Algorand Transaction Execution Approval Language

September 26, 2019

Abstract

Algorand allows transactions to be effectively signed by a small program. If the program evaluates to true then the transaction is allowed. This document defines the language and bytcode format.

Contents

1 Transaction Execution Approval Language (TEAL)

TEAL is a bytecode based stack language that executes inside Algorand transactions to check the parameters of the transaction and approve the transaction as if by a signature.

TEAL programs should be short and run fast as they are run in-line along with signature checking, transaction balance rule checking, and other checks during block assembly and validation. Many useful programs are less than 100 instructions.

1.1 The Stack

The stack starts empty and contains values of either uint64 or bytes. (`bytes` implemented in Go as a []byte slice) Most operations act on the stack, popping arguments from it and pushing results to it.

The maximum stack depth is currently 1000.

1.2 Scratch Space

In addition to the stack there are 256 positions of scratch space, also uint64-bytes union values, accessed by the `load` and `store` ops moving data from or to scratch space, respectively.

1.3 Execution Environment

TEAL runs in Algorand nodes as part of testing a proposed transaction to see if it is valid and authorized to be committed into a block. If an authorized program executes and finishes with a single non-zero uint64 value on the stack then that program has validated the transaction it is attached to.

A program is an authorized program one of two ways: The SHA512_256 hash of the program (prefixed by "Program") is equal to the transaction Sender address; OR the program (prefixed by "Program") has a valid signature or multi-signature from the transaction Sender address.

The TEAL bytecode plus the length of any Args must add up to less than 1000 bytes (consensus parameter LogicSigMaxSize). Each TEAL op has an associated cost estimate and the program cost estimate must total less than 20000 (consensus parameter LogicSigMaxCost). Most ops have an estimated cost of 1, but a few slow crypto ops are much higher.

The TEAL program has access to data from the transaction it is attached to, any transactions in a transaction group it is part of, and a few global values like the current Round number, block Timestamp, and some consensus paramaters.

1.4 Constants

Constants are loaded into the environment into storage separate from the stack. They can then be pushed onto the stack by referring to the type and index. This makes for efficient re-use of byte constants used for account addresses, etc.

The assembler will hide most of this, allowing simple use of `int 1234` and `byte 0xcafed00d`. These constants will automatically get assembled into int and byte pages of constants, de-duplicated, and operations to load them from constant storage space inserted.

Constants are loaded into the environment by two opcodes, `intcblock` and `bytecblock`. Both of these use proto-buf style variable length unsigned int, reproduced here. The `intcblock` opcode is followed by a varuint specifying the length of the array and then that number of varuint. The `bytecblock` opcode is followed by a varuint array length then that number of pairs of (varuint, bytes) length prefixed byte strings. This should efficiently load 32 and 64 byte constants which will be common as addresses, hashes, and signatures.

Constants are pushed onto the stack by `intc`, `intc_[0123]`, `bytec`, and `bytec_[0123]`. The assembler will typically handle converting `int N` or `byte N` into the appropriate constant-offset opcode or opcode followed by index byte.

1.5 Operations

Most operations work with only one type of argument, uint64 or bytes, and panic if the wrong type value is on the stack.

This summary is supplemented by more detail in the opcodes document.

Some operations 'panic' and immediately end execution of the program. A transaction checked by a program that panics is not valid.

1.5.1 Arithmetic

For one-argument ops, `X` is the last element on the stack, which is typically replaced by a new value.

For two-argument ops, `A` is the previous element on the stack and `B` is the last element on the stack. These typically result in popping A and B from the stack and pushing the result.

Op	Description
`sha256`	SHA256 hash of value, yields [32]byte
`keccak256`	Keccak256 hash of value, yields [32]byte
`sha512_256`	SHA512_256 hash of value, yields [32]byte
`ed25519verify`	for (data, signature, pubkey) verify the signature of the data against the pubkey => {0 or 1}
`rand`	push random uint64 to stack
`+`	A plus B. Panic on overflow.
`-`	A minus B. Panic if B > A.
`/`	A divided by B. Panic if B == 0.
`*`	A times B. Panic on overflow.

Op	Description
`<`	A less than B => {0 or 1}
`>`	A greater than B => {0 or 1}
`<=`	A less than or equal to B => {0 or 1}
`>=`	A greater than or equal to B => {0 or 1}
`&&`	A is not zero and B is not zero => {0 or 1}
`||`	A is not zero or B is not zero => {0 or 1}
`==`	A is equal to B => {0 or 1}
`!=`	A is not equal to B => {0 or 1}
`!`	X == 0 yields 1; else 0
`len`	yields length of byte value
`itob`	converts uint64 to big endian bytes
`btoi`	converts bytes as big endian to uint64
`%`	A modulo B. Panic if B == 0.
`|`	A bitwise-or B
`&`	A bitwise-and B
`^`	A bitwise-xor B
`~`	bitwise invert value
`mulw`	A times B out to 128-bit long result as low (top) and high uint64 values on the stack

1.5.2 Loading Values

Opcodes for getting data onto the stack.

Some of these have immediate data in the byte or bytes after the opcode.

Op	Description
`intcblock`	load block of uint64 constants
`intc`	push value from uint64 constants to stack by index into constants
`intc_0`	push uint64 constant 0 to stack
`intc_1`	push uint64 constant 1 to stack
`intc_2`	push uint64 constant 2 to stack
`intc_3`	push uint64 constant 3 to stack
`bytecblock`	load block of byte-array constants
`bytec`	push bytes constant to stack by index into constants
`bytec_0`	push bytes constant 0 to stack
`bytec_1`	push bytes constant 1 to stack
`bytec_2`	push bytes constant 2 to stack

Op	Description
`bytec_3`	push bytes constant 3 to stack
`arg`	push LogicSig.Args[N] value to stack by index
`arg_0`	push LogicSig.Args[0] to stack
`arg_1`	push LogicSig.Args[1] to stack
`arg_2`	push LogicSig.Args[2] to stack
`arg_3`	push LogicSig.Args[3] to stack
`txn`	push field from current transaction to stack
`gtxn`	push field to the stack from a transaction in the current transaction group
`global`	push value from globals to stack
`load`	copy a value from scratch space to the stack
`store`	pop a value from the stack and store to scratch space

Transaction Fields

Index	Name	Type
0	Sender	[]byte
1	Fee	uint64
2	FirstValid	uint64
3	LastValid	uint64
4	Note	[]byte
5	Receiver	[]byte
6	Amount	uint64
7	CloseRemainderTo	[]byte
8	VotePK	[]byte
9	SelectionPK	[]byte
10	VoteFirst	uint64
11	VoteLast	uint64
12	VoteKeyDilution	uint64
13	Type	[]byte
14	TypeEnum	uint64
15	XferAsset	[]byte
16	AssetAmount	uint64
17	AssetSender	[]byte
18	AssetReceiver	[]byte
19	AssetCloseTo	[]byte
20	GroupIndex	uint64
21	TxID	[]byte

Additional details in the opcodes document on the `txn` op.

Global Fields

Index	Name	Type
0	Round	uint64
1	MinTxnFee	uint64
2	MinBalance	uint64
3	MaxTxnLife	uint64
4	TimeStamp	uint64
5	ZeroAddress	[]byte
6	GroupSize	uint64

1.5.3 Flow Control

Op	Description
`err`	Error. Panic immediately. This is primarily a fencepost against accidental zero bytes getting compiled into programs.
`bnz`	branch if value is not zero
`pop`	discard value from stack
`dup`	duplicate last value on stack

2 Assembler Syntax

The assembler parses line by line. Ops that just use the stack appear on a line by themselves. Ops that take arguments are the op and then whitespace and then any argument or arguments.

"`//`" prefixes a line comment.

2.1 Constants and Pseudo-Ops

A few pseudo-ops simplify writing code. `int` and `byte` and `addr` followed by a constant record the constant to a `intcblock` or `bytecblock` at the beginning of code and insert an `intc` or `bytec` reference where the instruction appears to load that value. `addr` parses an Algorand account address base32 and converts it to a regular bytes constant.

`byte` constants are:

```
byte base64 AAAA...
byte b64 AAAA...
byte base64(AAAA...)
byte b64(AAAA...)
byte base32 AAAA...
byte b32 AAAA...
byte base32(AAAA...)
byte b32(AAAA...)
byte 0x0123456789abcdef...
```

`int` constants may be `0x` prefixed for hex, `0` prefixed for octal, or decimal numbers.

`intcblock` may be explictily assembled. It will conflict with the assembler gathering `int` pseudo-ops into a `intcblock` program prefix, but may be used in code only has explicit `intc` references. `intcblock` should be followed by space separated int constants all on one line.

`bytecblock` may be explicitly assembled. It will conflict with the assembler if there are any `byte` pseudo-ops but may be used if only explicit `bytec` references are used. `bytecblock` should be followed with byte contants all on one line, either 'encoding value' pairs (`b64 AAA...`) or 0x prefix or function-style values (`base64(...)`).

2.2 Labels and Branches

A label is defined by any string not some other op or keyword and ending in ':'. A label can be an argument (without the trailing ':') to a branch instruction.

Example:

```
int 1
bnz safe
err
safe:
pop
```

3 Encoding and Versioning

A program starts with a varuint declaring the version of the compiled code. Any addition, removal, or change of opcode behavior increments the version. For the most part opcode behavior should not change, addition will be infrequent (not likely more often than every three months and less often as the language matures), and removal should be very rare.

For version 1, subsequent bytes after the varuint are program opcode bytes. Future versions could put other metadata following the version identifier.

3.1 Varuint

A ‘proto-buf style variable length unsigned int’ is encoded with 7 data bits per byte and the high bit is 1 if there is a following byte and 0 for the last byte. The lowest order 7 bits are in the first byte, followed by successively higher groups of 7 bits.