Software Assurance: Defect Localization

Austin Moreau
William Bogardus
Duy Nguyen
Pablo Salinas

admoreau@uh.edu
whbogardus@uh.edu
djnguyen3@uh.edu
plsalinas@uh.edu

Research Report

Prepared for

The University of Houston
And
Johns Hopkins University

5/4/2018

Developing a Better Static Code Analysis Tool

Keywords: Machine Learning, Cybersecurity, Bioinformatics, Software Vulnerabilities, Static Code Analysis

The aim of this project is to utilize machine learning methods developed for genomic sequencing to more effectively analyze binary executables for feature in code which will reliably indicate security vulnerabilities. First, by translating the binary to a DNA base analogue then, uploading this information to the online genome sequencing algorithm mVista, and finally processing these results the researchers were able to discern a set of features. Then with this information were able to identify the levels of occurence of the features in a set of 10 safe sets of code and a set of 10 unsafe pieces of code. Then through data analytics, the researchers were able to discern the unsafe from the safe code with 92% accuracy.

EXECUTIVE SUMMARY:

As long as humans are involved in the process of writing code there are bound to be errors in their code. As the code has continued to increase in length and complexity so too have number of bugs. When the code for a single project can have over 1 million lines or more, it will be impossible for individuals or a team to comb through it to find security vulnerabilities manually. This issue has led to the development of static code analyzers and analysis techniques designed to automate the process of objectively investigating and reporting vulnerabilities in code. It is our hypothesis that by compiling the source code into machine code and then letting machine learning algorithms identify a set of features indicative of code quality, that it will improve on the current state of these static code analysis tools by identifying a specific set of data that can be understood to have a strong relation to the security of a code base.

The process begins with the translation of binary object files from the Juliet Test Suite, a balanced code base containing code and functions known to be a security risk as well as the ameliorated code with this risk removed. The binary code created from the Juliet Test Suite files were then translated to DNA base pairs so as to format it with the necessary file format, a format known as FASTA which is typically used for genome sequencing programs, to then be uploaded to the online version of the mVista genome sequencing software suite. Once this was accomplished the researchers had to develop web scraping tools to pull the resulting data from the mVista software and create a database of weak similarities between individual code sequences. Next, these pieces of code were compared to one another to find a set of features that

were similar to every other feature returned by the mVista software at a desired level. Using these features the researchers created a machine learning program that could effectively classify the float representation of the occurrence of these strings in a set of test cases.

Software Assurance: Defect Localization - Research developed for The University of Houston and Johns Hopkins University as part of the INSuRE cybersecurity research project. This Software Assurance: Defect Localization Research was carried out by Austin Moreau, Pablo Salinas, William Bogardus, as well as Duy Nguyen. All of these researchers are students at The University of Houston in the Computer Science program. This research will be carried out to further the development of technology to protect against potential threats from security vulnerabilities in source code using machine learning methods initially developed for use in bioinformatics as well as open source compiler software.

TABLE OF CONTENTS:

INTRODUCTION:

Software defects within the scope of this research project are any security vulnerabilities in code commonly created as a human oversight in developing and maintaining code. Such errors are easy to make across large projects and should be both expected and subsequently accounted for. However, when tasked with debugging tens of thousands or more lines of code, it is not feasible to rely on human analysis. In fact the average software developer is unable to effectively search a code base for bugs in a timely and effective manner. [1] A number of tools exist for static code analysis but none of them are 100% effective and in world increasingly dependent on software, bad actors will seek to exploit faults in these tools or vulnerable software as well as seek to develop new tools or methods that will circumvent known security analysis methods.

Statement of Purpose:

The primary goal of this project is to identify a technique for classification of code as either secure or insecure with the eventual desire to develop a framework and methodology with the intent of creating one or more machine learning algorithms which will possess the ability to be deployed to real world code analysis in an overarching and singular program. The final format of

the proposed software will be adequately effective when tasked to discern whether or not a section of source code will contain a potential security vulnerability with more accuracy than known static analysis systems by utilizing trainable machine learning algorithms capable of software vulnerability recognition at the machine code level. This will allow any necessary human oversight to more effectively debug massive amounts of code in a more timely manner.

Motivation:

Computers and software have become ubiquitous throughout society and this trend of the world becoming steadily more reliant on computers is not hesitating. With this exponentially accelerating deployment of computing technology comes an equal if not more rapid deployment of software to run these machines. One of the most used software repositories on Earth, the Google codebase [2], contains billions of lines of code alone. Analyzing this amount of code for security vulnerabilities or bugs, could take hundreds or thousands of man hours and is not guaranteed to be successful as the shortcomings of human oversight of code security is well documented [1].

Lloyd's Emerging risk report 2017, produced with assistance from Cyence [3], which created and studied a scenario where several levels of losses from compromised computer systems were simulated with the intent to better understand the potential loss a single company would experience due to security vulnerabilities from poorly developed and insecure code. The report states that, in the extreme case, losses incurred can cost up to $28.7 billion and take up to 248

days to remediate effectively. This was the case for the issues one single company could expect to realize.

The authors of [4] tested open source bug detection softwares SPLINT, ARCHER, BOON, UNO, and Polyspace over 44 vulnerable functions of which they had false alarm rates up to 50%. Many of these programs rely on Natural Language Processing and are thus constricted to either a certain language or only being able to detect certain types of bugs. Lastly, said programs leave many gaps in bug detecting due to the methodology of NLP which can be confused by function and variable naming. Based on a study in 2017 from [5] in the table below, the programs individually were unable to detect 60% of the bugs accounted for in the test suite provided by Toyota ITC**.**

Bugs Detected in Static Analysis Programs

	CppCheck	Splint (weak)	Splint (standard)	Clang (core)	Clang (alpha)	CodeSonar	Facebook Infer
Tool version	1.72	3.1.1	3.1.1	5.0.0	5.0.0	4.4p0	v0.12.0
Detected	132/1915	79/1915	344/1915	125/1915	326/1915	651/1915	149/1915
Extra	42	229	2809	125	1406	649	39
Total analysis time (seconds)	5.80	2.33	1.88	293.62	580.84	1346.86	159.52

This research seeks to improve on the areas mentioned above by using machine learning techniques and statistical methods to more reliably find software bugs. By compiling and processing code down to binary, which achieves the largest data set available with the highest level of consistency, it was found that the machine learning algorithms will have the opportunity to perform static code analysis to detect software vulnerabilities.

It is the belief of the researchers that by the very nature of computer code, it is not comparable to a natural language. What is meant by this is that, the only reason code is even close to anything the average person can understand is because it has to be for humans to be capable of altering and working with it in a reasonable way. True machine code such as Assembly and further the binary executables actually executed by the machines are extremely difficult or impossible for the layman or even advanced users to understand. This reason is the primary issue behind the fact that binary code can not be decompiled. Due to this, the researchers hypothesized that, in this state, binary should possess the most comprehensive and verbose explanation of the processes occurring within the functionality of software. With the language of binary code represented in two numerical characters and its comparability to DNA which is represented in 4 bases, the researchers believed that methods used to understand and classify DNA would be far more applicable if indeed the binary does contain consistent patterns capable of indicating its functionality.

Broader Impact:

It can be assumed that should a program, a program which is capable of discerning potential security risks in the source code of any developed software automatically and with great precision, be created through this research the world will become more secure. Even if software were to stagnate in its deployment and proliferation throughout the world, any major risk posed to these systems can be considered massively significant both economically and socially. Should

this program be developed, the amount of automation and security assurance which could be introduced to the software development industry could free up many hours developers would have required for finding and remediating any bugs this developed program will be capable of preventing, again this prevention could be autonomous and ideally require little to no human oversight.

In the event that this software is in fact not sufficiently effective in the prevention of the creation of insecure code bases, this research would also create a learning opportunity. The application of machine learning algorithms developed for genomics and bioinformatics to binary computer code for cybersecurity purposes can be considered novel and worth devoting research to. In summation, if this research is successful, the world of insecure software where billions of dollars are lost, massive insecurity on the part of every user, and in general a lack of confidence in any institution that uses computer programs rife with bugs and poorly developed, will be a world of the past. The future with the algorithms developed from this research will be secure and confident in the quality of the code that governs everything from federal databases, social interaction, banking systems, commerce, industrial and manufacturing, to the potentially infinite applications yet to be developed.

LITERATURE REVIEW:

The road forward for accomplishing a satisfactory test of the hypothesis will include the compilation of higher level languages into the binary executables. Finding features analogous to

security vulnerabilities through analysis of the binary executable by algorithms initially developed for relating genotypes and phenotypes which will be repurposed for application to binary code instead of DNA base pairs. After any features are recognized as malicious, the features will be categorized and made available for reference by a machine learning algorithm that will be developed to analyze a test case for any similar code structure. Due to the nature of the software defects, occasionally a security vulnerability will be the result of a missing function call or undeclared variable, in other words code absent from the source code may lead to issues later on. Testing for these cases based on pattern matching algorithms may be difficult at best and at worst nearly impossible. For these cases it is believed that the development of similar pattern matching algorithms developed to recognize proper code structuring as opposed to dangerous structuring will circumvent ineffective code analysis. These algorithms will be used in tandem.

Justification for the strategy outlined previously is related to previous research developed for the use of machine learning in static code analysis. [6][7][8] It is believed that machine code will contain patterns that can be analyzed to more effectively identify security vulnerabilities. Also it is assumed that the genomic sequencing algorithm that will be repurposed for binary may be able to recognize patterns in the binary executable that a human observer would either not be capable of identifying or may overlook with the belief that it is not worth concern.

The researchers decided to utilize the popular genome alignment program, Shuffle-LAGAN presented in [9] for finding similar strings in separate instances of the translated binary created

from the Juliet Test Suite. Shuffle-LAGAN is a combination of global and local methods called glocal alignment, where one algorithm creates a map that transforms one sequence into the other while allowing for rearrangement events. Pairwise alignments of genomic sequences come in two main classes; global alignment where one string is transformed into the other with a combination of edits and local alignment where all locations of similarity between the two strings are returned. The advantage of Global alignment is that it is less likely to establish false homology because each letter from one sequence can be aligned with only one letter from the other sequence. Local alignment is good with dealing with rearrangements between sequences of different chromosome that are incompatible with each other by identifying similar regions in the sequences. However, local alignment does not take into consideration overall conservation maps which leads to a higher false positive rate.

The Shuffle-LAGAN algorithm is made up of three stages. In the first stage, the local alignments between the two sequences are found using the CHAOS tool. In the second stage, the maximal scoring subset of the local alignments under certain gap penalties is picked to form a 1-monotonic conservation map which means that the map must be non-decreasing in only one sequence. Finally, the local alignments in the conservation map that are part of a common global alignment are joined into the maximal consistent subsegments, which are aligned using the LAGAN global aligner. The subsegments are aligned by sorting the local alignments in the 1-monotonic map by their start coordinates, taking the first alignment to be the start of a consistent subsegment, and adding additional local alignment while they are all consistent.

The Shuffle-LAGAN is a glocal alignment algorithm based on the CHAOS [10] local alignment algorithm and the LAGAN [11] global aligner to align long genomic sequences. The CHAOS algorithm works by chaining together pairs of similar regions, one from each of the two input DNA sequences, called seeds. A seed is pair of words of length k with at least n identical base pairs. Seeds can be chained together whenever they are "near" each other which is defined by both a distance and a gap criteria. The final score of a chain is the total number of matching and mismatching pairs in it. CHAOS throws away chains with a score below a specified threshold. The LAGAN algorithm aligns pairs of sequences in three steps. First, it will generate a local alignment between two sequences using the CHAOS algorithm, which are each assigned a weight. Next, the highest weighted chain of local alignments is computed using the Increasing Subsequence algorithm. These two steps are applied recursively for all alignments in chain. Finally, the global alignment is computed using Needleman-Wunsch-like dynamic programming within the subset of cells in the alignment matrix.

The ideas being explored in this research are in all likelihood not foolproof. Even the best developed machine learning algorithm will not work one hundred percent of the time. Regardless the necessity of the exploration of alternative or new techniques to solve problems can be summed up in a quote from Edison on learning from failure with respect to the development of a better light bulb "I have not failed 700 times. I've succeeded in proving 700 ways how not to build a lightbulb." It is a common strategy in finding solutions to problems to break the problem down into smaller pieces. If breaking down code into its binary form for analysis fails to yield better results than other methods, then it can be surmised that drastically different thoughts and

methods may need to be developed for finding a solution to the issues for which a resolution is sought in this paper.

METHODS AND PROCEDURES:

Characterization of Methods:

A combination of both constructive and exploratory research methods are applied in this research. This research is considered by the researchers to be exploratory in that the application of genomic sequencing alignment algorithms on the binary format of the source code in an attempt to discover new methods for identifying and classifying features is considered to be an exploratory process. Furthermore the research is considered to be constructive in that the proposed and studied method of bringing the source code to binary format would transform the binary into a DNA analogue and subsequently implement a genomic sequencing algorithm to identify the features in this analogue. As far as the researchers are aware, there has been no previous research done which consists of implementing genomic sequencing algorithms to find similarities in binary format code in order to identify features to be used in classification.

Procedures/Plan Overview:

1) Develop a method of taking a high level computing language from the cases contained within the Juliet Test Suite and translating it into a consistent binary machine code representation with the proper functionality.

2) Create library of training data from these binary files.

3) Repurpose known genome sequencing program mVista for the purposes necessary for this research. To do this, the binary must first be translated into a DNA representation and formatted with the FASTA format necessary for use with the mVista online program.

From this library of translated binary sequences, upload as large a sample as possible will be processed with the mVista software.

4) As the online version of mVista does not have a simple retrieval method for the results detailing the subsequences of the uploaded sequences which contained similar strings, a webscraper must be developed to automate this retrieval and consolidate the mVista results into easily accessible text files.

5) Once a database of text files containing comparisons made between individual sequences is made. A method was created to compare a single sample from the comparisons to every other sample contained within the files returned by the mVista program to create a final set of features which are all at least 90% similar to one another.

6) The first sample uploaded to mVista were cases considered to be secure. Steps 3 through 5 were then repeated with cases considered to be insecure so as to create two sets of features indicative of both secure and insecure code.

7) A sample of 20 cases from the translated binary sequences which were not uploaded to mVista was created where 10 of these cases were considered insecure and 10 were considered secure.

8) In each case from the 20 samples, every substring of comparable length, of each of these cases was compared to each each feature one by one using a window that began at the beginning of the sample from the 20 files and ended at the end of that sample and moved one character at a time. The highest level of similarity, stored as a float value, between the feature created from previous steps and the sample being examined was stored in an array where the final array value was a classification value. If this float value was less than .5, the number 0 was stored instead.

9) Using these arrays as training and testing values, a machine learning program was created to attempt to classify these arrays as either secure or insecure cases.

Deliverables:

The following deliverables are all available on the PURR Website in 2018 INSuRE

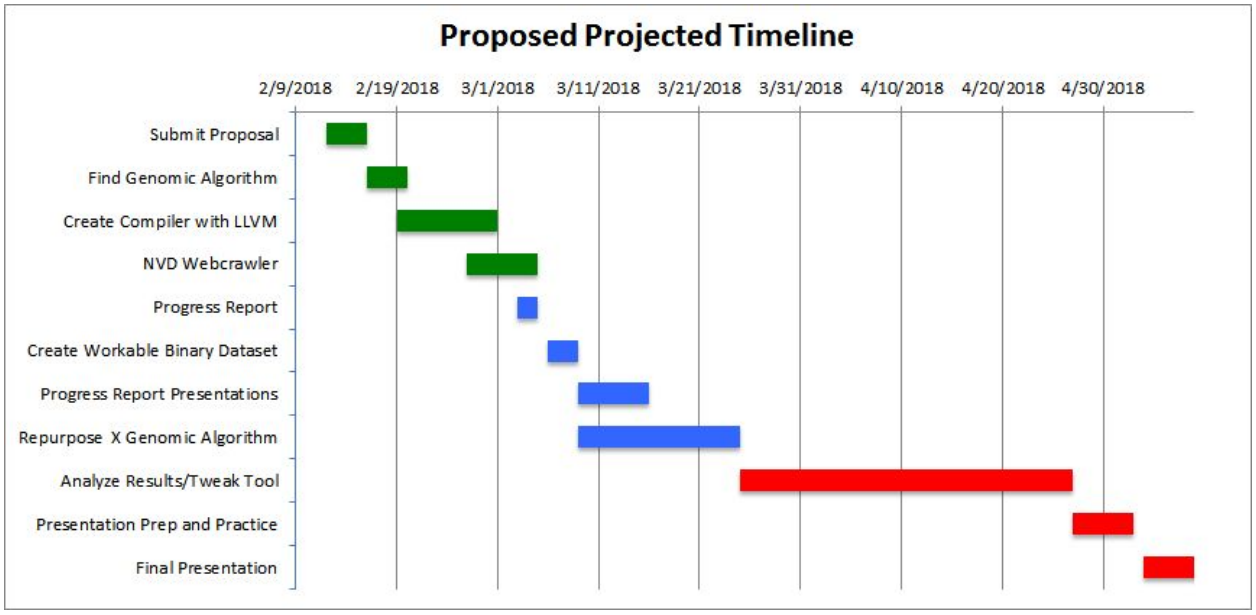
1. Weekly Progress Reports presented to University of Houston Professor Rakesh Verma

2. Midterm Progress Report submitted to John Hopkins March 9th

3. Progress Report presentations delivered to John Hopkins University March 9th and 16th

4. Final Report submitted to University of Houston and John Hopkins May 4th including all code developed as described below
5. Final Report presentations delivered to John Hopkins University May 4th
6. Poster presentation delivered to University of Houston May 10th

Schedule:



Limitations and Delimitations:

mVista's Shuffle-LAGAN algorithm places a couple of constraints on the project. The program can only be used to compare a maximum of 100 sequences. This can potentially create issues as the subsequent features used for classification testing are limited to what sequence matches Shuffle-LAGAN finds. Shuffle-LAGAN is not symmetric, and therefore each sequence and its

inverse pair must be checked in the event of different results. For example, sequence pair 1-52 and 52-1 may both find different alignments.

Time was a limiting factor which narrowed the potential for greater and broader research goals and resulted in a limit to more extensive testing. Graph tracing and decompiling back to high level language were both too extensive to develop algorithms for in the time allotted. This research focused on the Command Injection vulnerability in the Juliet Test Suite and on the C++ language due to the computationally expensive feature extraction time.

Although there are many different compilers that each drastically change the binary code, this research used the Microsoft C++ Compiler to standardize all the compiling and because the Juliet Test suite was specifically designed to be compiled with it.

We ultimately chose the Shuffle-LAGAN tool over other genome extracting tools due to popularity in the Bioinformatics field. Mauve was also considered but due to the cubic performance on sequences was cut out to reduce time on genome extracting.

FINDINGS:

Findings Overview:

Converting the binary to a DNA base analogue then uploading the resulting DNA sequences to the mVista online tool and finally processing these results the researchers were eventually able to

identify a set of potentially useful features. Using these features, the researchers were able to detect the levels of similarity between the features and a sample of 10 safe cases from the Juliet Test Suite and a sample of 10 unsafe cases and create a data set which could be used with classification techniques. Then through the Scikit learn voting method which included the three voting classifiers, random forest, extra random trees, and linear discriminant analysis, the researchers were able to identify and distinguish the unsafe from the safe code with 92% accuracy. With these results in hand it can be assumed that in fact, potential software vulnerabilities contained within software functions can potentially be identified by the methods described and explored through this research. More research is necessary to explore the ability of these methods in real world scenarios and a broader scope.

Detailed Findings:

Through the methods, developed, hypothesised and tested by the research team, a set of features were discovered. These features were subsequences of code identified as either malicious or benign. In total 6 features were found that indicated malicious code and 11 features were found for secure code. Float values representing the level of similarity between the created set of features and a total of 20 test cases of code (10 secure and 10 insecure) with respect to the discovered features were stored in arrays of float values. These values were then fed into a Machine Learning program developed by the researchers. This program was created in python using scikit learn and a three voter classification method. The three voters were a random forest with 20 estimators and a max depth of 500, an extra trees classifier with 100 estimators, a max depth of 40 and used entropy as a criterion, and finally a linear discriminant analysis classifier

using a least squares delimitation technique. With this program was able to achieve a maximum classification precision of 92%, recall of 90%, an f1-score of 90% and a support of 10 out of 10 .

Due to time and computational limitations, the online version of mVista was used instead of software that can be downloaded and used on local machines. Though this software could produce some results, the results produced by the online version were far more verbose and applicable. Though the online mVista program produced the most usable results this method was limited by the lack of an effective retrieval method to get the results from the mVista online servers to a format more usable by the researchers. For this reason a web-scraper had to be designed by the researchers to not only copy the output but strip all unnecessary html information from the pages where it was kept.

To use these algorithms and the mVista software the binary had to first be translated to DNA base pair representation and then this had to be formatted into what is known as FASTA format to be in proper form for submission to the mVista program. Initially the researchers attempted a direct representation where two numerals would be represented by a single letter. The Shuffle-LAGAN program was not capable of understanding long strings of consecutive characters created by the long sequences of zeros that are common in code. For this reason a more nuanced approach was developed where consecutive ones and zeros would be translated to three alternating characters instead. This was accepted by mVista and Shuffle-LAGAN and produce the data used in the final steps. Two sets of functions formatted to be uploaded to mVista where one set represented secure code and one set represented insecure code and with

each set containing one hundred functions, were submitted to mVista for comparisons. One hundred was the number uploaded as this was the maximum allowed. The researchers wanted to use as large a pool as possible to increase the possibility of finding relations with as strong a correlation as possible.

From the massive amounts of relations found amongst the submitted code bases, the researchers assumed there were a set that were of a similar level of comparability to one another and that would subsequently make for a much more useful and stronger set of indicative features to be used in the final steps. It was also assumed that a smaller set of features would be necessary to keep the entire project from becoming too unwieldy with respect to size and computing power. These features were found and the massive sets of data were reduced to extremely small (by comparison) sets of data. For the 90th percentile in similarity across code that was considered insecure, an initial set of features that were similar to one single string contained 740 individual elements. After inter-comparison to an equivalent level of confidence, this number was reduced to 6.

Due to the extremely small set of features and number of test cases, machine learning algorithms had to be developed to handle this scope. A majority voting method was developed that contained a random forest classifier with extra random trees and a linear discriminant analysis technique. Random forests were used because the final array of floats were extremely similar through all cases, i.e. the numerical values contained differences that were extremely small which caused extremely dense clusters that no kernel function was really able to differentiate

effectively. This caused support vector machines and neural networks to perform far worse than 50% in most tests. The linear discriminant analysis technique was chosen for its ability to work well in situations where the number of features outnumber the number of test cases. The results of these methods have been described previously. The high precision of these methods is assumed to be a strong indicator of an affirmation of the researchers' initial hypothesis.

PreProcessing - 95 Hours:

- Code to Binary to DNA O(n) - 1 Hour
- Shuffle-LAGAN $\Omega(gn^2)$ - 36 Hours
- Scraping Vista $O(n^2)$ - 40 Hours
- 90% Confidence Testing on Features $O(gn^2)$ - 18 Hours

Training and Testing - 6 Hours

- Random Forest O(dn * log(n))
- Extremely Randomized Trees O(dn * log(n))
- Linear Discriminant Analysis $O(nd^2)$

There are n number of sequences where g is the average length of a sequence and d is the number of features. In our case, d is 17 and n is roughly 1,000 except in Feature Selection (Shuffle-LAGAN, Vista, and Confidence Testing) where n is 200.

ISSUES:

Issues with respect to the initially proposed plans came from six areas: Finding a suitable dataset, Juliet Test Suite and LLVM, Web Scraping, Feature Extraction, Binary Tracing, and Time Constraints.

Previously the training data was acquired by scraping known exploit databases. However, the CVE data which was initially planned to be the source of vulnerabilities proved difficult to classify autonomously and provided less data points than expected for any one coding language. Instead the NIST Juliet Test Suite was better suited as test data including 64,000 classified cases. Each and every case includes both the exploit, and code to remediate the vulnerability. Such balanced cases help more effectively determine exploits such as the buffer overflow due to being able to train on the presence of a buffer. While the Juliet Test Suite solved one problem, it created another, which was that it was designed to be compiled with Microsoft Visual Studios instead of LLVM/Clang. Due to the time limitations, the decision was made to compile the code with the Microsoft C/C++ Optimizing Compiler Version 18.00.40629 for x86 from a single machine to preserve the standardization of the code that was necessary.

The research relied on the mVista genome alignment tool to provide initial features to later be processed for the machine learning algorithm. mVista came with limitations as a tool being used from the Vista Website. The mVista web tool is not particularly suited for large amounts of data, being able to compare at maximum 100 sequences together. The mVista web tool also returns the data in a tree of web pages containing many javascript elements, making web scraping the data both essential and difficult. In order to actually web scrape the web pages we had to perform link

parsing on the URL's which took a considerable amount of debugging. Lastly the mVista tool came with an unexpected problem which returned errors on long strings of zeros or ones. The researchers surmise that the tool was not made to handle very long strings of single characters and this mechanism was built to stop unforseen loops. However, the binary would quite often contain long strings of zeros. In order to work around this problem we categorized the strings 00 and 11 as being a combination of genes ATT and TTA instead of A and T respectively. Many of these problems could be resolved with the Vista code on hand, but unfortunately the researchers were unable to re-apply the program in way that was conducive to the needs of the research in a timely manner.

Tracing Binary back to C++ to find the exact location in the code that the error is located was quite difficult and inaccurate. Many compilers tend to optimize the source code of programs by moving around the code, removing chunks or completely changing the code. This results in changes to the structure of the code as compilers reorder expressions to take advantage of the processor pipeline to inlining whole functions and generating code that takes advantage of faster instructions. Another issue with decompiling is that it removes the function names from the binary. Making it difficult to recover the source code but still maintaining the same functionality as the original source code.

Previously in the Proposal it was indicated that the scope of the project may have needed to be narrowed down due to time constraints. Therefore, the framework has been limited to the C/C++ coding languages which is reflected in the Juliet Test cases. The analysis has been further

narrowed down to one vulnerability at a time. We chose to start with CWE 78, OS Command Injections.

CONCLUSIONS AND RECOMMENDATIONS:

Conclusions:

The research presented here should be considered a proof of concept for a novel and previously untested static code analysis method, as opposed to a fully fleshed out framework for static code analysis. Though the research was a success and the hypothesis that features could be found in the binary executables of programs through the use of programs designed for genome sequencing, features which could be used to indicate the possibility for security vulnerabilities being present, the limitations of the research must be acknowledged. Namely the code used for the testing, the Juliet test suite is a small subset of the code in the world and is closer to being considered a toy example of security vulnerabilities as opposed to real world cases. The methods developed here were also only tested on one kind of vulnerability as well. With the understanding of the limitations of this research and the final results in mind as well, it is of the opinion of the researchers that more research should be done in this field to understand the true viability of these methods when applied in real world cases. The researchers do believe however, that this future research would prove successful in replication of the results found here.

Future Work:

More comprehensive testing on the subject is required to obtain a holistic view that could be considered practical in real world applications. Standardized methods for reapplication of the process outlined in this paper need to be developed for this to be possible as well. For instance each compiler will have a different interpretation of how a higher level code will be translated to binary, which could open up new patterns in code that were not present in our approach. Also, another possible route for future research would include using different sets of computer code considered to be vulnerable from a security standpoint, for testing to possible further validate the hypotheses of the researchers when applied to broader applications, as well as trying multiple different vulnerabilities from the Juliet Test Suite for similar reasons. Machine Learning methods used for classifying binary data sets may be able to detect some vulnerabilities better than others which will require a case by case redevelopment of programs similar to those developed here, for the new problem sets. Newer and more applicable genome sequencing algorithms or perhaps even completely redeveloped algorithms that were initially inspired by these algorithms, could also be developed and/ or tested to better understand the relation of binary computer code to DNA base pairs and what this necessitates from a sequence comparison program. This information could be used to create methods designed solely for parsing binary data instead of repurposing other methods. In general, optimization at nearly every level of the methods used in this research could be performed to provide a much more usable framework for potential future applications of similar methods to real world scenarios or for future research as well.

REFERENCES:

[1] Edmundson A., Holtkamp B., Rivera E., Finifter M., Mettler A., Wagner D. (2013) “An Empirical Study on the Effectiveness of Security Code Review”. In: Jürjens J., Livshits B., Scandariato R. (eds) Engineering Secure Software and Systems. ESSoS 2013. Lecture Notes in Computer Science, vol 7781. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-36563-8_14.

[2] Rachel Potvin and Josh Levenberg. 2016. “Why Google Stores Billions of Lines of Code in a Single Repository”. Commun. ACM 59, 7 (June 2016), 78–87. https://doi.org/10.1145/2854146.

[3] Maynard PhD, MSc, FIA, Trevor , and George Ng. *Emerging Risks Report*. Lloyd's, 2017, *Emerging Risks Report*, www.lloyds.com/news-and-risk-insight/risk-reports/library/technology/countingthecost.

[4] Zitser, M.; Lippmann, R.; Leek, T. “Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code”. ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 2004, pp. 97-106.

[5] A. Arusaie, S Ciobaca, V. Craciun, D. Gavrilut, D. Lucanu: “A Comparison of Static Analysis Tools for Vulnerability Detection in C/C++ Code”. Alexandru Ioan Cuza University & Bitdefender 2016, 1-8.

[6] Mokhov, Serguei A., et al. “The Use of NLP Techniques in Static Code Analysis to Detect Weaknesses and Vulnerabilities.” *Advances in Artificial Intelligence Lecture Notes in Computer Science*, 2014, pp. 326–332., doi:10.1007/978-3-319-06483-3_33.

[7] Static Analysis of Binary Executables - Steve Hanov. (n.d.). Retrieved March 9, 2018, from http://www.bing.com/cr?IG=3EA4E1A9D5A741A6AE22115CF47DD6C6&CID=17352997331E665A09B6223932B16772&rd=1&h=OdMj8UVRKPggy4PCw6hwtbWoB8629UoNpOlrlxcLpJk&v=1&r=http%3a%2f%2fstevehanov.ca%2fcs842_project.pdf&p=DevEx,5064.1

[8] Learning to Detect and Classify Malicious Executables in ... (n.d.). Retrieved from http://www.bing.com/cr?IG=38FDE886FD4B4E2D94BD48195EF94BED&CID=1B18F8731D5361430FF8F39C1CFC6012&rd=1&h=l5GofMqLju5bcr1Zr9dsZQ4pvXKmbhTfiIGYp21koTk&v=1&r=http://jmlr.csail.mit.edu/papers/volume7/kolter06a/kolter06a.pdf&p=DevEx.LB.1,5069.1

[10] Michael Brudno, Sanket Malde, Alexander Poliakov, Chuong Do, Olivier Courone, Inna Dubchak, and Serafim Batzoglou, Glocal alignment: finding rearrangements during alignment. Special Issue on the Proceedings of the ISMB 2003, Bioinformatics 19: 54i-62i, 2003.

[11] Michael Brudno, Chuong Do, Gregory Cooper, Michael F. Kim, Eugene Davydov, Eric D. Green, Arend Sidow and Serafim Batzoglou, LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA, Genome Research 2003 Apr;13(4):721-31.

[12] Michael Brudno, Michael Chapman, Berthold Gottgens, Serafim Batzoglou, and Burkhard Morgenstern, Fast and sensitive multiple alignment of long genomic sequences. BMC Bioinformatics, 4:66 2003.

Tasking:

Austin Moreau - Pre and post processing data, Working with Shuffle-LAGAN, and developed machine learning program.

Pablo Salinas - Find suitable algorithm(s) for repurpose, assist with Shuffle-LAGAN, develop web scraping script to extract data from mVista.

William Bogardus - Compiling, data preprocessing, postprocessing of Shuffle-LAGAN results, and assisting with web scraping.

Duy Nguyen - Compiling weekly reports, preprocessing Shuffle-LAGAN results.

Biographies:

Austin Moreau – Currently studying Computer Science as a post baccalaureate student in The University of Houston undergraduate Computer Science program, I previously received a Bachelor's of Fine Arts, Studio Art - Painting, from the University of Houston as well as a minor in mathematics. I was previously the Chairman of the Board for a nonprofit as well as a business owner. I am currently employed in the development of software to be deployed inside of the industrial internet of things for automation of the resource extraction industry. I am very interested in developing new machine learning methods with a concentration on the field of cyber security. I am also keenly interested in reapplying the application of methods designed for bioinformatics and gene-sequencing to new fields.

William Bogardus – I am currently a post baccalaureate student at the University of Houston seeking admission to the Masters Program in Computer Science. My area of focus is in cybersecurity. I have keen interest in programming, compilers, machine learning, and cybersecurity. I am hopeful that the skills I am learning in my Security Analytics class will allow me to pursue future research project in cybersecurity.

Pablo Salinas – As a Post-Baccalaureate student in the Computer Science Program at The University of Houston pursuing admission to the Masters Program in Computer Science, one of my areas of interest is cybersecurity. I previously received a Bachelor of Science in Industrial Engineering and I currently work as a Staff Engineer at Measuresoft. I look forward to contributing my skill set and being part of the that the team that ensures application confidentiality and integrity as a crucial part of security while also moving towards a better defense against software attacks.

Duy Nguyen – I am prospective graduate student enrolled in the University of Houston Computer Science Program. I have been involved in team projects including raspberry pi smart objects and presenting prototypes to MD Anderson. I will use this opportunity to explore the field of CyberSecurity and machine learning using my knowledge from my Security Analytics Course, while coordinating with the team to deliver the project on schedule.

Github repository for resources developed and utilized:

https://github.com/ADMoreau/Software-Assurance-Defect-Localization