Predicting Security Vulnerabilities using Machine Learning and Code Metrics

Baran Barut, Michael Harris, Curtis Helsel, Kyle Reid, Thomas Serrano

### Department of Computer Science, University of Central Florida, Orlando, Florida, 32816-2450

### *Abstract* — Software projects are inherently insecure, and security vulnerabilities residing in production systems results in a decrease in the reliability and performance of code that runs the modern world. We present a novel approach to predicting security faults by providing a cross-platform desktop application with a machine learning back-end, capable of making accurate predictions on the likelihood of a file containing a security fault exclusively using the metadata from the file. Also contained is a back-end data pipeline used to train our machine learning model that can extract metadata from project files that are provided to the application. By informing software developers of security fault probabilities for their files, we argue that the awareness for where faults are likely to occur in their projects and overall security posture of their projects is heightened.

### *Index Terms —* Application software, big data applications, machine learning, relational databases, software maintenance, statistical learning.

### I. Introduction

Security vulnerabilities and faults that impact production systems can be costly, especially in real-time systems like those used in vehicles and other critical embedded platforms. With the internet of things becoming real, it is crucial to emphasize how important it is that the code that runs our lives should be safe and secure. To that end, we have designed a portable solution that combines data modeling, machine learning, reinforcement training, and a robust interface to predict the likelihood of a security fault in files. As well as providing useful, targeted information to developers in an effort to produce more accurate code and increase awareness of security issues that may arise over time. The application improves the reliability and functionality of existing platforms, which have the net effect of making our lives safer and increasing the reliability of the technological solutions programmers and software engineers face in the field.

This project was inspired in part by previous work by Thomas Ostrand, Robert Bell, and Elaine J. Weyuker in “Where the Bugs Are,” which was an attempt at predicting faults in code based on metadata and information about the development cycle. Their work used a binomial regression model, and was able to predict correctly, on average, 83% of the files that contained a fault. The main focus in the research was identifying *what* code in a software project should be tested, as opposed to *how* to test code.

Due to the research-driven nature of this project, it is challenging to put quantitative specifications into place since fundamental design choices may constrain future results. However, we can glean specific objectives from the nature of the project’s scope, which has driven forward progress on the project.

Machine learning takes a vast amount of useful data to perform well with accurate results. Acquiring this data and storing it in a prescient manner was one of the most crucial elements of our project. Utilizing publicly available development tools like GitHub’s Application Programming Interface was necessary to identify projects that are good sources of data and yield the best results for our purposes. To facilitate this, we utilized open-source projects to build a realistic model which is then trained to see faults in the wild. Various metrics from code - or metadata - are critically important to making predictions and is what our model uses to find out which files contain security faults. Further, it was critical to provide a front-facing solution that is both easy to use and can quickly provide users and development teams with reliable information that can be used to improve their development cycle.

Machine learning tools, like neural nets, are an excellent way to build a statistical model that can make predictions based on previously seen data patterns. Experimenting to find the right model using different training sets and architectures was essential to get valid information about what is possible to predict through public APIs and bug information. Developing further data fields in the form of file-specific feature points was key to increasing the ability for the model to make predictions.

### II. Project Specifications

Due to the research-driven nature of this project, it is challenging to put quantitative specifications into place since they may constrain the future development of the project. However, we can glean specific objectives from the nature of the project’s scope.

Machine learning takes lots of useful data, so acquiring this and storing it in an easily retrievable manner is a must. Utilizing publicly available development tools, like GitHub APIs and metadata, are necessary to identify projects that are good sources of data and yield the best results for our purposes. To facilitate this, we should utilize open source projects and make the data store easily accessible by all parties. Code metrics, or metadata, are critically important to making predictions and is what our model uses to find out which files contain security faults, and which do not. Further, we need a front-facing solution that is easy to use and able to provide users and development teams with reliable information that can be used to improve their development cycle.

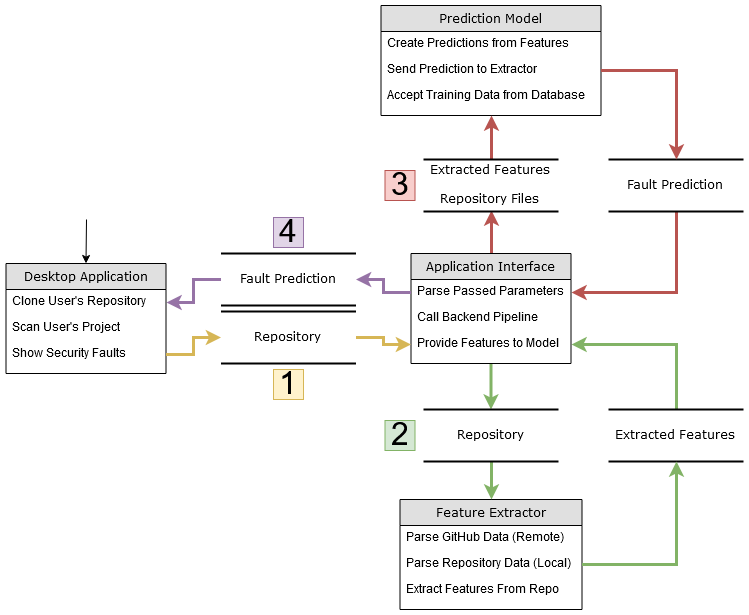
Machine learning tools, like neural nets, are an excellent way to build a statistical model that can make predictions based on data it has seen before. Experimenting to find the right model using different training sets and architectures is essential to get valid information about what is possible to predict through public APIs and bug information. Once development the model is complete, it is imperative to test it upon unseen data to develop a sense of how it can work in the “real world.” Evolving data fields and updating the network architecture are vital at the mid-stage of the project.

Fig. 1. Project Diagram Outline

The above project outline diagram shows the project’s explicit design summary, with the block diagram that follows it elaborating on each segment, and the UML diagram providing a high-level abstraction of how each party interacts with the project.

### III. Methods

Our general question posed is if it is possible to make accurate predictions on the likelihood of files in a project containing security faults using metadata from the project. The four major components necessary for this project to succeed in finding the answer to the question are data acquisition, a machine learning backend, a front-end desktop application, and a backend database and server. Below we describe our methodology using these four components to answer this question.

*A. Data Acquisition*

As mentioned above, making accurate predictions using machine learning requires an immense amount of data. Appropriately, the bulk of our work done lies in the acquisition of data that has been seen to positively impact the accuracy of our results.

Using Python to facilitate data acquisition, homing in on fields for data collection and storing the data to refine further are the critical processes for acquiring data. Once modules are completed, a fully functional script can be written to acquire all meaningful information and store it in the database.

Multiple Python libraries were used to find a way to collect data specific to the files of projects. The data collection was done using the commit log history which had detailed information on the number of files changed and messages related to security vulnerabilities that were needed as well as the metadata associated with the individual files we were parsing through, this data included number of lines, number of indentations, deepest level of indentations, number of subdirectories, etc.. With the commit information, we parsed through the data given and pull more specific data points such as the number of additions, deletions and overall changes to a file for use in our Machine Learning algorithm.

The “PyGithub” library is used to pull specific generic data from GitHub repositories, whereas “GitPython” focuses more on the file specific data over commits. Python’s built-in “OS” library was used extract more detail information from individual files and broader pieces of data on a directory level.

Another major effort undertaken by the data acquisition team was the encoding of data that we received. The data that we extracted through various means needed to be encoded in a way that it could be used to train our model. Each of the data points retrieved is its designated feature in the neural network. The features being used are either numeric or categorical.

Numerical variables (or quantitative variables) are variables that can be measured or counted and have a corresponding number value to them. This data type can be broken down further into discrete or continuous variables. Discrete variables are ones that can be counted but not measured or broken down into smaller components and remain as part of the count. For example, the number of methods in a file, or the number of times a specific function is called. It would be impossible to break down this data into smaller components. On the other side, continuous variables are ones that can be measured and cannot be counted. Examples of this type of data would include the time of day that a commit was added into the commit history. In our database, the discrete numerical variables are stored as integers, and the continuous numerical variables are stored as floats.

Categorical variables are variables that represent the characteristics of the data. This data type can also be broken down further into either nominal or ordinal data. Nominal variables are ones that represent individual values of the data but have no quantitative value to themselves. They have no order to them; they are just values that represent the data. In the case of our project, these types of values would include the programming language used or the author of the commit. Ordinal variables are like nominal variables in that they are a non-numeric way to represent a specific data type but have an ordering to them. For example, if the GitHub API stored information on the level of skill the programmer who created the commit, such as Senior, Junior, Entry-level, these would be considered ordinal variables.

For the project to build a successful model using a neural network, it needs to have all features encoded as numerical values (integers or floats). Encoding is a simple process from the data we receive from the GitHub API that is already numeric as we can store them in their predefined data type. For categorical data, it needs to be modified in a way that can be represented as numerical data.

The process of changing categorical data into numerical data can be broken down into two different processes. The first one being integer encoding and the second called one-hot encoding.

Integer encoding is a straightforward process that mirrors the logic behind ordinal data. It involves assigning an integer value to the categorical value. The encoding is a straightforward process and is easily reversible if the values are consistent among multiple different features. An example as to how we would encode different levels of programming skill into features by labeling each level is as follows:

1. Trainee/Intern Programmer
2. Junior Programmer
3. Middle Programmer
4. Senior Programmer

This type of encoding would be fine for the neural network because it justifies the average competence of the programmer that submits the commit. However, this is not going to be the best encoding process because our data does not contain any variables that can be ordered by significance. We would not want to use this encoding for something like the type of programming language used because we cannot put a value on the order of the language. Personal preference aside, we cannot objectively state that one programming language is better than another and therefore integer encoding is not the way to go with our categorical data.

The other process of encoding categorical data is called One-Hot Encoding. Since we don’t want to include an ordinal pattern to categorical data, we can resolve this by doing a binary representation for each of our categorical data. This process is done in a matrix of the data type as the feature set where the number one (1) represents the value is active, and zero (0) represents that the value is not active. An example of this done on a subset of possible programming languages is shown below in Fig. 2.

|  |  |  |
| --- | --- | --- |
| Java | Python | C |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

Fig. 2. An example subset of possible programming languages when doing One-Hot Encoding

In this table, we can map whether a language is being used but not putting a higher numbering system on languages higher in the list. Representing the data this way prevents the neural network from putting an incorrect emphasis on the language type that happens to be encoded as a higher integer as we would with integer encoding.

We have two sources for our data collection. One is from the GitHub API that pulls information regarding at the repository and commit level. Due to the unique problem we are faced with to determine the file at which the security vulnerability occurs, we need to create additional methods in which to gain features about the file.

*B. Machine Learning*

In the process of acquiring the data, it was quickly discovered that the process of searching for keywords to determine the files which contained faults led to a heavily skewed dataset favoring non-fault files. This was due to the fact that when parsing the commit message to determine if a keyword was found, the majority of the time the keyword was not found or the message itself didn’t provide enough information to ensure that the file being committed actually contained a security fault. The process for pulling this data provided a dataset where nearly 90% of the files claimed to have no faults. This skewed dataset invariably provided a model that would essentially predict that every file would not likely have a security fault. Upon validation, this showed a 90% accuracy in prediction but only because 90% of the dataset was likely to be without faults.

Due to the skewness of the original dataset, a different procedure was implemented in order to gain an equal number of files that would be considered either containing a security fault or not. This new procedure abandoned the process of looking through commit logs for keywords and implemented a process of parsing the CVE database for already known security faults based on user submissions. The database provided URLs for the commit hash where a fault file would lie within the commit log. This allowed us to pinpoint specific cases of security vulnerabilities without requiring us to search through the entire log.

Since we were able to pull out individual hashes from the database, we had to construct our new dataset to ensure that there were equal number of fault files with non-fault files. We did this by first validating whether the commit hash provided contained only a single file. If the commit hash contained multiple files, the files on the commit were set aside as neither valid fault files and unconfirmed non-fault files, internally dubbed ‘grey’ files. Once the list of confirmed fault files and unconfirmed files was created, extensions were extracted from the fault file list and found a similar file in close time proximity of the fault file commit. This was done so that the features extracted from both the fault and non-fault files would not have a large variance in time committed since it’s possible the changes over time may affect the actual features of the file.

The process above gave us an even split of data between both fault and non-fault data. The data for both types of files contained features related to the GitHub repository, such as watchers, stargazers, and contributors, and file only features, number of insertions, deletions, total adjustments. These features were parsed and combined into a single feature set for each of the files to be trained. Through many different architecture changes, the dataset only produced a model that gave a 50% accuracy on the validation set. This accuracy did not change regardless of alterations to the network parameters, such as number of layers, number of neurons per layer, and batch size for the training model. Looking at the feature set, we could see that the features implemented were heavily geared toward repository data but not enough on the file specific data. This could essentially provide some insight on whether a specific project would have a security fault but did not necessarily provide a clean look at the actual file fault prediction, due to the repo fields not being representative of file-specific metrics.

The realization that the features being extracted may not produce desirable results led to eliminate the repository features completely and train solely on the file features. When doing this, the validation accuracy raised to 60% which gave cause to believe that there exists some correlation between the features of the file and its likelihood to contain faults. Currently in progress is the process of training of 21 additional file features to produce a better validation accuracy on fault prediction.

Much consideration was taken when determining the best option for training our data to produce worthwhile results. The type of network that was finally used was a feed forward neural network using backpropagation with a sigmoid function on the final activation.

The sigmoid activation takes a real value input and outputs a value between zero and one. This activation is nonlinear and has a smooth gradient which allows for it to be an excellent activation function for classifying data. A drawback of this function is that outputs that are close to either zero or one tend to respond very little to the changes in backpropagation which causes the training process to be very slow. In the overall model, the sigmoid activation will be the last layer of the network and will classify if a file contains or does not contain a security fault.

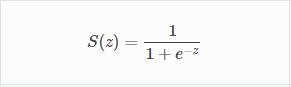


Fig. 3. Sigmoid Activation Function [1]

To gauge the effectiveness of the model, a binary cross entropy function was used for the loss function during backpropagation. This loss function was chosen specifically because the output of the model only fell within two categories: fault or non-fault file.

Feature scaling (also known as normalization) is the process of standardizing data so that more significant valued features do not overpower smaller valued ones. For example, if we look at the two features, the number of authors on a file and the number of lines in a file, obviously the number of lines would be considerably larger in value and would potentially weigh more in our model and skew the accuracy. For this reason, we need to set the magnitude of all features to be within the same level.

In our system, the Standard Scaler function provided from the Scikit-learn preprocessing toolkit. The Standard Scaler method takes the values and replaces them with their z-scores. This process is done by subtracting the mean of the feature from the value and dividing by the standard deviation for the feature. Standardization gives a normal distribution of the data about its mean value set at zero (μ = 0) with a standard deviation of one (σ = 1). The formula for standardization is as follows:

https://lh3.googleusercontent.com/5Ai6JrTsiZdOnzyS19EkeczYA6jKMuGsg4dFhUP215_xxo2TBEAo9D3PBm3LNL6nVLS6zwrGWcF7CGvRlsXRfsa0nuQ9EVz3QoWycMuSwZj9ib8uBma3qtN6heJTIg

Fig. 4. Standardization Formula for Feature Scaling [2]

This scaling technique has the feature values between (-∞, ∞) with a bell curve around the mean of zero which means that 99% of the data should be within three standard deviations of zero (-3,3).

Throughout the modeling process, great care was taken to ensure that the accuracy of the model was validated. Initially concerned with the amount of data we would be able to create we considered multiple options for validation including K-fold and Iterated K-Fold but finally settled on a technique called Hold-out Validation.

Hold-out validation is the simplest of all the cross-validation methods. This process consists of shuffling the training data and splitting the training data at an arbitrary point into two separate sets: training and validation. The neural network is trained on the training set and the model produced is evaluated on the validation set. The point of the split is arbitrary, but in standard practice, the resulting training data is larger than the validation data. In our training, the validation split was 80% training and 20% validation.

The model itself was written in Python using the Keras library on top of TensorFlow. These tools are the logical choices in creating neural networks as they are fully vetted by many machine learning experts and well-known quantities in the industry. The final architecture of the network consists of two hidden networks containing both 64 neurons as this produced the best overall validation accuracy for the model.

*C. Front-End Application*

It is not enough to hand a user a machine learning model and a data acquisition pipeline and tell them to make predictions on their project code. We strive to make our work available for as many users as possible with a focus on simplicity and ease of use. Therefore, we have developed a desktop application written in C#, fully equipped with a user-friendly interface that allows the user to complete the entire scanning process - start to finish - with ease.

Upon starting the program, you are greeted with the “Scan” tab. On this tab you can specify which type of repository you’d like to scan, and any exclusions you’d like to make for things you don’t want to scan. We desired for the user to be able to scan both local and remote repositories to allow for both offline and online scanning. What information must be supplied by the user depends on the type of repository that will be scanned. The following information must be supplied for each type of repository:

1. Local Repository: All that is required of them is the path where the repository is contained on their system.
2. Remote (Public) Repository: We require that the user supply the GitHub URL to the project, and the path on their system that they’d like to clone the project to.
3. Remote (Private) Repository: Besides the already mentioned requirements for a remote (public) repository, we require that the user authenticate with GitHub by supplying their GitHub username and password. These credentials are solely used to authenticate with the platform.

After properly cloning the repo locally (if the project is remote), the user is able to specify the types of exclusions they’d like to make to items that they don’t want scanned by the model. The three types of exclusions the user is able to make are:

1. File Names: To exclude a specific file by name, an example would be “filename.txt” for a file named “filename” with the file extension “.txt”.
2. File Extension: To exclude all files that have a specific file extension. An example being “.txt”, which would exclude all files with that extension.
3. Folders: To exclude folders by name. This was done to allow users to exclude folders that may contain data that does not need to be scanned, or folders that may not pertain to their project but exist within the same folder. This folder exclusion works recursively as well, completely excluding all files and folders contained within the excluded folder.

The user is then able to load the project into our program and begin the scan. Depending on the type of repository selected, the application decides what data to send to the backend data pipeline. By spawning a new Python process and running the script to interface between the application and the backend, the application is essentially able to “communicate” with the backend. Data such as the project location is passed from the application to the interfacing script, and the interfacing script then sends the data to all necessary scripts. Once a prediction is made, the interfacing script provides the output to the application and the application presents that information to the user in a friendly and understandable manner.

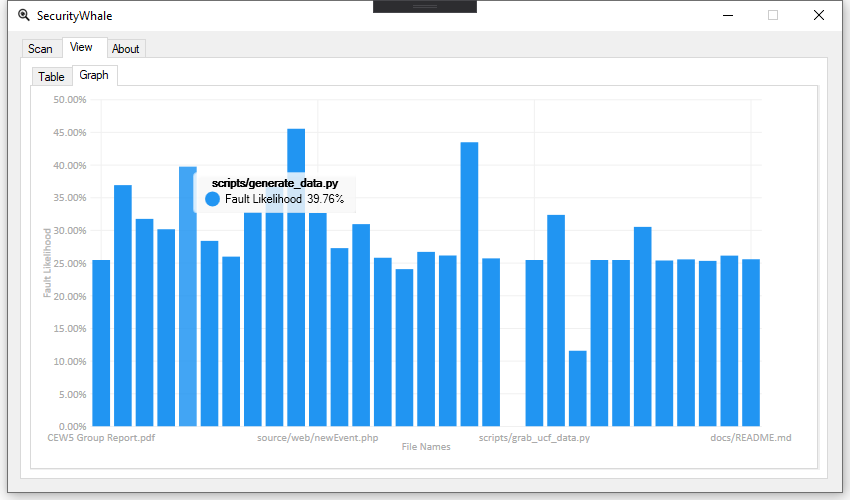
Predictions made on the files from the project are presented in the form of a table and a graph. The table provides each file’s name, and the probability (as a double from 0 to 1, with 0 being 0% and 1 being 100%) of that file containing a security fault. The graph shows this data in the form of a bar graph, which allows users to quickly compare to see which files may be more problematic than others.

Fig. 5. Sample Output from Our Program Showing File Names and Security Fault Likelihoods

*D. Backend Database and Server*

Initially, the first database design contained fields which were specific to the repository that the file originated from. These included such fields that were relevant and common to the repository but were later dropped once it was discovered that they reduced accuracy. Fields like ‘milestones’, ‘issues’, ‘open issues’ were not necessarily relevant of any file specific metrics, but rather were indicators of what the repo as a whole consisted of. Further, since these metrics were applied to each file record, they were over-represented in the modeling and reduced accuracy. At this stage, some of the now present file-specific features existed, such as ‘total inserts’, ‘total deletions’ which indicate how much the file changed through commits. After the introduction of the dual path intake pipeline, it was apparent that the quality of data being pulled by the data collection utility was not going to be up to snuff for the needs of the project. At this stage, we introduced the CVE database parsing, which reliably linked us to problematic commits in repos. This allowed us to identify further file-specific features and write code to allow this information to be stored in the database, as a prelude to training the new model. A comprehensive list of these file fields follows:

Total inserts, total deletions, deletion averages, total lines, line averages, total access, recent time delta, age of file, lines per file, words per file, characters per file, average words per line, average characters per line, total indents in file, number of lines with indents, deepest indentation level in file, number of files, number of subdirectories, number of directory levels, and finally average number of files per directory. The above fields represent the last push for a higher accuracy, and clearly include many metric points about how the file has changed over time and what it now contains.

### https://lh6.googleusercontent.com/G8dKMcr2zvg4l2q4mR2XfeQOe7VvlGMRwDLCdBqKqKzl7naPEDCnhWQDGNB5Ffmw4C7GuPbnfGsCHYX2aYP2uuKyLXzMVAUac-oxS7UICzIKbDs3-1Ze0rFtjItHaOBeQ0w-Q3sVW7IIV. Design Summary

Fig. 6. Python Back-End Diagram

Fig. 6 above shows the flow of how the Back-End handles the information received from the user using our application. Our script calls the application gathering all the information from the user. Once the info is received the script shall collect both GitHub data such as number of contributors and file data such as number of lines in the individual files. After all the data is collected and it is sent to the function that predicts based on the data the results of the prediction are returned back to the application for the user to see.

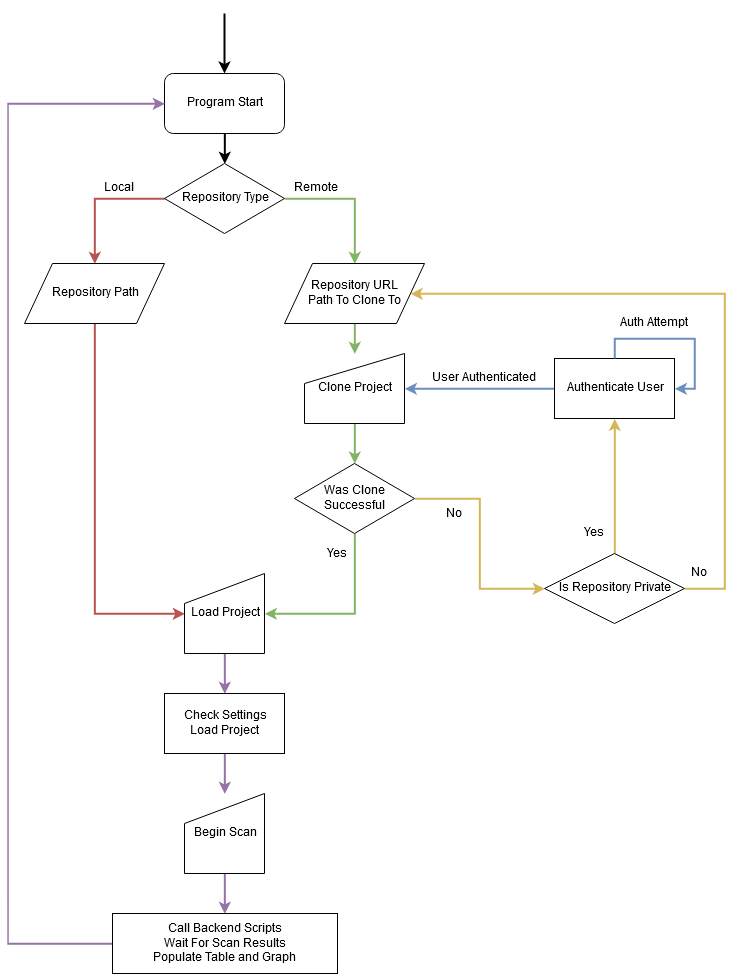
Fig. 7 shown below lays out the front-end application diagram, describing in detail the control flow for the desktop application. The user is presented with the option of a local or remote repository, and then goes through the necessary steps to load the project into the program. Once the project can be loaded, the program will check their settings and load the project. After the user clicks “Begin Scan”, the program will call the backend Python scripts to begin the scanning process and waits for the scan results. When the backend provides the scan results to the program, it will take that information and populate the table and graph in the “View” tab with any relevant information.

Fig. 7. Front-End Application Diagram

### IV. Results

We were able to make predictions as to where security faults may lie in projects with a 60% accuracy. On a per-file basis, our machine learning model with data pipeline backend can correctly predict that a file contains a security fault six-out-of-ten times. This is better than a coin flip and can be used by QA testers or software engineers to obtain a general overview as to where they should focus their security vulnerability finding efforts. We argue that our project should be used in a manner that works in-tandem with current programs and processes that aid in finding vulnerabilities in code and should not act as a replacement. By pairing our program with current vulnerability finding processes, developers can glean more information regarding the overall security posture of their projects. This increase in the amount of information at the developer’s disposal can only beneficial and it is our goal that this helps developers in securing their projects.

Below, Fig. 1 is demonstrating the output that a user would see after scanning their project with our application. As you can see, the application provides the user with a general overview, showing which files may be more problematic and which files may need to be looked at first. Due to the accuracy being “better than a coin flip”, this is beneficial. We see software engineers then taking this information and aiming other security vulnerability scanners at those problematic files to dive deeper into what may be causing those percentages

### V. Conclusions

The most important – and most difficult – portion of this project was collecting features from repositories for use in the machine learning. The greatest barrier to any machine learning project is good data, and after examining available avenues for file-specific features, we were still left wanting to investigate further, either by gathering new features or performing additional analysis and discarding features that weren’t as useful to model accuracy. Additionally, much of the time spent working on this project was spent setting up algorithms to process and format these features and assembling the pipeline that connects each individual part of the project. However, once these tools were assembled, progress proceeded at a much quicker rate. Being able to test every piece of the project in order greatly accelerated development, but it also created new difficulties due to having to debug a much larger program. Given more development and refinement time, we believe the accuracy of our product would be able to provide much more accurate predictions. This project has demonstrated that fault detection through file metrics is possible, but this shows how much can still be done in this area to hone in on a higher prediction accuracy. Ultimately, the team was able to answer the question “is it possible to make accurate predictions for security fault likelihoods using code metrics and machine learning?” We found that yes, it is possible to make accurate predictions as to where security faults lie in git repos, and that by utilizing our program, one can gain insight into a high-level directional overview as to where faults are likely to occur.

### VI. Legal, Ethical, and Privacy Issues

Since our front-end Application can run locally without an internet connection, this avoids any potential legal pitfalls by preventing any storage of code or code derivatives in the form of generalized metrics. It is impossible for a server to be breached or source code to leak as a result of running our Application. Information stored in the database is about specific public repos which have publicly disclosed security vulnerabilities, so the database or model do not store anything which could be viewed as sensitive.

Ethically, the Application will only allow further examination into potential problematic files, so no dilemmas should be posed by the software as it exists currently. Like Dr. Weyuker’s work, there is no mechanism by which to assign responsibility for any issue, or potential issues, to a certain party. A high percentage ranking that a file may contain a fault only suggests that metrics may warrant a further look at this file. The only problem that could be posed is the thought that a bad actor would now have the ability to scan repos that it is looking to target with the aim of finding security vulnerabilities. We argue that any sophisticated attacker would already have tools like this at their disposal and that this only benefits the overall security posture of software developers. By giving the “good guys” tooling that may be used to attack their code, they can use such tooling to protect their code from such attacks.

In an early phase of the project, a feature was planned where a repo scanned by the Application could be submitted to the database for future training iterations, which would necessitate some sort of privacy agreement with the user, but due to the inability to guarantee that claimed bugs were actually problematic files, the threat of contaminating the data set with false positives was deemed the highest priority, so this feature was deemed unfeasible for this iteration of the program.

### Acknowledgement

The authors wish to acknowledge the assistance and support of their advisor, Dr. Paul Gazzillo, and the contributions and expertise of Dr. Elaine Weyuker. As well as the continued support from their Senior Design professor, Dr. Mark Heinrich.

### References

[1] Avinash Sharma V and Avinash Sharma V. 2017. Understanding Activation Functions in Neural Networks. (March 2017). Retrieved April 19, 2019 from https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[2] Sebastian Raschka. 2014. About Feature Scaling and Normalization. (July 2014). Retrieved April 13, 2019 from https://sebastianraschka.com/Articles/2014\_about\_feature\_scaling.html