TxVM
A New Design for Blockchain Transactions

Bob Glickstein, Cathie Yun, Dan Robinson, Keith Rarick, Oleg Andreev
{bobg, cathie, dan, kr, oleg}@chain.com

Chain

March 2018

Abstract

We present a new design for blockchain transactions called TxVM, the transaction virtual machine. TxVM seeks to achieve the expressiveness and flexibility of an imperative contract model such as Ethereum's while maintaining the efficiency, safety, and scalability of a declarative transaction model such as Bitcoin's. TxVM defines a stack machine for manipulating plain data items like strings, integers, and tuples, but also special types: *values*, each with an amount and asset type; and *contracts*, programs that lock up values and other data. Rules governing the handling of these types provide guarantees about integrity and security. Each transaction is a TxVM program evaluated in isolation from the blockchain state, and whose output is a deterministic log of proposed state changes for the blockchain. Transactions can be therefore be validated in parallel. Their logs can be applied to the blockchain in linear time. Our implementation of TxVM is currently used in production in our hosted ledger service, Sequence. The implementation and specification are available as an open-source project on GitHub.

1 Introduction

A transaction in a blockchain protocol is a proposal to update the blockchain's global state. Depending on the blockchain, this could mean consuming some value tokens and creating others, or updating the balances in one or more accounts. Participating nodes in the blockchain network reach consensus on whether the transaction is valid and should be applied. If so, the proposed state changes are made.

Different blockchain protocols take different approaches to representing transactions, each with its own strengths and weaknesses.

1.1 Prior art: Bitcoin

In Bitcoin [1] and similar systems, including earlier versions of the Chain Protocol, a transaction is a static data structure with fields that must be interpreted and validated using an ad hoc collection of rules. One or more "inputs" identify existing tokens to be redeemed from earlier transactions. Each input includes the data needed to authorize access to those tokens. The accessed value is divided among one or more "outputs." Each describes how to secure its value. This is typically done by specifying the public key of a payee, usually in the form of a short program for a specialized virtual machine. The program verifies a digital signature against that public key. Some future "input" must supply that signature to "spend" the output.

This model is **declarative**. The transaction's data structure declares its proposed state changes directly. No user-defined program runs except for the predicates attached to the previous outputs to authorize their use, and there are no effects on the blockchain state other than the ones discoverable by a simple inspection of the transaction's fields. Furthermore, tokens are immutable. Once created, they persist in the same state until consumed.

These properties make declarative transactions efficient and secure. Transactions can be validated in parallel before any of their effects have to be applied to the blockchain state.[1] It is easy to avoid transactions with unexpected side-effects. And the global blockchain state does not need to contain much more than a cryptographic hash committing to the contents of each existing token.

On the other hand, Bitcoin's transaction model limits the power of its smart contracts. This is partially a result of Bitcoin's restricted virtual machine, which by design is not Turing-equivalent. But mainly it's because Bitcoin's scripts are simple predicates evaluated independently of each other. This makes it unduly difficult to express multiple contracts interacting, even after adding more powerful opcodes, as in earlier versions of the Chain Protocol. We found that modeling more sophisticated flows of value required unwieldy contortions in the best case and were occasionally impossible to do securely.

1.2 Prior art: Ethereum

In Ethereum [2] and similar systems, value resides in contracts that also embody mutable state and program logic. A transaction is a message sent to a contract, causing its program to execute.

This is an **imperative** model, in which the effects of the transaction on the blockchain are not known until the contract logic finishes running, during which time it may alter its own state as well as send messages to other contracts, which can update their own state.

Contract logic may be highly sophisticated, and indeed a wide variety of

[1] One transaction's application to the state may render another transaction inapplicable, as when each tries to spend the same token. For our purposes, this is a separate step from validation, and is considered acceptable as long as such conflicts can be resolved in approximately linear time.

novel flows of value have been demonstrated using imperative blockchain contracts, from decentralized autonomous organizations to cryptocurrency-based virtual pets.

However, interaction with the global blockchain state during execution means that there can be no meaningful optimization by parallelizing validation. Transactions must execute serially, in a deterministic order. Since one contract might alter the state of any other contract, it is easy for execution to have unexpected side-effects, and it can be difficult to reason about a contract's state even during the lifetime of a single transaction. This can be catastrophic, such as in the June 2016 hack of the "DAO" contract [4], which resulted in the theft of around $50 million, and the November 2017 Parity bug [7], which froze wallets containing around $150 million.

1.3 A combined approach

TxVM is the basis for the protocol used in Sequence [6], Chain's blockchain-based hosted ledger service. TxVM stands for "transaction virtual machine."

With TxVM we seek to combine the respective strengths of the declarative and imperative approaches to representing blockchain transactions, while avoiding their weaknesses. It takes advantage of lessons we learned from our own previous design, ChainVM [3], and from developing Ivy [5], our higher-level smart-contract language, which compiles to ChainVM and also to Bitcoin Script. TxVM is designed to be an ideal compilation target for Ivy.

A TxVM transaction is an **imperative** program that produces a **declarative** log of proposed blockchain state changes when executed. Execution happens in isolation from the global blockchain state. Running in isolation means TxVM programs cannot have unexpected side effects in other contracts, and that they can be run in parallel.

2 Operation of the virtual machine

TxVM defines a stack-based virtual machine to execute transaction programs. Programs are expressed as strings of bytecode. The TxVM instruction set includes a `jumpif` instruction, making it Turing-complete. It also includes operations for manipulating various types of data, introspecting aspects of the VM state, computing cryptographic hashes, verifying signatures, and more. The complete instruction set is shown in figure 1.

Certain instructions cause records to be appended to the *transaction log*, a VM data structure that is the primary output of running a TxVM program. It contains the blockchain state changes proposed by the transaction. After the transaction program finishes, the log may be applied to the blockchain.

A *contract* is a unit of program execution that contains a program (a string of bytecode) and a stack. While a contract is running, its stack serves as the VM's *current contract stack*, where most stack-based operations take place. A contract may also suspend its execution in a variety of ways, passing control to

	00	10	20	30	40	50	60	70
0	0/false	16	int	nonce	verify	eq	1 byte	17 bytes
1	1/true	17	add	merge	jumpif	dup	2 bytes	18 bytes
2	2	18	neg	split	exec	drop	3 bytes	19 bytes
3	3	19	mul	issue	call	peek	4 bytes	20 bytes
4	4	20	div	retire	yield	tuple	5 bytes	21 bytes
5	5	21	mod	amount	wrap	untuple	6 bytes	22 bytes
6	6	22	gt	assetid	input	len	7 bytes	23 bytes
7	7	23	not	anchor	output	field	8 bytes	24 bytes
8	8	24	and	vmhash	contract	encode	9 bytes	25 bytes
9	9	25	or	sha256	seed	cat	10 bytes	26 bytes
a	10	26	roll	sha3	self	slice	11 bytes	27 bytes
b	11	27	bury	checksig	caller	bitnot	12 bytes	28 bytes
c	12	28	reverse	log	cprog.	bitand	13 bytes	29 bytes
d	13	29	get	peeklog	timerange	bitor	14 bytes	30 bytes
e	14	30	put	txid	prv	bitxor	15 bytes	31 bytes
f	15	31	depth	finalize	ext	0 bytes	16 bytes	32 bytes

small ints	stack ops	crypto ops	control flow	data ops
int ops	value ops	tx ops	extension	pushdata

Figure 1: The TxVM instruction set.

some other contract. At such times the suspended contract's stack is preserved until it is reinvoked.

When control passes from one contract to another, data may be passed between them on the VM's shared *argument stack*.

The overall transaction program forms an implicit top-level contract.

Stacks may contain plain data items: strings, integers, and tuples. They may also contain contracts, and they may contain *values*, each of which is a specific amount of a specific asset type. Values and asset types are discussed in further detail in the next section.

In addition to their bytecode, transaction programs specify an integer *runlimit*. Each instruction costs a nonzero amount to execute, and the total cost of running the program must not exceed the specified runlimit. This prevents abuse of the network in a similar way to the role played by Ethereum's "gas." A network can agree to reject transactions with runlimits that are too high.

In order to be valid, a transaction program must execute to completion and leave no data on any stack.

2.1 Values

A TxVM blockchain is used to track the issuance, ownership, and transfer of values of different types and amounts, e.g. "5 USD" or "3 shares of AAPL." A value is a first-class item on the stack.

Inside a value object is an amount, an *asset ID*, and an *anchor*. A new value may be created only by the `issue` instruction, which populates the asset ID field with a cryptographic hash computed from the currently running contract—the one containing the `issue` instruction. An issuance contract thus uniquely determines the asset type it issues. No other contract may issue units of the same asset.

The `issue` instruction populates the anchor field with a hash derived from earlier values in the blockchain, guaranteeing uniqueness and non-replayability.

Finally, `issue` adds a record of the new issuance to the transaction log.

Once created, a value may be `split` into two values with the same asset ID and sum, and may be `merge`d with other values with the same asset ID, in all cases producing values with new anchors. A value with an amount of zero is useful in some cases for its anchor alone.

Values may be destroyed with the `retire` instruction, which also creates a transaction log record.

Unlike plain data, values may not be duplicated or dropped from a stack.

2.2 Contracts

A contract contains a program and a stack. It is created with the `contract` instruction. Its stack is initially empty, and its program is set to the bytecode string that `contract` takes as an argument.

Contracts are invoked with `call`, which causes the VM's current contract stack to be saved away and replaced by the called contract's stack. The saved stack is restored when control returns to the caller.

When a contract reaches the end of its program with an empty stack, it is *complete*. It is removed from the VM and control returns to the caller. It is an error for a contract to reach the end of its program while items remain on its stack. If it is the implicit top-level contract, the argument stack must also be empty.

A contract that has not yet completed may suspend its own execution in one of three ways:

- It may execute the `yield` instruction, placing the contract on the argument stack while returning control to the caller.

- It may execute the `output` instruction, writing an "output" record to the transaction log. That record contains a cryptographic hash committing to a snapshot of the contract's state, including the contents of its stack. This *snapshot hash* will be added to the blockchain's global state as an unspent output contract. The contract may be reconstituted in a later transaction, and its execution resumed, with the `input` instruction.

- It may execute the `wrap` instruction, which is like `yield` but makes the suspended contract "portable." Portability of contracts is not discussed here; for details please see the TxVM spec. [9]

In order to reconstitute a contract from the global blockchain state (placed there with `output`), a program first creates a plain-data depiction of the contract: a tuple that includes the contract's program string and all the items on its stack.[2] The `input` instruction turns that tuple into a callable contract object while adding an "input" record to the transaction log. The input record contains a snapshot hash computed from the tuple. Later, when the log is applied to the blockchain, that snapshot hash is checked against the global state to ensure that the stipulated contract actually exists to be consumed.

Like values, contracts may not be duplicated or dropped from a stack. Thus, all contracts created during a transaction must run to completion or be persisted to the global blockchain state with `output` in order to be cleared from the VM. It follows too that all values (other than those that are destroyed with `retire`) must end up in the stack of a contract persisted with `output`, or else be left in the VM, preventing successful completion.

2.3 The transaction log

The transaction log is the primary result of running a TxVM transaction. It is also the source of a transaction's unique ID, which is computed from a hash of the log's contents.

A transaction usually includes one or more signature checks. The message being signed is typically the transaction's ID, possibly in combination with other data. This creates a chicken-and-egg problem: the transaction is not complete until it includes the necessary signatures, but the signatures require the transaction ID, which requires running the transaction.

To solve this problem, TxVM includes the `finalize` instruction. This freezes the transaction log, prohibiting further changes to it. It also makes the `txid` instruction available for querying the transaction's ID. Every transaction must execute `finalize` exactly once. It is possible to run a transaction program up to its `finalize` instruction, in order to compute the ID of that transaction. Signature-checking contracts presumably still remain on the stack at this point. Once the ID has been computed, it's possible to compute any required signatures. These can now be added to the transaction program as arguments to those contracts, together with the `call` instructions that will invoke them and clear them from the VM.

To ensure the uniqueness of each transaction and each transaction ID, the `finalize` instruction consumes an anchor (a value with an amount of zero) from the stack.

[2]Since only the snapshot hash is stored in the blockchain state, the user has the responsibility to remember or retrieve sufficient information about the contract to reconstruct it. This may involve monitoring the blockchain for recognizable contract patterns and parsing those, communicating out-of-band with the contract's creator, or other techniques.

3 A typical transaction

In this section we present a simplified TxVM transaction, in which Alice wishes to pay 10 units of some asset to Bob. Alice's transaction inputs two contracts from the blockchain, one containing 5 units and the other containing 7. The transaction combines those values and then resplits them, creating one output of 10 for Bob and a "change" output of 2 for Alice.

This example uses TxVM assembly-language notation, in which certain operations have a simplified depiction. For instance, "pushdata" instructions are implicit, tuple literals (delimited as `{...}`) abbreviate the steps needed to construct them, and a sequence of assembly-language instructions enclosed in square brackets (`[...]`) denotes the bytecode string they produce when assembled.

Here is the transaction, with some details elided for clarity.

```
{...} input call get get
{...} input call get get
```

Marshal two contracts from the global blockchain state, call them, and move their results—a value and a signature-check contract each—from the argument stack to the current contract stack.

```
2 roll merge
```

Put the two values (a 5-unit value and a 7-unit value) next to each other on the stack and merge them into one 12-unit value.

```
10 split <Bob's pubkey> put put [get get ... output] contract call
```

Split the 12-unit value into one 10-unit value and one 2-unit value. Add Bob's pubkey to the stack. Move it and the 10-unit value to the argument stack. Construct and call a contract whose program consumes the value and pubkey, then `output`*s itself.*

```
2 split <Alice's pubkey> put put [get get ... output] contract call
```

Split the 2-unit value into one 2-unit value and one zero-unit value (which will be used as an anchor by `finalize`*). Add Alice's pubkey to the stack. Construct and call a contract whose program consumes the value and pubkey, then* `output`*s itself.*

```
finalize
```

Consume the zero-value anchor and freeze the transaction log. At this point, the two signature-check contracts (for authorizing the `input`*s above) remain on the stack.*

```
<a signature by Alice of this transaction ID> put call
<a signature by Alice of this transaction ID> put call
```

Supply a signature to each signature-check contract and call it to clear it from the VM.

The next sections take a look at some of the details elided from the example above.

3.1 Checking signatures

Here is a simple signature-checking program.

```
txid <pubkey> get 0 checksig verify
```

The steps of this program are:

`txid`	Get the transaction's ID and push it on the stack (only possible after `finalize`);
pubkey	Push the pubkey (of a value's owner, or an asset's authorized issuer, etc.) on the stack;
`get`	Move a data item (the signature) from the argument stack to the current contract stack;
`0`	Push a 0 on the stack (signaling the `checksig` instruction to use the Ed25519 signature scheme);
`checksig`	Compute the validity of the signature with respect to the transaction ID and pubkey;
`verify`	Fail execution if `checksig` did not produce a true value.

3.2 Unspent output

Here is a simple program for an unspent output contract that already contains a value and a payee's pubkey on its stack.

```
put [txid swap get 0 checksig verify] yield
```

The `put` instruction releases the contract's value to the argument stack. The `yield` instruction, with a signature-checking program as an argument, suspends this contract's execution (with the payee's pubkey still on its stack) and places *it* on the argument stack. Of course, the suspended contract will need to be `call`ed again to clear it from the VM. This is a *deferred* signature check, which can run only after `finalize`, since it uses `txid`.

Note that this version of the signature-checking contract differs slightly from the example presented above. In that example, the pubkey appears literally in the program. In this example, the pubkey is already on the stack and is moved into its proper place (with `swap`[3]) after the transaction ID is placed on the stack.

Knowing this program, it is possible to flesh out some of the tuple passed to `input`:

```
{'C',
  <seed>,
  [txid swap get 0 checksig verify],
  <payee pubkey>,
  {'V', <amount>, <asset ID>, <anchor>}
}
```

[3]Which is TxVM assembly-language shorthand for `1 roll`.

Here, `C` and `V` are type codes (for "contract" and "value" respectively), and "seed" is the contract's seed, a unique identifier (not discussed here). The `input` instruction turns this structure into a contract object, meanwhile computing a snapshot hash for the transaction log that must match the snapshot hash from an earlier `output` instruction.

3.3 Pay-to-pubkey

Finally, here is the contract used to lock up value with a payee's pubkey. Note how the unspent-output contract is a latter phase of this contract, and the signature check is a latter phase of *that*.

```
get get [put [txid swap get 0 checksig verify] yield] output
```

This consumes two arguments from the argument stack, a pubkey and a value. It then `output`s itself as an unspent-output contract. When next called (after `input`), it will release the value and defer a signature check against the pubkey.

3.4 Scratching the surface

The example in this section is a simplified transfer of a single type of value from a single sender to a single recipient. In Sequence such transfers are made with slightly more elaborate versions of the contract programs presented here. Those programs include provisions for M-of-N signatures and for attaching user-supplied reference data to payments, among other things.

Beyond that, it should be evident that much more is possible. A transaction can involve multiple parties trading multiple asset types simultaneously and atomically. A contract can lock up zero, two, or more values, not just one. An asset's issuance contract can be designed to constrain how units of it may be spent.

A discussion of TxVM's full power is beyond the scope of this paper; indeed we are still discovering it ourselves. In the coming months we will be retargeting our compiler for the Ivy high-level smart contract language to TxVM. We expect to show how use cases such as escrowed payments, collateralized loans, second-price auctions, bond coupons, and even decentralized autonomous organizations and cryptocurrency-based virtual pets may be expressed in Ivy and compiled to compact TxVM programs.

4 Further reading

We have a full specification and implementation of TxVM available as an open-source project on GitHub, at `github.com/chain/txvm`.

One of us (Yun) presented TxVM at the Stanford Blockchain Protocol Analysis and Security Engineering (BPASE) 2018 conference. [8] The talk and slides are available online. [10] [11]

TxVM is currently being used in production in our ledger service, Sequence. For more information, visit `Chain.com` and read our blog post, “Introducing Sequence.” [6]

5 Conclusion

We have presented TxVM, a transaction model and virtual-machine design that combines the usability and power of Ethereum-like contracts with the safety, efficiency, and scalability of Bitcoin-like transactions.

We designed TxVM to serve as the core of our platform for financial applications. We have a full specification and open-source implementation of TxVM in Go, currently deployed in production.

Beyond continuing to use TxVM in production and developing it further, we are interested in applying these ideas to other blockchain protocols. Constructive transaction programs may provide novel ways to build and serialize transactions in Bitcoin-like protocols, while declarative deterministic effect logs may improve the safety and scalability of Ethereum-like platforms.

References

[1] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Nov. 2008. URL: https://bitcoin.org/bitcoin.pdf.

[2] Gavin Wood. *Ethereum: A Secure Decentralized Generalized Transaction Ledger*. 2014. URL: http://yellowpaper.io/.

[3] Chain. *Chain Protocol Whitepaper*. Oct. 2016. URL: https://github.com/chain/chain/blob/main/docs/1.2/protocol/papers/whitepaper.md.

[4] David Z. Morris. *Blockchain-based Venture Capital Fund Hacked for $60 Million*. June 2016. URL: http://fortune.com/2016/06/18/blockchain-vc-fund-hacked/.

[5] Chain. *Announcing Ivy Playground*. May 2017. URL: https://blog.chain.com/announcing-ivy-playground-395364675d0a.

[6] Chain. *Introducing Sequence - Move Fast. Break Nothing.* Oct. 2017. URL: https://blog.chain.com/introducing-sequence-e14ff70b730.

[7] Alex Hern. *$300m in Cryptocurrency Accidentally Lost Forever Due To Bug*. Nov. 2017. URL: https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether.

[8] *Blockchain Protocol Analysis and Security Engineering*. Stanford Cyber Initiative, Jan. 2018. URL: https://cyber.stanford.edu/bpase18.

[9] Chain. *TxVM specification*. Mar. 2018. URL: https://github.com/chain/txvm/blob/main/specifications/txvm.md.

[10] Cathie Yun. *TxVM talk - BPASE '18*. Jan. 2018. URL: https://youtu.be/qY_0MJDMBNY.

[11] Cathie Yun and Oleg Andreev. *TxVM slides - BPASE '18*. Jan. 2018. URL: https://cyber.stanford.edu/sites/default/files/txvm_stanford_jan24.pdf.