a higher-level language

While programming languages or virtual machines are a suitable programming medium for some, in other domains more specialised, domain-specific languages are used in practice. A key example is that of business process description, and there is work on translating specifications written in such languages into blockchain scripting languages, such as Solidity.

6.1.1 BPMN: The OMG Business Process Model and Notation

BPMN is used for describing real-world business processes, such as a supply chain, which require the collaboration of a number of entities. Tracking and validating such a

---

[3]Note, however, that the manager cannot affect the result of the execution of scripts other than by aborting in the middle of the process, see Section 7.5.

process can be achieved with DLT, using blockchain as both a synchronisation primitive and an immutable audit trail. Systems like these are typically described in a language like BPMN [8]. Recent work [49] shows how BPMN specifications can be translated into smart contracts in Solidity, which form part of a larger runtime system that can monitor and orchestrate the execution of the process, as well as providing other services to the process, such as payment escrow.

6.1.2 Data-Aware Processes

Further to the work in the previous section, another evaluation of the possibilities of using blockchain in this area [26] argues that this sort of 'artefact-based' approach is suitable for data-rich real-world processes, too, and in particular it can be seen to provide possibilities for rich conceptual modelling and verification. In particular, the authors argue that realistic systems involve data, taking their models beyond the purely finite-state machine, requiring machine-assisted proof or automated technology such as model checking. These approaches, which are already used to verify BPM systems, should be extensible to blockchain-based systems.

6.2 Finite State Machines

A DSL for FSMs describing blockchain transactions is discussed in a Nxt forum [19]; it is not clear that this has been taken any further, but may link with the Bamboo proposal in Section 7.2.

6.3 Hyperledger: Dockerized chaincode

The Hyperledger [27] consortium, which involves IBM and the Linux Foundation among other organisations, aims to build general purpose DLT technology, with making many aspects of the system pluggable, including, for example, confidentiality and (even) the consensus mechanism.

Scripting in this model uses *chaincode*, which first had a binding in the Google Go language (and is set to have Java and JavaScript bindings too). This makes the model similar to that of Nxt, but the difference in Hyperledger is that chaincode is isolated into Docker containers, thus providing some guarantees automatically. Each chaincode instance can define persistent state variables that are stored on the blockchain, and are updated when the transaction is invoked; the totality of this storage is called the *world state*.

6.4 Logic rules

There is a history of using a variety of logics to describe contracts, and [28] builds on this to examine the feasibility of using logic to describe smart, blockchain-based, contracts. This is done through exploring a typical example, first in pseudocode and then in Formal Contract Logic (FCL), a deontic, defeasible logic, implemented in the defeasible logic engine SPINdle. Once the approach is established, the paper also examines the trade-offs between performing (parts of) the computations on and off the

blockchain. It concludes by arguing that a case for feasibility has been made, but notes that a substantial challenge to adoption is the relatively inefficiency of logic execution mechanisms.

6.5 Process Calculi

Other languages have emerged, as well. Synereo's Rholang is based on a more formally well-understood system, Process Calculus, and it has a published specification [41]. This makes it potentially amenable to formal verification with tools such as Coq, and to rich typing systems such as linear session types. However, Rholang is also reflective[33], in the sense that data can be represented by processes, which may pose problems for both those options. Additionally, many programmers do not have any familiarity with process calculus, which would add to both learning overhead and also comprehension overhead.

On the other hand, use of communicating processes in Rholang underlines the point (also made by the proposal) that "social" contracts are inherently concurrent artefacts, and so it is desirable that this concurrency is not lost through sequentialisation when modelling them.

7 Critiques of existing systems

When programming, developers usually focus on normal usage; however, in a scenario where money is involved and the programs developed have to interact with potentially malicious agents, it is crucial to understand the unanticipated possibilities that the language affords, including any default behaviours of the programs, and any potentially unspecified aspects.

In this section, we consider a series of observed security problems that follow this pattern.

7.1 Reentrancy

Reentrancy is the correct behaviour of a function or method in the event of it being called in the middle of its own execution. Reentrancy is guaranteed for those functions that are referentially transparent, that is, they do not have side effects or access to global variables.

In cases where a function calls another function over which we do not have control, we must consider the possibility that the remote function will call our function back again, thus we cannot assume that our function call will act atomically [3, 31].

In Ethereum, payments are implemented through a call to an unknown function. Because we usually do not intend the called function to execute any code, we want to limit the gas of the call to a low value. Failing to do so, allows a malicious recipient to execute arbitrary code after the payment.

One example of when this would be a problem is if we have a "withdraw" function that retrieves money from an account and does so by first sending money to the user and then subtracting the amount from the balance. A malicious user could cause the

payment call to execute the "withdraw" function recursively and he would be able to do so until the bank runs out of funds since the balance will not be updated. This security issue has been infamously illustrated by the DAO attack [31].

There are several problems in this example: one is the lack of reentrancy, another one is the unintended execution of arbitrary code.

Nevertheless, we can imagine that isolating the logic from the side-effects would help avoid this kind of problem.

7.2   Bamboo: improving Ethereum-style reentrancy

Yoichi Harai has developed an improved language Bamboo, [23] based on the Ethereum approach to contracts; in particular, his language improves its reentrancy behaviour. He attributes the deficiencies of Ethereum to the absence of message queues to contracts, as implemented in Erlang, for instance.

> "In EVM, when contract A calls contract B, contract B can call back into contract A. At this point, two executions of contract A exist, which do share the storage but do not share the program counter. So the reasoning is almost as complicated as having two threads on the same program. Both EVM and Solidity have this nasty property."

> "In Bamboo the program execution is always at a single point, even in case of reentrancy. The snippet

```
sleep_after_calling(B)
    with reentrancy { ... code_reentrant ... };
... code_continued ...
```

> sends a message to the callee B. Usually, when the callee B returns, the execution continues in `code_continued`. If the callee B calls back our contract, the execution is trapped into `code_reentrant`. This prevents some code far away from changing the state while our contract is calling callee B." [24]

While this approach allows the called code to be more well-behaved, the problem of the callee reentering an already called function remains and, arguably, this is still something to be addressed.

7.3   Implicit runtime exceptions

Exceptions represent behaviour which is not the intended one. Because of their nature, exceptions are good candidates to cause behaviours in programs that were not considered by their developers. If we want to help developers make their code predictable we must study how to help them be aware of all kinds of exceptions that can occur.

Ethereum's EVM handles exceptions by simply returning 0 whenever a call fails [38]. Contract developers are expected to verify return values in the calling function.

In this section, we study some examples of exceptions that have allowed attacks in Ethereum in the past and we suggest some ideas that scripting languages may implement in order to mitigate this kind of problem.

7.3.1 Stack overflow

In Ethereum, the act of sending a message (which can be caused by a function call in Solidity) increases the size of the stack. Because the execution of a function can be the result of another function call, we cannot make assumptions about the amount of stack space remaining, thus, any call can potentially produce an exception. Furthermore, if the exception is not automatically propagated and the programmer forgets to check for errors, the execution will continue as if it was a normal execution, which allows for potential malicious attacks [15].

7.3.2 Out of gas exception

As we saw in Section 3.1.1, the execution effort of contracts in Ethereum is controlled by gas. When you issue a transaction, you may be able to predict the gas that is going to be needed (assuming there are no race conditions). But code that is part of a contract stored in the blockchain may have any amount of gas available when called by an external agent. Thus, execution can potentially suffer an out-of-gas exception at any point.

Ethereum implements a good fallback for this cases, since transactions are rolled back when they run out of gas. But it introduces uncertainty when calling foreign functions which may not succeed if insufficient gas is provided [15]. We cannot know for sure how much gas is going to be necessary for a foreign call to be executed. This suggests that it could be useful to find mechanisms to limit the execution time without using gas, and that it would reduce uncertainty to be able to guarantee preconditions on the execution of transaction (so as to prevent race conditions).

7.3.3 Dealing with exceptions

There are several ideas that a scripting language could implement in order to try to make errors derived from unexpected runtime exceptions more difficult, some of these solutions affect the design of the low level scripting language since high order languages may not be able to enforce atomicity [31]:

- [31] suggests propagating the exceptions through callers and reverting the state, and providing a proper mechanism to control exceptions.

- We may want to have a way of describing the way exceptions will be handled for each scenario. We may want the execution to carry on and ignore failing operations, or we may want to rollback everything if any operation fails, but we probably want to define this explicitly or at least have a fallback that is easy to predict.

- We may want to force the developer to define explicitly the exceptional behaviour whenever there is a chance for it to occur.

7.4 No reimbursement

A possible issue that, while less severe, may be just as frustrating for users, is incomplete handling of preconditions. Again, it is usually the case that developers focus on the normal execution flow, and when preconditions of a contract are not met, even if the execution is aborted, the user may not be returned the money invested in the contract [15, 38]. Furthermore, even if the user checks that the preconditions are met before sending a transaction, it is possible that a race condition changes this fact, thus forcing the user to face a risk when issuing a transaction.

Making transaction execution deterministic [31] can certainly avoid the uncertainty for users, but there is no easy way of enforcing that this kind of errors do not occur. One possible path of research would be to find a way of writing programs that analyses the sum of incomes and outcomes for each user and makes variations to this sum explicit.

7.5 Unilateral abortion

In complex multi-stage contracts, it is possible for users to stop collaborating on the execution of contracts if, for example, doing so would be against their interest [15]. Because of this, contracts must be designed with the possibility of eventual non-cooperation in mind for each step. Usually, non-cooperation can be punished by storing a deposit for each participant that is forfeited if they choose to not cooperate, and by establishing deadlines for each step of the protocol [30].

On the other hand, deadlines necessarily pose a risk for users, that may suffer a DoS that prevents then from cooperating and, thus, loose the deposit even if they are willing to cooperate.

Again, there is no obvious way of preventing this mistakes from happening, but it should be possible to find a way to design contracts that allows to specify deposits and timeouts for users and automatically punishes users when they do not cooperate and, for example, rollback the whole protocol for the rest. Even though any of these solutions would probably require many considerations and calibrations specific to the scenario.

7.6 Unpredictable state

Because transactions are processed in chunks (blocks), the order in which unprocessed transactions will be processed is unknown. The order is decided by the miner that is successful at a certain point in time. Since smart-contracts may depend on context (the state of the blockchain), it is not unusual for their result to be unpredictable [15, 3].

One possible solution to the unpredictability could be to establish a particular context as a precondition for transactions to be executed, or to use the expected output as a precondition [31]. But this could potentially slow down highly interactive contracts and force users to resend transactions many times.

Another consequence is that, if there is any benefit on being the first in creating a transaction that contains a secret, an attacker could intercept a transaction with the secret and make their own transaction by using the secret, and potentially get their transaction to be recorded first in the blockchain. We can imagine this would be an important issue for a system that registers names like Namecoin (see Section 5.2.2)

since attackers could see that a user is trying to register a name and then register it themselves and later charge a ransom to the user for releasing it.

The reasons why attackers may be able to get a copy of a transaction to appear earlier in the blockchain than the original one include: luck, paying higher fees, and controlling enough network or mining power and using it to discard or difficult the propagation of the user's transaction.

But that is not the only consequence of unpredictability, some properties can be directly manipulated by miners without the need of controlling the network, like is the case of timestamps [31] in Ethereum, which are allowed a margin of up to 900 seconds due to the latency and potential lack of synchronisation between the different nodes. Allowing the language to provide this kind of unreliable information can trick developers into a false sense of security which may lead to bugs like using timestamps as seeds for randomness [31], so it may be a wise design decision not to provide them. [31] suggests using the block height number as a replacement and translating the expressions that use timestamps in terms of block height numbers (since the expected time between blocks is known).

The problem of secret stealing can be solved by using cryptographic commitments or zero-knowledge proofs (see Section 8.4).

7.7 Secrecy

Nor Ethereum, nor Bitcoin provide anonymity to users by default, other than the option of using pseudonyms, but that does not usually prevent by itself the use of analysis techniques to analyses data.

In addition, many contracts implementing multi-player games, require that some fields are kept secret for a while: for instance, if a field stores the next move of a player, revealing it to the other players may advantage them in choosing their next move.

In such cases, to ensure that a field remains secret until a certain event occurs, the contract has to exploit suitable cryptographic techniques [3]. Some of these techniques have been mentioned in Sections 5.4 and 8.4.

7.8 Immutable bugs

Immutability can be both seen as desirable and as undesirable depending on the circumstance. Clearly, the fact that data in the blockchain is immutable is a conscious design choice, but it becomes a problem when what is immutable is a bug [3]. The challenge is not really technical but bureaucratic, in that it would be necessary to find a convenient and convincing way of deciding how to decide about the changes, upgrades and fixes of contracts, and in general of the rules in a decentralised crypto-system. We have already seen in Tezos (see Section 5.3) an approach to the later, the default solution provided for fixes in contracts was to have a designated manager for contracts. But we can see that having a manager is a limitation if we want to trust no-one, or maybe we are interested in trusting a scheme of the type N-out-of-M, which would require some script that does the validation.

7.9   Lost ether

Another consequence of immutability is currency sent to addresses whose key is not known [3] due, for example to a mistake. If standard cryptographic assumption hold, this currency is statistically impossible to recover and is lost forever (burned). In addition to the economical loss, this produces a burden on the blockchain because the unspent currency cannot be forgotten, since there is no proof that it is actually unspendable (as far as everyone knows someone could actually have the key).

In order to reduce the likelihood of this happening, in the context of Bitcoin, Base58Check is often used which adds redundancy to addresses, so that a small typo cannot produce a different valid address [2].

7.10   Non-randomness

Some scripts require a safe source of randomness during execution [3]. When the execution environment is a blockchain, the problem translates into finding a source of randomness that is globally available, independently verifiable, and unpredictable.

The blockchain seems to be a good source for such randomness, since the hashes of new blocks are difficult to predict. But using it as source of randomness opens the gate for potential manipulation by the miners. Thus, finding a safe way of using the entropy in the blockchain as a source of randomness is an open and actively researched problem [40].

7.11   Turing completeness or not?

A Turing complete scripting language is capable – in principle at least – of allowing all computable functions to be scripted in the language. This means that the execution of some scripts will not terminate (for some inputs), and even for those that do terminate, execution times can be indefinitely large. So, it would appear that choosing a Turing incomplete language would be appropriate. While this is true, Turing incompleteness is not sufficient in itself.[4] For example, computation times can be super-exponential even for primitive recursion (with higher types), as evidenced by the Ackermann function. Even without recursion, putting *a priori* bounds on execution time can be problematic in scenarios where contracts can invoke each other, and calls can be symbolic.

8   Technologies

8.1   Verifiable computation

Verifiable computation [37] is a technique that allows the generation of proofs of computation – that is proofs that the evaluation of a program has a specified result – that can be verified faster than the time that it would take to do the actual computation. This allows the outsourcing of computation to untrusted parties.

---

[4]This argument is made in the context of Ethereum in the white paper: `https://github.com/ethereum/wiki/wiki/White-Paper#computation-and-turing-completeness`

In Pinoccio [37], authors claim to have achieved a "nearly practical" way of doing this, as opposed to previous ways that, while asymptotically would work for some input, in practice they have an infeasible constant factor (hundreds of years for small computations).

In Geppetto [10], authors describe additional techniques for improving the efficiency of the approach. The work in the paper is for a practical C compiler, based on LLVM, and introduces a number of advanced techniques that aim to reduce power overhead and increase the scope of the work. The Geppetto work is built on top of the Pinocchio system, which also underlies zero knowledge proofs (see Section 8.4).

Verifiable computation has, at least, one interesting application to smart contracts: they would allow miners to do the computations only once and generate a proof of computation, and verifiers would only need to verify that the proof is correct. This would allow for more expensive contracts to be run, since they would not increase the time required to validate the blockchain proportionally.

Potentially, computations could be carried out directly by the users sending the transactions, but this would potentially require transactions to be recomputed in case their context changes (which could happen as a result of a race condition in the process of adding them to the blockchain, see Section 7.6).

8.2   Verifiable bounds and reusable libraries

Even if we cannot give a proof of computation, it would be useful to supply a proven upper bound for the amount of computation that a program will require. This way, we can calculate a minimum amount of gas that would guarantee the contract to execute, which would avoid the out-of-gas exception completely (with the security implications this has, see Section 7.3.2).

Contracts are usually designed and reused many times with small changes to their parameters and inputs. Even if the automatic verification of general contracts is not possible, it may be possible for some contracts to manually prove or provide formulas that verifiably calculate the amount of time required to execute a given contract, or even to prove that execution is correct (as is the case with NP problems, whose solutions can be verified in polynomial time). Such reusable and verifiable formulas can be stored in the blockchain.

8.3   Multiparty computation

Secure multiparty computation (MPC) protocols, allow a group of mutually distrusting parties to compute a joint function f on their private inputs. Typically, the security of such protocols is defined with respect to the ideal model where f is computed by a trusted party [1].

The aspects that can be guaranteed in different settings vary, the basic protocols allow simple emulation of the computation, but Bitcoin has been used successfully [1, 4] to enforce the results of computations and penalise in cases of non-cooperation (see Section 7.5).

8.4 Zero knowledge

Zero knowledge proofs are a cryptographic mechanism of proving the possession of a specific bit of knowledge without revealing that bit of knowledge. It has been shown [20] that it is possible to interactively demonstrate with zero knowledge the possession of a solution for the 3-colouring problem of a graph and by extension any NP-complete problem.

Zerocoin [34] and Zerocash [45] use zero knowledge proofs to add anonymity to bitcoin transactions without adding any trusted parties. The algorithm allows users to mint a number of zero-coins into a pool and to later redeem different ones, but ensures than each user cannot redeem more coins that he or she has minted without revealing any links between the inputs and the outputs.

Zero knowledge proofs are also integrated in Hawk (see Section 5.4)) to ensure anonymity.

8.5 Proof-carrying code / proof of computation

In *proof-carrying code* [36] program code – in the original example, machine code – is accompanied by a proof that the code satisfies certain properties, such as terminating in a given period of time. The proof is defined in such a way that it can be checked by the recipient, and such a check is substantially cheaper than the recipient making the proof for themselves. Moreover, the recipient need not trust the sender: the proof provides the essential evidence that something is the case, rather than having to trust the veracity of another agent in the system. Assuming that the proof format is published, the recipient can indeed write a proof checker for him or herself, and need not trust a third party to supply one.

This does have a major drawback, however: not only will implementors of the blockchain system need to understand and implement a programming language, they will also need to understand and implement a full blown proof checker. Just using proof assistants like Coq is described by programming language implementors as requiring a PhD, nevermind constructing one. Now, while this is not really true, it does convey the complexity of the task. On the other hand, this may be a good thing, as it creates a hurdle for those wanting to develop a validator, and may deter all but the most determined (and skilled), thus avoiding the production of poorly designed validator software that could threaten the blockchain network.

A related concept is *proof of computation*, where a proof accompanies a value: in such a case the value is purported to be the result of evaluating a given expression, and the proof provides evidence that this is the case.

8.6 Use of combinators

Combinators are higher-order functions, which could be in an untyped functional language (like the λ-calculus) or a typed language (such as Haskell). Sets of such functions form combinator libraries, and they have been used to define small *domain specific languages* (DSLs); well-known examples include parsing libraries and hardware description. [39] studies how combinators can be used to represent typical financial contracts.

It aims to define a combinator language that is closer to the terminology that is used in financial contexts.

The model is of contracts with two parties: the holder and the counter-party. Seen as existing over time, with a horizon (expiry date), and potential constraint on the acquisition date. Contracts can be combined through conjunction, disjunction and one after the other (sequentially), as well as being able to "swap polarity" as it were. Shown how complex contracts can be described using the DSL: including "American"- and "European"-style options.

The paper [39] presents, in addition to the combinator library itself, a valuation of contracts, based on stochastic machinery, and a concrete implementation of this valuation machinery.

8.7 Use of polymorphic dependent types and algebraic side-effects

Dependent types allow users to declare more expressive types for their programs. Indeed, through the "Curry-Howard" isomorphism, types can be identified with propositions in predicate logic, and members of those types with proofs of those propositions. For these reasons, an expressive language might be an appropriate vehicle for guaranteeing that smart contracts have certain properties by design. The paper [38] investigates this, and in particular examines how Idris, with its combination of dependent types and algebraic effects, can be used to good effect in this space. In particular, it manages to model handling of both gas and global state in a safer way.

In a little more detail, it proposes using dependent and polymorphic types, and algebraic side-effect declarations to prevent them. The intention is to avoid unexpected outcomes by declaring the expected outcomes and side-effects explicitly. The paper also provides a translation from Idris to Serpent, the Python-style high-level language for Ethereum. Finally, the paper looks at some common problems of contract development, but these are taken directly from [15], discussed in Section 7. In evaluating the work, the paper concludes that process calculi, plus behavioural types, might provide a better solution.

8.8 Formal verification of contracts

"Proving programs correct" has been discussed for more than thirty years, but is coming of age. Some work has been done on applying this to smart contracts, too, with two notable examples of verification of Ethereum contracts.

The first approach [5] uses F*[5], which is an ML-like functional programming language aimed at program verification, to verify Ethereum contracts, written in Solidity. It presents three different ways of attacking the problem.

- First it presents a system Solidity*, written in OCaml, and which translates programs written in Solidity to F*, (constituting a "shallow" embedding of the language). It is then possible to verify properties of the translated functions. This

---

[5]https://www.fstar-lang.org

work detected some re-entrancy problems, as well as unchecked calls to send, and it was evaluated on a small number of source-level scripts.[6]

- EVM bytecode can be decompiled into F*, using another tool called EVM*, which is also written in OCaml. It is possible then verify properties of the code, e.g. gas consumption, running in a lightweight interpreter of a subset of the instruction set.

- Given the two translations, it is also possible to show the equivalence of the Solidity* and EVM* translations, at least when Solidity source code is available.

The Ethereum Virtual Machine (EVM) has also been formalised [25] in the Isabelle/HOL proof assistant, and aspects of the behaviour of a typical contract – the Deed contract, which is a part of the Ethereum Name Service – have been verified. In particular, the safety property that "only the registrar can decrease the balance" has been formally established.

As should be expected, one of the benefits of such a formalisation is that [25] contains a clear statement of the set of assumptions under which the result holds, and these represent the result of a serious conceptual modelling exercise. The report points to further work,[7] such as validating the EVM implementation and formalisation of Gas, so that liveness properties ("something good happens") can be proved in addition to the existing safety property ("nothing bad happens").

8.9 Static analysis of contracts

Some errors can be easily checked by applying static analysis tools. This approach has been explored extensively for programming languages like C. But in particular we would like to highlight its application as a quick fix to Ethereum EVM languages through the Oyente tool [31], which can be used both for finding errors in contracts that are written by the user, and to allow users to avoid using buggy contracts that are already deployed.

8.10 Merkelized Abstract Syntax Trees

MAST (Merkelized Abstract Syntax Trees) [44] is a proposal for allowing Bitcoin transaction validation scripts to be stored in partially-hashed form.

The existing structure of a Bitcoin validation script is simply a Script program consisting of a sequence of Script operations. However, there is a logical structure to Script programs, as in all programs, which has tree form. In particular, every conditional operation causes the program to branch into two logical subprograms, one for the case that the condition was true, and one for the case that the condition was false.

For any given script, a successful validation involves passing down to a leaf block of code in such a tree, having passed or failed the tests accordingly. All other branches of the tree are irrelevant for the validation in question.

---

[6]Note that the vast majority of Ethereum scripts on the blockchain are only available as EVM code.

[7]More is said about this on reddit: `https://www.reddit.com/r/ethereum/comments/59wr6w/formal_verification_of_deed_contract_in_ethereum/`

The MAST proposal takes advantage of this fact, and proposes to replace all of the irrelevant branches with their hashes, thereby eliminating from validation scripts all portions of the script that are not required to perform the validation.

Implementing MAST in a completely new blockchain ought to be simple. There is a major caveat, however. MAST relies on the fact that evaluation of code involves simply running through a list of operations to perform once, in the obvious order. The possible execution paths correspond directly to the program branches. Richer programming constructs such as loops, recursion, defined functions, etc. make it harder, possibly impossible, to use MAST. This arises because the parts of the program that are relevant to a computation, and therefore to a validation, cannot be delineated at the outset as one of a handful of paths to a leaf node in the code tree. Instead, the relevant paths are determined by execution and may involve many scattered parts of the program. It may still be possible to use MAST in this setting, however, but it's an open question as to how precisely to do that.

9   Conclusion

In this paper, we have surveyed some representative examples of the advanced use of cryptocurrencies and blockchains beyond their basic usage as a payment method, and we have focused in existing scripting solutions, their strengths and weaknesses, and some existing solutions for known problems with them.

We have seen that, while there have been many diverse efforts in different directions, there are still many open questions, no universal solutions, and lots of room for future research and experimentation.

References

[1] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In 2014 IEEE Symposium on Security and Privacy, pages 443–458. IEEE, 2014.

[2] Andreas M Antonopoulos. Mastering Bitcoin. O'Reilly Media, 2014.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. Cryptology ePrint Archive: Report 2016/1007, `https://eprint.iacr.org/2016/1007`, 2016.

[4] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In International Cryptology Conference, pages 421–439. Springer, 2014.

---

[8]`https://iohk.io`

[5] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS '16. ACM, 2016.

[6] Block size limit controversy. `https://en.bitcoin.it/wiki/Block_size_limit_controversy`, 2010. [last accessed 21-11-2016].

[7] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. `https://eprint.iacr.org/2015/261.pdf`, 2015. [last accessed 27-11-2016].

[8] BPMN. `http://www.bpmn.org`, 2016. [last accessed 21-11-2016].

[9] Colored Coins. `https://en.bitcoin.it/wiki/Colored_Coins`, 2014. [last accessed 21-11-2016].

[10] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In 2015 IEEE Symposium on Security and Privacy, pages 253–270. IEEE, 2015.

[11] Counterparty. `http://counterparty.io/`, 2014. [last accessed 21-11-2016].

[12] Counterparty at Wikipedia. `https://en.wikipedia.org/wiki/Counterparty_(technology)`, 2014. [last accessed 21-11-2016].

[13] Karl Crary and Michael J Sullivan. Peer-to-peer affine commitment using bitcoin. ACM SIGPLAN Notices, 50(6):479–488, 2015.

[14] Data Storage Methods at Colored Coins wiki. `https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification/wiki/Data%20Storage%20Methods`, 2015. [last accessed 21-11-2016].

[15] Kevin Delmolino, Mitchell Arnett, Ahmed E Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. IACR Cryptology ePrint Archive, 2015:460, 2015.

[16] Design Rationale at Ethereum Wiki. `https://github.com/ethereum/wiki/wiki/Design-Rationale`, 2014. [last accessed 23-11-2016].

[17] Disabled opcodes at Bitcoin client source code. `https://github.com/bitcoin/bitcoin/blob/57b34599b2deb179ff1bd97ffeab91ec9f904d85/src/script/interpreter.cpp#L288`, 2010. [last accessed 21-11-2016].

[18] Distributed ledger technology: beyond block chain. `https://www.gov.uk/government/news/distributed-ledger-technology-beyond-block-chain`, 2016. [last accessed 27-11-2016].

[19] FSM and Nxt. `https://nxtforum.org/automated-transactions/ats-with-fsm-based-dsl/`, 2014. [last accessed 21-11-2016].

[20] O Goldreich, S Micali, and A Wigderson. Proofs that yield nothing but the validity of their assertion. Preprint, 1986.

[21] LM Goodman. Tezos—a self-amending crypto-ledger White paper. `https://tezos.com/pdf/white_paper.pdf`, 2014.

[22] LM Goodman. Tezos: A Self-Amending Crypto-Ledger Position Paper. `https://tezos.com/pdf/position_paper.pdf`, 2014.

[23] Yoichi Hirai. Bamboo: an embryonic smart contract language. `https://github.com/pirapira/bamboo`, 2016. [last accessed 24-11-2016].

[24] Yoichi Hirai. Ethereum reentrancy. Private communication, 2016.

[25] Yoichi Hirai. Formal Verification of Deed Contract in Ethereum Name Service. `https://yoichihirai.com/deed.pdf`, 2016. [last accessed 24-11-2016].

[26] Richard Hull, Vishal S. Batra, Yi-Min Chee, Alin Deutsch, Fenno F. Terry Heath, III, and Victor Vianu. Towards a Shared Ledger Business Collaboration Language based on Data-Aware Processes. In Proc. Intl. Conf. on Service Oriented Computing (ICSOC), 2016.

[27] Hyperledger. `https://www.hyperledger.org/blog`, 2016. [last accessed 21-11-2016].

[28] Florian Idelberger, Guido Governatori, Régis Riveret, and Giovanni Sartor. Evaluation of logic-based smart contracts for blockchain systems. In International Symposium on Rules and Rule Markup Languages for the Semantic Web, pages 167–183. Springer, 2016.

[29] Is there a maximum size of a scriptSig/scriptPubKey? at Bitcoin StackExchange. `http://bitcoin.stackexchange.com/questions/35878/is-there-a-maximum-size-of-a-scriptsig-scriptpubkey`, 2015. [last accessed 21-11-2016].

[30] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. University of Maryland and Cornell University, 2015.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 254–269. ACM, 2016.

[32] Maximum number of op_codes in script at Bitcoin StackExchange. `http://bitcoin.stackexchange.com/questions/38230/maximum-number-of-op-codes-in-script`, 2015. [last accessed 21-11-2016].

[33] L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. Electr. Notes Theor. Comput., 141(5):49–76, 2005.

[34] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In Security and Privacy (SP), 2013 IEEE Symposium on, pages 397–411. IEEE, 2013.

[35] Namecoin. `https://namecoin.org/`, 2011. [last accessed 21-11-2016].

[36] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Mobile Agents and Security, pages 61–91, London, UK, UK, 1998. Springer-Verlag.

[37] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In Security and Privacy (SP), 2013 IEEE Symposium on, pages 238–252. IEEE, 2013.

[38] Jack Pettersson and Robert Edström. Safer smart contracts through type-driven development. `https://publications.lib.chalmers.se/records/fulltext/234939/234939.pdf`, 2106.

[39] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP '00. ACM, 2000.

[40] Cécile Pierrot and Benjamin Wesolowski. Malleability of the blockchain's entropy. Cryptology ePrint Archive, 2016/370, `https://eprint.iacr.org/2016/370.pdf`, 2016.

[41] Contracts, Composition, and Scaling The Rholang 0.1 specification. `https://docs.google.com/document/d/1gnBCGe6KLjYnahktmPSm_-8V4jX53Zk10J-KFQl7mf8/edit?pref=2&pli=1#heading=h.k4bk2akncduu`, 2016. [last accessed 11-12-2016].

[42] Rootstock. `http://www.rsk.co/`, 2014. [last accessed 21-11-2016].

[43] Rootstock White Paper. `http://www.the-blockchain.com/docs/Rootstock-WhitePaper-Overview.pdf`, 2015. [last accessed 21-11-2016].

[44] Naik Manali Rubin, Jerry and Nitya Subramanian. Merkelized Abstract Syntax Trees. `http://www.mit.edu/~jlrubin/public/pdfs/858report.pdf`, 2016. [last accessed 11-12-2016].

[45] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In 2014 IEEE Symposium on Security and Privacy, pages 459–474. IEEE, 2014.

[46] Script – Bitcoin Wiki. `https://en.bitcoin.it`, 2010. [last accessed 21-11-2016].

[47] Serpent at Ethereum Wiki. `https://github.com/ethereum/wiki/wiki/Serpent`, 2014. [last accessed 21-11-2016].

[48] Solidity documentation. `https://solidity.readthedocs.io/en/develop/`, 2016. [last accessed 21-11-2016].

[49] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, and Alexander Ponomarev andJan Mendling. Untrusted Business Process Monitoring and Execution Using Blockchain. In Intl. Conf. Business Process Mgmt. (BPM), 2016.

[50] What are Forks, Alt-coins, Meta-coins, and Sidechains? `https://coincenter.org/entry/what-are-forks-alt-coins-meta-coins-and-sidechains`, 2015. [last accessed 21-11-2016].

[51] Where do contract event logs get stored in the Ethereum architecture? at Ethereum StackExchange. `http://ethereum.stackexchange.com/questions/1302/where-do-contract-event-logs-get-stored-in-the-ethereum-architecture`, 2013. [last accessed 21-11-2016].