June 2020

SECURITY AND OTHER VULNERABILITY PREDICTION USING NOVEL DEEP REPRESENTATION OF SOURCE CODE WITH ACTIVE FEEDBACK LOOP

Krishna Sundaresan

Anshul Tanwar

Prasanna Ganesan

Sriram Ravi

Sathish Kumar Chandrasekaran

See next page for additional authors

Inventor(s)
Krishna Sundaresan, Anshul Tanwar, Prasanna Ganesan, Sriram Ravi, Sathish Kumar Chandrasekaran, and Parmesh A.

SECURITY AND OTHER VULNERABILITY PREDICTION USING NOVEL DEEP REPRESENTATION OF SOURCE CODE WITH ACTIVE FEEDBACK LOOP

AUTHORS:

Krishna Sundaresan
Anshul Tanwar
Prasanna Ganesan
Sriram Ravi
Sathish Kumar Chandrasekaran
Parmesh A.

ABSTRACT

Since the cost of fixing vulnerabilities can be thirty times greater after an application has been deployed, it is recognized that properly-written code can yield potentially large savings. Accordingly, approaches presented herein apply machine learning and Artificial Intelligence (AI) techniques to improve developer experience by enabling developers to avoid introducing potential bugs and/or vulnerabilities while coding. Billions of lines of source code, which have already been written, are utilized as examples of how to write functional and secure code that is easy to read and to debug. By leveraging this wealth of available data, which is complemented with state-of-art machine learning models, enterprise-level software solutions can be developed that have a high standard of coding and are potentially bug-free.

DETAILED DESCRIPTION

Traditionally, the discovery of security and other vulnerabilities is performed at the end of a development cycle. Thus, vulnerabilities are typically found during the testing phase, the deployment phase, or even after an application is deployed. However, finding a security vulnerability in these stages incurs more cost and can be damaging to the credibility for a product. The techniques discussed herein employ shift-left error and vulnerability detection strategies to identify potential vulnerabilities as early as possible in the software development life cycle (i.e., during code development).

When software engineers write code for various features, various types of errors (e.g., coding, logic, semantic, etc.) can be introduced that may not be caught by compilation tools. Companies may thus be required to allocate a lot of resources in terms of both money and time in finding and fixing bugs that could have been avoided if coding was properly performed. In some cases, same or similar bugs that were fixed in past (although in different modules) can nevertheless be introduced in production code.

Undetected flaws in software can lead to security vulnerabilities that potentially allow attackers to compromise systems and applications. Many traditional approaches include static and dynamic analyzers that utilize rule-based approaches to error detection; thus, these approaches are limited to the employed rules. These tools can further introduce false positives, causing critical findings to be buried in a sea of warnings. Moreover, conventional approaches fail to take application-specific field errors into account during future analyses. For example, when a possible issue is identified, no further insights on the issue's impact are provided to the developer, nor are potential fixes.

There is no AI-based system to that uses a deep representation of code to aid developers in identifying potential bugs and other issues that might exist in newly-developed code based on the history of issues seen. Thus, information such as a list of similar functions and any references to bugs found on those functions could provide helpful insights to developers.

Presented herein is a novel AI-based system that identifies potential bugs that are introduced at the time of development itself. For example, as a developer writes new code, the tool can integrate with the integrated development environment (IDE) as a plugin that works in the background, listing already-available similar functions or code-segments and any associated bugs in those functions. This feedback enables developers to incorporate suggestions immediately at the time of development, rather than waiting for a quality assurance technician or a customer to identify a defect. Also included herein is a novel approach of data cleaning, data preparation, and a feedback loop that utilizes as input two discrete data sources.

To prepare the data, source code from a codebase is converted to an Abstract Syntax Tree (AST) representation. Using the AST, each method is represented with the set of encoded path context. The number of path context representations for a given function are

then computed. Initially, all path context representation from the AST are considered, and a ranking model is applied to eliminate some of the representations that occur commonly across the functions, as well as path contexts that are found in very few functions. Thus, overfitting of the model is avoided by increasing the nodes in the first layer of the model to account for the input dimensionality later. The minimum number of the occurrences of a path context is one of the hyper-parameters of this model. This method is found to work very effectively compared to considering all of the path context representations.

Let **P** represents the set of path contexts in an AST. Each path context **(p)** will be of format $\{n_i - p_{ij} - n_j\}$, where n_i and n_j represent the encoded node values and **p** represents the encoded path values. Node encodings are numerical representations of each node in an AST. Similarly, path encodings are the numerical representation of each path in an AST. These numerical encodings enable conversion of the text to a number which can be then be used as input for the model being trained.

Each path context that is obtained will then be filtered to remove the path contexts that occur very frequently (as those contexts do not aid in uniquely distinguishing the AST or the function of interest), along with the path contexts that occur very rarely (to avoid higher-dimensionality encodings). Equation 1 indicates how path contexts are filtered:

$$\text{Filtered path contexts } (P_F) = \forall\, p \in P\,, if\ \min \leq count(p) \geq max$$

Equation 1

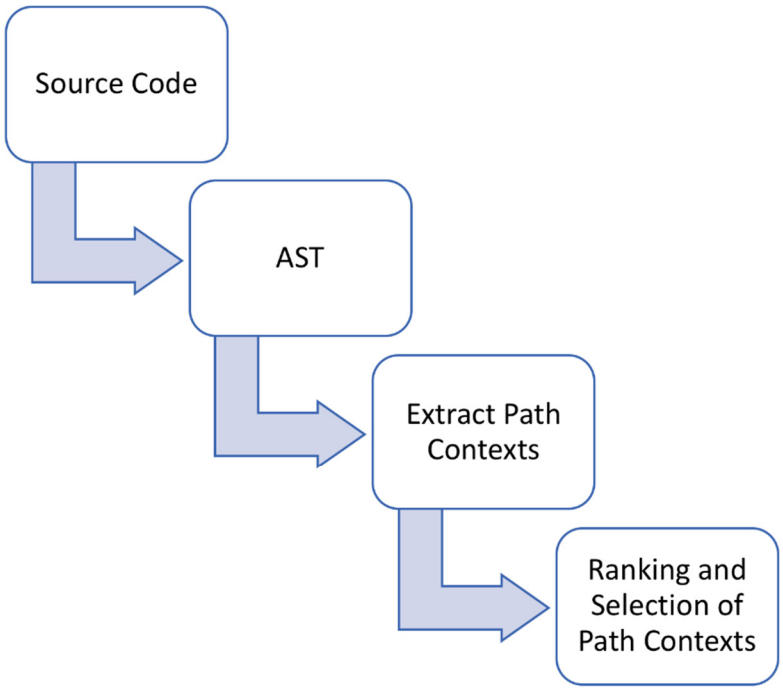

Figure 1

Figure 1 depicts a process for processing source code to select path contexts. A single vector representation (also referred to as code embedding or a code vector for a function) is obtained from the set of path contexts. A path context- and attention-based model is trained to learn the code vectors, which are the weighted average of the path embedding concatenated with the weighted average of the node embedding. Node embeddings and path embeddings are learned from the encodings during model training. The attention weights are learned from training the model itself. Thus, unlike conventional techniques, this approach uses both path embeddings and node embeddings to obtain path contexts, and selects them using an attention model.

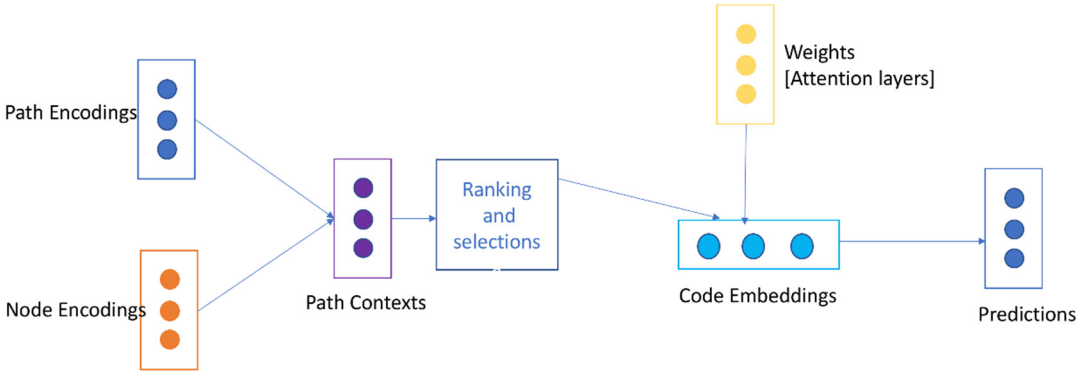

Figure 2

Figure 2 depicts a method of obtaining code embeddings, applying weights, and applying a trained model. For each **p** in **P**, the path context embedding, c, is learned during model training for each path according to the relation:

$$c_i = embeddings(n_i, p_{ij}, n_j)$$
$$ci = [node_{embeddings(n_i)} \ \ path_{embeddings(p_{ij})} \ \ node_{embeddings(n_j)}]$$

The weighted average of all embeddings **c** is computed to obtain a single embedding representation for a given function. The weights are learned from the attention layers according to Equation 2.

$$code\ embedding = \sum_{i=1}^{n} \alpha_i * c_i$$
$$where\ \alpha_i = \frac{\exp(c_i^T . a)}{\sum_{j=1}^{n} c_j^T . a}$$

Equation 2

As depicted in Equation 2, ***a*** denotes the global attention vector which is initialized randomly and learned simultaneously with the network.

Thus, a vectorized representation for the code embeddings of each function present in the codebase is obtained. This representation is further enhanced by the addition of function names followed by their corresponding code vectors. As mentioned previously, both node and path embeddings are considered in order to arrive at the final vector representation. In one example, the node and path embeddings have been configured to each be 128-dimension vectors, and the final code vector is a 384-dimension vector.

The code embeddings obtained above are at the function level of granularity and do not include the information about any enclosed function calls within them. To incorporate the complete functionality of a function, the embeddings of the functions that are being invoked from the main function of interest must also be considered. To achieve this, a combination property of code embeddings is used, and a call graph is formed to add the embeddings of the invoked functions as well. The function call stack depth is limited to three. The value of this limit can act as a hyper-parameter; expanding the value to three levels in the stack gives good results while not becoming too complex.

```
void int sample_composite_function(struct sample_instance *sinstance)
{
     <Standard initialization code>
     ...
     app_logic_function_helper1(...)
     ...
     app_logic_function_helper2(...)
      }
}
```

Figure 3

In the example code depicted in Figure 3, the code embeddings for each individual functions are initially obtained:

sample_composite_function = E1,

app_logic_function_helper1 = E2

app_logic_function_helper2 = E3

Assuming that there are no other functions being invoked from *app_logic_function_helper1* and *app_logic_function_helper3*, the new embedding for "sample_composite_function" can be defined as:

$$sample\_composite\_function\ r = E1 + E2 + E3$$

with the addition operator indicating vector addition.

Next, the dataset is extended to include historic bugs associated with each function. For this, the Cisco Defect Tracking System (CDETS) data source is utilized to fetch the fixes associated with each bug, which are parsed to find all the functions that were modified (i.e. fixed) in the dataset. Open-source datasets may also complement the dataset that is used to represent the historic bugs. For example, the publicly available Draper Vulnerability Detection in Source Code (VDISC) Dataset can be used, which consists of the source code of 1.27 million functions mined from open source software, and labelled by static analysis for vulnerabilities. The dataset is analyzed and cleaned to remove any incorrect markings of common weakness enumeration (CWE) labels. Thus, the code embeddings are obtained for each function, along with and the number of bugs associated with them.

Function Name	Embeddings	CDETS
function1	300D Embeddings	Bug ID's
function2	300D Embeddings	Bug ID's

Figure 4

Figure 4 depicts an example table of functions, embeddings, and associated bugs.

Using the above trained "code2vec" model, the code vectors for every function present inside a dataset, such as the Draper VDISC dataset, are extracted. This acts as the initial labelled dataset, and consists of code vectors as independent variables and the marked CWE labels as dependent variables.

Two sets of three-layer neural networks can be used to train the vectors: one set is trained with "vanilla" code embeddings and another is trained with composite code embeddings obtained by adding the embeddings of the functions invoked within them. The

composite embeddings can include the complete functionality of each module, which helps to identify errors that are spread across multiple functions. The other neural network helps to identify semantic and other errors within the block of code of a function. Next, a simple logistic regression model is used to combine the results of both models. The models are trained for a classification task using their input data.

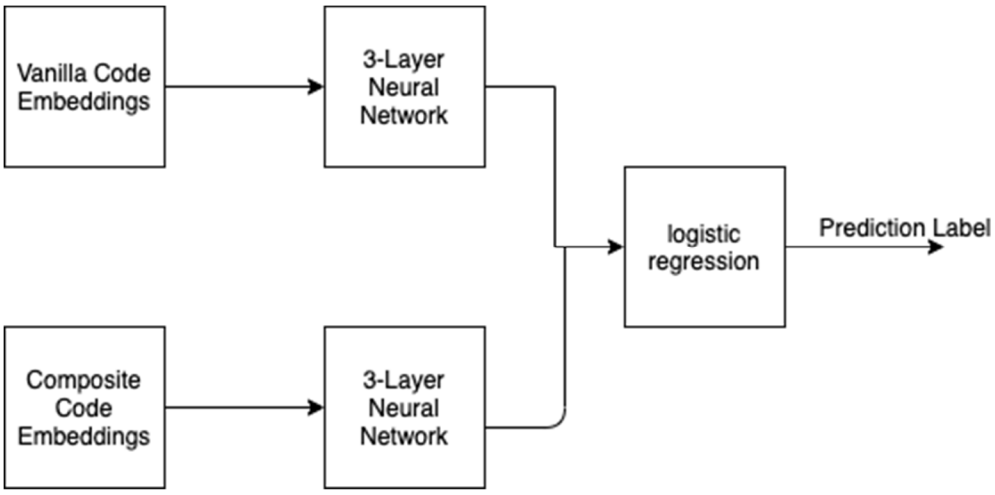

Figure 5

Figure 5 depicts an example of how two neural network models are trained for a classification task. Historical vulnerability data is considered to evaluate the performance of the trained model and to further fine-tune the model. For each function obtained, the model is applied to predict the probability that the function has any CWEs or bugs in general.

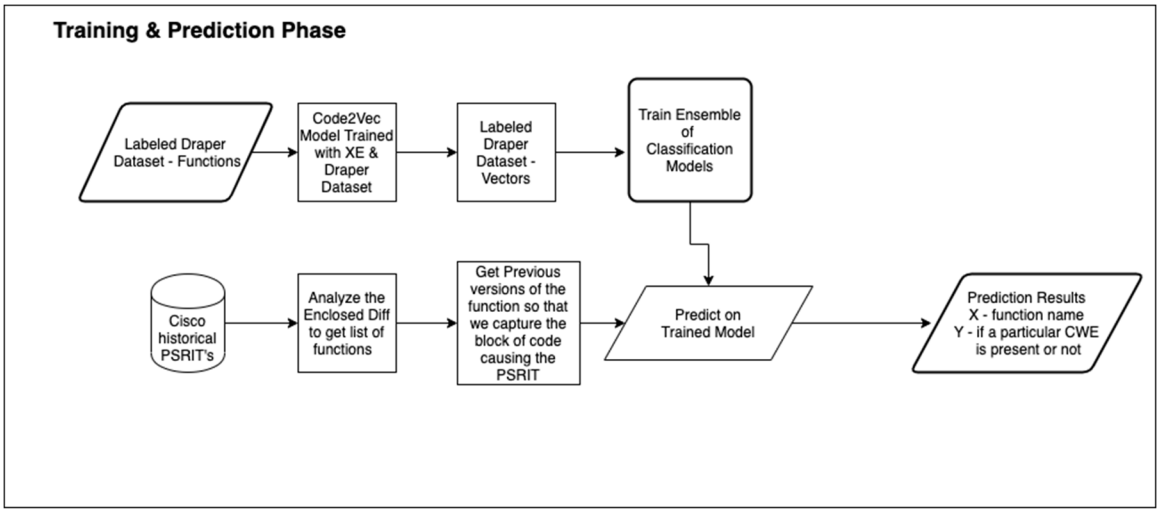

Figure 6

Figure 6 is a flow diagram depicting a training and prediction phase.

The results obtained by the models can be validated using two approaches: manual evaluation, and evaluation using an automated tool. When manual validation is performed on each predicted result, developers can also provide an additional label indicating whether the code contains issues such as application logic error other than CWEs. Additionally or alternatively, a set of security tools, such as Flawfinder, CPPCheck, Coverity, Clang, and the like, can be applied to determine whether the tools can identify the predicted CWEs. Using the feedback from either of these approaches, tune the model's parameters are fine-tuned to result in a finalized model that will be used for CWE prediction.

Finally, the model is retrained using a dataset containing historical vulnerability data. During retraining, application logic errors are included as another dependent variable.

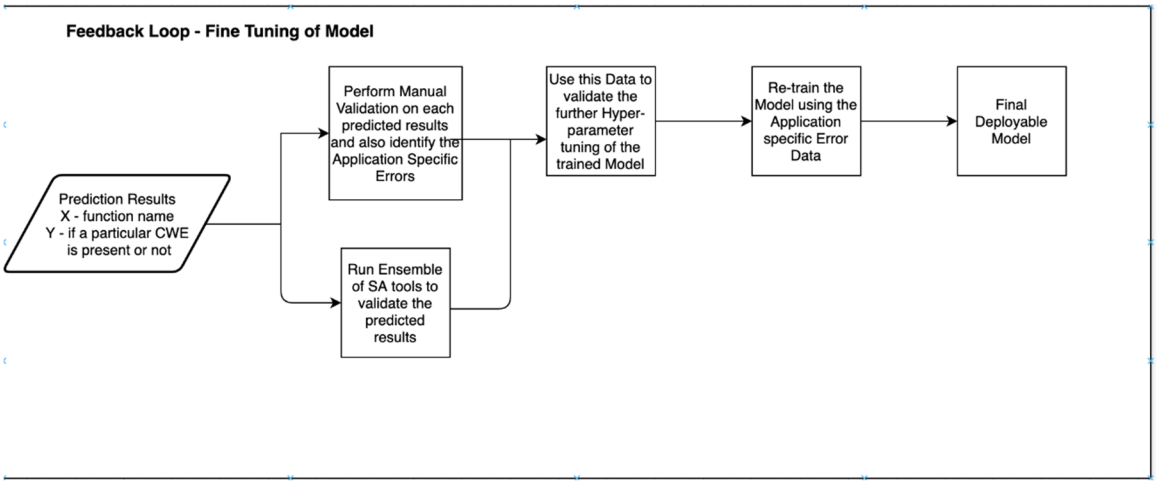

Figure 7

Figure 7 is a flow diagram depicting a fine-tuning of a trained model. This fine-tuning eliminates many false positives and also enables application logic errors to be included in the findings of the model.

After fine-tuning, the model can be deployed. The model can be deployed into production using a batch mode approach or a real-time approach. In the batch mode approach, the predictive model is run for every function in a codebase, and the results and insights are shared with a developer. In a real-time approach, the model is integrated as a plugin in an IDE so that whenever a developer writes a new block of code, details can be provided regarding the probability of the code being buggy, and probable fixes can be suggested when possible. Suggested corrective actions can be selected by looking for the similarity of the given function to a function in the historical vulnerability database and then applying the fix that was done to close the bug. The similarity check can be performed in the n-dimensional space model using the code2vec representation.

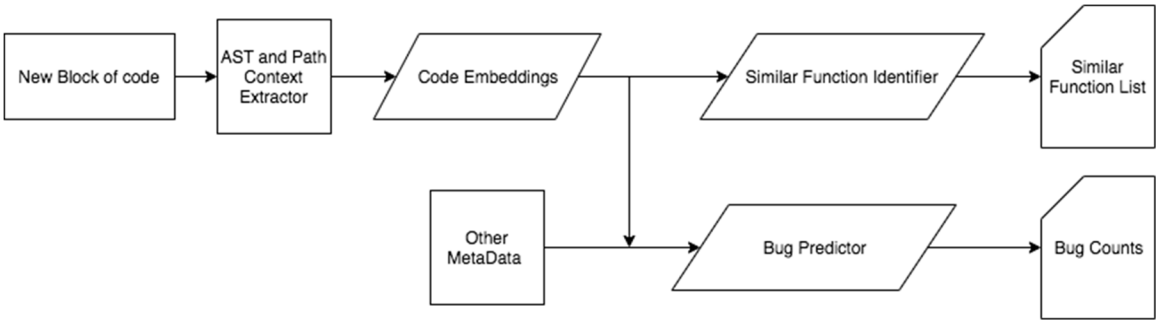

Figure 8

Figure 8 is a flow diagram depicting the processing of a block of code to identify similar functions and corresponding bugs and fixes.

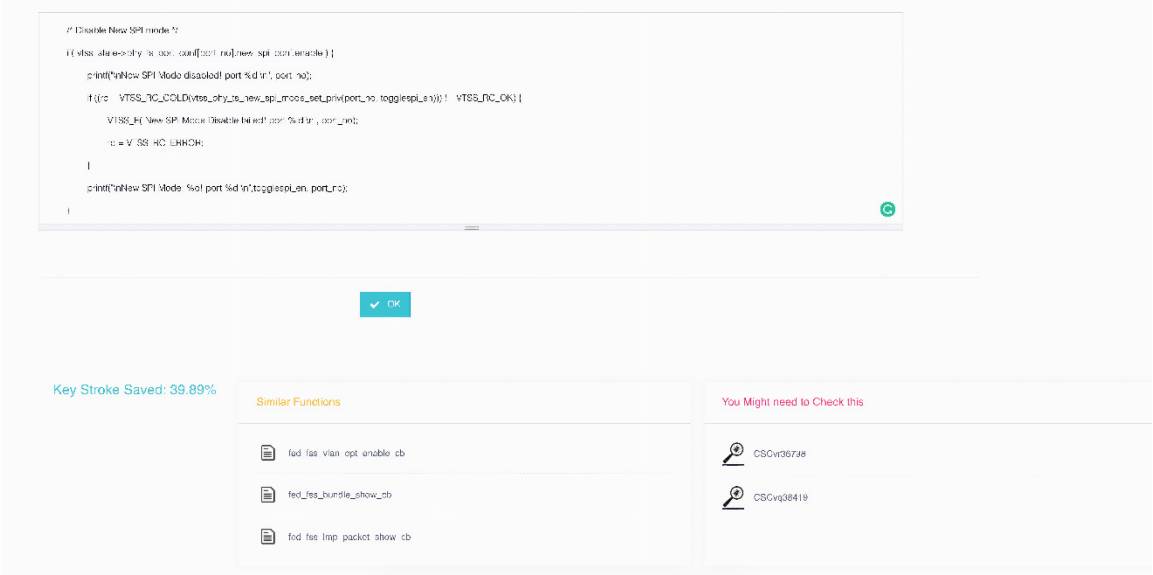

Figure 9

Figure 9 depicts a user interface of a model being applied to identify vulnerabilities and other errors in code. In an example use case, a developer can use an editor to type in the code as usual. While the developer is coding and/or once a function block is completed, details like potential bug count and similar functions and its associated bugs in the past.

The model can also include a feedback loop for active learning. The developer may be presented with an option to provide direct feedback to the model about potential bugs that are being discovered by the tool to ensure that the system's results do not stagnate over time. Advantageously, this data can be used to train new versions of the model.

If F represents a new function block that a developer is coding and L is defined as its label or the function name, code embeddings (E) can be extracted for the function of interest from the trained model according to the relation:

$$E_L = model(F)$$
$$L \equiv E_L$$

To extract the embeddings, an AST representation is obtained, and each path context is encoded using the vocabulary set of training data. Based on the embedding E, functions are identified in a database that include embeddings (e_D) closer to E_L. The distance between the two embeddings should be less than the predefined threshold (t) for selection, as represented by the relation:

$$Result = \forall\ e_D\ in\ Database, for\ which\ e_D \sim E_L \leq t$$

To incorporate the feedback, the code embedding that the model estimated for a given function is modified based on user feedback. For positive feedback from a user, the code embedding of the new function being developed is moved closer to the function tagged with the predicted vulnerabilities by a certain distance in n-dimensional space. The distance moved can be proportional to the logarithm of the number of positive votes of users. For negative feedback, the code embedding of the new function is moved farther away from the embedding associated with the function tagged with the predicted vulnerability. Again, the distance moved may be proportional to the logarithm of the number of negative user votes.

If the model predicts that a function with label L and embedding E_L is similar to one of the functions in the database having embedding as e_D, and p is defined as the number of positive votes and n is defined as the number of negative votes received for this

prediction, then the new embedding for the function with label L is defined according to equation 3:

$$E_L = e_D - \Delta$$
$$Here\ \Delta = \log(K * (p - n)) * unit\ vector\ , if\ (p - n) \neq 0$$
$$where\ K\ is\ a\ scaling\ constant\ and\ for\ now\ its\ value\ is\ kept\ as\ 1$$
$$Also - indicates\ the\ vector\ subtraction$$

Equation 3

In addition, the model can be retrained from scratch by using the current embeddings as the initial weights for every function in the database that includes the embedding for the recently added functions to maintain the model using the up-to-date codebase. The frequency of training can be decided by the domain experts and may be, for example, retrained on a monthly basis.

One unique aspect of this model is the taking the composite code embeddings of each method for computation and prediction in addition to just the vanilla code embeddings of the function. To proceed using the composite vectors, it must first be demonstrated that the vectors are additive in nature, a hypothesis represented by Equation 4:

If vector(funcA) + vector(funcB) = vector(funcC), this implies that C is doing the functionality of both A and B.

Equation 4

Approximately 250 subject functions were selected to serve as positive and negative use cases for this hypothesis. As a result, the cosine similarities were found to be greater than 0.9 for positive cases, thus confirming the hypothesis.

For the task of identifying similarity between two functions, a threshold distance of less than 0.4 was selected for two functions to be similar; this threshold provided an accuracy of 95%.

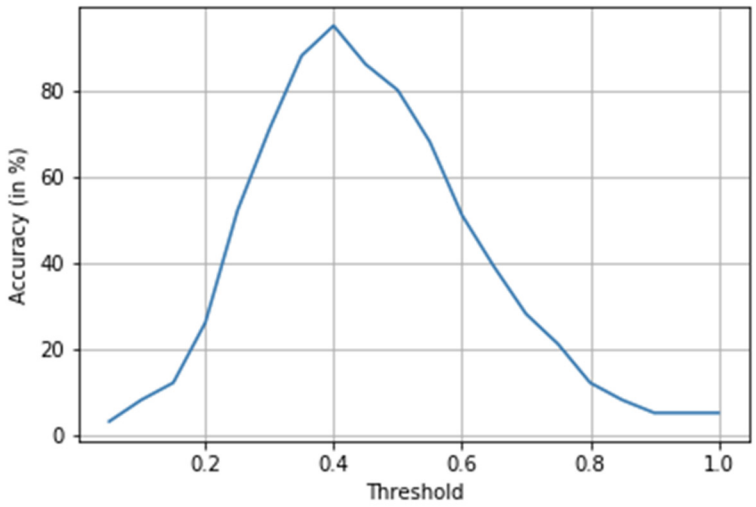

Figure 10

Figure 10 depicts the trending of observed accuracy values at various threshold values. Values less than 0.4 resulted in higher false negatives, and values higher than 0.4 produced more false positives. Thus, a threshold value of 0.4 was selected as an optimum threshold to achieve the desired results.

The model's ability to identify vulnerabilities was tested in two iterations. In the first iteration, composite code embeddings were not considered, and the additional logistic regression was not performed. The results for the Bug Prediction task were obtained using historical bugs to see if the model predicts them beforehand on the functions associated with them, resulting in an accuracy of around 70%, a precision value of 0.74 and a recall of 0.77.

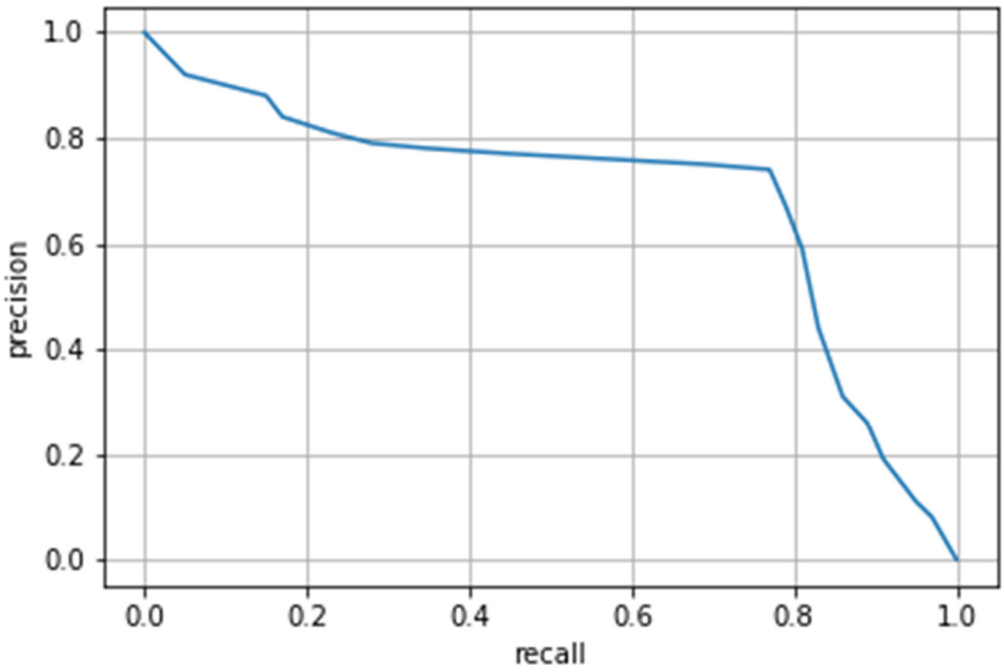

Figure 11

Figure 11 depicts example results of the second iteration, which included composite code embeddings. This iteration resulted in a significant boost in the performance of the same bug prediction task described above, with an accuracy of around 78%. The new precision value was 0.81 with a recall of 0.82.

In summary, techniques are presented herein that apply machine learning and AI techniques to improve developer experience by enabling developers to avoid introducing potential bugs and/or vulnerabilities while coding. Billions of lines of source code, which have already been written, are utilized as examples of how to write functional and secure code that is functional, secure, and contains fewer vulnerabilities. By leveraging this wealth of available data, which is complemented with state-of-art machine learning models, enterprise-level software solutions can be developed that have a high standard of coding that contains fewer bugs or is potentially bug-free.