Group 3
John Blodgett, Harshit Mittal, Shubham Sengar
Professor Anderson
CSC 466

Cryptographic Hash Algorithm Identifier - Final Report

Project Overview and Relevance:

A key component that determines the degree of security of a cryptographic hash algorithm is its ability to produce hash strings that are indistinguishable from random. The goal of the project is to use machine learning to train models that take in as input a cryptographic hash string and details about the file that it was generated from and attempt to identify the cryptographic hash algorithm that was used to generate the hash string. This in turn extends to being able to determine the pseudo-randomness of a hash string. If we are able to correctly predict what hashing algorithm was used to generate a hash, that may expose a huge security risk in the algorithm as it narrows down attack methods that an adversary might use against them. This will help us assess the strength of a cryptographic hash algorithm and figure out which ones are the most secure to use.

Data Collection:

For the purposes of this project, we needed lots of bit sequences (structured or unstructured) and their associated hashes. To obtain the bit sequences, we opted to download lots of files from online sources. We found datasets on GitHub and Kaggle that contained multiple files in different formats. For example, this dataset contains a lot of .txt files that are available for download. We found similar datasets for .pdf, .txt, .json and .jpg files. We wrote a script to get the metadata and hashes of these files and appended the data to a csv file for data processing. To get the metadata associated with each file, we used the os python library, in addition to writing some code to calculate the file distribution and properties like the maximum consecutive count. Finally, we also used the hashlib library to get the SHA1, SHA2, SHA3 and MD5 hashes of all the files that we had downloaded. Of course, we decided not to store the files since the contents of the files were not directly relevant to our project, and storing them required inordinate amounts of memory. Nevertheless, we did store the names of the files, so that we could obtain them again if necessary at any point.

Data Processing:

After having generated the “raw” data, we needed to process it. However, first, we distinguished between processed and raw data and kept each in separate directories. Finally, we generated a script that processed unprocessed data as follows. We started with randomly choosing a single hashing algorithm for each row of the raw data. We did this because we didn’t want our files to overlap when we were comparing hashes from different algorithms. Next, we wanted to take our file meta data and use as many different pieces as possible. We found the total distribution of 0s and 1s and each hex value in the hash, as well as its length, counted the number of 0 and 1 bits, and also directly used our file distribution of bytes to calculate the total file size. We then dropped duplicate values and removed unnecessary or redundant columns. The previous steps were all carried out on raw data, in order to create a corresponding processed data file. Finally, we had to combine all the processed data into an overall total_data.csv file, which would represent our dataset. We accomplished this by appending dataframes to each other using pandas, and writing the conclusive dataframe to a csv. As mentioned, all of these steps were combined into a “script” ipynb that processed the unprocessed raw data files and finally used all the processed files to generate one large overall file representing our dataset to be used for our models.

Exploratory Analysis:

We wanted to examine whether hashes had different properties by performing some exploratory data analysis on our total data. The purpose of this was to inform and motivate our strategies when we approached the machine learning block of the project. First, we determined if the number of 0 and 1 bits in the bit representation of the hashes differed between hashes created by different algorithms. Unsurprisingly, there were differences, largely due to the fact that hashes produced by different algorithms varied in length, and thus a hash with a greater overall length would also obviously have more overall 0/1 bits. However, when we normalized the number of 0/1 bits by the length of the hash, we saw no difference between the distributions among all the hashes; they were all approximately 50%/50%. We also decided to examine whether there was any interaction between the hash algorithm and the file type the hash was generated from on the distribution of the hash. Once again, we found no significant difference. However, we did find a difference in distributions of the normalized hex values between the hashing algorithms, with some hex values differing from the expected relative frequency by almost two orders of magnitudes greater than other hex values. This indicated that the distribution of hex characters in the hash should be a

feature we included in our machine learning models. Lastly, we decided to see if separating the hashes into chunks would aid in discerning distinctions between the hashing algorithms. Unfortunately, we did not see any important distinctions, so we decided not to chunk the hashes while creating our classifiers.

Initial Model Building:

After exploratory analysis, we decided to advance to the model building stage. Because of the nature of our data (its proclaimed randomness, as well as the strong correlation between multiple sets of columns of our data), we were unsure of what algorithm would be best suited to classifying hashes to hashing algorithms. As a result, we decided to create mini models based on multiple different algorithms (sklearn implementations) in order to determine how each classifier was doing. Of course, we followed standard best model construction practices, including splitting our dataset into training and testing portions. We implemented a pipeline where we could train whichever model we chose while performing k-fold cross validation. We also tuned the hyperparameters each model had to get the lowest possible cross-validation error and plotted those results to show the performance of the classifier as a function of its hyperparameters. For example, for a KNeighborsClassifier, we varied the number of neighbors parameter and noted the fall/decline of the model's performance based on the value of the hyperparameter. As previously stated, we did this cross-validation, hyperparameter optimization, etc. for multiple different classifier algorithms. Ultimately, we ended up trying the KNeighborsClassifier, MLPClassifier, DecisionTreeClassifier, GradientBoostingClassifier, RandomForestClassifier, and AdaBoostClassifier. Naturally, we had to optimize different hyperparameters for the different algorithms. Furthermore, before creating the models, we decided to perform some crucial, decisively advantageous preprocessing to data before it was fed into the models. We scaled some of the numerical features in our dataset, which was necessary since some of the classifier algorithms we were using calculated distances. Furthermore, we needed to incorporate our file distribution of bytes into the models. Earlier, we had simply dropped it from our training/testing data, since it was a list of 256 numerical elements for each observation (hash), and expanding it would lead to the curse of dimensionality and also greatly increase model training time. Therefore, we decided to reduce the dimensionality of that feature itself by expanding it into 256 columns and performing a principal component analysis on it. Luckily, we were able to capture 93% of the variance in the file distribution of bytes in just 2 components, so we opted to replace the file distribution feature with 2 features

representing the 2 principal components instead. These preprocessing steps greatly increased our models' efficiencies, raising the accuracy of the KNeighborsClassifier by over 25% and of the MLPClassifier by over 60%!

Model Finalization and Performance:

Finally, we had to pick an algorithm and generate a final model. We noticed that all the algorithms delineated earlier were converging to roughly the same accuracy/F1 scores, around 74%. This demonstrated that we could essentially pick any classifier we wanted while guaranteeing the same performance. However, we wanted the model to be efficient (fast) to train, simple to interpret, and one whose hyperparameter tuning reflected real, significant contrasts in model accuracy. The KNeighborsClassifier met all three of these conditions, so we decided to continue with it. From here, we performed some more hyperparameter tuning, including varying k to higher ranges, as well as varying the power of the Minkowski distance metric. After we found the confluence of those hyperparameters that had the least cross-validation error, we started to perform feature trimming on the model. We decided to perform feature trimming since models with a smaller number of features are usually simpler and therefore more interpretable, and also tend to reduce overfitting. The biggest reason we had not implemented substantial feature trimming prior to this stage was due to the limited breadth of data we had - after all, our dataset really only contained the hash, properties of the hash, broad properties of the file, and the 2 PCA components created earlier. Due to how limited and closely related the features were, we had decided to not trim until the final stages of the project. Our method of trimming involved running feature importance on the current model, and then noting all of the important features in that model. Any features not listed as important would subsequently not be selected for the next model, and this process would continue iteratively until all the features in the model were deemed important. Of course, this comes with the caveat that features that were created by dummifying a column must either all remain in the model or all be removed, which is the practice we followed. After all the aforementioned stages had been completed, we had arrived at our final hash algorithm classifier, which is described below.

Model Insights and Results:

Our final model predicted based on the length of the hash (in bits), the number of 0 bits in the hash, the number of 1 bits in the hash, the maximum consecutive count of any byte in the file, as well as dummified variables associated with the corresponding byte with the

maximum consecutive count. Furthermore, the model was mostly relying on the length of the hash in order to predict, and would perfectly predict hashing algorithms with unique hash length outputs, but fail to repeatedly, accurately distinguish between hashes with lengths that were outputs of multiple algorithms. Nevertheless, the existence of significant columns that were not related to the hash length and the fact that the hash length only model had a drastically lower F1 score than our final model implies that there might be some relationship to be gleaned. Specifically, the fact that the consecutive byte was important signifies that perhaps hashing algorithms fall into some patterns, or at the very least differ from randomness, when provided "periodic" (includes the same byte consecutively multiple times) data. Still, the fact that the hash length was the most important predictor leads us to declare that hashes produced from the algorithms included in our dataset (SHA1, SHA2, SHA3, and MD5) are indeed indistinguishable from random. Thus, although our model isn't very complex and doesn't quite have stellar performance, at least we know that those algorithms are cryptographically secure as determined by our model. (Never mind the fact that some have already been broken - the human mind really is better than computers!). Surprisingly, and also somewhat reassuringly, the file distribution and file size were not important predictors while classifying. This is reassuring since it does indicate that the final hash and both the contents of the file as well as its size do not interact to leak information about the hashing algorithm it was generated from. This is a natural extension of the guarantee of one-way-ness, which is another central hash algorithm property. Therefore, our model also indicates that the algorithms are one-way.

Future Work

There were a few interesting takeaways from our final model. When we started this project, we didn't expect to see any promising results from our final model, but since our model was actually able to predict with some sense of accuracy, and not just from the hash length, there are vulnerabilities with these hashes that we could look into in the future. One of the more interesting relationships that we found was that the consecutive byte count was a key feature used to predict these hashes. As previously mentioned, with more time, we would like to further analyze this relationship. Another issue we were having was the amount of computing power we had. During the hyperparameter tuning step, we were noticing that the error of our MLP Classifier was slowly decreasing, but with each iteration, the code was taking longer and longer. If we had more computing power, we could let this run until it reached its minimum error. The last thing we had talked about doing in the future was

adapting our model so we could use it as a benchmark to determine if future hash algorithms are good. We could also try incorporating advanced NLP methods and see if they would be able to increase our model performance.