A Mathematical Analysis of Catcher/Pitcher Encryption Schemes

{mlancast, samird, mtwu}@mit.edu

May 16th, 2018

1 Introduction

For those unfamiliar with the game of baseball, there are a few important actors that play a role in every single play of the game. The **pitcher** throws the baseball to his teammate, the **catcher**, while a member of the opposing team, the **batter**, attempts to hit the thrown ball. Pitchers use a variety of different types of **pitches** in an attempt to prevent the batter from hitting the ball. There are many different types of pitches, each of which moves with different speeds, curves, slides, or dips. The four main pitches (and the ones we will refer to in this paper) are

- The **fastball** - thrown straight and fast
- The **curveball** - thrown with spin so that it dives or curves
- The **slider** - thrown with spin so that it tails laterally
- The **changeup** - thrown slowly

Before throwing a pitch, the pitcher and catcher need to agree on the pitch to be thrown. Usually, the catcher will indicate to the pitcher which pitch he thinks should be thrown using a series of hand signs. The pitcher will simply shake or nod his head in (dis)agreement. The standard mapping used by catchers to indicate pitches is one finger for fastball, two for curveball, three for slider, and four for a changeup. Obviously the catcher cannot simply tell the pitcher out loud which pitch to throw, because then the batter will gain a competitive advantage in that he knows which pitch to expect. Instead, the catcher will use hand signs, as seen in Figure 1 below, to indicate which pitch to throw.

Now that we have explained one of baseball's basic constructions, we will move on to explain how this is related to cryptography and frame this problem as a cryptography one. There is a specific scenario in the game of baseball in which the above scheme for communicating pitches between the pitcher and catcher is insufficient. If a batter successfully reaches second base, he has a clear

Figure 1: Catcher giving sign for fastball

view of the catcher giving signs to the pitcher. Given that the standard mapping (1=fastball, 2=curveball, 3=slider, 4=changeup) is known by everybody, he can then relay the signs to the batter. Standard ways of doing this include placing his hands on his knees, shuffling his feet, touching his helmet, etc. These details are less relevant, but the main point is that the pitcher/catcher communication scheme is no longer secure.

To mitigate this, the pitcher and catcher must modify their communication scheme such that neither the current batter, or the batter on second base can determine which pitch will be thrown in advance. They typically do this using an encryption-like scheme that is designed to hide the value of the pitch from the batter on second base. There are a number of such schemes, most of which use a sequence of hand signs from the catcher to the pitcher. When we discuss these sequences, we will use the term **index** to refer to the location of a number in the sequence. For example, the index of the number 3 in the sequence 21431 is 4. In order to be consistent and to bound our calculations, we restricted the number of signs in a sequence for all of the encryption schemes to 5.

It will be helpful to think about the original value of the pitch (1, 2, 3, or 4) as the plaintext message, and the sequence of signs as the ciphertext. To be clear, the schemes used by the pitcher and catcher to securely communicate pitch values are not encryption schemes in the traditional sense. They do not use mathematical keys to decrypt, instead the keys are simply knowledge of the encryption algorithm, which allows the pitcher to invert the encoding to retrieve the original pitch value. In order to be usable in an actual baseball game by the pitcher and catcher, these schemes must have a few characteristics:

1. They must be non-deterministic, since the catcher will be encrypting the same plaintext many times

2. They must be simple to "decrypt" by the pitcher, since he only has a

couple of seconds to invert the encryption and decide if he wants to throw that pitch

We can formalize this problem as a Chosen Plaintext Attack security analysis on the encryption-like schemes used by a pitcher and catcher to communicate pitch information. We consider two different scenarios:

1. Consider the adversary to be any member of the opposing team on second base. Let n represent the number of times an adversary reaches second base. If we assume the average at-bat is approximately 5 pitches, the adversary will get to see $5n$ (ciphertext, plaintext) pairs to use to crack the scheme.

2. Consider an adversary watching recorded replays of a team's games. If the team uses the same scheme from game to game (which teams do), the adversary has essentially a limitless pool of (ciphertext, plaintext) pairs to use to crack the scheme.

In the above scenarios, the adversary sees both the encryption sequence generated by the catcher, and the decrypted plaintext in the form of the pitch that gets thrown.

In this paper, we will do a mathematical analysis on the relative hiding capabilities of 9 of these schemes. We will then combine our analysis with a scoring of each of the schemes using some homegrown metrics such as usability and (in)consistency between at-bats. These terms will be given more context later. To help compare and rank the different schemes, we will also present a neural network that is trained to make predictions about pitch values given ciphertext inputs. Using what we learn from this analysis, we will develop our own pitch-encoding scheme and present that at the end.

Before we dive into the details of the metrics, there are a few more definitions that will make the following analysis much easier to digest for a reader without baseball-specific knowledge. First, the basic unit of a pitcher/catcher face-off is an **at-bat**. Each batter takes one at-bat at a time against the pitcher. Within an at-bat, we refer to the **count**, which indicates the state of the at-bat by tallying the number of **balls** and **strikes** the pitcher has thrown. A pitcher can throw up to 4 balls or 3 strikes in an at-bat, whichever comes first.

2 Defining the Metrics

There are many aspects to each scheme that can make it more or less secure. From being easy for the pitcher and catcher to understand to being tough for the opposing team to crack, there are many factors and scenarios to consider. We have come up with five metrics to help quantify the security and practicality of each scheme

- **Usability** - whether the pitcher can instantaneously determine the pitch from the catcher's signs with minimal thinking

- **Hardness** - how many at-bats an adversary needs to see to be able to accurately predict the pitch
- **Inconsistency** - the variation of the scheme between pitches
- **Encodablility** - the number ways a given pitch can be encoded
- **Decodability** - the number combinations a scheme has if the adversary knows the scheme (number of variations of the specific rule based on parameters that are set beforehand by catcher and pitcher)

2.1 Usability

Some schemes require little to no thinking from the pitcher to decrypt the pitch from the catcher. The easier the transfer of knowledge is, the less likely it is that a miscommunication or misinterpretation will occur. For example, if the pitch is always the first sign, the pitcher simply has to look for that first sign and can ignore the rest of the scheme. On the other hand, a scheme like counting the number of 1s in a series of signs requires the pitcher to see all the signs and perform a calculation on that value. Thus, the key difference in usability is whether or not the pitcher has to observe all the signs to decrypt the code or if he simply needs to wait for one sign of the sequence. We say a scheme has usability 0 if it does the former and usability 1 if it does that latter.

2.2 Hardness

The longer it takes to crack a scheme, the harder that scheme is by our definition. To objectively obtain a measure of how hard a scheme is to break, we built a MLP Classifier (Multi-layer Perceptron) neural network and observe the number of at-bats necessary in the training set for the model to learn to accurately predict a pitch from a given sequence. We built a generator function for each scheme that randomly picked sequences of length 5, and identified what the correct pitch should have been if each of those sequences was shown to a pitcher. We ensured that the probability that the sequence would correspond to a given pitch was equal across all four potential pitches. These sequences of signs, each appended by the number of outs, pitch count (balls and strikes), jersey number and whether the batter is left or right-handed at the time the sequence was given, constituted an input vector to the model. The output label for each input vector was simply the pitch that this sequence corresponded to (the pitch that should have been thrown). We trained each scheme's model on varying training set sizes, and deemed a scheme to be "cracked" when the neural net could consistently achieve over 80% accuracy on the test set. So for each scheme, its hardness metric is equivalent to the number of at-bats necessary for our neural net to "crack" it.

As a basic overview, a neural network iteratively trains/learns on a training data set, and its performance is measured on a test data set of which it has not seen any of the data. In the training set, it is given many data points, each

of which has an input vector of **features** to learn from, and a corresponding output label. In the test set, it is only given features and must make a prediction of the output label, and the percentage of the test set that it correctly predicts is its accuracy.

An at-bat was the granularity we used to determine the hardness of a scheme, and we generated a random sequence for each pitch in the at-bat. Each pitch has a 50% probability of being a strike or ball, and the at-bat ends when there are either 4 balls accumulated or 3 strikes, and we start the next at-bat from a 0-0 count. The reason we choose to use the at-bat as our smallest unit of granularity was because some of the schemes we analyze change the rule every at-bat, not every pitch, and we wanted to capture the same encoding rules in the same unit of measurement.

We trained our model incrementally, in order to best pinpoint the number of at-bats needed in the training set (size of the training set) to achieve a prediction accuracy of 80% in the test set which was of equal size. More tangibly, we first trained a classifier with 2 generated sequences using the scheme in question, then 3, then 4, and so on, until we reached a point where 10 consecutive training set sizes yielded test accuracy over our 80% threshold once trained. The 80% threshold was derived from the fact that the expected value of pitches in an at-bat under our scheme was 5, and we considered the neural network to have "cracked" the scheme successfully if it accurately predicts all but one of the pitches thrown in a standard at-bat (4/5). Further, we searched for the first sequence when 10 models in a row attained satisfactory training accuracy because since we restarted the training of each model from scratch, there was significant variation between trials that differed by 1 in the number of at-bats in the training set, and we wanted a meaningful measure of hardness, not one that would be widely subject to random variations.

2.3 Inconsistency

Another metric that is important to consider is the inconsistency a scheme shows between pitches and at-bats. While a neural network does not make a distinction between consecutive pitches and pitches thrown very far apart, the limited memory of humans makes pitches thrown next to each other much more prominent. So when cracking a scheme, variability in the scheme from pitch to pitch and at-bat to at-bat will make it harder to decrypt. We give schemes with variance between pitches in an at-bat a Pitch Inconsistency of 1 and those without a Pitch Inconsistency of 0. Similarly, schemes that change between at-bats get an At-Bat Inconsistency of 1 and those that don't get an At-Bat Inconsistency of 0.

2.4 Encodability

Schemes start to break down when the opposing team recognizes a pattern between the pitch and signs each time a type of pitch is thrown. When asked to throw a specific pitch, the catcher has options of different sequences to encode

the pitch with because the schemes are non-deterministic. The more ways a pitch can be encoded, the more secure it is because the same series of numbers is not being used over and over to communicate a pitch. Recall that a deterministic encryption scheme is a scheme in which a given plaintext is always encrypted to the same ciphertext. For a catcher looking to encrypt the same pitch many different times throughout the course of a game, determinism is bad. Therefore, we evaluate a measure of how non-deterministic a given scheme is by giving each scheme an Encodability metric equivalent to the average number of ways it can encode a given pitch.

2.5 Decodability

When the opposing team (the adversary) believes that they might have cracked one part of the code, they must then figure out exactly what the scheme is. For example, the adversary may suspect that the scheme involves counting a certain number but they then have to determine which number is being counted. A higher number of possible schemes can delay the opposing team from determining exactly what the scheme is. It might even be confusing enough that the adversary believes its hunch is incorrect and moves on to other ideas. Thus, we give each scheme a Decodability metric equivalent to the number of possible combinations of schemes once the core idea of the scheme is discovered.

3 Scheme Analysis

From both scheme research and our own experiences, we have compiled a list of nine schemes that are commonly used in baseball. For each one, we will calculate each of the metrics we discussed in the previous section.

3.1 No Encoding

In this scheme, a single sign indicates the pitch to be thrown. Examples are shown below below.

$$\mathbf{2} \rightarrow \mathbf{2}$$
$$\mathbf{1} \rightarrow \mathbf{1}$$
$$\mathbf{3} \rightarrow \mathbf{3}$$

It is very usable because the pitcher only looks for that one sign for a usability of 1. It has a hardness metric of 4 because an adversary simply needs to see each of the four pitches once to reliably predict future pitches. It is has no inconsistency because the pitch is always just a single sign so pitch and At-Bat Inconsistency scores of 0. When encoding any pitch, there is only 1 way to do so for an Encodability of 1 and if the opposing team catches on, there is only 1 way this scheme could be constructed for a Decodability of 1.

Usability: **1**	Hardness: **4**
Pitch Inconsistency: **0**	At-Bat Inconsistency: **0**
Encodability: **1**	Decodability: **1**

3.2 Constant Index

For this scheme, the pitcher and catcher choose a sign index (first, second, third, fourth, or fifth index) where that index in the sequence of signs is always used. Below are examples for when we are using the second sign index.

$$\textbf{12345} \rightarrow \textbf{2}$$
$$\textbf{12433} \rightarrow \textbf{2}$$
$$\textbf{31224} \rightarrow \textbf{1}$$

This scheme is usable because the sign index is known so the pitcher simply has to wait to see the sign at that index for a usability of 1. Our neural net found it took 90 at-bats to crack the constant index scheme, as shown in the graph in Figure 2. Since we always use the same index for each pitch and each at-bat, the inconsistency score for both is 0. For Encodability, the catcher has a predetermined pitch he wants to throw so he simply needs to sign that pitch at the agreed-upon index in the sequence. For the other four signs, the catcher can put down whichever numbers he wishes which means this scheme has a total of $4^4 = 256$ encodings. For Decodability, given that the adversary knows a constant index is being used, the index could be the first, second, third, fourth, or fifth index for 5 total decodings.

Usability: **1**	Hardness: **90**
Pitch Inconsistency: **0**	At-Bat Inconsistency: **0**
Encodability: **256**	Decodability: **5**

3.3 Righty/Lefty

In this scheme, the pitcher and catcher encrypt pitches depending on whether the batter is left handed or right handed. Below are examples where a right handed batter indicates a first sign index and left handed batter indicates a second sign index. Consider a right handed batter and the following ciphertext encodes 2=curveball:

$$\textbf{23412} \rightarrow \textbf{2}$$

If it was a left handed batter it encodes 3=slider for the same ciphertext:

$$\textbf{23412} \rightarrow \textbf{3}$$

This scheme is usable because the sign index is known after the pitcher determines the right or left handedness of the batter, yielding a Usability of 1. Our neural net found it took 138 at-bats to crack the Righty/Lefty scheme, as shown in the graph in Figure 3. Assuming players do not switch bat handedness, the pitcher will be using the same sign index between pitches so the

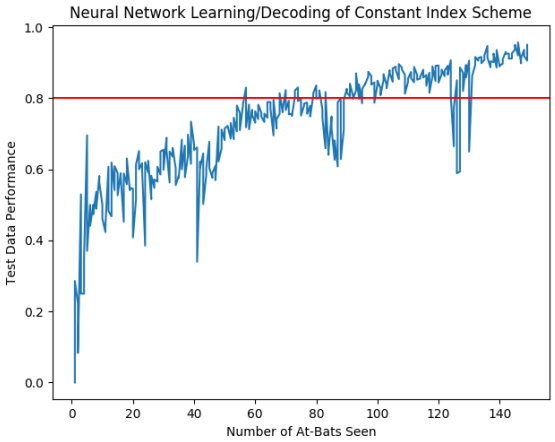

Figure 2: Accuracy of MPLClassifier model on Predicting Pitch given by Constant Index Scheme. 90 at bats is the critical point at which the accuracy of the network surpasses 80% by our definition above. Specifically, it is the first at bat at which a sequence of 10 consecutive trained models exceed this threshold test accuracy.

Pitch Inconsistency is 0. Between at-bats, different batters may have different bat handedness which changes the index the pitcher is looking for. This means that At-Bat Inconsistency is 1. For number of encodings, we could have either a right or left handed batter for 2 combinations for selecting the sign index. Given that sign index, the catcher can put down whichever numbers he wishes for the other four signs in the sequence for 4^4 encodings per sign index. This yields a total of $2 * 4^4 = 512$ encodings. For decodings, even if an adversary knows the scheme uses bat handedness, there are still 20 different possible schemes. This is because right handedness could be associated with any of the five indices and then there are four indices for which left handedness could be associated with for $5 * 4 = 20$ decodings.

Usability: **1**	Hardness: **138**
Pitch Inconsistency: **0**	At-Bat Inconsistency: **1**
Encodability: **512**	Decodability: **20**

3.4 ABE

The ABE scheme is an acronym for Ahead-Behind-Even, which refers to different states in the current count from the pitcher's perspective. A count is considered "ahead" if there are more strikes than balls (0-1, 0-2, 1-2), "behind" if there are more balls than strikes (1-0, 2-0, 3-0, 2-1, 3-1, 3-2), or "even" if there are equal numbers of balls and strikes (0-0, 1-1, 2-2). Now, an "ahead" count maps to

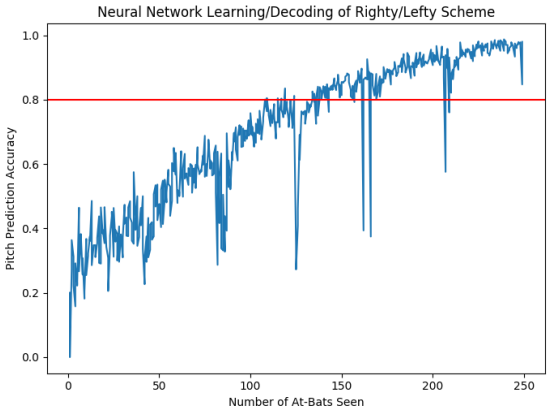

Figure 3: Accuracy of MPLClassifier model on Predicting Pitch given by the Righty/Lefty Scheme. The critical number of at bats (measure of hardness) is 138.

the first index in the sequence, "behind" maps to the second index, and "even" maps to the third index. In a simpler representation: A(head)=1, B(ehind)=2, E(ven)=3. Below is an example encoding that corresponds to 4=changeup for a count with 1 ball and 2 strikes:

$$\mathbf{41223 \rightarrow 4}$$

And another example corresponding to 1=fastball for a count with 2 balls and 2 strikes.

$$\mathbf{32124 \rightarrow 1}$$

For Usability, ABE scores well because the pitcher can precompute the index for which he needs to look in the sequence. Before throwing the pitch, the pitcher knows the count, and therefore he also knows whether the count is considered "ahead", "behind", or "even" so he simply needs to look for the first, second, or third index in the sequence. The neural network used 384 at-bats to reliably predict pitches encoded under the ABE scheme, as shown in Figure 4. From here on, we will not explicitly reference the figure in which the hardness of a scheme is depicted, but all figure are captioned and can be matched by the reader.

For Pitch Inconsistency, the ABE encryption scheme changes with each pitch, so it gets that point. Since the sequence of counts an adversary would observe changes from at-bat, ABE also gets the point for At-Bat Inconsistency. The Encodability for ABE can be calculated similarly to Righty/Lefty. For any given pitch, the count could be ahead, behind, or even with each scenario corresponding to one of three pitch indices. From that index, the catcher has four other signs in the sequence to choose for 4^4 encodings per sign index and a

total of $3 * 4^4 = 768$ encodings. When calculating Decodability, we can assume the adversary knows the scheme uses ABE. However, he does not know which sign index each of A, B, and E corresponds to. We have five indices to associate A with, then four indices left to associate B with, and finally three indices for E for a total of $5 * 4 * 3 = 60$ decodings.

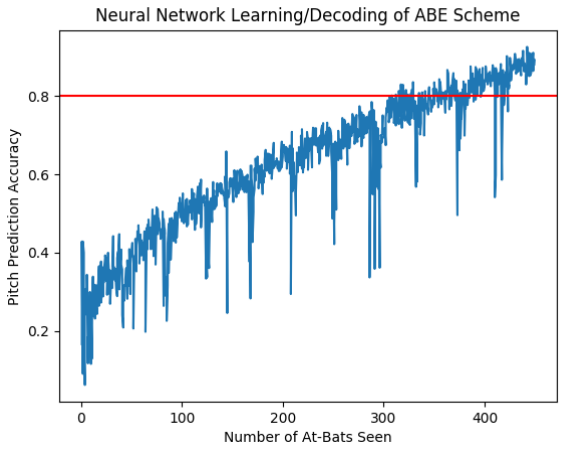

Figure 4: Accuracy of MPLClassifier model on Predicting Pitch given by the ABE Scheme. The critical number of at bats (measure of hardness) is 384.

Usability: **1**	Hardness: **384**
Pitch Inconsistency: **1**	At-Bat Inconsistency: **1**
Encodability: **768**	Decodability: **60**

3.5 Outs+1

Outs+1 is another scheme which uses external information to communicate the index at which the pitch value is located. In this case, the pitcher and catcher add 1 to the current number of outs (zero outs=1, one out=2, two outs=3) to determine the sign index. Here's an example encoding with two outs for 2=curveball:

$$\textbf{11231} \rightarrow \textbf{2}$$

Another example with 1 out encoding 3=slider:

$$\textbf{23241} \rightarrow \textbf{3}$$

Again, Outs+1 scores well for Usability because all of the required information can be precomputed by the pitcher. He knows the number of at-bats before he receives the ciphertext from the catcher, so he knows exactly which index to

look for. The Outs+1 scheme received a hardness score of 270 from our neural network. Now, since the number of outs generally stays the same throughout an individual at-bat, the Outs+1 scheme does not get the point for Pitch Inconsistency. It does, however, get At-Bat Inconsistency because the number of outs generally varies from at-bat to at-bat. Encodability for Outs+1 is almost identical to ABE in that there are three possible pitch indices and 4^4 encodings per sign index for a total of $3 * 4^4 = 768$ encodings. Decodability is also extremely similar as there are five possible pitch indices three of the indices must correspond to zero outs, one out, and two outs. As in ABE there are $5*4*3 = 60$ ways for this to happen for a Decodability of 60.

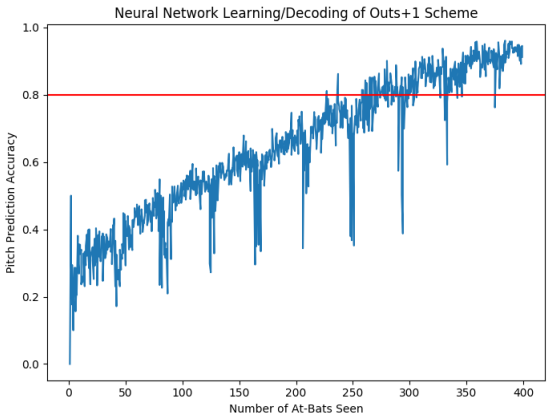

Figure 5: Accuracy of MPLClassifier model on Predicting Pitch given by Outs+1 Scheme. The critical number of at bats (measure of hardness) is 270.

Usability: **1**	Hardness: **270**
Pitch Inconsistency: **0**	At-Bat Inconsistency: **1**
Encodability: **768**	Decodability: **60**

3.6 Jersey Number

For this scheme, the pitcher and catcher observe the jersey number of the batter and take it modular 5. The resulting value will determine the sign index for this at-bat (note: any number that would be 0 (mod 5) will be correspond to the fifth pitch because there is no zeroth pitch). Consider an example where the batter's number is 9. Since 9 (mod 5) = 4, his at-bat will use the pitch at the fourth index:

$$\mathbf{12235 \rightarrow 3}$$

In another example, let the batter be number 20. With 20 (mod 5) = 0, we will use the fifth index as shown below:

44241 → 1

This scheme gets a 1 for Usability because the pitcher knows what index he is looking for since he's already done the jersey number calculation. Our neural net found it took 1135 at-bats to crack the Jersey Number scheme. This scheme has a Pitch Inconsistency of 0 since the index being used is set for each at-bat based on the batter's number. However, it has an At-Bat Inconsistency of 1 because every batter has a different number and we will see different indices between batters when their numbers yield difference values when taken modular 5. The Encodability of this scheme is the highest of any scheme we've looked at. Since the sign index used can range from the first sign to the fifth sign, there are possible pitch indices and 4^4 encodings per sign index for a total of $5 * 4^4 = 1280$ encodings. Given that the adversary knows that the pitcher and catcher are using a Jersey Number scheme, the only way to alter the scheme is to change what we are taking modular 5. The only schemes that could be possible decodings are $JerseyNumber \pmod 5$, $JerseyNumber + 1 \pmod 5$, ... , $JerseyNumber + 4 \pmod 5$ for a total of 5 decodings.

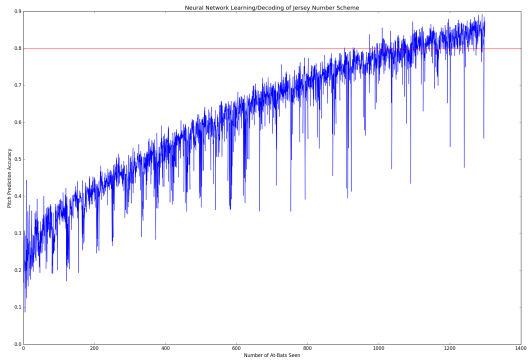

Figure 6: Accuracy of MPLClassifier model on Predicting Pitch given by the Jersey Number Scheme. The critical number of at bats (measure of hardness) is 1135.

Usability: **1**	Hardness: **1135**
Pitch Inconsistency: **0**	At-Bat Inconsistency: **1**
Encodability: **1280**	Decodability: **5**

3.7 Follow the Leader

This scheme gets its name because the basic idea is that the number shown in the first index determines which of the subsequent indices represents the value of the pitch. If the catcher shows a 2 as the first sign, the pitcher then counts

the next two signs and knows that that second sign is the pitch value. Here are some examples:

24132 → 1
41232 → 2
31232 → 3

Follow the Leader does not get the Usability point because the pitcher does not know which index he needs to look for beforehand. He must wait until he sees the first sign, and then count the number of subsequent signs. This is more error-prone than some of the other schemes, especially if the catcher is giving the signs quickly. The neural network needed to see 675 at-bats in order to reliably predict pitches using the Follow the Leader scheme. It does, however, get the points for Pitch Inconsistency and At-Bat Inconsistency because the encoding is completely determined by the catcher and does not rely on any information present across pitches or at-bats. For calculating Encodability for this scheme, we first notice that there are four values to give the first sign that will determine which index will contain the actual pitch. The actual pitch is determined but the remaining three signs in the sequence can be chosen by the catcher. There are $4^3 = 64$ ways to do this for each sign index so there are a total of $4 * 4^3 = 256$ encodings for this scheme. The Decodability for this scheme is actually 1. Since it is so specific, knowing that the first sign indicates the number of pitches to wait leaves only one possible scheme.

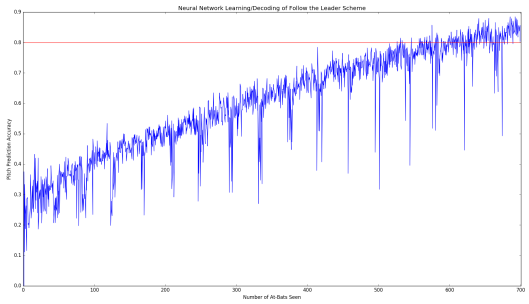

Figure 7: Accuracy of MPLClassifier model on Predicting Pitch encoded using the Follow the Leader scheme. The critical number of at bats (measure of hardness) is 675.

Usability: **0**
Pitch Inconsistency: **1**
Encodability: **675**

Hardness: ***
At-Bat Inconsistency: **1**
Decodability: **1**

3.8 Chase the 2

The Chase the 2 scheme essentially uses what's known as an "indicator". The pitcher and catcher agree on a number before the game (or before the inning, if they want to improve this scheme) and they use that number to sign the value of the pitch. For example, they could agree to use 2 as the indicator in which case, the value of the pitch to be thrown is whatever number follows the 2 in the sequence. They can use any of the four numbers as the indicator. Here's an example where the pitcher and catcher are using a Chase the 3 scheme:

$$\mathbf{21431 \rightarrow 1}$$

And an example with Chase the 2, where we assume that the first instance of a 2 is used as the indicator:

$$\mathbf{24123 \rightarrow 4}$$

As for Usability, the Chase the 2 scheme does not earn that point because the pitcher does not know the specific index for the encoded pitch value. In addition, the pitcher has to pay attention to two signs, the indicator and then the pitch value. Our neural network used 179 at-bats to crack an encoding using the Chase the 2 scheme. It's both Pitch Inconsistent and at-bat Inconsistent because the location of the indicator can change after every pitch. To calculate Encodability for this scheme, we first notice that there are four indices for the actual pitch to go where a 2 can precede it. Since the actual pitch and the 2 are deterministic, there are then three other signs to be chosen by the catcher for a total of $4 * 4^3 = 256$ encodings. Given that the adversary knows the scheme involves chasing a number, they could either chase a 1, 2, 3, or 4 for a Decodability of 4.

Usability: **0**	Hardness: **179**
Pitch Inconsistency: **1**	At-Bat Inconsistency: **1**
Encodability: **256**	Decodability: **4**

3.9 Pumps

The Pumps scheme is an interesting scheme that encodes a pitch value by counting the number of "pumps" in the sequence. The pitcher and catcher agree on a pump value beforehand, and then the catcher encrypts a pitch by generating a sequence which includes the pump value exactly the number of times which corresponds to the pitch they would like to throw. For example, if the pump is agreed to be 1, and the catcher wishes to encrypt a 3=slider, he will include three 1s in the ciphertext. Here are some examples to make it more clear. With a pump value of 1:

$$\mathbf{12143 \rightarrow 2}$$
$$\mathbf{12343 \rightarrow 1}$$

And an example with a pump value of 4:

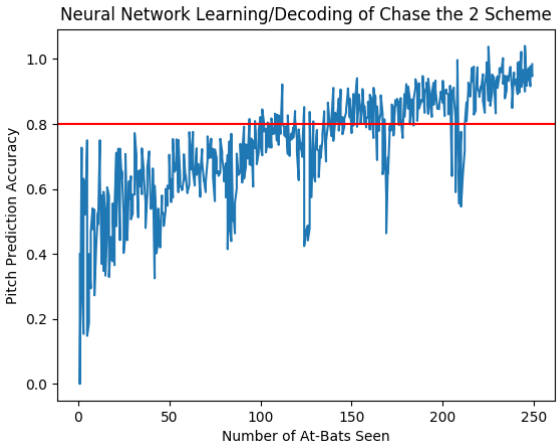

Figure 8: Accuracy of MPLClassifier model on Predicting Pitch given by Chase the 2 Scheme. The critical number of at bats (measure of hardness) is 179.

44244 → 4
14344 → 3

The Pumps scheme is not considered Usable because, firstly, the pitcher is not simply looking for a location in the sequence which represents the pitch value, and, secondly, the pitcher needs to count the number of instances of the pump value. Basically, the Pumps scheme can not be precomputed before the ciphertext is seen. This scheme does fairly well against our neural net, requiring 801 at-bats before reliably predicting pitches. Pumps gets both Pitch Inconsistency and At-Bat Inconsistency points because it does not rely on any repeating information across pitches. The Pumps scheme requires us to calculate the encodings for each pitch and then average the expected number of encodings. If the pitch is a fastball and we need the sign sequence to contain just one 1, there are $\binom{5}{1}$ ways to place that 1. The other four sign of the sequence can be another number that is not a 1 for $3^4 = 81$ possibilities. This means that there are $\binom{5}{1} * 81 = 405$ ways to encode a fastball. The logic is similar for the other pitches. For a curveball with two 1's, we have $\binom{5}{2}$ to place the ones and 3^3 possibilities for the rest of the numbers for $\binom{5}{2} * 3^3 = 270$. With three 1's, we have $\binom{5}{3}$ to place the ones and 3^2 for the rest for $\binom{5}{3} * 3^2 = 90$ and with four 1's, we have $\binom{5}{4}$ to place the ones and 3^1 for the rest for $\binom{5}{4} * 3^1 = 15$. Averaging over the four pitches yields an expected value of $(405 + 270 + 90 + 15)/4 = 195$ encodings for the Pumps scheme. Just as with Chase the 2, if the adversary knows the scheme involves counting a digit, they could count either 1, 2, 3, or 4 for a Decodability of 4.

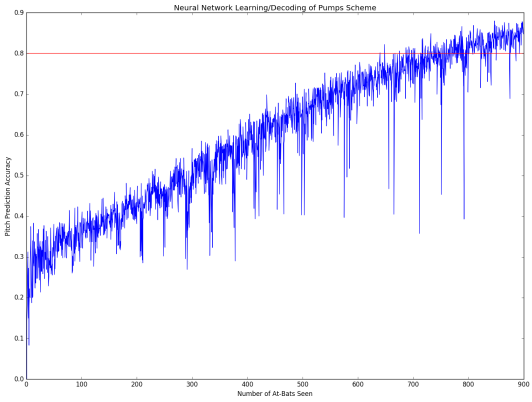

Figure 9: Accuracy of MPLClassifier model on Predicting Pitch given by the Pumps Scheme. The critical number of at bats (measure of hardness) is 801.

Usability: **0**	Hardness: **801**
Pitch Inconsistency: **1**	At-Bat Inconsistency: **1**
Encodability: **195**	Decodability: **4**

4 Evaluation Equation

4.1 Normalization of Metrics

After analyzing the Usability, Hardness, Inconsistency, Encodability, and Decodability of these nine schemes, we look to develop an equation that could give an overall score to each scheme. This score allows us to quantitatively compare the effectiveness of one scheme versus another. We look to create a weighted linear equation where the Scheme Score is comprised of the normalized metrics where each w_k is a weight reflecting the importance of each metric:

$$SS = w_1U + w_2H + w_3PI + w_4API + w_5E + w_6D$$

To use this equation, we first took a table of our raw metric scores and then normalized by dividing each metric by the maximum of that metric across all nine schemes. Our reason for normalizing over the maximum ensures that each metric would be on the same 0 to 1 scale and the weights will take care of the differences in the importance of each metric to the overall scheme score. Table 1 below shows the raw scores and Table 2 has each data point divided by the maximum value of its given metric.

Table 1: Raw Metric Data of Schemes

Scheme	Use	Hard	Pitch	At Bat	Enc	Dec
No Encoding	1	4	0	0	1	1
Constant Index	1	90	0	0	256	5
Righty/Lefty	1	138	0	0	512	20
ABE	1	384	1	1	768	60
Outs+1	1	270	0	1	768	60
Jersey Number	1	1135	0	1	1280	5
Follow the Leader	0	675	1	1	256	1
Chase the 2	0	179	1	1	256	4
Pumps	0	801	1	1	195	4

Table 2: Normalized Metric Data of Schemes

Scheme	Use	Hard	Pitch	At Bat	Enc	Dec
No Encoding	1	0.004	0	0	0.001	0.017
Constant Index	1	0.079	0	0	0.2	0.083
Righty/Lefty	1	0.122	0	0	0.4	0.333
ABE	1	0.338	1	1	0.6	1
Outs+1	1	0.238	0	1	0.6	1
Jersey Number	1	1	0	1	1	0.083
Follow the Leader	0	0.595	1	1	0.2	0.017
Chase the 2	0	0.158	1	1	0.2	0.067
Pumps	0	0.706	1	1	0.152	0.067

4.2 Adjusting Weights

In order to have an appropriate equation that properly takes each metric into account, we must select weights that reflect the importance of the metric to the overall security of a scheme. Weights can be customized for each coach's preferences but we have looked to create weights that are both unbiased and relevant to the context of each metric.

When analyzing Usability, we recognized that the difference between a 0 and a 1 was whether or not the pitcher had to observe all the signs to decrypt the code or just one sign. However, having to see all versus one of the signs does not make a scheme unusable, just slightly more error prone. While it is nice for a pitcher to have a little less to think about when reading signs, it is not as important as the security of the scheme. In reality, teams use some of the schemes we have given a 0 in Usability. Since our 0 for Usability means harder to use and not unusable, we are giving Usability a weight of $\frac{1}{3}$.

We feel that hardness is one of the most reliable metrics we have created. The simplicity in the creation of the neural net provides an unbiased approach to how long it will take to crack a scheme. In order to provide additional separation for schemes of varying hardness, we have chosen to increase the

weight of Hardness to 2.

For the rest of these schemes, we found it difficult to provide a strong argument that one was more important than another. Both Pitch and At Bat Inconsistency can cause a scheme to be broken if a pattern is allowed to emerge. In addition, Encodability and Decodability can mask the secret and it feels necessary to reward schemes that have more random noise to throw off the adversary. As a result, we decided to keep the weights for these last four categories at 1. This gives us a final equation of:

$$SS = \tfrac{1}{3}U + 2H + PI + API + E + D$$

4.3 Ranking of Schemes

Now that we have an evaluation equation, we can deduce a ranking for our nine schemes. Adding a Score column to a weighted data table below allows us to see the components that sum to each scheme's score.

Table 3: Weighted Metric Data and Scheme Score Calculation

Scheme	Use	Hard	Pitch	At Bat	Enc	Dec	Score	Rank
No Encoding	0.333	0.007	0	0	0.001	0.017	**0.358**	**9**
Constant Index	0.333	0.159	0	0	0.2	0.083	**0.775**	**8**
Righty/Lefty	0.333	0.243	0	0	0.4	0.333	**2.310**	**7**
ABE	0.333	0.677	1	1	0.6	1	**4.610**	**1**
Outs+1	0.333	0.476	0	1	0.6	1	**3.409**	**4**
Jersey Number	0.333	2	0	1	1	0.083	**4.417**	**2**
Follow the Leader	0	1.189	1	1	0.2	0.017	**3.406**	**5**
Chase the 2	0	0.315	1	1	0.2	0.067	**2.582**	**6**
Pumps	0	1.411	1	1	0.152	0.067	**3.630**	**3**

We can then rank and plot these scores to see how they compare to one another. We see that the ranking of schemes (most effective to least effective) is ABE, Jersey Number, Pumps, Outs+1, Follow the Leader, Chase the 2, Righty/Lefty, Constant Index, and No Encoding. Another interesting observation is that there are clear-cut clusters of schemes such that the top two schemes (ABE, Jersey Number), have decent separation using the score from the next three schemes (Pumps, Outs+1, Follow the Leader), which have scores in a much different range than the next two schemes (Chase the 2, Righty/Lefty), and the last two schemes (Constant Index and, No Encoding) have a score way below all the others.

The schemes at the top of the ranking make sense to us. Since ABE is based on the current count, it is constantly changing and hard to decrypt. In addition, ABE has a great Inconsistency metric and a large number of possible encodings and decodings. Jersey Number also does well because by making it possible to have the actual pitch hidden at any of the five sign indices, the variation between at-bats greatly changes.

The next three schemes are all pretty secure but have a flaw that keeps them from being a top scheme. With both Follow the Leader and Pumps, we see that they are hard to crack due to their complexity. However, since these schemes are based off of only the sequence of signs and not at all on the external factors of the game, they become very easy to decrypt once a pattern emerges to the adversary. Outs+1 struggles from being predictable over pitches and innings. It's not a top-tier scheme since each at bat uses the same scheme and the outs go from 0 to 1 to 2 each inning.

The next two schemes, Chase the 2 and Righty/Lefty lack complexity and thus struggle compared to other schemes. Chase the 2 has little variation because it requires a 2 in every sequence, and Righty/Lefty is essentially only scrambled by a binary variable, the handedness of the batter.

The last two schemes are correctly at the bottom of our ranking. As our base cases, Constant Index and No Encoding are both expected to be very easily decrypted.

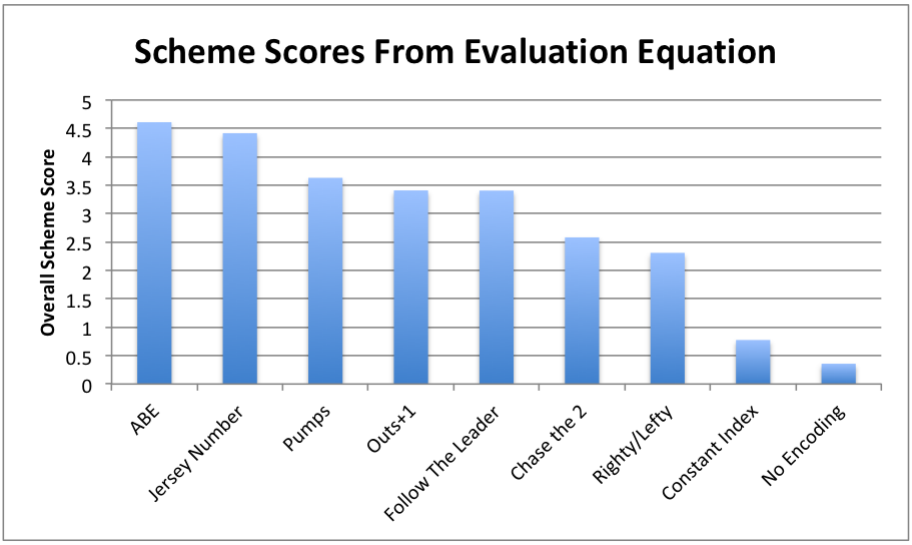

Figure 10: Graph of scheme scores calculated from our evaluation equation

5 Our Scheme

To create our optimal scheme, we looked to take the best elements from the schemes we've seen. We wanted a scheme that was usable, difficult to crack, and appears as randomized as possible. To achieve this, we look to devise a scheme that could be in any of the five pitch indices like Jersey Number and also varied on every pitch like ABE. At the same time, the scheme had to be simple enough as to not confuse pitchers or catchers with things like tricky math.

5.1 Jersey Parity + Num Balls

To achieve the best, across-the-board metrics we combined the uniqueness of the batter's jersey number with a factor that changes with every pitch, the number of balls in the current count. This scheme is a direct mapping from the parity of the batter's jersey number summed with the number of balls in the current count. We map odd-numbered jerseys to 1, and even-numbered jerseys to 2. Then, we sum that number with the number of balls in the current count which varies from 0 to 3 to give us a final value in the range 1 to 5. This value is used as the index in the encryption sequence. Here are some examples with a batter numbered 27 and 2 outs:

$$\textbf{12141} \rightarrow \textbf{1}$$
$$\textbf{23413} \rightarrow \textbf{4}$$

And an example with a batter numbered 16 and 0 outs:

$$\textbf{42312} \rightarrow \textbf{2}$$

The Usability of our scheme is 1 because the pitcher knows exactly which index to look for before the signs are shown as jersey parity and number of balls are predetermined. After altering the step size of our neural net from 1 to 5,000, we were able to approximate a hardness metric of 16,250. Since the number of balls changes during the at-bat, the sign index can change between pitches giving us a Pitch Inconsistency of 1. In addition, the parity of the batter's number will switch between at bats giving us an At-Bat Inconsistency of 1 as well. For Encodability, the sign index could be anywhere from 1 to 5 as discussed above and with four signs for the catcher to freely choose, there are a total of $5 * 4^4 = 1280$ encodings. For Decodability, given that the adversary knows that the number of balls and parity of jersey is being used, they still do not know what mapping odd and even valued jersey numbers have. There are five ways to choose the mapping for the odd number and four ways for the even for a total of $5 * 4 = 20$ decodings.

Usability: **1**	Hardness: **16,250**
Pitch Inconsistency: **1**	At-Bat Inconsistency: **1**
Encodability: **1280**	Decodability: **20**

We can also plug the metrics of our own scheme into our evaluation equation. To do so, we must first normalize Hardness, Encodability, and Decodability. We will simply normalize Hardness to 1 since it is so much greater than the rest of the schemes. To normalize Encodability, we divide our scheme's Encodability of 1280 by the maximum scheme Encodability of 1280 to get $1280/1280 = 1$. We do the same for Decodability to get $20/60 = 0.333$. We then plug our scheme's normalized metrics into the evaluation equation:

$$SS = \tfrac{1}{3}U + 2H + PI + API + E + D = \tfrac{1}{3} * 1 + 2 * 1 + 1 + 1 + 1 + 0.333 = 5.666.$$

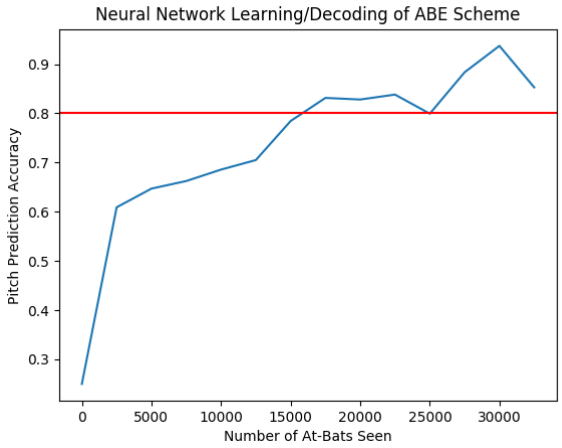

Figure 11: Accuracy of MPLClassifier model on Predicting Pitch given by this new scheme that we devised (Jersey Parity + Num Balls). The critical number of at bats (measure of hardness) is about 16,250.

This scheme score is over a point higher then the best of the nine schemes we tested. While it takes slightly more math to be used in practice, its incredibly high Hardness and Inconsistency metrics make it extremely secure and thus, favorable. There may be other creative schemes that can yield similar results but this was the scheme we found to be both tough to crack and simple to implement.

6 Conclusion

In summary, we have presented a novel, proof-of-concept analysis of 9 of the most common methods of communication between the pitcher and catcher in the game of baseball. We have framed the problem as a cryptography problem by thinking of the desired pitch as a plaintext to be communicated from a catcher to a pitcher, and the sequence of signs given by the catcher as the corresponding ciphertext. Our adversary is a member of the opposing team or somebody trying to gain a competitive advantage in knowing which pitch will be thrown beforehand. We have assumed that our adversary has access to an arbitrary number of plaintexts and their corresponding ciphertexts, possibly by watching replays of the scheme being executed.

We have also used what we learned in our analysis of those 9 commonly used schemes to generate our own scheme, dubbed Jersey Parity + Number of Outs, which outscores all of the other schemes.

For those wondering how likely and/or realistic it is for an adversary to go to such great lengths to crack these schemes, we leave you with a few (perhaps,

comical) examples from the history of sign stealing in the game of baseball.

In 1948, Hall of Fame pitchers Bob Feller and Bob Lemon set up a relay system using a military-grade gun sight to view the catcher giving signs to the pitcher. With this hack, they ended up winning 19 of their last 24 games to clinch a playoff berth.

In the 1950s, Comiskey Park in Chicago was outfitted with a scoreboard that had a platform from which an employee with binoculars could spy on the opposing catcher, but a hidden light – visible by the batter – that flashed in accordance with the upcoming pitch.

Although our analysis is not a cryptanalysis in the truest sense, we hope that our work can be viewed as a unique and novel proof of concept, especially for those with a penchant for baseball.

7 Appendix

Below is an example of the code file that we used to generate sequences for the ABE scheme. This file generates random sequences, determines what the actual thrown pitch would have been, and simulates the at bats and innings in a baseball game. Further, it takes as argument numAtBats, and iterates through all possible dataset sizes from 2 to the value of numAtBats. It trains a neural network based from half the number of at bats, and tests it on the other half. We pick the value at which the average of 10 consecutive neural network trainings first exceeds our threshold of 0.8. We have a similar file for each of the 9 schemes (including our new one, but excluding the no encoding scheme). We then graph all this data and use this graph in the figures throughout this paper. All the other code from the project resembles the same structure as from this scheme and can be found at: https://github.com/maxlancaster/857-final-project.

```
import random
import matplotlib.pyplot as plt
import matplotlib.axes
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

def abeGenerator(balls, strikes, scheme, outs):
        sequence = []
        for i in range(5):
                sequence.append(random.randint(1,5))
        actualPitch=0
        if balls>strikes:
                actualPitch = sequence[scheme[1]]
        elif strikes>balls:
                actualPitch = sequence[scheme[0]]
        elif strikes==balls:
                actualPitch = sequence[scheme[2]]
        sequence.append(balls)
```

```
        sequence.append(strikes)
        sequence.append(outs)
        return sequence, actualPitch

def pitch(count):
        result=random.randint(0,1)
        count[result]+=1
        return count

def generateData(atBats):
        scheme = [0,1,2]
        data = []
        actualPitches=[]
        outs=-1
        for i in range(atBats):
                count = [0, 0]
                outs += 1
                if outs==3:
                        outs = 0
                while count[0]!=4 and count[1]!=3:
                        abe=abeGenerator(count[0], count[1], scheme, outs)
                        data.append(abe[0])
                        actualPitches.append(abe[1])
                        count = pitch(count)
        return data, actualPitches

numAtBats=900
X, y = generateData(numAtBats)
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5)
clf=MLPClassifier()
clf.fit(X_train,y_train)
print numAtBats, clf.score(X_test, y_test)

datapoints=[]
for i in range(2,numAtBats):
        X, y = generateData(i)
        X_train, X_test, y_train, y_test = \
                train_test_split(X, y, test_size=0.5)
        #print X_train, X_test, y_train, y_test
        clf=MLPClassifier()
        clf.fit(X_train,y_train)
        datapoints.append((i/2, clf.score(X_test, y_test)))
        print i, clf.score(X_test, y_test)
fig, ax = plt.subplots()
plt.plot(*zip(*datapoints))
```

```
ax.axhline(y=0.8, c='r')
ax.set_xlabel("Number of At-Bats Seen")
ax.set_ylabel("Pitch Prediction Accuracy")
ax.set_title("Neural Network Learning/Decoding of ABE Scheme")
# 384 at-bats needed
plt.show()
```