Current Issues

Currently Taiga's data protocol is specified through various Rust structs that contain formats from crates such as halo2

and pasta_curves

. Although these crates have formats such as Borsh, these are Rust specific and don't offer any obvious BNF (Click Specification, the "BNF" showed is not a real BNF). What would be ideal is a binary format for public facing data types into Taiga itself.

This format, should be unambiguous and simple for people to create parsers back and forth.

Most of the suggestions here are actionable today, with the caveat that the binary format is not yet specified

General Layouts

Instead of relying upon layers of Rust structs, it would be better if we just laid out the bytes plainly, if the curve points for a transaction are a series of 4 256 bit numbers, then lay it out as such. Thus a note would look something like this

+---------------+---------------+---------------+---------------+...+ | 256 bit point | 256 bit point | 256 bit point | 256 bit point |...| +---------------+---------------+---------------+---------------+...+

where each number is specified in little endian.

The data we are concerned with are the encoding formats for the revised public API that this post covers. Thus transactions

and partial transactions

Most data for these structures are fixed size and can be encoded with fixed length encodings, except for the circuit representation, executor/verifier hints, and number of partial transactions in a transaction.

Taiga Data Revisions

Currently, in the Taiga Codebase we use a dyn trait to represent the kinds of circuits we can verify, with each instance requiring implementing some functions to fulfill this role. This makes it hard to create a serialization format for, as you need to serialize Rust functions. Further, this is the only bit of dynamic data, thus making any solver's job hard, as they have little insight into the intents expressed within.

Thus I propose the following:

We have two dynamic fields in the bytesteam, a verifier hint

and anoma byte code

.

The verifier hint

is an s-expression expressing whatever intent is expected:

EX:

(and (or (trade-ratio (btc 10) (eth 20)) (trade-ratio (btc 5) (xtz 100))) (limit (btc 100)))

might mean for this intent we accept the ratio of 10 btc for 20 eth, or 5 btc for 100 xtz, with a limit of trading at 100 btc.

The size of this s-expression is not known before hand, and thus will be encoded as an s-expression on the wire.

The anoma bytecode

represents the logic that will be ran in a circuit.

this will contain a list of bytecode instructions to be ran. This size is not fixed, and we should chose a strategy for encoding that is safe.

Why The Verifier Hint

One important question one may have is why have the verifier hint

at all? A counter argument could very well argue the following:

Imagine some bytecode system S

, and high level language L

. in S

you can encode the intent such that, when loading the code into Anoma OS, the bytecode adds a definition d

such that when viewed with L

one can natively program over it, effectively loading it into the system, allowing one to program over it

This is a fairly solid counter argument, however due to the freedom in the format, the structure of the bytecode would fall down to convention on where the hints would live and thus cause some issues on where to load it into L

, whereas with it being a separate field, a solver written in Anoma OS, can simply just load the s-expression format into their system wherever they desire, effectively losing nothing.

Putting the types together

With these formats somewhat describes we could see a format as follows

Partial Transaction +----------------+---------------+----+---------------+-----+----------+-----+ | page of fields | 64 bit amount| ... | verifier hint | '(' | bytecode | ')' | +----------------+---------------+----+---------------+-----+----------+-----+

Where we separate the byteode by ()

's to denote the start and end. Note that bytecode likely have known bytes, meaning they are separated with no extra work.

Further the ...

's are fields that can be fleshed out with proper care by the Taiga Team

Further in a transaction, we can abuse this trick again to denote a list of partial transactions, a context free specifying method ought to be used for this.

Additional Benefits

If this format is specified, this not only lets people bind to taiga, without first having to go into inner crates, but it also gives us a good feeling on where we are wasting space.

With a defined format, we can find ways to maybe compress the data needed or rethink strategies, as it should be important to minimze the size of data sent to and from Taiga.