# Truebit Light's Incentive System

Christian Reitwießner
chris@ethereum.org

Blockchains can reasonably ensure that programs are executed correctly in a decentralized setting. They do not guarantee that a transaction is accepted, nor do they guarantee that an accepted transaction remains accepted, but the probability of a rollback decreases as time progresses.

The main problem of blockchains is their scalability: The amount of computation that can happen per time is more or less fixed. It especially does not increase the more participants join the network, it is rather limited by the slowest node in the network.

TrueBit tries to solve this problem through interactive verification. It allows more or less arbitrarily complex computations to be performed under the assumption that there is at least one honest participant. It does not require that participant to be altruistic, though. TrueBit also includes some drawbacks, especially the drawback that transactions usually take more time to be accepted and they can also be delayed arbitrarily by an attacker, as long as this attacker has enough financial resources. The system favours correctness over liveness, i.e. as long as there is at least one honest participant, it is impossible to include an incorrect computation/transaction, but an attacker can cause arbitrary delays for correct computations/transactions to be included.

This article wants to specifically address the Dogecoin-Ethereum bridge, which requires blocks from the Dogecoin blockchain to be verified inside an Ethereum smart contract. This verification is too expensive to be done directly, and because of that, we utilize TrueBit to take the computation off-chain.

Having said that, all analyses are equally applicable to bridges between blockchains where the availability of blocks can be reasonably assumed. This means that it can be used to e.g. offload processing volume from the main Ethereum chain to another blockchain (which might even be a proof-of-authority chain) as long as all participants in that chain rightfully assume that block data is available to all potential challengers.

We call TrueBit-light the protocol that does not make an effort to bring honest participants to the network. We assume that an honest participant is present who is altruistic to a certain degree. This includes keeping up with the Ethereum network, paying the gas fee and having a certain amount of money to pay for an initial deposit.

We also simplify the system to a degree to only allow one parallel task. This might be extended to a constant amount of tasks but would also make the presentation here more complex.

All timeouts are lower bounds and have to be extended in case of Ethereum network congestion. This means that if you want to claim a timeout to take effect, you have to provide a proof that

several previous blocks had enough capacity to include a potential response by the other party. Since TrueBit never makes a claim that state transitions take effect in a finite amount of time, this is still consistent with the theory presented below.

The TrueBit contract has the following properties:

TrueBit consists of the fact claiming component and the verification game. In both cases, we fix a mathematical function $f\colon \Sigma^n \mapsto \Sigma^n$ which can be implemented on a given machine in $s$ elementary computation steps. This is not a big limitation, since $f$ can be an interpreter for another machine, thus allowing to run arbitrary programs, which have a certain finite running time. Limiting the running time is a crucial component, although this limit can be magnitudes higher than what is possible to compute in a single block of the underlying blockchain.

We will start with describing a language that helps us treat the smart contract systems.

**Definition 0.1.** *Let $\mathcal{A}$ be the set of all accounts and $\mathcal{T}$ the set of timestamps (e.g. the natural numbers). We model a smart contract $C$ as a state machine that can receive inputs from $\mathcal{T} \times \mathcal{A} \times I$ (timestamp, sender and input) and acts on this input by changing to a state from $S$ and producing an output from $O$. Here, $I$, $O$ and $S$ are specific to each smart contract type. We identify the smart contract with its state transition function $C\colon S \times \mathcal{T} \times \mathcal{A} \times I \to S \times O$. The function is a partial function, i.e. the machine is able to reject certain inputs. Smart contracts reject any input whose timestamp is not larger than the previously non-rejected input. We also identify $C$ with the iterated state transition function $C\colon (\mathcal{T} \times \mathcal{A} \times I)^* \to S \times O$, where we assume an implied initial state $s_0$ such that $(s_0, o_0) = C(\varepsilon)$. The iterated state transition function is then defined inductively as $C(I_n, (t, a, i)) = C(s', t, a, i)$, where $(s', o') = C(I_n)$.*

*As a shorthand, we write $C\colon s \underset{t,a}{\overset{i}{\Longrightarrow}} s' \mid o$ if $C(s, t, a, i) = (s', o)$.*

*A* strategy *for a player $a \in \mathcal{A}$ is a function from $S \to (\mathcal{T} \times \mathcal{I}) \cup \{\bot\}$ (where $\bot$ means that the player does not make a move). For a strategy assignment $\mathcal{S}\colon \mathcal{A} \to (S \to (\mathcal{T} \times \mathcal{I}) \cup \{\bot\})$, a* game *$g$ in a smart contract $C$ according to $\mathcal{S}$ is a sequence of moves, i.e. $g \in (\mathcal{T} \times \mathcal{A} \times \mathcal{I})^*$ such that $C(g)$ is defined and either $g = \varepsilon$ (the empty game) or $g = g' \cdot (t, a, i)$ such that $g'$ is a game in $C$ where $C(g') = (s, o)$, $\mathcal{S}(a)(s) = (t, i)$ and there is no $a' \in \mathcal{A}$ such that $S(a') = (t', i')$, $t' < t$ and $C(g' \cdot (t', a', i'))$ is defined. The length of the game is called the number of* rounds.

*For a strategy assignment $\mathcal{S}\colon \mathcal{A} \to (S \to (\mathcal{T} \times \mathcal{I}) \cup \{\bot\})$, we write $C \rightsquigarrow_{\mathcal{S}} o$ if for any game $g$ in $C$ according to $\mathcal{S}$ there is some $s$ such that $C(g) = (s, o)$. For a single strategy function $s\colon S \to \mathcal{T} \times \mathcal{I} \cup \{\bot\}$ for a player $a \in \mathcal{A}$ we write $C \rightsquigarrow_s o$ if $C \rightsquigarrow_{\mathcal{S}} o$ for any $\mathcal{S}$ that satisfies $\mathcal{S}(a) = s$.*

**Theorem 0.2.** *For any function $f$ taking $s$ steps to compute, there is an interactive game with two participants $a$ and $b$ implemented by a smart contract $G[a, b, \cdot, \cdot, \cdot]$ with the following properties:*

1. *it takes at most $1 + 2\log_2 s$ rounds and at most $t_G \log_2 s$ time (assuming no network congestion) for some intra-round timeout $t_G$*
2. *for any $x$ and $y$, there is always a strategy $s$ for player $a$ such that $G[a, b, x, f(x), y] \rightsquigarrow_s f(x)$*
3. *for any $x$ and $y$, there is always a strategy $s$ for player $b$ such that $G[a, b, x, y, f(x)] \rightsquigarrow_s f(x)$*

*Proof.* The game will keep the invariant that both players agree on the internal state of the computation at some step $l$ but disagree about the state at step $h$. Note that we can also work with hashes of internal states, so the data sent in each round is not very large.

Initiall, $l = 0$ and $h = s$ and the game halves the distance $h - l$ with every second message (round).

Let $t_0$ be the timestamp at which the game is created. The initial state is

$G[a, b, x, y_a, y_b, t_0](\varepsilon) = (t_0, (0, x), (s, y_a, y_b))$

All following messages have to have a timestamp larger than the one in the state, i.e. we have an implicit requirement that $t > t_p$. Furthermore, we will omit the parameters of $G$ in the following. We will use $\alpha$ for a generic accounts that can be either $a$ or $b$.

If $h - l > 1$, we ask both participants to submit what they think is the internal state at step $\lfloor \frac{h-l}{2} \rfloor$:

$$G[\dots]\colon t_p, (l, s_1), (h, s_a, s_b) \underset{t,\alpha}{\overset{s_2}{\Longrightarrow}} t_p, (l, s_1), (h, s_a, s_b), (\alpha, s_2) \quad \text{for } \alpha \in \{a, b\} \tag{1}$$

$$G[\dots]\colon t_p, (l, s_1), (h, s_a, s_b), (a, s_2) \underset{t,b}{\overset{s_2'}{\Longrightarrow}} \begin{cases} t, (\lfloor \frac{h-l}{2} \rfloor, s), (h, s_a, s_b) & \text{if } s = s' \\ t, (l, s_1), (\lfloor \frac{h-l}{2} \rfloor, s_2, s_2') & \text{otherwise} \end{cases} \tag{2}$$

$$G[\dots]\colon t_p, (l, s_1), (h, s_a, s_b), (b, s_2) \underset{t,a}{\overset{s_2'}{\Longrightarrow}} \begin{cases} t, (\lfloor \frac{h-l}{2} \rfloor, s), (h, s_a, s_b) & \text{if } s = s' \\ t, (l, s_1), (\lfloor \frac{h-l}{2} \rfloor, s_2', s_2) & \text{otherwise} \end{cases} \tag{3}$$

If $h - l = 1$, the smart contract can actually perform the computation:

Let $f(s, i, p)$ be the internal state of the algorithm that computes $f$ after running a single step starting from step number $i$ and internal state $s$ taking into account auxiliary proof data $p$ (the value is undefined if $p$ is malformed or invalid).

$$G[\dots]\colon t_p, (l, s_1), (l + 1, s_a, s_b) \underset{t,\alpha}{\overset{p}{\Longrightarrow}} \bot \mid y_\alpha \text{ if } f(s_1, l, p) = s_\alpha \tag{4}$$

Furthermore, at a certain time $t > t_p + t_G$, a timeout can be claimed:

$$G[\dots]\colon t_p, \cdot, \cdot, (\alpha, s_2) \underset{t,\cdot}{\Rightarrow} \bot \mid y_\alpha \tag{5}$$

$$G[\dots]\colon t_p, \cdot, \cdot \underset{t,\alpha}{\Longrightarrow} \bot \mid y_\alpha \tag{6}$$

Let us now analyze the number of rounds of the game in the worst case. Note that timeouts (i.e. mesages of type (5) or (6)) can directly end the game from any state. A message of type (1) followed by either (2) or (3) reduce $h - l$ roughly by half. Apart from timeouts, these are the only messages possible until $h = l + 1$. At that point, only message (4) is possible.

This means that if there are no timeouts, the game will require $1 + 2 \log_2 s$ messages.

Note that the timeouts for messages of type (1), (2) and (3) all start at the same time. This mean that both parties have $t_G$ time to perform the magnitude reduction of $h - l$. If this takes longer

than $t_G$, anyone can step in and end the game. This means that the game will take at most $t_G \log_2 s$ time (assuming there is an actor who will trigger the timeout).

Finally, we argue why both players have a strategy to end the game with $f(x)$. Due to symmetry, we only argue for player $a$.

Obviously, by responding in time, $a$ can always avert the situation that the game ends with a timeout in a state different from $f(x)$.

If the current state of the game is $t_p, (l, s_1), (h, s_a, s_b)$, the strategy is to send a message that contains the internal state of the algorithm computing $f$ at step $\lfloor \frac{h-l}{2} \rfloor$. In doing so, the smart contract will end up with a state $t_p, (l, s_1), (l+1, s_a, s_b)$ where $s_1$ is the state at step $l$ and $s_a$ is the state at step $l+1$. Since $s_b \neq s_a$ and the algorithm computing $f$ is deterministic, $b$ cannot use a message of type (4) to turn the smart contract into state $y_b$. Instead, $a$ uses (4) to make the smart contract output $y_a = f(x)$. □