LLVM intermediate representation for code weakness identification

Shannon K. Gallagher
CERT, Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
skgallagher@sei.cmu.edu

William E. Klieber
CERT, Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
weklieber@sei.cmu.edu

David Svoboda
CERT, Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
svoboda@sei.cmu.edu

***Abstract*—Recent effort for code weakness identification focuses on training statistical machine learning (ML) models on source code text as the feature space in addition to more structural features like abstract syntax trees. LLVM intermediate representation (IR) can aid ML models through standardizing code, reducing vocabulary size, and removing some context sensitivity regarding syntax and memory. We investigate the benefit of LLVM IR to train statistical and machine learning models including bag-of-words models, BiLSTMs, and a few varieties of transformer models. We compare these LLVM IR-based models to models trained on source C-based models on two different sets of data: synthetic data and more natural data. We find that while using LLVM IR features does not result in more accurate models than their C-based counterparts, we are able to identify context-specific LLVM IR and C tokens that help indicate the presence of weaknesses. Additionally, for a given data set, we find that bag-of-words models can be powerful indicators whether any statistical or ML model is beneficial for code weakness identification before using more complex and time consuming models.**

***Index Terms*—intermediate representation, LLVM, vulnerability identification, neural nets, RoBERTa,**

I. Introduction

Vulnerabilities in code are defined by the National Institute of Standards and Technology (NIST) as "a weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability" [1]. If left alone, vulnerable code can lead to malicious attacks or other negative consequences. Similarly, a code weakness is defined as "a piece of code that may lead to a vulnerability" [2]. As such, weakness and vulnerability identification are important and well-studied issues in cyber security. A number of open-source static analysis tools (e.g. clang, Cppcheck, Rosecheckers [3], [4], [5]) were developed to automatically determine whether functions were vulnerable or not. Despite having some success in predicting code vulnerabilities, these tools rely on approaches to identify weaknesses and vulnerabilities that result in a number of false positives. The high proportion of findings that are reported but not actually weaknesses or vulnerabilities can consume and waste valuable developer time. The approaches also use indicators that may not be effective at identifying new types of vulnerabilities."

Recently, a number of works have studied the use of statistical and machine learning (ML) models to predict whether code is vulnerable or not. Examples of such work include Sestili et al. (2018) [6] who use deep learning to examine whether C functions are vulnerable. Zeng et al. (2020) [7] present a survey of current work in this field and identify four 'game changers' who were trendsetters in using statistical and ML models on different levels of source code. Additionally, they discuss some of the challenges involving obtaining good data for benchmarks in this field.

In addition to using source code text to train models, many researchers have tried to incorporate the implicit and explicit structure of code in the feature space. This idea is pursued in Zhou et al. (2019) [8] which uses graph neural networks to model code structure. Buratti et al. (2020) [9] instead use a transformer (C-BERT) to learn properties about the source code instead of trying to explicitly learn the graph features of the code. They show that the transformer-based classifier performs comparably to the graph neural network. Similarly, Feng et al. (2020) [10] introduce the transformer CodeBERT and release the model for public use. The authors in Coimbra et al. (2021) [11] learn path-context from abstract syntax trees for C functions.

Zhou et al. (2019) [8] identify another key issue for statistical and ML tools for vulnerability identification: test data. A number of data sets (e.g. Juliet and STONESOUP [12]) have been synthesized to test static analysis tools, but it is unclear whether statistical and ML models that perform well on these synthetic data will actually perform well on more natural, 'wild' data. To this end, Zhou et al. (2019) [8] release a new series of labeled data sets ("Devign" data) that should be more representative of more natural data. Lu et al. (2020) [13] expand upon this by consolidating a number of data sets, including the Devign data, for code-based machine learning tasks and release them publicly.

Instead of training our model on source C features, abstract syntax trees, or graph-based neural networks, we investigate the use of LLVM intermediate representation (IR) for code weakness classification. LLVM IR is an intermediate step

between source C code and machine level code [3]. LLVM IR may benefit ML model performance by standardizing code and reducing vocabulary by removing local variable names. Additionally, LLVM IR removes some context-sensitivity regarding syntax and memory, can avoid some syntactic edge cases, and introduces a number of special tokens. Moreover, LLVM IR has the potential to be used in a variety of higher level languages (e.g. C, python) where code can be first compiled to this IR.

In particular we seek to answer the following questions:

- Is LLVM IR beneficial in statistical/ML models for weakness/vulnerability identification?
- How do LLVM IR-based models compare to source C-based models?
- Can we learn how LLVM IR features are related to the weakness/vulnerability identification?
- How do statistical and ML models perform on synthetic data compared to more natural data?

The rest of this paper proceeds as follows. In Section II we describe the data pre-processing pipeline from source C code to LLVM IR functions along with our data sets: a synthetic and two more natural ones. In Section III we describe the different models we use for weakness/vulnerability identification on LLVM-based features, and in Section V we summarize our results and comment on future work.

II. Data

In this work, we explore the use of LLVM IR in two types of data sets: a synthetic one where weaknesses have been intentionally injected into the code and two more natural data sets that were manually labeled as vulnerable or not by security researchers.

A. Data processing pipeline

Before any modeling can be done, our data undergo a number of pre-processing steps, which are summarized by the flowchart in Figure 1. For both the synthetic and more natural data sets we use, we first compile source C files with clang (see [3]) into LLVM IR. Following that, we extract each function from the resulting LLVM IR files and keep track of its label (vulnerable or not). During this extraction process, we also pre-tokenize the function text, a process which preserves a number of standard library commands. Once the function text is extracted, we prepare our feature matrix (or tensor) $\mathbf{X}$. We tokenize the function text by splitting the text into base-level chunks (i.e. tokens), the set of which form our vocabulary. For the non-sequence dependent models of logistic regression, support vector machine, and random forest, we form weighted document term matrices from the tokens where the weights are the term-frequency inverse-document frequency (TF-IDF). In addition, we include the log number of tokens as a feature for each function. The sequence dependent models of the bidirectional long short-term memory (BiLSTM) model and the transformers are trained on features where the text sequences are preserved.

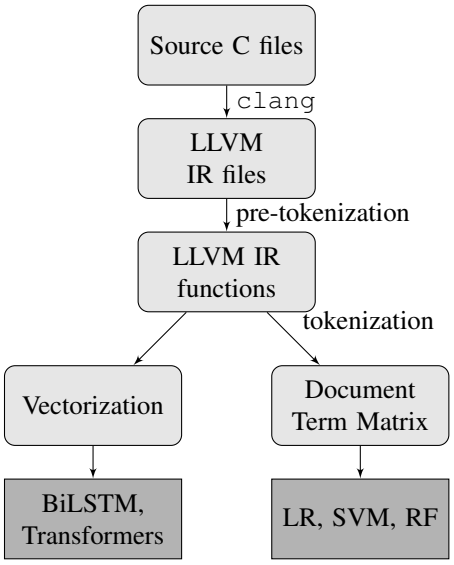

Fig. 1. Flowchart for data processing from source C files for use in both sequence-dependent (left) and sequence-independent (right) models. Here, BiLSTM refers to the bidirectional long short-term memory model, LR to logistic regression, SVM to support vector machine, and RF to Random Forest.

B. Synthetic Data

The first data set we use to test the usefulness of LLVM IR in weakness/vulnerability identification models is IARPA STONESOUP Phase 3 VM [12]. These data contain over 4000 source C files, which were originally created to test static code analysis tools. As mentioned above, we use clang to compile source C to LLVM IR and then extract each function ID, label (weak/not weak), and function text. The result of this process is over 5GB consisting of over 266,000 functions where 40% are labeled as not weak and the remaining as weak. The 40/40 split makes this a fairly well-balanced data set. In these data, the median (25th, 75th quantile) for the number of tokens per function overall are 661 (364, 1779) but are 528 (445, 1672) for the weak group and 753 (282, 1869) for the not-weak group (see Fig. 2).

The STONESOUP data have some attributes that make them difficult for use in statistical and machine learning models. For example, a large proportion of functions labeled as weak include the token "stonesoup". These attributes are obvious indications that the functions are weak, and while the token's appearance should not influence static analysis tools, this token will very likely be used by statistical and machine learning models to predict whether a function is weak or not. We attempt to mitigate this issue by removing all occurrences of the string "`stonesoup`" in the LLVM IR. However, we acknowledge this mitigation is *ad hoc* and does not alleviate the entire problem. Moreover, we are reluctant to further modify the text as changes could move the code further away from its original intent. As such, we expect that the results of any statistical and ML models may not be representative of code weaknesses for other repositories. Another attribute of STONESOUP is that only some of the functions labeled

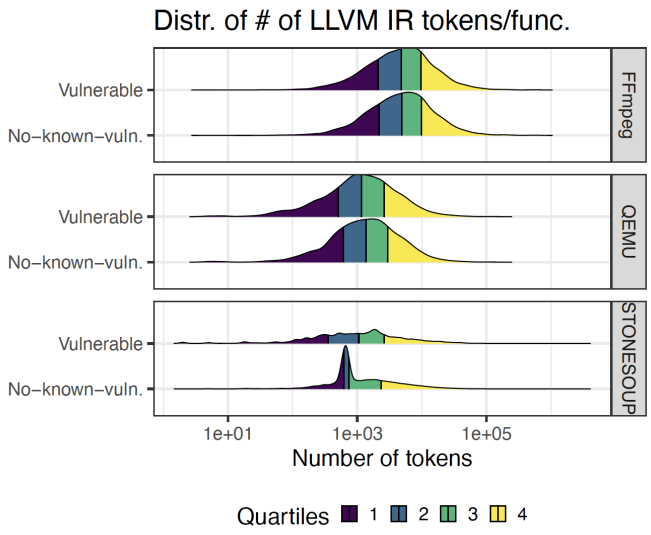

Fig. 2. Distribution of number of tokens for the three different data sets. STONESOUP is a synthetic data set whereas FFmpeg and QEMU are more natural.

as 'weak' are weak in the traditional sense. The labels in STONESOUP are assigned at the *source file* level, and we extrapolate them to the function level. While this extrapolation is useful for our purposes, again it is not necessarily indicative of more natural weak functions.

C. More natural data

The second sets of data ("Devign data') we analyze were originally released by Zhou et al. (2019) [8], were further analyzed by [9] Buratti et al. (2020), and consolidated by Lu et al. (2020) [13]. We have access to git commits in two open-source repositories: FFmpeg and QEMU. The former repository is a "cross-platform solution to record, convert and stream audio and video," and the latter is "a generic and open-source machine emulator and virtualizer" [14, 15]. Security researchers first identified relevant commits and then determined if these commits dealt with a security vulnerability or some other vulnerability or weakness. They further assume that one commit typically corresponds to one function. If the latest commit fixed a security-related vulnerability in a function, the same function from the *previous* commit was labeled as vulnerable. We label the same function in the fix-commit as non-vulnerable. If the same function appeared in several commits, we only recorded the last commit as non-vulnerable. We are directly comparing the same functions to one another: pre- and post-fix. Note that our labeling is a different scheme than what was done in Zhou et al. (2019) [8] which instead separates function fixes into two categories: those which correspond to *security* vulnerabilities and those which correspond to other vulnerabilities or weaknesses. We believe our task to be a more difficult problem as both the vulnerable and non-vulnerable examples contain the same set of functions, albeit with differences in the source code.

There are approximately 2×12460 total C functions (and source files) that are labeled as vulnerable in the previous commit and not-vulnerable in the fix-commit. We compile the C functions with clang into LLVM IR, a number of functions could not be compiled due to a variety of issues such as . These issues were similar to compilation issues faced by Zhou et al. (2019) and Buratti et al (2020) [8, 9]. For FFmpeg, the result is 0.5GB consisting of 6611 functions with a 3072/3539 (53.5/46.5)% vulnerable/not-vulnerable split. In these data, the median (25th, 75th quantile) for the number of tokens per function is 4808 (2153, 9791) but is 4775 (2096, 9691) for the vulnerable group and is 4860 (2187, 9860) for the non-vulnerable group. For QEMU, The result is 0.5GB consisting of 5127 functions with a 2798/2329 (54.6/45.4)% vulnerable/not-vulnerable split. In these data, the median (25th, 75th quantile) for the number of tokens per function are 1259 (554, 2763) but are 1162 (512, 2598) for the vulnerable group and 1373 (611, 2950) for the non-vulnerable group. The distribution of the number of tokens per function are shown in Fig. 2. In comparison to the STONESOUP data, the differences in empirical distribution of number of tokens are much less apparent between the vulnerable and not vulnerable group for both QEMU and FFmpeg. We also notice that FFmpeg functions generally seem to have more tokens than QEMU functions.

III. Methods

Once the data are prepared, we fit a number of models to determine if using LLVM IR-based features is useful for predicting code weaknesses or vulnerabilities. These models can broadly be split into two classes: those where token sequence order does not matter and those where token sequence order matters.

We train/implement a variety of models on the LLVM IR-based feature space and compare those to models that are trained/implemented on the source C-based feature space. Models where sequence order does not matter are commonly referred to as bag-of-words models and include logistic regression (LR); random forests (RF); support vector machines (SVM)). The second set of models were sequence-dependent and include recurrent neural nets (BiLSTM; LLVM IR-trained transformer (TF); LLVM IR-trained Longformer (LF)). The source C-based feature models are LR, RF, SVM and additionally, BiLSTM, and CodeBERT. What follows is a high level overview of the different methods. Please see Appendix A for details on how to reproduce our results.

For the models trained on synthetic data, we first split the data into train (60%), test (10%), and hold-out sets (30%). We build the models on the training set and evaluate them on the test set. The hold-out set is currently being reserved for future analyses. Since there are about an order of magnitude fewer functions in the more natural data sets, we use 80% of the data for training and the remainder for testing.

For the sequence-independent data we choose logistic regression, random forests, and support vector machines due to their practical success in binary classification problems.

Additionally, we chose logistic regression and random forests because some of their features have practical interpretations. In these methods, we model the probability of a function being vulnerable as

$$P(Y_i = 1|\mathbf{X}_i) = f(\mathbf{X}_i),$$

where $Y_i = 1$ is the event of function i being vulnerable, $\mathbf{X}_i$ is a *vector* of sequence-independent features for function i, and f is a map from the feature-space to [0,1]. Functions with estimated probabilities > 0.5 are given a vulnerable label and the rest given a non-vulnerable label. For these models, $\mathbf{X}_i$ is derived from the document-term matrix (DTM) corresponding to the function text. For a vocabulary consisting of N_V tokens, the DTM is a $n \times N_V$ matrix $\mathbf{D}$ where entry $\mathbf{D}_{ij}$ corresponds to the number of times token j occurs in function i. In the synthetic data our vocabulary size is from the pre-tokenization step is about 72000 tokens. We reduce the dimensionality of the feature space by examining tokens occurring in at least 5% (an arbitrary choice) of all functions, which reduces the vocabulary size to 284. Additionally, for any pair of perfectly correlated features (e.g. in a function, for every '[' that occurs, there exists a corresponding ']'), we remove one of the features. We perform a similar transformation for the more natural data. The raw document term matrix is then weighted by the term-frequency inverse-document-frequency (TF-IDF) which increases the weight of frequent tokens within a function and simultaneously decreases the weight of tokens common to all functions. This transformation places emphasis on more common tokens, but only if the tokens occur in a small number of documents. This weighting, for example, does not place much weight on the token 'ret' which occurs in nearly every function and so should be very useful in the classification process despite its high occurrence. In addition to the weighted DTM, we also include the log number of tokens as a feature in $\mathbf{X}_i$.

Of these bag-of-words models, LR and RF allow for (limited) interpretation of features. In logistic regression, we obtain the odds ratio (OR) for each feature. This value is the odds of the function being predicted as vulnerable compared to not, for a one unit increase in that feature and adjusting for the other features. Values >1 are associated with being more likely to be predicted as vulnerable and values <1 are associated with being predicted not-vulnerable. For RF, we compute the permutation importance index for each feature. For a given feature, this index is computed by randomly permuting the given feature values of the functions, passing them through the fit RF, and determining the mean difference in prediction accuracy (see Breiman (2001) [16]). The result of this process is a ranking of features with regards to their importance in prediction, although the index does not tell us *how* the feature is associated with vulnerable or non-vulnerable predictions.

Besides sequence-independent models, we also examine a few sequence-dependent models, namely the BiLSTM recurrent neural network and a few varieties of transformers. In these models, the feature space $\mathbf{X}_i$ maintains the order of the text. The LSTM was initially introduced by Hochreiter and Schmidhuber (1997) [17] and learns token embeddings sequentially (in one direction) which can then be used for vulnerability identification. The BiLSTM uses both directions to learn embeddings and is specifically designed to carry information about past tokens over long distances.

Transformers are neural networks closely associated with encoder-decoder architecture where both the inputs and the outputs are used to estimate token embeddings (see Zhang et al. (2021) [18]). Moreover, they are based on the idea of self-attention layers to learn token embeddings in a pre-training step. Attention layers pool together features using a summary of the queries (Q), keys (K), and values (V). In self-attention, the queries, keys, and values are the same ($Q = K = V$). Once the embeddings are learned, transformers are fine-tuned, i.e. the embeddings are used as features in another neural net trained to the research problem. We use the transformer RoBERTa (see Liu et al. (2019) [19]), which is implemented by the Hugging Face API (Wolf et al. (2020) [20]). We first pre-train RoBERTa from scratch on LLVM IR functions where the training task is masked language modeling (MLM). Specifically, the training data are LLVM IR functions from 13 open source repositories including doxygen, FFmpeg, git, hexchat, lmms, obs-studio, openh264, openvpn, qemu, redis, systemd, wireshark, and xen. We then fine-tune the model for vulnerability identification, a binary classification problem. To compare the results from the LLVM IR-based feature space to the source C-based feature space, we use the pre-trained CodeBERT (Feng et al. (2020)[10]).

Transformers like BERT (Devlin et al. 2018) [21] were initially motivated by language translation (e.g. English to French) and were designed to work on relatively short token sequences (<128-512 tokens). This creates an issue for us because for LLVM IR function text sequences, at least half the functions exceed 500 tokens and sometimes even exceed 100,000 tokens. A number of methods were proposed to alleviate this problem of working with longer sequences of text ranging from an optimized model to deal with longer token sequences up to length 4096 (the longformer (Beltagy, Peters, and Cohan 2020) [22]), using a BiLSTM model on top of N sequence chunks, or guidance on how to choose the tokens to be included in the embeddings. Here, we explore the first approach, the longformer in addition to a base model using only the first 256 tokens for prediction.

For comparison, we use the Rosecheckers static analysis tool to find specific problems with the code on the source line level [5]. The Rosecheckers tool was developed specifically to enforce the guidelines in the SEI CERT C Coding Standard (available at https://wiki.sei.cmu.edu/confluence/display/c).

IV. Results

We show the results of our methods which use a LLVM IR-based feature space and compare those to each other along with previous results on a source C-based feature space. We show the estimated accuracy (%), F_1 score, precision, and recall on the test set. We also report LLVM IR and C tokens

that were considered important in weakness/vulnerability identification.

A. Synthetic data

In Table I we show the accuracy on both the train and test sets of the LLVM-based methods for the STONESOUP data. All methods perform better than the naive guess of assigning all functions as vulnerable (60% accuracy). The methods of RF, BiLSTM, and the two transformers have over 90% accuracy on both the train and test set. From this table we also see that none of the models are overfitting to the data as shown by the small drop in accuracy between the train and test sets. The base transformer using only the first 256 tokens outperforms the longformer using the first 4096 tokens, which may seem surprising. However, given enough training epochs, we would expect the longformer's results would improve.

In Table II, we show the precision, recall, and F_1 score of the methods on the test set. The base transformer is the most precise method with 99% of functions being predicted as vulnerable actually being vulnerable. Both RF and the longformer have recall values of 1.00 meaning there were no false negatives. For F_1, RF has the highest score of 0.99, and BiLSTM, the base transformer, and longformer all have F_1 values > 0.95. Overall, we can conclude that the best methods using this LLVM IR- based feature space on the STONESOUP data are very useful for vulnerability identification. In fact, the best performer in terms of accuracy, recall, and F_1, RF, is a bag-of-words model which is much simpler to implement and run than the sequence-dependent neural nets. The RF also allows for limited interpretability of the feature space by examining the permutation importance (see Section III). We plot this graph in Figure 3, which shows the top 20 features in prediction accuracy. For example, the token 'i64' is the most important feature in the RF with regards to prediction accuracy. This token is only found in LLVM IR and not in the original source C code. In fact, a number of these top tokens are LLVM IR special tokens, and more information about them can be found at https://llvm.org/docs/LangRef.html. Notably, the token '<ret>' is not a LLVM-special token and was inserted to account for instruction separation in LLVM IR.

Given that the 25th and 75th quantiles for number of tokens in LLVM function code are 364 and 1779 and the base transformer only uses the first 256 tokens in its classification, it is very concerning that this method does so well and may indicate there is something more structural about how the STONESOUP vulnerabilities are injected into the code as opposed to our models learning generally applicable features. Moreover, some of the most important tokens from RF are numbers like "3", "4", and "5" which are references to other objects in LLVM IR. However, there is no reason to believe that reference indices should be associated with the vulnerability identification outcome, and so we have evidence that our methods are fitting to spurious features.

Conclusions drawn from this STONESOUP data must be viewed cautiously due to the synthetic nature of the data set. STONESOUP data were designed for testing static analysis tools and as a result may not be representative of data found in the 'wild.' That said, we find it useful to examine this data set for 1) a sanity check for our statistical and ML models, 2) an indicator of whether LLVM IR can be useful for weakness identification, and 3) a way to test whether training models on the readily available and labeled STONESOUP data is useful for weakness/vulnerability identification on more natural data.

TABLE I
ACCURACY (%) OF VULNERABILLITY IDENTIFICATION ON THE STONESOUP TRAIN AND TEST DATA. THE METHODS ARE LOGISTIC REGRESSION (LR), RANDOM FOREST (RF), SUPPORT VECTOR MACHINE (SVM), BIDIRECTIONAL LONG SHORT-TERM MEMORY NEURAL NET, LLVM IR-BASED TRANSFORMER USING THE FIRST 256 TOKENS (LLVM-TF), AND LLVM IR-BASED LONGFORMER USING THE FIRST 4096 TOKENS (LLVM-LF).

Set	LR	RF	SVM	BiLSTM	LLVM-TF	LLVM-LF
Train	83.8	**98.9**	87.2	97.5	97.4	94.8
Test	83.6	**98.8**	87.0	97.3	96.9	94.7

TABLE II
RESULTS OF MODELS ON THE STONESOUP TEST DATA. THE METHODS ARE LOGISTIC REGRESSION (LR), RANDOM FOREST (RF), SUPPORT VECTOR MACHINE (SVM), BIDIRECTIONAL LONG SHORT-TERM MEMORY NEURAL NET, LLVM IR-BASED TRANSFORMER USING THE FIRST 256 TOKENS (LLVM-TF), AND LLVM IR-BASED LONGFORMER USING THE FIRST 4096 TOKENS (LLVM-LF).

Model	Precision	Recall	F1
LR	0.81	0.95	0.87
RF	0.98	**1.00**	**0.99**
SVM	0.85	0.95	0.90
BiLSTM	0.93	0.98	0.96
TF	**0.99**	0.94	0.96
LF	0.95	**1.00**	0.97

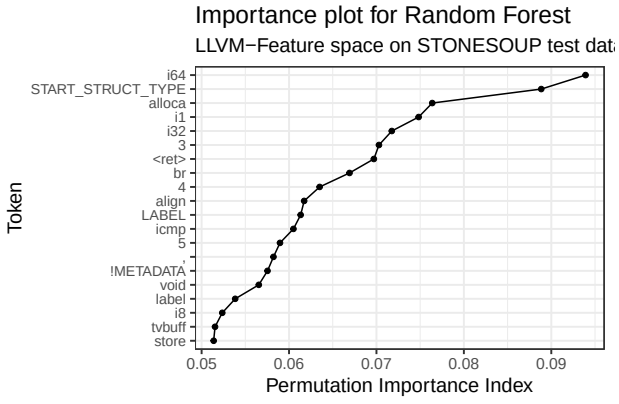

Fig. 3. Permutation importance index for the top 20 features from RF in the STONESOUP test data. The features are LLVM IR tokens and each entry is the weighted token count in each function. The token '<ret>' is not a LLVM token and was inserted to account for instruction separation in the LLVM IR. Information about LLVM special-tokens may be found at https://llvm.org/docs/LangRef.html.

TABLE III
RESULTS OF METHODS FOR VULNERABILITY IDENTIFICATION ON THE FFMPEG AND QEMU TEST DATA SETS. THE METHODS ARE RUN ON BOTH SOURCE C CODE AND LLVM IR. FOR THE LLVM IR-BASED METHODS, BOTH THE BASE TRANSFORMER AND THE LONGFORMER ARE TRAINED-FROM-SCRATCH USING LLVM IR FROM 13 OPEN-SOURCE REPOSITORIES. THE BILSTMS USE EMBEDDINGS FROM GLOVE, AND THE BASE TRANSFORMER USED ON THE C SOURCE CODE IS CODEBERT. THE BASE TRANSFORMER (BASE TF) IS THE TRAINED-FROM-SCRATCH LLVM-BASED TRANSFORMER USING THE FIRST 256 TOKENS, THE LF IS THE LLVM-BASED LONGFORMER USING THE FIRST 4096 TOKENS, LR IS LOGISTIC REGRESSION, RF IS RANDOM FOREST, SA STANDS FOR STATIC ANALYSIS TOOL, WHICH HERE IS ROSECHECKERS, AND SVM IS SUPPORT VECTOR MACHINE.

Data	Source	Method	Accuracy	Precision	Recall	F_1
FFmpeg	C	CodeBERT	**0.546**	**0.560**	0.693	0.620
FFmpeg	LLVM	Base TF	0.510	0.513	0.433	0.470
FFmpeg	C	BiLSTM	0.537	0.543	0.034	0.063
FFmpeg	LLVM	BiLSTM	0.502	0.501	0.438	0.467
FFmpeg	LLVM	LF	0.512	0.513	0.505	0.509
FFmpeg	C	LR	0.537	0.540	0.914	0.678
FFmpeg	LLVM	LR	0.534	0.535	**0.973**	**0.690**
FFmpeg	C	RF	0.545	0.556	0.738	0.634
FFmpeg	LLVM	RF	0.537	0.543	0.840	0.660
FFmpeg	C	SA	0.531	0.538	0.873	0.666
FFmpeg	C	SVM	0.542	0.553	0.740	0.633
FFmpeg	LLVM	SVM	0.542	0.553	0.729	0.629
Data	Source	Method	Accuracy	Precision	Recall	F_1
QEMU	C	CodeBERT	**0.640**	**0.638**	0.796	**0.708**
QEMU	LLVM	Base TF	0.548	0.589	0.462	0.518
QEMU	C	BiLSTM	0.635	0.601	0.569	0.584
QEMU	LLVM	BiLSTM	0.537	0.508	0.764	0.611
QEMU	LLVM	LF	0.525	0.525	**0.998**	0.688
QEMU	C	LR	0.567	0.575	0.809	0.672
QEMU	LLVM	LR	0.545	0.564	0.769	0.651
QEMU	C	RF	0.576	0.602	0.674	0.636
QEMU	LLVM	RF	0.606	0.623	0.724	0.670
QEMU	C	SA	0.482	0.500	0.412	0.452
QEMU	C	SVM	0.517	0.535	0.913	0.675
QEMU	LLVM	SVM	0.512	0.650	0.251	0.363

B. More Natural Data

In addition to showing the result of classification success on the synthetic STONESOUP data, we show the results from two more natural data sets: FFmpeg and QEMU (see Table III). For these results, we ran our methods on both LLVM IR and additionally on the source C code.

For FFmpeg, CodeBERT (run on the first 256 source C tokens) was the most accurate and precise model and logistic regression had the best recall and F_1 score. Notably, *none* of the methods resulted in an accuracy much more than the naive guess of 'vulnerable' (53.4% accuracy), and even the most accurate method is not better than chance (p-val. =0.102) for any reasonable α-level. Overall, all the methods we tried for vulnerability classification in FFmpeg perform poorly with respect to accuracy.

We had more success for the QEMU repository where three methods had over 60% accuracy on the test data (the naive guess would be 55.2% accurate). We can be 95% confident that these methods are doing better than chance (p-val = 0.0006). Again CodeBERT had the largest accuracy and precision and additionally the largest F_1 score. The drop in accuracy for CodeBERT between the train and test sets was 19%, indicating that the model is overfitting to the training data. The longformer run on LLVM IR (4096 tokens) had the largest recall.

For both applications, we find that the stochastic methods we use do no worse than the static analysis tool Rosecheckers, and in QEMU, all stochastic methods perform better in terms of accuracy, precision, recall, and F_1 score.

We see that the methods trained on the source C code are generally more effective in all metrics compared to methods trained on LLVM IR. In fact, the sequence-dependent models never perform better on LLVM IR than source C with respect to accuracy or precision. One explanation for this is that the these methods tend to perform better on shorter text. Only random forest has better accuracy on LLVM IR than source C in the QEMU application.

Finally, we notice that the sequence-dependent models tend to do better in any of the metrics than the bag-of-words models. However, if there is a statistically significant signal from the code for weakness/vulnerability classification, then random forest and often the other bag-of-words models indicate the presence of this signal. For example, the top three C tokens according to the permutation importance were 'true', 'DeviceClass', and '*dc' for QEMU. Additionally, in the best logistic regression trained on QEMU C code, after adjusting for six other tokens ('buffer', '*err', 'size', 'NULL', '1', '+'), for a unit increase in the TF-IDF weighted token 'true;' we expect the odds of a function being classified as vulnerable to be 0.86 (95% CI: 0.80, 0.93) times less.

V. CONCLUSION

Overall, we find that while there are a few benefits of using LLVM IR for code weakness/vulnerability identification, using source C code generally results in better models with respect to accuracy, precision, recall, and F_1 score. Moreover, there are currently several limitations to LLVM IR-based methods. Many of these limitations stem from the fact that the number of tokens in LLVM IR functions is about an order of magnitude more than in source C code and so models cannot be easily computed on small GPUs. Other practical concerns include the hundreds of computer hours it took to first pre-process the data into LLVM IR, the fact that not all source C could be successfully compiled in the first place, and the fact that LLVM IR specific line references have no reason *a priori* to be correlated with function weakness classification.

Despite not finding LLVM IR particularly useful for code weakness and vulnerability identification, we were able to explore other issues including: synthetic vs. more natural data, directly identifying pre- and post- function fixes, identifying interesting tokens, and the overall usefulness of bag-of-words models. We immediately found that any of the stochastic methods we used could successfully identify weak functions from the synthetic STONESOUP data with high accuracy. This may be an indication that statistical and ML tools may be more helpful in identifying intentionally malicious vulnerabilities as opposed to unintentional vulnerabilities but needs to be further validated. To test more natural data, we use Devign data like in

several previous approaches, but unlike these approaches we directly take on the task of identifying whether a vulnerable function has been fixed. Perhaps surprisingly, the results of our task are largely comparable to the other approaches even with the differing classification task. In fact, differences in accuracy seem more dependent upon the specific code base as opposed to the specific classification task. For example, our results show that our best models do about 10% better than chance at prediction for QEMU functions, a result consistent with [9] and [8]. Conversely, in FFmpeg our models are unable to estimate vulnerabilities better than the naive guess of vulnerable. This result differs from [9] and [8] who are able to perform better than chance. Notably FFmpeg functions tend to have more tokens than QEMU functions, which may have effected this result. Finally, we recommend using bag-of-words models as a first step in any weakness classification task. These methods can be very simply implemented in common tools such as python and R on one's own laptop, which was not the case with the neural nets. Additionally, since logistic regression and RF have interpretable features, they can be used to guide further models. We found that the bag-of-words methods could indicate the presence of some statistical signal by producing a classifier with more accuracy than random chance or by identifying statistically significant tokens.

Finally, like many previous approaches, we find that code vulnerability classification is a difficult problem that, currently, is only somewhat improved upon by using statistical and ML models trained on expensive labeled data. As such, in future work we would like to study the differences between human-based and machine-based vulnerabilities and how statistical and ML models trained on code (or code IR) may be affected by their presence.

REFERENCES

[1] "Vulnerabilities," https://nvd.nist.gov/vuln, accessed: 2021-09-08.

[2] "Definitions," https://samate.nist.gov/BF/Enlightenment/Definitions.html, accessed: 2021-12-09.

[3] "Clang: a C language family frontend for LLVM," https://clang.llvm.org/, accessed: 2021-09-08.

[4] Marjamaki, "Cppcheck," https://github.com/danmar/cppcheck, 2021.

[5] CERT, "CERT Rosecheckers," https://github.com/cmu-sei/cert-rosecheckers, 2021.

[6] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with ai," 8 2018. [Online]. Available: http://arxiv.org/abs/1808.09897

[7] P. Zeng, G. Lin, L. Pan, Y. Tai, and J. Zhang, "Software vulnerability analysis and discovery using deep learning techniques: A survey," *IEEE Access*, vol. 8, pp. 197 158–197 172, 2020.

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

[9] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rossiello, A. Morari, J. Laredo, V. Thost, Y. Zhuang *et al.*, "Exploring software naturalness through neural language models," *arXiv preprint arXiv:2006.12641*, 2020.

[10] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," 2020.

[11] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu, and H. Erdogmus, "On using distributed representations of source code for the detection of c security vulnerabilities," 6 2021. [Online]. Available: http://arxiv.org/abs/2106.01367

[12] "Resources from the Software Assurance Reference Dataset," https://samate.nist.gov/SRD/around.php, accessed: 2021-09-08.

[13] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021.

[14] FFmpeg, "FFmpeg," https://www.ffmpeg.org/, 2021.

[15] QEMU, "Qemu," https://www.qemu.org/, 2021.

[16] L. Breiman, "Random forests," pp. 5–32, 2001.

[17] S. Hochreiter and J. . U. Schmidhuber, "Long short-term memory." [Online]. Available: http://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf

[18] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into deep learning," *arXiv preprint arXiv:2106.11342*, 2021.

[19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[20] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[21] J. Devlin, M.-W. Chang, K. Lee, K. T. Google, and A. I. Language, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018. [Online]. Available: https://github.com/tensorflow/tensor2tensor

[22] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 4 2020. [Online]. Available: http://arxiv.org/abs/2004.05150

[23] R Core Team, *R: A Language and Environment for*

Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2021. [Online]. Available: https://www.R-project.org/

APPENDIX

Below we provide the details in our models to reproduce our results. The data were first randomly split into train, test, and hold-out sets. All models are built on the train set and then evaluated on the test set unless otherwise specified.

Sequence Independent Models. These models were fit in `R` version 4.1.0 [23].

Tokenization. Function code sequences were first pre-tokenized as described in the main text to maintain LLVM special tokens and global variables. We used the `R` package `tm` to convert the code sequences into a document term matrix (DTM) and weight them by the TF-IDF. Sparse terms occurring in less than 10% of the functions were removed to reduce the vocabulary size (and hence feature space).

Perfectly correlated features were removed from the DTM using the function `caret::findCorrelation`.

Logistic regression. We used the package `glmnet` to select features by minimizing the the mean square error of a binomial logistic regression model with a L_1 penalty. Practically this was done using the function `cv.glmnet`. We used 10-fold cross-validation and selected the best model as the one standard error minimum lambda (`lambda.1se`). The model matrix used was the weighted DTM described above in addition to the log number of tokens and the label. Due to model complexity, features were first selected on the *test* data, and the final model with those features were fit on the *train* data.

Random Forest. RandomForest models were fit in `R` using the `ranger` package. We used the `ranger` function on the aforementioned modeling matrix of the weighted DTM in addition to log number of tokens. We use the arguments of `min.node.size` = 5, `importance` = `"permutation"`, and `probability` = `TRUE`.

Support Vector Machine. We use the `ksvm` function from the `kernlab` package in `R`. Due to the computational complexity of SVM, we use a random subset of 10,000 functions in the train set to build the model. The model is evaluated on the full test-set. This time the weighted DTM was subset to the features with the top 15% of weight according to the column sum of each feature.

Sequence Dependent Models.

These models were run on NVIDIA PYTORCH GPU with CUDA Version 11.3.

BiLSTM. To implement the BiLSTM, we used pytorch. The pre-tokenized function code sequences were inputted into a three-layer recurrent neural network consisting of a two-layer LSTM (input vocabulary size = 14,4777, embedding size = 100, dropout = 0.2, bidirectional = True) each with output size 32, a fully connected layer with input features dimension 64 output features dimension 1, and a sigmoid activation layer. The model is summarized in code below.

```
classifier(
  (embedding): Embedding(14477, 100)
  (lstm): LSTM(100, 32, num_layers=2,
  batch_first=True,
  dropout=0.2, bidirectional=True)
  (fc): Linear(in_features=64, out_features=1,
   bias=True)
  (act): Sigmoid()
)
```

The model has 1,507,157 trainable parameters.

We use the Adam optimizer with default parameters along with binary cross-entropy loss. We use four training epochs. We limit all code sequences to the first 10,000 words.

Base Transformer. We further tokenize the pre-tokenized sequences using the BpeTrainer with a vocabulary size of 1000. We use size 256-blocks of tokens. We then pre-train the transformer from scratch on the train set using the model `RobertaForMaskedLM`. We have the following non-defaults in the config file:

```
config = RobertaConfig(
    vocab_size=1000,
    max_position_embeddings=512,
    num_attention_heads=12,
    num_hidden_layers=6,
    type_vocab_size=1,
)
```

We pre-train this model with four epochs. We then initialize the fine-tuned model, `AutoModelForSequenceClassification` using our pre-trained model with two labels. We again train the model for four epochs. *Longformer*.

The longformer is implemented very similarly to the Base Transformer. We use the model `LongformerForSequenceClassification` from the Hugging Face API. Due to its computational complexity, we only train the model for two epochs. For this model, the first 4096 tokens are used.

CodeBERT. We use the pre-trained model and further train it on source C code for the classification task.

ACKNOWLEDGMENT

We would like to thank Bob Schiela for his fantastic guidance and input. Additionally, we would like to thank Tom Scanlon and Mark Sherman for their suggestions.