# Provable Fairness

Lujing Cen, Gloria Fang, Andrea Jaba
{lujing, yfang, adjaba}@mit.edu

## 1 Introduction

Our project focuses on provable fairness, a relatively new emergence in the domain of online gambling that has not been well studied. The necessity for provable fairness arises from the fact that it would be very easy for an online gambling service to gain an unfair advantage through manipulating the outcome of a supposedly randomized game.

We seek to survey the current landscape of provable fairness on the Internet and analyze them accordingly in a generalized protocol. We will show a few practical and theoretical attacks against existing provable fairness schemes or relaxed versions of those schemes from the perspective of both the client and the server. Our analysis methods will mostly rely on a site's public protocol.

In addition, we propose a provable fairness scheme that addresses some of the issues that we have uncovered while surveying existing protocols. We will prove that our scheme satisfies a strict definition of provable fairness and demonstrate how a few common game types could be efficiently represented in this scheme.

### 1.1 Significance

With advances in cryptography and technology, many people have migrated from traditional gambling to online gambling with cryptocurrencies. In fact, the size of the Bitcoin gambling market is estimated to be over 4 billion dollars since its rise in 2014 [1]. A similar emergence has been seen in the gaming industry, where the markets for loot boxes and skin gambling are estimated to hit 50 billion dollars by 2022 [2].

For these sizable markets, the consequences of cheating could be significant. Most gamblers only play because they feel that they have a fair chance of winning – or at least aren't at an unexpected disadvantage. Likewise, the gambling sites would like to believe that they are able to make some profits in expectation. This market exists on a certain amount of trust between the players and the house. The games being played should be fair (with regards to the advertised house edge) and have enough randomness in their outcomes.

The challenge arises in developing schemes for different game types such that no party can gain an unfair advantage. These schemes must also present some form of uncertainty so that players are incentivized to participate. A vulnerability in these schemes that allows any party to gain even a slight advantage could result in huge wins or huge losses over the course of many transactions, which can happen more anonymously and in much larger volumes compared to traditional gambling.

## 1.2 Background

Most gambling sites that want to establish trust with their players use cryptography and the idea of provable fairness. An example exchange is as follows. Suppose you are playing on a server. First, the server commits to a random seed. The server does not reveal the seed immediately, but you want to be sure the server does not change its commitment after the exchange is over. To prove that the seed is not changed during a given transaction, the server gives you $h$(server seed), where $h$ is a hash function.

Next, you pick a client seed. This client seed could be generated randomly by the server, but you must have the option of modifying it to whatever you want. The important observation at this step is that server cannot know of this client seed beforehand, otherwise it may choose a server seed that maximizes its profits unfairly.

Once the server receives the client seed, it will compute another function that uses both the client seed and server seed to determine the outcome of the game. This function must be deterministic and publicly verifiable.

At the end of the game, the server will publish the result of the game as well as its seed. You can verify that the server did not change its initial commitment by hashing the server seed. Furthermore, you can apply the deterministic function to ensure that the outcome is consistent with the given server seed and client seed [3].

Some benefits of this system include being able to verify that both parties are not cheating and helping detect security breaches. For example, if the server's published seed does not match the hash value it provided initially, it is possible that the server is not trustworthy or that a third party has tampered with the results.

Of course, this notion of provable fairness relies on the security of the hash function, as well as the generation of initial seeds. There are many possible exploitations during every step of the exchange. For example, if the client seed being generated is predictable, the server could select an initial seed value that deliberately puts the client at a disadvantage.

## 1.3 Previous Works

There have been studies on how to create a provably fair gaming algorithm. A paper by Podulszló [15] splits the seed into two: `hostSeed` and `publicSeed`. Before the start of each game, the host will have to commit to a `hostSeed` and display the commitment to all players, who only afterwards will generate and contribute to the `publicSeed`. A combination of these two seeds will then be used as an initialization parameter for randomization to reduce opportunities for manipulation. According to this paper, a provably fair algorithm is also a deterministic algorithm that will return the same output with the same input. It must ensure that the integrity of the `hostSeed` can be verified by all participants and is public to every participant of the game. In generating a sequence of random outputs, a cryptographic nonce should be used – once for each seed set, to produce unique outputs for consecutive bets using the same seeds.

While provably fair schemes guarantee that the server does not change seeds based on client seeds, it is not a guarantee that the server seeds are generated at random. Shufflepuff is a theoretical tool that casinos can use to optimize winnings against players, and it operates on the fact that there are initial seeds that give more advantage (in addition to the advertised house edge) to the server regardless of the value of the client seed. These advantageous seeds are easily found by optimizing for different winning scenarios when the seed space does not cover the arrangement space [13].

Weak random number generators can also undermine provably fair algorithms. In 2015, CSGOJackpot, a gambling website, was exploited in such a way that the client can determine the "winning percentage," which is analogous to the server seed. This is because the server gives the client $hash$(blinding, "winning percentage") before each game as part of provable fairness, and both blinding and "winning percentage" came from calls to their weak random number generator. Since the winning ticket is simply the "winning percentage" multiplied by the total number of tickets, an attacker with this information can manipulate the game to his/her advantage, which is also not fair [14].

## 2 Definitions

We first seek to formally define provable fairness in the context of cryptography. Let Alice, who is the server/house, be denoted by $A$. Let Bob, who is the client/player, be denoted by $B$. Note that there are games involving multiple players, but our project is mainly concerned with two-player games.

Alice picks some seed $s_A$ and commits using a proof $p_A = C(s_A)$ using some public function $C$. Bob receives $s_A$, and chooses some seed $s_B$. Then, Bob sends $s_B$ to Alice, who computes the result of the game $r = G(s_A, s_B)$ using some deterministic and public game function $G$. Finally, Alice sends Bob $r$ and $s_A$. Bob will verify that $V(p_A, s_A) = 1$ using a verification function $V$ that returns 1 if and only if $p_A$ is a valid proof for $s_A$. In addition, he will check that $r = G(s_A, s_B)$, then conclude that the game was fair.

One might notice that $C$ resembles the commit step of a commitment scheme. However, all existing provable fairness schemes simply choose $C$ to be a hash function even though a provable fairness scheme is certainly possible with another commitment scheme. One potential explanation is that $C$ needs to be simple to verify, and many commitment schemes do not have easy-to-use online tools for verification.

The seeds for both parties are always chosen from seed spaces $\sigma_A$ and $\sigma_B$, each with finite size, such that $s_A \in \sigma_A$ and $s_B \in \sigma_B$. The seed spaces are always defined by the protocol and can vary greatly. Although it is not strictly necessary for $|\sigma_A| = |\sigma_B|$, a protocol is only provably fair if is computationally infeasible for one party to select a seed that produces noticeable deviations from the expected house edge when the other party chooses their seed uniformly at random.

This criterion ensures that if one party adopts a uniformly random strategy for selecting seeds, the outcome of the game cannot be unfairly skewed in favor of the other party. However, this criterion does not necessarily force the outcome of the game to be random. We consider a scheme to be provably fair if it satisfies this criterion.

## 3 Existing Schemes

In this section, we present our analyses of existing schemes which are of some interest in that they are exploitable from the perspective of the client or the server. Many of these schemes use an incrementing nonce $n$ in addition to $s_A$ and $s_B$. Thus, the server reveals $s_A$ only after a certain $n$ that is usually decided by the client. This means that $s_B$ contains the nonce and that $r = G(s_A, s_B)$ is a sequence of verifiable results rather than a single result. Other variations of the generalized scheme will be presented in the individual sections.

### 3.1 Proof Evasion

Suppose Alice chooses a function $C(s_A)$ but never actually reveals $s_A$ or the method by which to verify $C(s_A)$. Then, Alice convinces Bob to compute $V(p_A, s'_A)$ where $s'_A \neq s_A$. Furthermore, she convinces Bob to accept the fact that there is no public game function at all. While this oversight is evident in this context, it is less evident to users who are not well versed in provable fairness. One such service that uses this exploit is MysteryBrand [4].

We provide a more concrete version of the protocol [16] as used by the service. Let $H$ be the MD5 hash function. The server commits to a value $C(s_A)$, likely to be $H(s_A)$. The client provides the server $s_B$. The server may compute the result using a game function, but this function is not revealed. The player is only given the result along with $H\left(s_B \mid\mid H(s_A)\right)$, an irrelevant proof. While it is certainly possible to verify that the proof resulted from hashing the concatenation of the two values, it by no means proves the outcome of the game since there is no public game function.

In effect, this protocol is not very different from one without any provable fairness. The criterion of provable fairness is violated because the server can simply choose any $s_A$ and game function. Whatever the expected house edge is, the server can easily manipulate the results as the result computation is entirely opaque to the client.

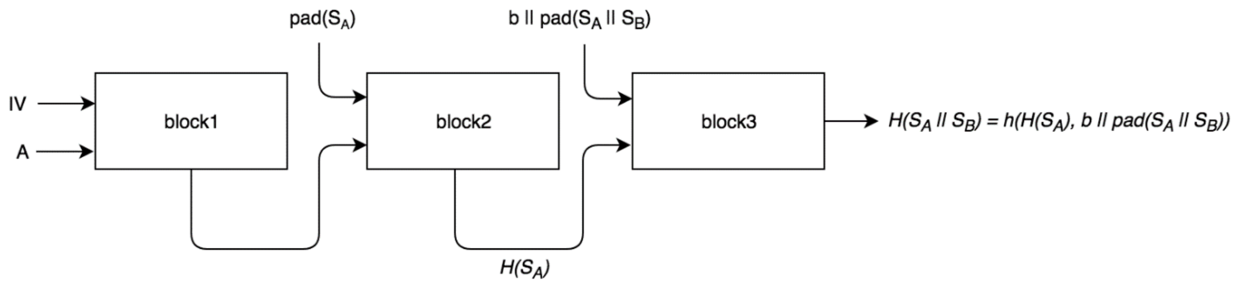
## 3.2 SHA Length Extension

Let $H$ be the SHA-512 hash function. Suppose Alice chooses $\sigma_A$ to be the set of all possible 128-byte strings and $C(s_A) = H(s_A)$. She also chooses $G(s_A, s_B) = H(s_A \mid\mid s_B)$. This is a relaxed version of the scheme used by Bitsler [5]. We show that when $\sigma_B$ permits strings containing more than 128-bytes, Bob can efficiently manipulate the result of the game by exploiting the Merkle–Damgård construction of SHA-2 [6].

This client-based attack is possible if $|s_A|$ is known and fixed, which is generally the case in provably fair schemes. We will let $h(a, b)$ be the single-block SHA-512 hash function that does not apply any padding to its inputs. In the context of the Merkle–Damgård construction, the two inputs to $h$ are the previous block's output hash and the current message block. Note that the block size for $h$ is 1024 bits. We work through the steps of the game interaction.

First, the server sends the client $C(s_A) = H(s_A)$. However, by the specifications of SHA-2, when $s_A$ is less than a block size, it will be padded with `0b100…0` followed by a 64-bit integer representing the message length. When $s_A$ is exactly a block size, there will be an entire block of padding using the same scheme. Given that $|s_A|$ is exactly one block, the value sent to the client is $H(s_A) = h\big(h(IV, s_A), \text{pad}(s_A)\big)$ where $IV$ is the public initialization vector of SHA-2.

To perform the exploit, the client will select $s_B = \text{pad}(s_A) \mid\mid b$ where $b$ is an adversarial seed chosen specifically to skew the result of the game in favor of the client. The client knows that the server will compute the game function as $H(s_A \mid\mid s_B)$ and will not change its commitment to $H(s_A)$. Thus, $H(s_A \mid\mid s_B)$ becomes $h\big(H(s_A), b \mid\mid \text{pad}(s_A \mid\mid s_B)\big)$ under the adversarial seed. Note that it is easy to compute $\text{pad}(s_A \mid\mid s_B)$. Namely, the client controls the length of $s_B$, thereby controlling the length of $s_A \mid\mid s_B$, which is at least two blocks long. We assume that the padding of the entire message along with $b$ does not require a fourth block.



The client simply enumerates different $b$ values until the result is desirable. In expectation, under the random oracle model, it should take no more than a few tries in simple dice games to get a roll below or above a certain threshold. The outcome of the dice roll is usually determined by interpreting some bits of the output hash as a number, which should be more or less uniformly distributed as $b$ varies.

One potential defense which is employed by Bitsler is to insert a separator in between the server seed and client seed. For example, the game function is computed as $H(s_A \mid\mid , \mid\mid s_B)$. This prevents the attack described above, because the binary representation of a comma does not begin with a 1, and thus the client is unable to compute $H(s_A \,||\, , \,||\, s_B)$ using $H(s_A)$ since the client is not able to inject $\text{pad}(s_A)$ directly after $s_A$. Another potential solution is to simply restrict $\sigma_B$ to contain only alphanumeric characters (since none of these characters can constitute the first byte of padding) or to restrict the length of the seed to be small enough such that $|\sigma_B \,||\, \text{pad}(\sigma_A \,||\, \sigma_B)|$ is less than 128 bytes. However, concatenating the client seed after the server seed is somewhat risky, and could present vulnerabilities if incorrectly implemented.

## 3.3 HMAC Key Exposure

Let $H$ be the SHA-512 hash function. Suppose Alice again chooses $C(s_A) = H(s_A)$, but naively chooses $\sigma_A$ to be the set of all possible $a$-byte strings where $a > 128$ bytes. She also chooses the function $G(s_A, s_B) = MAC(s_A, s_B)$, where $MAC$ is the hash-based message authentication code (HMAC) that uses SHA-512 and follows the specifications of RFC 2104 [7], which is used in most implementations. This scheme is a relaxed version of the scheme used by Fair-Dice [8] and BitDice [17]. Bob can efficiently manipulate the result by exploiting the fact that Alice has effectively leaked the key to the HMAC function.

The definition of the HMAC function given a key $K$ and message $m$ is shown below.

$$MAC(K, m) = H\big((K' \oplus opad) \,||\, H(K' \oplus ipad) \,||\, m\big)$$

$$K' = \begin{cases} H(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

A server employing this scheme will usually choose $G(s_A, s_B) = MAC(s_A, s_B)$. If the client observes $H(s_A)$ and knows that that $|s_A| > 128$ bytes, then the client now has knowledge of $K'$ in the HMAC function. The values of $opad$ and $ipad$ are public. Thus, the client has obtained all the necessary values of the game function to select an advantageous client seed $s_B$. Like the previous attack, the client simply brute forces a few values of $s_B$ until the output of the game function is desirable.

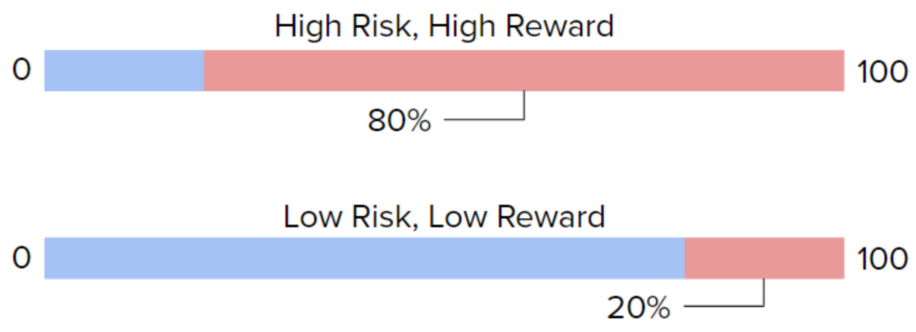
There are a few different defenses to this attack. For one, the server can simply choose $|s_A|$ to be exactly the block size of the hash function, which in this case is 128 bytes. Assuming $s_A$ is derived from a cryptographically secure source, there is no reason to choose a seed which is above 128 bytes, since the hash function only provides around 64 bytes of security. Another security mechanism that is usually employed is to select a different hash function $H'$ and use that as the commitment function $C$. Even if the size of $s_A$ is above 128 bytes, if $H(s_A)$ cannot be easily computed from $H'(s_A)$, then this attack is ineffective.

It is also possible to use $G(s_A, s_B) = MAC(s_B, s_A)$, since $H(m)$ contains padding and the length of $s_A$. Revealing $H(m)$ does not permit the client to easily compute $MAC(s_B, m)$, since the client cannot force the contents of the outer hash function to be the same length as $m$.

## 3.4 Reversed Commitment Order

Suppose Alice manages to convince Bob to provide $s_B$ before she provides $C(s_A)$. We will show in this case how Alice can gain a slight advantage over Bob (in excess of the advertised house edge) using very little computational power. One example of a site employing this flawed scheme is FortuneJack [11].

One additional aspect of the scheme used by FortuneJack is that the client can define what constitutes the winning outcome. After providing $s_B$ and receiving $C(s_A)$, the client can freely choose the corresponding winning range and thereby the payout. The server fixes a house edge, so that defining one of the winning range or the payout automatically fixes the other value such that in expectation, the house edge is achieved.



The client will generally pick between two game variants. The first is a high risk, high reward game. In this game type, the client wins in less than 50% of the output space. The other option is a low risk, low reward game where the client wins in more than 50% of the output space. An example of this can be seen in the image above. By observing the public log for some time, we noticed that a vast majority of players choose the high risk, high reward game, which is perhaps more in the spirit of gambling.

Undoubtedly, the server has more clear data regarding client preferences. If the server can target clients who play the high risk, high reward game, it is able to skew results in its favor by selecting an advantageous $s_A$ given its observation of $s_B$. The reason is that if the client wins in less than 50% of the output space, there is a range in the middle for which regardless of which side the client chooses, the server will always win. For the example in the image, the client wins if $G(s_A, s_B) \in [0,20)$ or $G(s_A, s_B) \in [80,100)$.

The client does not always switch its seed after every roll. Instead, the game function is computed with an incrementing nonce over many rolls. This complicates the attack slightly, but still allows for noticeable deviations assuming a reasonably bounded maximum nonce value. The server can simply compute the average maximum nonce value for a given client and use that as is target nonce.

Given both the client's preference for the high risk, high reward game and a bounded nonce $n^*$, the server simply tries a few different hash values and computes the function below. Note that $G(s_A, s_B)$ will output a list of rolls of length $n^*$.

$$v(s_A, s_B) = \sum_{i=0}^{n^*-1} (G(s_A, s_B)[i] - 50)^2$$

$$\arg\max_{a} f(x) = \{\, v(a, s_B) \mid a \in \sigma_A \,\}$$

In effect, the server is looking for a seed value $a = s_A$ such that for a given $n^*$ and $s_B$, there are more numbers close to 50 than would be in expectation. The value function $v(s_A, s_B)$ is just one example of what the server might be optimizing for. A more sophisticated attacker would use the behavior of the client to adjust the value function.

It is not feasible to get the most optimal value $a \in \sigma_A$, since the space of all seeds is too large. However, it is enough for the server to select a small subset of $\sigma_A$ and look for the best $a$ within that subset. To reduce suspicion, the server could simply generate random $a$ values and compute the value functions for these seeds up to a certain time threshold, so that it is harder to detect foul play.

There is not much the client can do to defend against this attack aside from choosing to play the low risk, low reward variant of the game. Proving that the server is employing such a scheme would be equally difficult without access to the entire history of rolls, which is not available to the public. Testing the random number generator would require a large amount of cryptocurrency, which might be out of reach for most clients.

## 3.5 Skewed Seed Space

Suppose Alice restricts or convinces Bob to select from seed space where $|\sigma_B|$ is very small compared to $|\sigma_A|$. This is the case with BGaming [18], whose client seed space is around 300, and Diceum [12], whose default client seed space is around $10^{10}$. We will show that Alice can select a subset of $\sigma_A$ that performs strictly better assuming Bob chooses a seed uniformly at random from $\sigma_B$, which violates the criterion of provable fairness.

We will first consider the provably fair poker scheme in use by BGaming. The server commits to an initial ordering of the deck. Note that in this case, there are 6 standard decks used in the shuffle. Thus, $|\sigma_A|$ is around 312!, including duplicate permutations. The server permits the client to cut the deck once. Thus, the seed space of $\sigma_B$ is only 312. We can immediately see that it is very difficult for the client to brute force all $s_A \in \sigma_A$ in search of an optimal client seed. However, it is relatively easy for the server to perform the same attack.

The server assumes that the client is using a relatively optimal Basic Strategy for blackjack and is uniformly choosing a seed from $\sigma_B$. Furthermore, the server will reshuffle the entire deck after every game. The rules of the game are as follows: dealer stands on soft 17, double down after splitting is allowed, and 3:2 payout. This attack does not rely on a specific rule set, although for the purposes of demonstrating an actual implementation, we will use this set of rules. Note that it has been shown that with the Basic Strategy, the house edge can be reduced to around 0.42% [20].

We used a blackjack simulator [21] with the Basic Strategy from [19] and adapted it to show an example attack. The server will randomly choose 100 initial shuffles of the deck and evaluate each deck against a player employing the Basic Strategy for all possible $s_B \in \sigma_B$. For each deck, the server computes the expected house edge. Finally, the server selects the best ordering from the 100 initial shuffles that maximizes the expected house for an $s_B$ chosen uniformly at random.

We first tested the implementation using just random initial shuffles to ensure that the house edge was sufficiently close to the actual house edge. Indeed, the result comes out to slightly less than 0.42%. Under our attack however, the server can consistently choose a deck ordering that seems random but achieves a house edge of over 10%. Of course, the server can perform even better if it could simulate more initial shuffles in a short period. For 100 shuffles, our Python implementation takes around 1 second, which means that such an attack is very feasible.

This problem is not unique to deck shuffles. Any sufficiently small seed space should be exploitable. For the scheme employed by Diceum, if the client chooses a seed that is within the seed space of size $10^{10}$, we estimate that the server could distinguish the better of two hashes using around two hours of computation time. Another interesting property of this attack is that it can be precomputed and does not rely on a specific client seed as the input. Thus, the server could simply precompute many optimal server seeds and select from those more often, thereby biasing the outcome from the expected house edge.

## 3.6 Mersenne Shuffle

Suppose Alice uses the Mersenne Twister pseudorandom number generator (PRNG) as one component in the game function to compute an initial shuffle of a deck. We will show how Bob can exploit these shuffles to gain a significant advantage in games, often by fixing $s_B$ to be a value such that the initial deck ordering is recoverable. A few sites that are potentially using this scheme include BGaming [18], Coin Royale [22], and Nitrogen Sports [23].

Consider the code fragment below in JavaScript, which represents one implementation of a shuffle using the Mersenne Twister. Note that lines 3 through 6 represent the Fisher-Yates shuffle [9], which swaps different cards by querying the PRNG.

```
1. function shuffle(deck, seed) {
2.   let mt = new MersenneTwister(seed);
3.   for (let i = deck.length - 1; i > 0; i--) {
4.     let j = mt.int32() % (i + 1);
5.     [deck[i], deck[j]] = [deck[j], deck[i]];
6.   }
7. }
```

There is one distinct problem with the shuffle method as above. Each seed corresponds to only one permutation of the deck. Thus, if the seed space is sufficiently small, the client is effectively able to brute force for the initial seed given an observation of a few cards in the deck. Many implementations still rely on the Mersenne Twister with a 32-bit seed. However, $2^{32}$ is very feasible to enumerate on a modern CPU within a few hours.

We propose the following attack that the client can employ. First, observe the first six cards for a given initial shuffle. Then, go through all $2^{32}$ seeds to determine which seed is consistent with the observed ordering. If the Mersenne Twister in combination with the Fisher-Yates shuffle generates decks that are somewhat uniformly random in the output space, then there should only be one seed consistent with the observation.

$$\frac{52!}{(52-k)!} \geq 2^{32} \wedge k \in \mathbb{Z} \Rightarrow k \geq 6$$

We tested this idea using a C++ implementation. We seeded the Mersenne Twister with the number 42 and searched through all $2^{32}$ possible initial seeds to see how many were consistent with the first 4, 5, and 6 cards respectively. The computation took around 4 hours to run on a single thread. The results are shown in the table below. Indeed, we see that for 6 cards, only one seed is consistent with the observation.

| Matches First | Number of Seeds |
|---|---|
| 4 | 666 |
| 5 | 14 |
| 6 | 1 |

One defense against this attack is to seed the Mersenne Twister using a 64-bit seed, which will make it more difficult for the client to brute force the seed space. However, a slightly better alternative would be to fully seed the Mersenne Twister's internal state with a cryptographically secure source. However, this is less practical, as the Mersenne Twister has a relatively large state buffer of 2.5 KiB, which might be challenging for the server because entropy is not cheap when there are many players.

However, extracting the Mersenne Twister to a global state does not fully solve the problem. Rather, it permits a different kind of attack. Consider the alternative implementation of the shuffle method below, which relies on a Mersenne Twister instance which shares state with past shuffles.

```
const MT = new MersenneTwister(SEED);
function shuffle(deck) {
  for (let i = deck.length - 1; i > 0; i--) {
    let j = MT.int32() % (i + 1);
    [deck[i], deck[j]] = [deck[j], deck[i]];
  }
}
```

The Mersenne Twister has a large period of $2^{19997} - 1$. If the initial seed was chosen from a cryptographically secure source and provides enough bits of security, it is infeasible for the client to brute force the initial seed given that they do not know how many shuffles have taken place.

However, this does not prevent the client from recovering the internal state of the Mersenne Twister and using it to predict all future shuffles. The Mersenne Twister uses 623 32-bit numbers for its internal state. Previous attacks have already shown that by observing 624 32-bit outputs from the PRNG, it is possible to uncover the internal state since the Mersenne Twister is not cryptographically secure, and all its internal operations are invertible [24].

We build on this idea by observing that the output of a Fisher-Yates shuffle is invertible given the initial configuration. We will walk through an example to see how an adversary might recover the state of the Mersenne Twister by observing deck outputs. Consider a simplified deck $D$ below along with a shuffle of $D$ which we will call $F$.

$$D = [1, 2, 3, 4, 5]$$
$$F = [3, 4, 2, 5, 1]$$

$$[1, 2, 3, 4, 5] \Rightarrow_0 [5, 2, 3, 4, 1] \Rightarrow_0 [4, 2, 3, 5, 1] \Rightarrow_1 [4, 3, 2, 5, 1] \Rightarrow_0 [3, 4, 2, 5, 1]$$

Given $D$ and $F$, we can recover the 4 truncated numbers used to generate the shuffle $F$. We observe that the Fisher-Yates shuffle swaps in a known order. It swaps the last card with another number first. Once that swap is made, the last card's position is fixed. By applying this idea iteratively on all cards, we can effectively reverse the shuffle. By themselves, these numbers are not very useful. However, some of the numbers contain enough information to permit a side-channel attack.

The bit relation of the truncated numbers is difficult to reason about. However, when a number is taken modulo $2^n$, the $n$ lowest-order bits are revealed. For a deck containing 52 cards, there are various opportunities to observed leaked bits at positions 32, 16, 8, 4, and 2. This represents a total leakage of 15 bits for a single shuffle.

An implementation already exists for taking truncated 8-bit outputs of the Mersenne Twister and recovering its internal state [25]. This side-channel attack can be simplified to solving a large system of equations involving the bits of the internal state and the outputs. While the observations do not necessarily need to be contiguous, it is more difficult to perform this attack in practice if it is not known how many shuffles occurred between the last observation and the current observation.

## 4 Proposed Scheme

We propose a scheme that satisfies the criterion of provable fairness and is relatively easy to implement for a variety of game types. First, we will describe our protocol which builds on working primitives from the existing schemes. Next, we will prove that our scheme satisfies the strict definition of provable fairness and describe some of the properties that it has. Finally, we will demonstrate how to apply the scheme to different game types.

The main idea behind the proposed scheme is that all game types which we have surveyed can be reduced to a set of finite game states, each of which occur with equal probability. Most game types compute expectations of house edge based on a uniform distribution of the finite game states. An ideal scheme would be able to choose a state uniformly at random from the set of possible states for a game, which would prevent bias towards either party.

Let $S$ be the set of all game states. Our protocol requires that the client and server seed spaces be the set of integers such that $\sigma_A = \sigma_B = [0, |S|)$. For a given seed pair $s_A \in \sigma_A$ and $s_B \in \sigma_B$, the game function is defined as $G(s_A, s_B) = M\big((s_A + s_B) \bmod |S|\big)$ where $M$ is the public mapping function for a given game type. Note that $M$ should be a bijective mapping between $[0, |S|)$ and the game states. This ensures that each integer input to $M$ corresponds to exactly one game state.

It is recommended that the server commits using some randomness $r$ in addition to its seed. The reason is that $|S|$ may not be very large, and thus committing using a deterministic hash function would reveal the value of $s_A$. If $H$ is a secure hash function like SHA-512, committing using $p_A = C(s_A) = H(s_A \mid\mid r)$ where $|r| \geq 128$ bits should provide sufficient security under the random oracle model. To verify, the client would receive $s_A$ along with $r$ and verify that $p_A$ is the result of hashing the concatenation of $s_A$ and $r$. Furthermore, the client should check that $G(s_A, s_B)$ is indeed the output of the game.

We will now prove that this scheme satisfies the criterion of provable fairness. Assume that the client chooses $s_B$ uniformly from $\sigma_B$. For any fixed seed $s_A$, $(s_A + s_B) \bmod |S|$ is uniformly distributed in the domain of $M$. This implies that the outputs are also uniformly distributed in $S$ due to the bijective property of $M$. Thus, it is impossible for the server to select a single adversarial seed that performs better than the expected house edge. A similar argument can be applied when $s_A$ is chosen uniformly from $\sigma_A$ using symmetry.

## 4.1 Coin Games

For coin games, the mapping function is relatively trivial because there are only two states: heads and tails. The seed space is $S = \{0,1\}$, where 0 corresponds to tails and 1 corresponds to heads. A satisfying mapping function would be $M(s) = s$.

## 4.2 Dice Games

A satisfying mapping for dice games is also simple – the states of rolling 1 through 6 can be mapped directly to the numbers 1 through 6. In other words, $S = \{1,2,3,4,5,6\}$. A provably fair game between the client and server would use the mapping function $M(s) = s + 1$.

<u>Example Game</u>

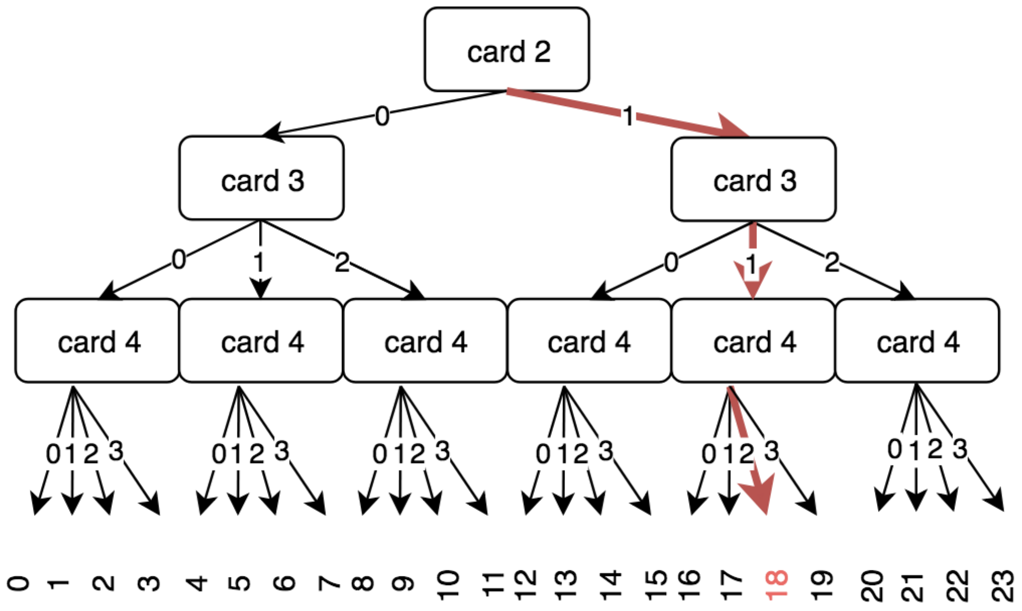
**Server:** Chooses $s_A = 4$. Sends client $p_A = H(4 \mid\mid r)$ where $r$ is some randomness.
**Client:** Chooses $s_B = 5$. Sends $s_B$ to server.
**Server:** Computes result from the game function as $G\big((4+5) \bmod 6\big) = M(3) = 4$ to decide if client won. Note that the criteria of winning are determined by the specific game type. The server then sends the client $s_A$ along with $r$.
**Client:** Verifies that $p_A$ matches. Verifies that the output of the game matches.

## 4.3 Card Games

For card games, we would like a mapping function that converts a number from 0 to $n! - 1$ to an ordering of the $n$ cards. We will call the number corresponding to a single permutation as the "permutation number" of the deck.

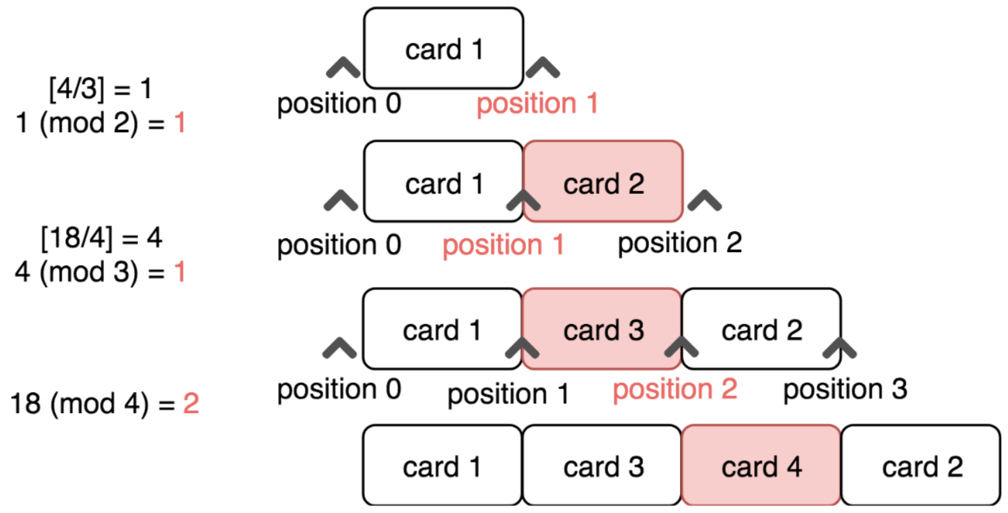
Define an ordering of cards from 1 through $n$. This ordering should be public and fixed between shuffles. First, working backwards, note that we can place the cards down starting from card number 1. The first card has exactly 1 position that it can go in. Then we can place down card number 2. Card 2 has two possible positions: left or right of card 1. We name these positions "position 0" and "position 1". For each arrangement of card 1 and card 2, card 3 will have 3 positions: before card 1, between card 1 and card 2, or after card 2. These three positions are labelled 0, 1, and 2 respectively. Continuing like this, for the $m$-th card, there are $m$ positions labelled 0 through $m - 1$.

If we draw the corresponding tree of outcomes, we will obtain a diagram like the one shown below. The diagram shows all possible shuffles for 4 cards. The permutations of cards are labelled 0 through $4! - 1$.



The red arrows 1, 1, 2 refer to the fact that card 2 was inserted in position 1, card 3 in position 1, and card 4 in position 2.

More specifically, we see this placement in the following figure.



Thus, we have a mapping from each permutation of cards to a permutation number. This mapping is clearly 1 to 1 because each permutation is accounted for in the tree by looking at all possible placements of the card at each step.

However, when we define the mapping function, we are performing the reverse computation of converting a permutation number to a card ordering. Given a permutation number such as permutation 18, we want to reverse-engineer the placement of each card. Starting from card 4 (still using the example of $n = 4$ cards), the last card must have been in position $18 \pmod 4 = 2$. Next, we divide by 4 to find which node in the second to last layer of the tree the permutation came from. Since $\lfloor 18/4 \rfloor = 4$ and $4 \pmod 3 = 1$, we know card 3 must have been in position 1. Similarly, divide again and use $\pmod 2$ to find that card 2 should have been placed in position 1.

Below is a Python function that converts any permutation number `combo_num` to a physical ordering of the deck with cards numbered 1 through `num_cards`.

```python
def map(num_cards, combo_num):
  cur_num = combo_num
  positions = []
  cards = []

  for card in range(num_cards, 1, -1):
    position = cur_num % card
    positions = [position] + positions
    cur_num = cur_num//card

  cards = [1]

  for card in range(2, num_cards + 1):
     position = positions[card-2]
     cards = cards[0:position] + [card] + cards[position:]

  return cards
```

Since we now have a 1 to 1 mapping for permutations (shuffles) of $n$ cards to numbers 0 through $n! - 1$, we can play any game with an arbitrary number of cards. In the example of a 52-card game, there are $52! - 1$ outcomes. Since $\log_2 52! < 226$, we can represent all the states (i.e. all the shuffles of the deck) in 226 bits. This is a feasible amount of entropy to generate for each game, since many existing schemes use around 128 to 1024 bits of entropy per game.

Defining the mapping function this way yields several benefits. First, the sizes of seed spaces are not skewed, so all card games where expected house edges are computed based on a uniform distribution over all shuffles will satisfy the criteria of provable fairness if one party chooses their seed uniformly at random. Second, it is possible under this mapping function for all permutations of the deck to appear, unlike some existing schemes where it is harder to prove that all permutations can occur with a given initial seed.

## 4.4 Slot Games

Slot games can also be transformed into a set of discrete states with uniform probability. Let $n$ be the number of reels and let $b$ be the number of states on each reel. There is a total of $nb$ different states, each occurring with probability $(nb)^{-1}$. We can imagine that $b$ is the base and $n$ is the number of digits. Thus, the mapping function is simply performing a base conversion from a base 10 number to an $n$-digit number in base $b$.

## 5 Conclusion

Through surveying a variety of different online gambling sites, we have determined a wide range of potential vulnerabilities that can be exploited from the perspective of the client and the server. Some of these vulnerabilities are easily patchable, while others represent deeper flaws in the design of the provable fairness scheme. We proposed a stronger definition of provable fairness and presented a generalized scheme that satisfies this definition. We also showed how to extend this proposed scheme to a few common game types.

## 6 References

[1] https://news.bitcoin.com/bitcoin-gamblers-wagered-4-5-billion-btc-2014/
[2] https://www.gamesindustry.biz/articles/2018-04-17-loot-boxes-skins-gambling-to-hit-usd50-billion-by-2022-report
[3] https://dicesites.com/provably-fair
[4] https://mysterybrand.net/en/provably-fair
[5] https://www.bitsler.com/en/provably-fair
[6] https://en.wikipedia.org/wiki/Length_extension_attack
[7] https://tools.ietf.org/html/rfc2104
[8] https://fair-dice.com/
[9] https://www.worldcat.org/title/art-of-computer-programming-volume-2-seminumerical-algorithms/oclc/85975465
[10] https://bitcointalk.org/index.php?topic=1700680.0
[11] https://fortunejack.com/dice
[12] https://diceum.com/bet
[13] https://bitcointalk.org/index.php?topic=1494470.0
[14] https://jonasnick.github.io/blog/2015/07/08/exploiting-csgojackpots-weak-rng/
[15] https://cryptogambling.org/whitepapers/provably-fair-algorithms.pdf
[16] https://repl.it/@mysterybrand/provably-fair-check
[17] https://www.bitdice.me/help
[18] https://www.bgaming.com/provability_explained.html
[19] https://wizardofodds.com/games/blackjack/strategy/4-decks/
[20] https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=1528&context=gradreports
[21] https://github.com/seblau/BlackJack-Simulator
[22] https://coinroyale.com/provablyfair
[23] https://blog.nitrogensports.eu/casino/was-your-blackjack-hand-provably-fair/
[24] https://know.bishopfox.com/blog/2014/08/untwisting-mersenne-twister-killed-prng
[25] https://github.com/fx5/not_random

## 7 Appendix

We show a few code fragments used in this report to construct theoretical attacks. The full code is available at https://github.com/bobbyluig/provable-fairness.

Code for computing Mersenne-based deck shuffles consistent with an initial described in section 3.6 is shown below.

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <random>
#include <vector>

int main() {
  static const unsigned int kIndices = 52;

  std::mt19937 initial_mt(42);
  std::vector<uint32_t> initial_indices(kIndices);
  std::iota(initial_indices.begin(), initial_indices.end(), 0);
  std::shuffle(initial_indices.begin(), initial_indices.end(), initial_mt);

  for (unsigned int &x : initial_indices)
    std::cout << x << ',';
  std::cout << std::endl;

  std::vector<uint32_t> indices(kIndices);
  std::iota(indices.begin(), indices.end(), 0);;

  for (uint32_t i = 0;; i++) {
    std::vector<uint32_t> indices_copy = indices;
    std::mt19937 generator(i);
    std::shuffle(indices_copy.begin(), indices_copy.end(), generator);

    if (std::equal(initial_indices.begin(), initial_indices.begin() + 4, indices_copy.begin())) {
      std::cout << i << ":";
      for (unsigned int &x : indices_copy)
        std::cout << x << ',';
      std::cout << std::endl;
    }

    if (i == std::numeric_limits<uint32_t>::max()) {
      break;
    }
  }
}
```

Code for selecting the best initial shuffle for a blackjack game described in section 3.5 is shown below.

```
def exploit():
  best_cards = None
  best_value = float('inf')

  for _ in range(100):
    game = Game()
    deck = game.shoe.cards.copy()
    shoe = game.shoe

    winnings = []
    bets = []

    for i in range(SHOE_SIZE * 52):
      cut_deck = deck[i:] + deck[:i]
```

```python
      shoe.cards = cut_deck
      game = Game(shoe)
      game.play_round()

      winnings.append(game.get_money())
      bets.append(game.get_bet())

    loss = sum(winnings) / sum(bets)

    if loss < best_value:
      best_value = loss
      best_cards = deck

  return best_cards, best_value
```