The *CryptoPolitics* chest opening, a pseudo-random, fair, traceable and reproducible card selection process

Jérémie Albert and Pierre Etchemaïté

inBlocks

March 3, 2022

Abstract

In the domain of online gaming in which randomness has a major impact on in-game events, the player acceptance is probably highly correlated to the reliability of processes in which randomness is used. The understanding of the in-game processes which involves randomness and the ability to reproduce those processes is a key factor to reach very large players acceptance. The trust ensuing this acceptance is yet more critical when the game and its in-game processes are creating digital assets that can exist on a distributed ledger (a.k.a blockchain technology) and that can bring value to its owners. In this paper we describe how, in the context of the game *CryptoPolitics*, the cards selection algorithm that is run at chest opening time and, more importantly, how players (or anyone actually because data is made publicly available) can repeat the operation by himself on its own execution environment to reproduce what happened online. We also show how pseudo-randomness is set and how consistency between talon operations can be verified (still being impossible to anticipate for players). Card selection algorithm is described here and all the results of its online computation is stored into a Precedence register to make it immutable, provable and undeniable using the inBlocks SaaS platform.

Keywords: cryptopolitics, blockchain, fairness, randomness, traceability, nft

Contents

1 Definitions

CryptoPolitics[1] is a card game, it features a talon and cards owned by the different players. Owned cards can change according to game rules, but return to their original state when reinserted into the talon. We will only focus on the talon life-cycle here:

- A talon contains several classes of cards: common, rare, epic, legendary.
- Each class is made of several series of cards (each being related to a politic personality, typically).
- Each series contains a given number of cards, numbered starting with #1.
- Players do not take cards directly from the talon, but instead get chests that contain sets of cards whose classes are determined by the type of the chest. The exact cards in the chest are only known when the chest is open though. At that time, the cards it contains are determined, taken off the talon and given to the player.
- As stated before, players can put cards back into the talon, in their original state.

2 Notations

Let's name $\mathbf{U}$ the set of all the cards and $\mathbf{C}$ the set of card classes:

$$\mathbf{C} = \{common, rare, epic, eternal, legendary\}$$

Given C a class of cards: $C \in \mathbf{C}$
C contains card series S_i^C with $i \in (1, m)$; Within a series, a card is identified by an index:

$$S_i^C = \{c_1^S, c_2^S, ...c_n^S\}$$

All the series of the class C partition that class:

$$\begin{cases} C = \bigcup\limits_{i=1}^{m} S_i^C \\ i \neq j \implies S_i^C \cap S_j^C = \emptyset \end{cases}$$

So, the set of all the cards is exactly the union of all the card series

$$\mathbf{U} = \bigcup_{C \in \mathbf{C}} \bigcup_i S_i^C$$

A time t, the talon $T(t)$ is a subset of the existing cards: $T(t) \subseteq \mathbf{U}$

$T(t)$ can also be partitioned in classes and series:

$$T(t) = \bigcup_{C \in \mathbf{C}} \bigcup_i I_i^C(t)$$

with $I_i^C(t) = \{i_1^S, i_2^S, ...i_n^S\} \subseteq S_i^C$

3 Talon transformations

A talon changes from one state to the next by applying a sequence of transformations. The current version of the game requires four transformations:

$$t_i \in \begin{cases} init : \emptyset \times \mathcal{P}(\mathbf{U}) \to \mathcal{P}(\mathbf{U}) \\ extend : \mathcal{P}(\mathbf{U}) \times \mathcal{P}(\mathbf{U}) \to \mathcal{P}(\mathbf{U}) \\ draw : \mathcal{P}(\mathbf{U}) \times \mathbf{C} \times r : \mathbf{N} \to \mathcal{P}(\mathbf{U}) \\ reject : \mathcal{P}(\mathbf{U}) \times c : \mathbf{U} \to \mathcal{P}(\mathbf{U}) \end{cases}$$

$$T(t+1) = t_n\left(t_{n-1}\left(t_{n-2}...t_1\left(T\left(t\right)\right)...\right)\right)$$

[1] https://cryptopolitics.com/

$init$ and $extend$ are two special transformations; $init$ is used to create a talon (so $init$ can and should only be used once, as the first transformation):

$$init(\emptyset, T_0) = T_0$$

$extend$ merges new cards in an existing talon:

$$extend(T, T') = T \cup T'$$

To $draw$ a card, a card selected based on a random seed r is removed from the talon:

$$selection : \mathcal{P}(\mathbf{U}) \times \mathbf{C} \times r : \mathbf{N} \rightarrow \mathbf{U}$$
$$draw(T, C, r) = T \setminus \{selection(T, C, r)\}$$

$reject$ transformation just consists in putting a card back into the talon. In the *CryptoPolitics* game, this operation is also called $unmint$.

$$reject(T, c) = T \cup \{c\}$$

We want to be able to draw cards off a given card class, preserving the probability distribution of series of drawing cards uniformly from the class, but always selecting order numbers drawn so as to maximize the rarity of low card order numbers.

First, we execute a random weighted selection of a series based on the number of cards of each series present in the talon; Given a random integer r picked uniformly in the interval $0 \leq r < |T(t)|$,

$$\exists s \text{ unique such as } \sum_{i=1}^{s-1} |I_i^C| \leq r < \sum_{i=1}^{s} |I_i^C|$$

Cards of a given series are then selected in increasing order number until they have all been dealt, and from then on we always select cards (which must now be cards that were rejected) with highest order number first. This can be implementing by adding some "counter" k_S to each series; when it has reached the last card of the series, previously rejected cards start being selected:

$k_S(0) = 0$ then

$$\begin{cases} k_S(t) < |S| \iff \text{selection} = k_S(t) + 1 \text{ and } k_S(t+1) = k_S(t) + 1 \\ k_S(t) = |S| \iff \text{selection} = max\{j | e_j^C = 1\} \text{ and } k_S(t+1) = k_S(t) \end{cases}$$

4 Provability

To demonstrate the lack of bias in the way cards are selected during chest opening, anybody should be able to reproduce it, using its own execution environment. To achieve this, several information are made public:

- The talon history to date;
- The sequence of transformations executed between each state;
- The algorithm used to get provable random seeds;
- The source code of the transformations used.

Data should not only be public, but also irrefutable. Nowadays this is best achieved using blockchain technology. At inBlocks we use the open-source solution we designed name Precedence[2] that is embedded in our platform. This way, the data is immutable, timestamped and we can provide an undeniable proof-of-existence.

[2] https://github.com/inblocks/precedence

4.1 Talon history

Since the whole talon history must be persisted and made public, a compact representation of each talon state is important.

The set of cards left in the talon can be subdivided by class and by series; The set of cards left in a series can be described in a compact way using a binary notation:

Given $E_i^C(t) = \{e_1^C, e_2^C, ..., e_n^C\}$ with $e_j^C \in \{0, 1\}$ such that

$$\begin{cases} s_j^C \in I_i^C \implies e_j^C = 1 \\ s_j^C \notin I_i^C \implies e_j^C = 0 \end{cases}$$

All $E_i^C(t)$ are sufficient to determine $I_i^C(t)$, and can be represented as a bitmap.

The counter k_S used to select the next order number to deliver from each series is also part of the talon state.

$k_S(0) = 0$ then

$$\begin{cases} k_S(t) < |S| \implies \text{selection} = k_S(t) + 1 \text{ and } k_S(t+1) = k_S(t) + 1 \\ k_S(t) = |S| \implies \text{selection} = max\{j|e_j^C = 1\} \text{ and } k_S(t+1) = k_S(t) \end{cases}$$

4.2 Sequence of transformations

The transformation(s) being executed between each consecutive states must have be made public, with a reference to the talon state they start from, their order, their parameters, the random seed used, their results, and a reference to the resulting talon state.

With all those information but the resulting talon state itself, it is possible to recompute and check the results; We provide a endpoint that will do just that, and return whether there's a match.

It is also possible to compute the resulting state and compare it against the next persisted state and the consecutive talon state can be verified by running the card selection software available online[3].

4.3 Source of randomness

The chest type to card classes mapping and the selection algorithm depend on a source of (pseudo) randomness, proving that cards are drawn fairly require the use of a source of randomness that's both uncontrollable and publicly registered.

Some services provide such public entropy sources, like NIST Randomness Beacon[4]. It is also possible to use public immutable information, like confirmed blocks of a blockchain, as a public source of pseudo-randomness.

However, sources listed above publish new entropy at a slow pace, and in general depending on an external resource adds unavailability risks. For those reasons, we decided to go with a simpler solution, and use repeated application of the SHA-256 hash on an initial high quality random seed kept secret; The sequence will be then served in reverse order so as to not be guessable by third parties, yet provable (each new seed will simply hash to the previous one using SHA-256). This scheme has been suggested before[5], and has some known limitations:

- one has to set in advance how many seeds will be generated;
- the generator will eventually loop, but using a cryptographic hash like SHA-256 it's likely to be a very long cycle;
- there's some loss of entropy with repeated hashing, but the total loss is logarithmic and only a long term concern[6].

We decided to go with 10M seeds, which should be enough for several years of playing *CryptoPolitics*, and is way below the number of iterations when either of the last two limitations should become an issue.

Computing and keeping 10M SHA-256 hashes in memory has proven astonishingly fast on modern hardware (around 6 seconds on a laptop), suggesting that a purely in-memory solution could be done, removing the complexity and risks of persisting secrets. However neither waiting 6 seconds for each request nor using 320MB of memory for storing all precomputed hashes was adequate for a micro-service. Instead, we compute all hashes when the service is started, but only persist some sampling of the generated seeds, typically 1 in 1000 seeds; This reduces the memory requirements by a factor 1000, but puts an upper bound for the cost of computing any seed from the closest sampled

[3] https://github.com/cryptopolitics/talon-oracle

[4] https://beacon.nist.gov/home

[5] See for example https://bitcointalk.org/index.php?topic=922898.0

[6] https://crypto.stackexchange.com/questions/24660/the-effect-of-truncated-hash-on-entropy/24672#24672

seed to 999 hashes, that can be done in around 0.6ms; Even keeping only a sample of the hashes in memory reduced the initialization time compared to keeping every hash by more than a half. As a extra refinement, since seeds will be served in reverse order, the result of computing a seed from closest sampled seed can be cached to serve the next seeds.

A standard `java.util.Random` object seeded using the generator described above is then used through the execution of all the transformations from one persisted talon state to the next: selection of the number of cards of each class that a chest will contain, and weighted selection of a series in each card class, etc. for each transformation of the list in order that uses some randomness.

4.4 Source code

The Java implementation of the micro-service computing talon states can be found on GitHub[7]. Namely,

- The random seeding can be found in the class `SeedService`, called from method `SeedGeneratorContext::getRandom`;
- The chest type to card classes mapping is implemented by the method `OpenChestTransformation::getCardClasses`;
- the selection algorithm is separately implemented by the methods `CardClass::pickNextCard` (weighted selection of the series) and `CardSerie::pickNextCard` (initial deal of cards in increasing order number then selection of discarded card of highest order number remaining).

OpenAPI documentation can be build from this project. A prebuilt version is also available in the docs subdirectory. To note,

- the endpoint `/talon/applyTransformations` can compute the effect of a list of transformations on a talon (results, seed used if any, new talon state and new seed generator state);
- the endpoint `/talon/checkTransformations` can take a talon history document (say from a public repository, see section 5 below) and return HTTP status 200 if the results it contains match the computed results, or HTTP status 409 otherwise.

The project also contains a demo Bash shell script `deplete_deck_of_cards.sh` that takes all cards out of a standard deck of 52 cards (a unique class containing a serie for each color), then reinserts them in random order.

5 Public data

5.1 Cryptopolitics talon history

The transformations applied to the talon can be found in the inBlocks SaaS platform and is publicly available. Direct access to the last talon transformation[8] and to a specific transformation[9] is made possible.

The returned JSON documents contain:

- the `version` of the document, that is incremented with each set of transformations;
- the state of the `talonBefore` the transformations, subdivided by classes and series. For each serie, the document contains the `size` of the serie (the number of cards it contains), a base64 encoded `setBitmap` representing the cards of the serie still in the talon as described before, and an `initialDealIndex` used to deal all the cards at first;
- the list of the `transformations` to apply, in order, with their parameters;
- the parameters of the seed generator used for those transformations. If a seed was used, it is also made public here;
- the list of the `results` returned by each transformation. Results list is ordered like the transformation list.

[7] https://github.com/cryptopolitics/talon-oracle

[8] https://api.inblocks.io/civicpower/cryptopolitics/precedence/chain/talon_cryptopolitics/data

[9] https://api.inblocks.io/civicpower/cryptopolitics/precedence/record/{hash(N)}/data, {hash(N)} for $N \in \mathbf{N}$ being the hexadecimal representation of the SHA-256 hash of the string "talon_cryptopolitics_vN"

Note that since only the `talonBefore` the last transformations is persisted, the current state of the talon is only available thru computation by applying those last transformations.

The transformation `types` are

- `Init` ($init$), with parameters `setup` and `seedGenerator`
- `ExtendTalon` ($extend$), with parameter `additionalCards`
- `PickCards` (vectorized version of $draw$, unused), with parameter `cardClasses`
- `AddCards` (vectorized version of $discard$), with parameter `cards`
- `OpenChest` (specialized version of `PickCards`, with card classes determined by game rules based on its parameter `chestType`)

Being the source of truth for the Ledger, transformation parameters may contain extra information not used by the talon transformations themselves.

Precedence's proofs of ownership of those documents can be found by removing the `/data` suffix from URLs referring to the transformation[10],[11].

[10] https://api.inblocks.io/civicpower/cryptopolitics/precedence/chain/talon_cryptopolitics

[11] https://api.inblocks.io/civicpower/cryptopolitics/precedence/record/{hash(N)}, with {hash(N)} computed as before