Dependency-Based Neural Representations for Classifying Lines of Programs

Shashank Srikant
shash@mit.edu
CSAIL, MIT

Nicolas Lesimple
nicolas.lesimple@alumni.epfl.ch
EPFL

Una-May O'Reilly
unamay@csail.mit.edu
CSAIL, MIT

Abstract

We investigate the problem of classifying a line of program as containing a vulnerability or not using machine learning. Such a line-level classification task calls for a program representation which goes beyond reasoning from the tokens present in the line. We seek a distributed representation in a latent feature space which can capture the control and data dependencies of tokens appearing on a line of program, while also ensuring lines of similar meaning have similar features. We present a neural architecture, Vulcan, that successfully demonstrates both these requirements. It extracts contextual information about tokens in a line and inputs them as Abstract Syntax Tree (AST) paths to a bi-directional LSTM with an attention mechanism. It concurrently represents the meanings of tokens in a line by recursively embedding the lines where they are most recently defined. In our experiments, Vulcan compares favorably with a state-of-the-art classifier, which requires significant preprocessing of programs, suggesting the utility of using deep learning to model program dependence information.

1 Introduction

A recent direction of program analysis research infers properties of programs by learning statistical models of them with "Big Code" architectures [20]. In one example, DeepBugs develops an architecture to detect whether an entire function is buggy [19]. Typically, a "BigCode" architecture relies upon a corpus of programs and has two (or more) neural networks in sequence. The code is preprocessed and input to the first neural network where the network transforms it, non-linearly, into a *distributed representation* using weights that are trained using statistical machine learning. A distributed representation refers to a latent space, in which the features of programs have comparative value i.e. similar program components have similar meaning and are close in the space. Next, the distributed representation is passed to a predictive network (model) that is trained with labels in a supervised manner. A variety of applications are well served by this approach including renaming poorly named variables to meaningful ones [5], detecting clones [26], and more. See Allamanis et al. [2] for a review on relevant literature.

The choice of an appropriate distributed representation of the corpus of programs is crucial to application success. A popular choice preprocesses the code by tokenizing it and presenting each line as a sequence of tokens as input to the architecture's first network which uses a recurrent learning architecture. The survey of [2], Table 1 presents multiple such examples. Tokens are convenient to use, however whether they are ideal is open to question. Programs have rich structural and contextual information which tokenization ignores. For example, a line of program `x = foo(a)+b` has different meaning depending on where it appears in a program, i.e. its context, and, the meaning of the statement depends on the most recent definitions of its right hand variables, and (recursively) on the most recent definitions of these recent definitions. These properties are not explicitly captured by token sequences. It could be argued that tokenization has traction largely because what it lacks in program expressiveness is, to some extent, compensated for by the power of the learning algorithm and non-linear capacity of the LSTM or graph neural network. See [11, 27] for a discussion on LSTMs capturing such dependencies in natural language processing (NLP) tasks. Regardless, tokenization also remains dependent on the application seeking a predictive property of the entire unit of code, e.g. a function in the case of DeepBugs, and not a single line.

In this contribution we seek a representation that serves single line classification. We ask whether a representation based on structural and contextual information is better than tokenization and up to the task of accurate line-level classification.

Our contribution is a novel neural architecture - Vulnerability Classification Network (Vulcan), that we demonstrate on the problem of vulnerability classification at the line level. Vulcan takes a much more nuanced approach to forming a distributed representation than tokenization. It extracts contextual information about tokens in a line and inputs them, as Abstract Syntax Tree (AST) paths, to a bi-directional LSTM with an attention mechanism. It concurrently represents the meanings of tokens in a line by recursively embedding the lines where they are most recently defined. It has multiple "helper" networks that transform variable length inputs to fixed lengths and sub-assembly steps performing concatenation. We experimentally evaluate Vulcan's performance and whether its distributed representation defines a latent

ArXiv Preprint, March, 2020
2020.

feature space where lines of similar meaning have similar features.

2 Representation Design

Our goal is a representation that will help us infer what a line means so that it is possible to classify it containing a vulnerability or not. The representation must capture the definitions of the different tokens that appear in a line and the context in which the line is executed. If a right hand side token is a variable, the representation will have to chain backward to retroactively include the meaning of the line where that variable is defined, and the context of that definition,. e.g. whether it is within a loop or if statement. Here, an update to a variable is also treated as a (re-)definition. For machine learning purposes, we need a network architecture that transforms the input representation of each line into a continuous valued vector v of some fixed dimension, t. The vectors of lines that are similar in meaning to each other should be close to each other in the vector space. This allows supervised machine learning models to pinpoint an accurate discriminatory boundary between label (presence, absence of vulnerability) classes during training.

Walking through a simple program snippet illustrates how a line can be represented. The program snippet in Figure 1

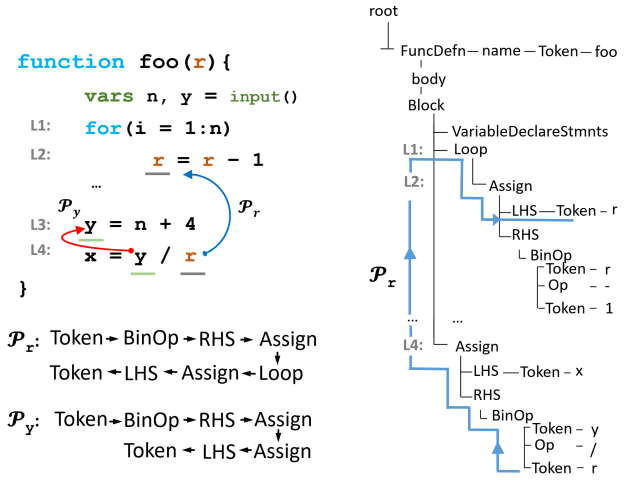


Figure 1. An example code snippet. We show how different tokens and their dependencies can be represented. Paths comprising the nodes of a program AST represent both control and data information. Here, paths $\mathcal{P}_r, \mathcal{P}_y$ connect the usages of variables `r` and `y` on line `L4` to their respective updates on lines `L2` and `L3`.

describes function `foo`. Variable `r` is updated in a loop, while variable `y` is updated on line `L3`. Both are used in line `L4`. How should we represent the meaning of `L4`: $x = y/r$? We know that `L4` updates variable `x` and the new value of `x` depends on variables `y`, `r` which are operands of the division operator. We need to represent this division expression and to do so we need the values of `y` and `r`. What, at `L4`, are the values of these variables? While these cannot be fully determined through static analysis, it is possible to go back to the line where each variable is most recently defined or updated, as well as to identify its control context. We refer to this process as retrieving the *define* and *context* information, respectively. The simple example is backtracking to `y`, and finding its most recent definition/update on `L3`. The assignment statement is not surrounded by control context that would influence the update of `y`, but we have one more detail to consider: the right hand side of expressions which assign values to `y` and `r` themselves have prior definitions and context. Thus, we have to recursively represent these until we finally recurse to the base case of their first definition, when we can express that directly.

The more involved example is backtracking `r`. It is updated on `L2` where `r`'s assignment is in the control context of a loop. In Section 4 we will use the AST of the program to extract this *context* by capturing the path between the two lines, and use a recursive algorithm to obtain a representation for the entire line `L2`.

Because operators are predefined we simply directly encode them with an arbitrary fixed representation that differentiates each from all others (a one-hot encoding).

Beyond this simple example, we need a way to encode function calls. They are effectively operators. If `L4` was instead `x = y/bar(r)`, for some function `bar`, we consider two cases: *(a)* `bar` is an in-built library, or *(b)* `bar` is a user-defined function. We treat calls to in-built libraries the way we treat operators - directly encoding them with a representation. We treat user-defined functions as a variable whose previous definition was the return statement in the function call. Hence, for a line `x = y/bar(r)`, we would, in all, encode the values and contexts of four tokens: `y, /, bar,` and `r`.

Algorithm 1 sketches this recursive enumeration routine to gather the (prior) definition and context of each token in a line. It starts with an empty line representation and iterates over the list of tokens to the right of the assignment operator. At each step of the recursion, it first locates the most recent definition. It then concatenates the context between the token and that location with a representation of that location. In the Methods section (section 4), we follow up by describing our network architecture and show, in three steps using a staging of neural networks, how a line is transformed starting from source code into v.

3 Related Work

We focus on works which use AST-based representations for program reasoning within "Big Code" approaches. Bielik et al. [5] correct improper variables names using probabilistic

graphical models (PGM) of features that capture AST edge information. This contrasts with how Vulcan employs a neural architecture to represent the AST edge information.

Both Hsiao et al. [7] and Srikant et al. [24] use program dependence graphs to reason about code-clone detection and bug finding, or automated assessments, respectively. However, they represent their entire programs as counts of edge information in the dependence graphs. They then build n-gram models based on these counts. Our work instead builds a distributed representation for such edge information in dependency graphs. Given its simplicity and effectiveness, however, we employ their approach as a baseline model in our work.

Alon et al [4] introduce the notion of paths - a data structure to capture the dependencies between different occurrences of a variable appearing in a program. They show this to be a generic representation suitable to model a variety of downstream tasks. We use this notion of paths as a building block in a larger representation scheme.

Allamanis et al [3] suggest using AST edge information in the graph networks they use to model programs. We capture the same inductive bias as that of a graph network, although we use a bi-LSTM over edges in program dependence graphs we extract. Moreover, we provide a hierarchical means of producing token-level and line-level representations, each building on the previous.

Some recent works have focused on detecting and classifying vulnerabilities through traditional program analysis techniques [17, 23, 28]. They use static analysis and fuzzing to detect vulnerabilities. In works employing machine learning, VulDeePecker [10], DeepBugs [19], and Russell et al. [21] are closest to the design we propose. We discuss them in detail.

VulDeePecker. VulDeePecker employs a bi-directional LSTM to model what they refer to as *code gadgets*. Each gadget starts with a line containing manually-identified constructs (like function and API calls) and lines containing variables which depend on these constructs, resulting in a set of lines of code governing the construct. Each code gadget has an associated label which the LSTM learns. A vector representation of a gadget is obtained by considering lexicalized tokens present in them, thus treating it as a paragraph containing strings of tokens. The main advantage of Vulcan over VulDeePecker is that it does not require elaborate gadgets to be designed. Vulcan extracts simple AST paths without any pre-processing that requires extracting slices over program dependence graphs. In follow-up work recently published on arXiv [9], they address two key limitations in VulDeePecker, namely, preparing gadgets for manually-identified constructs and not accounting for control dependencies. Their revised approach however again relies on an elaborate pre-processing step to identify *gadget* like code-blocks of interest, something which our approach does not need.

Russell et al. This work deals with C and C++ programs. They too use static analyzers to obtain their ground truth labels. However, they train a CNN on a bag of lexicalized tokens and then use a Random Forest classifier to predict whether an entire function contains a vulnerability or not. Our work instead focuses on line meaning. The features which our model learns contain control and data flow information between variables, a much richer set of features as compared to lexicalized tokens. We present models learned on a bag of tokens as a baseline to compare our model's performance against.

DeepBugs. The representation used in this work to detect bugs is token-level embeddings. These embeddings push tokens within a similar context close to each other in the chosen vector space. The work does not capture any dependency based information in an overt way through its underlying program graphs in any systematic way. Further, we were motivated to develop a method for a relatively low-resource setting, and hence chose to work with `Solidity`, a fairly recent programming language, where the number of usable scripts was in the order of 500K. DeepBugs trains on an order of a million samples. The architecture we propose does not require the magnitude of training data needed to learn unsupervised token embeddings.

4 Method

We describe our neural network architecture in this section. It consists of three stages. Its post-training input is the line being assigned a value, and its output is a distributed representation for the line, which is used to predict a label for the line. When in training mode, this line is accompanied by a label. We provide dimensions for intermediate and final outputs of the architecture in Figure 2. This architecture is sketched in Algorithm 1.

Stage 1. The input to Stage 1 is a tokenized line of code in a program and the corresponding abstract syntax tree (AST) [1] of the entire program. This stage retrieves tokens from the input line and prepares a representation for each one. Tokens here are variable names, function names, and operators.

Any operators or calls to library functions are represented with one-hot encoding over the space of such tokens seen in the training set. An `UNK` is used to handle out of sample tokens. User-defined functions are treated as variables, and are dealt with as described below.

A variable requires a pair of representations - *define* and *context*. For the first, we backtrack to identify the line of its most recent definition. We refer to this line as the variable's *end-point*. We retrieve the *end-point*'s recursively computed *define* representation. This is added to a list of *define* representations which is saved for later use in Stage 3. Hence, for each variable on the line of interest, we obtain a corresponding *define* representation. For the *context* representation, our

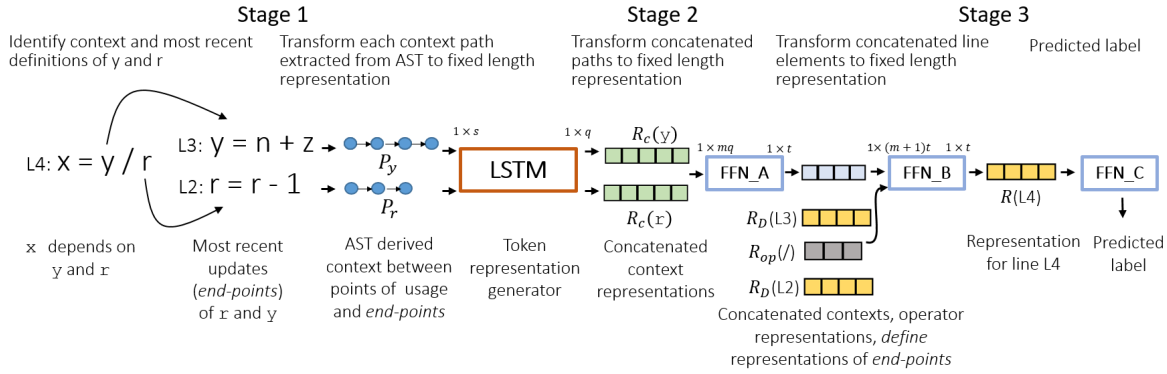


Figure 2. Overview of Vulcan. It uses the example of Figure 1. The inputs are the line of interest, i.e. L4, and the AST of the program. A representation for L4 is computed, which is used by a classifier to predict whether that line has a vulnerability. In Stage 1, we backtrack to the line where a variable used on the current line was most recently defined. In Figure 1, variables y and r were updated at L2 and L3. The path from the AST, expressing context in terms of control and data dependencies between line of interest and the most recent definition line, is then extracted. Each path, a sequence of length s, is then passed, one at a time, to a bidirectional LSTM with dot-product attention to obtain its continuous-valued representation of dimension q. These intermediate *context* representations are notated as $R_C(\cdot)$. In Stage 2 *context* representations $R_C(\cdot)$ of all tokens are concatenated and passed through a feed forward network FFN_A to obtain an intermediate representation (denoted in blue). In Stage 3, the intermediate representation is concatenated with the *define* representations (notated as $R_D(\cdot)$) of all the variables, and representations of operators. This concatenated vector is then transformed to a representation of dimension t using a feed forward network FFN_B, which is the final representation for line L4, R(L4). See Algorithm 1 for details. This is then passed to a classifier FFN_C to produce a binary value indicating presence of vulnerability

goal is to provide context with respect to the variable's most recent definition. We express the control flow that influences the variable, and the context of operators where it is an operand. For example, the loop enclosing the variable r in L2 in Figure 1 which exerts a control dependency, and the binary operator / on L4. Conveniently, these control and context dependencies are expressed by the program's AST via the AST path between the variable and its *end-point*. For example, in the snippet, for r in L2, we can use the path $\mathcal{P}_r$ where, in addition to the explicit data dependency modeled by the path when connecting to its usage in L4, the nodes LOOP, BinOp come up in the path as well. No other pre-processing or program slicing is needed to obtain this information.

Stage 2. The *context* of this variable, now a (*context*) path, is a variable length sequence of tokens. We next transform each variable's (context) path to a fixed length representation. Because a path is a sequence, we resort to a recurrent neural network for this transformation. We choose a bi-directional LSTM network to handle the long range dependencies in the sequence [8] (Network LSTM in Figure 2). The LSTM network has a dot-product attention mechanism [12], as it has been empirically shown to improve modeling of sequences.

We append the output of the LSTM to a list of the *context* representations for line of interest. Once all *context* paths, corresponding to each token on the line of interest, have been transformed, we pass this list through a simple feed forward model to obtain a single, fixed length representation of all the contextual information related to the line of interest (Network FFN_A in Figure 2).

Stage 3. The role of the next stage is to assemble the constituents of the line of interest. They comprise one-hot encodings for the operators, the *context* representation (Stage 2) and the list of *define* representations corresponding to *end-points* of each of the variables. We use a feed forward neural network to transform the aggregation into a final fixed length representation (Network FFN_B in Figure 2). It is this final representation of the line of interest we feed into a classifier for our downstream inference task. See lines 14, 15, 22, 24 in Algorithm 1 for how the line representations at *end-points* (which are the *define* representations) are used to form the final representation of line of interest.

Classifier Learning. Vulcan detects vulnerabilities on a given line of a Solidity smart contract. The final line representation produced by Stage 3 above is input to a feed-forward network that predicts the label - vulnerability or not (Network FFN_C, Figure 2). A cross-entropy loss between the predicted and true label trains the parameters of the entire architecture. Details on the dataset and the task setup are provided in the following section.

Algorithm 1 Algorithm to obtain line representations. (FFN is a Feed Forward Neural Network)

```
procedure RepresentLine(L, ast)
▷ L: Line number of current line in program P
▷ ast: AST object of program P
▷ Returns a t-dim representation of L
    ▷ Obtain RH tokens of expression on line L
    tokens ← RHS(L)
    defn_rs, cntxt_rs ← [ ], [ ]
    for tok ∈ tokens do
        (ep, pth) ← GetPath(tok, L, ast)
        ▷ Generate define representation (R_D(·), Fig 2)
        if pth ∈ ∅ then
            defn_r ← random(dim = t)
        else
            if ep ∈ ∅ then
                defn_r ← pth    ▷ One-hot-code of tok
            else
                defn_r ← RepresentLine(ep, ast)
        defn_rs ← [defn_rs defn_r]
        ▷ Generate context representation (R_C(·), Fig 2)
        ▷ See Fig 2 for LSTM, FFN_A, FFN_B
        cntxt_r ← LSTM(pth)
        cntxt_rs ← [cntxt_rs cntxt_r]
    ▷ Generate context representation ∀ tokens on L
    cntxt_rs ← FFN_A(cntxt_rs)
    ▷ Variable-length line representation
    line_rs ← [defn_rs cntxt_rs]
    ▷ Transform to fixed-length line representation
    line_rs ← FFN_B(line_rs)
    return line_rs
procedure GetPath(tok, L, ast)
▷ tok: Token on line L in program P
▷ L: Line number of current line in program P
▷ ast: AST object of program P
▷ Returns ep, the end-point– line number of most recent define of tok, and pth, path from ep to L
    if tok ∈ operators OR tok ∈ built-in func then
        ep ← ∅
        pth ← one-hot-encoding(tok)
    else
        if t ∈ user-defined func then
            ep ← line with return in tok's definition.
        else
            ep ← line where tok was last defined. ∅ if no previous definition exists.
        pth ← path in ast between token tok on line L and line ep. ∅ if no previous definition exists.
    return ep, pth
```

5 Experimental Setup

5.1 Vulcan

We train Vulcan, a classifier to predict vulnerabilities in `Solidity` programs. All our experiments are set up as binary classification tasks. We employ a weighted cross-entropy loss measure and sub-sample data from the training set to account for the highly uneven distribution of labels (details provided in the next subsection). For the attention mechanism, we implement Luong et al.'s dot-product attention.[12] We implemented all our models using `PyTorch version 1.0`. To ensure that all batches are of the same size, we limit the number of lines in a program to 128, number of variables in a program to 16, and the length of each variable's path to 32. These numbers are manually selected after observing their distribution on the train-set. The *context* and *define* representation dimensions (`q` and `t` in Figure 2) are 256 and 128 respectively. We use Adagrad as our optimizer and apply batch normalization. A URL to our source code will be released in the final draft of our work.

5.2 Dataset

We choose to work with `Solidity` because there are well documented recent cases of vulnerabilities leading to substantial financial losses. Multiple tools exists for detecting vulnerabilities in `Solidity` [13, 14, 16]. The most robust and popular of these tools uses symbolic analysis, which uses a SAT-solver to find erroneous program states [6]. However, this technique scales poorly. It requires experts to encode erroneous states and requires sophisticated software design that explores simulations of different program states.

Solidity and Ethereum. *Ethereum* is a popular public, decentralized, distributed ledger. It maintains transparent and immutable records which are programmable on the ledger. These are called *smart contracts*. Smart contracts enable program logic to be shared and executed by multiple parties. They are written in `Solidity`, a nascent programming language designed specifically for them. `Solidity` follows an object-oriented paradigm, is statically typed, and compiles to bytecode which can be executed on *Ethereum*'s Virtual Machine (EVM).

Vulnerabilities in `Solidity` programs. We analyze three vulnerabilities *(a)* Transaction order dependency (`TOD`) - these arise because of race conditions in the EVM which generate unreliable function call order, *(b)* State change after execution (`StateChange`) - these arise when function calls to third-party contracts hang, rendering all code written after the calls dead. *(c)* Integer Overflows, Underflows (`IntUnOv`) - these arise when the result of an arithmetic operation is larger than the word-size assigned by EVM. See Luu et al. [13] for examples of each of these vulnerabilities.

Dataset. We scraped publicly available `Solidity` programs from https://etherscan.io. As of May 2018, we scraped 28, 052 *verified* source files - files verified by Etherscan to be source

codes corresponding to their byte codes available on the *Ethereum* blockchain. 25, 813 of them were compilable. Among these, we selected only those which had at least two transactions recorded on *Ethereum*. This served as a proxy for filtering contracts involved in genuine transactions. We were left with 19, 023 files. In total, these files contained 69, 599 contracts, and a total of 487, 873 functions. We subsampled from this set by removing outliers and duplicates, reducing the total set to 194, 988 functions.

Labeling. Given the aim of this work is to evaluate a deep learning approach to program representation and vulnerability detection, we used `Mythril` [16], an open-source, symbolic analysis based vulnerability detection tool for smart contracts as a source of labels. `Mythril` provides line numbers of the vulnerabilities it detects. Lines not flagged by `Mythril` are considered benign. Our dataset had a total of 573, 251 lines of code. Of these, 12, 523 ($\sim$ 2.2%) were flagged as vulnerabilites by `Mythril`. The distribution of the three vulnerabilities `StateChange`, `TOD`, `IntUnOv` were 2750 (22%), 4830 (38%), and 4943 (40%) respectively. In our modeling process, each line of with code within every function was considered as an input to the model. A private correspondence with the authors of `Mythril` suggested that the tool has an error rate of close to 10-15%.

Error metrics. We use five error metrics to measure how well our classifier does, the same used by [10] - False positive rate (FPR = $\frac{FP}{FP+TN}$), False negative rate (FNR = $\frac{FN}{TP+FN}$), Recall (R = $\frac{TP}{TP+FN}$), Precision (P=$\frac{TP}{TP+FP}$), and F1-score ($\frac{2\times P\times R}{P+R}$) to evaluate how well our classifiers perform. Since we have much fewer vulnerable samples than benign samples, we want our classifier to be as precise as possible. Hence, what is desirable is low FPR and FNR, while having high recall, precision, and F1-scores.

6 Experiments & Results

We investigate Vulcan's performance as a vulnerability classifier using the metrics described in Section 5.2, and understand its components' contribution to its performance. Specifically, we ask -

> **RQ1.** Is Vulcan capable of detecting and flagging vulnerabilities in lines of programs?

Per Table 1, Vulcan has an F1-score of 60% compared to its closest and state-of-the-art approach Vuldeepecker, for which we train a model we call VULD-DeepLrn. VULD-DeepLrn has an F1-score of 51%. To obtain this comparison, we did our best to implement the Vuldeepecker approach as described in [9, 10] while applying the design to vulnerabilities in `Solidity`.[1] We heuristically identified arithmetic operations and function calls as *key points*, which the authors define to be "hotspots" for vulnerabilities. From these points, slices are made to generate *code gadgets* which are described by the authors as snippets of code which are inform or depend on the variables that interact at *key points*. We also observe that Vulcan's precision is better by 15% when compared to VULD-DeepLrn's, whereas the recall of both models is roughly equivalent.

Model	F1	P	R	FPR	FNR
Vulcan (This work)	60 (3)	59	60	2	40
VULD-DeepLrn	51 (2)	44	63	1	37
Tok-as-BOW	5 (0)	3	36	46	64
Only-AST-Nodes	18 (0)	10	70	22	29
Only-AST-Paths	30 (0)	61	20	0	80
VULD-LogRegr	23 (0)	17	35	1	65
Vulcan-NO_ENDPTS	52 (1)	45	60	3	40
Vulcan-PREV_LN	53 (3)	53	53	2	47
Vulcan-NO_ATTN	52 (2)	54	51	2	49

Table 1. Vulnerability classification of different models evaluated in our work. All values are percentages rounded to the nearest integer. This is a binary classification task of classifying whether a line has a vulnerability or not. The results are an average of 5 independent runs each. P, R stand for Precision and Recall respectively. For readability, we show standard deviations in brackets (·) only for F1-scores.

We expected Vulcan and VULD-DeepLrn would perform similarly. In principle, both approaches attempt to express similar information in programs. The relatively superior performance of Vulcan is likely due to a shortcoming in our implementation of Vuldeepecker. This shortcoming is prone to arising because of the complexity and heuristic judgement Vuldeepecker demands. Our approach, in contrast, requires far fewer design decisions. For instance, Vulcan needs no manual effort to identify *key points* to compute gadgets. Further, Vulcan uses AST paths while calculating gadgets requires program slicing. Vulcan achieves as much as Vuldeepecker while being a superior, seamless deep learning solution.

Reasoning at the granularity of lines is demonstrably hard - it demands a representation which accounts for the dependence information of the constituent tokens. Per Table 1, as expected, a naïve baseline of a bag of words of just the tokens appearing in a line does not discriminate presence of vulnerabilities (model Tok-as-BOW, F1-score of 5%). In Tok-as-BOW, a dictionary of all the unique tokens appearing in each line is populated and a count matrix is prepared, where each row corresponds to a line of program and the columns correspond to the set of unique tokens seen in the training set.

We also note that both Vulcan and VULD-DeepLrn perform modestly on the task of vulnerability classification.

[1] We did not communicate with the authors.

There is significant room for improvement. There could be several issues at play here. Two of the vulnerability classes in our dataset exploit *Ethereum*'s complex, concurrent architecture. Their precise meaning is tricky to express. Further, the dataset suffers from a class imbalance; just under two percent of the dataset is labeled with a positive class. Because this imbalance should be expected of real-world data, building models and techniques to deal with such settings is an important direction of future work.

On that note, very recent contributions in NLP [15, 18] have shown that despite high model performance, these models end up learning spurious correlations at best. This should be a call to our community to design programming tasks which truly can evaluate a machine's ability to comprehend them.

RQ2. What does each component of Vulcan contribute to its performance?

Vulcan has two key components - *context* and *define* representations. We investigate their respective contributions to Vulcan's ability to discriminate vulnerabilities. We proceed by considering models that isolate representation properties and by ablating Vulcan. We also investigate whether similar lines have similar line representations to lend confirmation that the architecture's representation space respects similarity.

Are *context* representations important? We would ideally want to answer this question by ablating just the *context* representations from the architecture (i.e. omitting `cntxt_rs` in Algorithm 1). This is not possible in the current setup since a token's *define* representation is recursively dependent on a line representation that is built from *context* representations. Hence, ablating the *context* representation would affect *define* representations as well. We instead train two simple bag of words classifiers using solely the *context* features to test whether they are predictive of program information. First, for a model named Only-AST-Nodes, we evaluate how much just the AST nodes appearing in Vulcan's paths, while ignoring other information which the entire sequence of nodes may provide, are predictive of the final task. We do this by training a bag of words on the names of unique AST nodes that appear in all of the variables' *context* paths seen training. Next, we train Only-AST-Paths, where we evaluate whether the sequential ordering of the nodes appearing in the paths adds additional value. We do this by learning a bag of words on all the unique paths, where a path is a string of AST nodes, of all the tokens seen in training. Only-AST-Nodes and Only-AST-Paths have F1-scores of 18% and 30% respectively. These two models suggest that AST node information and the sequential properties of the paths are important to the overall predictability.

In the spirit of Only-AST-Paths, we train model VULD-LogRegr, where we learn a bag of words model using the words extracted from all the gadgets of VulDeePecker seen in training. This gives a sense of how informative the code gadgets, which express a superset of the *context* paths, are by themselves. VULD-LogRegr has an F1-score of 23% placing its performance in between Only-AST-Nodes and Only-AST-Paths. This ranking could relate to our gadget design choices.

Are *define* representations important? We perform two ablations to our model to study whether the notion of *end-points* and their corresponding *define* representations add to the predictive ability of the model. First, we ablate the contribution of *define* representations completely. We name this model Vulcan-NO_ENDPTS. This corresponds to dropping `defn_rep` from being included in `line_rep` on line 22 in Algorithm 1. We expect ablating this aspect of the model to negatively affect the overall prediction since the model is left with only the contextual information present in the paths.

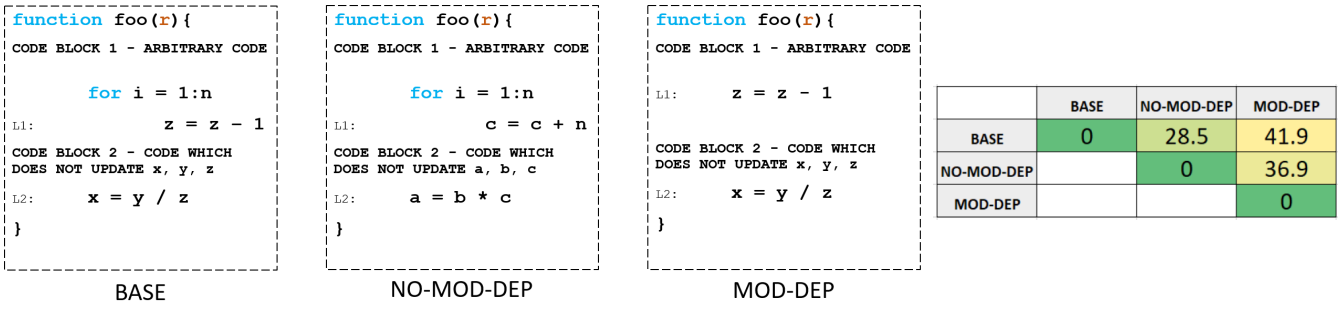
Second, we omit solely the *end-points* by selecting the *define* representations of the previous line instead of representations of the *end-points* of each token appearing on a line of interest. We name this modified model as Vulcan-PREV_LN. This corresponds to `ep` being assigned to `L-1` (line preceding `L`) on lines 11 and 13 in function `GetPath` in Algorithm 1. This is a tighter ablation as compared to Vulcan-NO_ENDPTS which compares the effect of just the *end-point* and its *define* representations.

Vulcan-NO_ENDPTS and Vulcan-PREV_LN have F1 scores of 52% and 53% respectively. This implies that the dependence information Vulcan captures of tokens appearing on a line of code accounts for a large part of its performance, as it rightly should. Comparing Vulcan-NO_ENDPTS and Vulcan-PREV_LN suggests that *end-points* are approximately as informative as previous lines. This merits future investigation to confirm if this lack of difference is seen across other tasks.

Overall, we find that the *context* and *define* representations we present in this work are important and contribute to the model's overall prediction.

Is attention important? We also evaluate whether the dot-product attention in Vulcan is effective. We name this ablated model Vulcan-NO_ATTN. This model has an F1-score of 52% versus Vulcan's F1-score of 60%. This worse value is expected because empirically, it has been shown that attention improves accuracy across most model architectures [22, 25]. We defer investigating the interpretability provided by attention to future work.

How informative are line representations? In designing Vulcan's architecture, our goal is finding distributed line representations that are similar for lines with similar contexts, and dissimilar for those without. To experimentally evaluate whether this is achieved, we set up the contexts of the tokens appearing in the lines of interest to be vastly different, while the lines themselves are identical. To proceed, we hand-craft three categories of simple `Solidity` programs -



	BASE	NO-MOD-DEP	MOD-DEP
BASE	0	28.5	41.9
NO-MOD-DEP		0	36.9
MOD-DEP			0

Figure 3. How informative are line representations? We set up three categories of synthetic `Solidity` programs containing 50 programs each. Categories NO-MOD-DEP and MOD-DEP modify unique programs in category BASE in a controlled and specific manner (details in Section 6). We compare the representations of specific lines of interest in programs from each of these categories as computed by a trained Vulcan. We compare the average L^2-distances of these representations among the three categories (right). Larger values indicate the representations are farther apart.

1. **BASE.** In this category we set up unique programs, each with a line of interest containing multiple tokens. One of these tokens is defined to have an update in a specific context, e.g. in a loop or within an if-branch, while arbitrary code can exist between the line of interest and the line of update of one of its tokens. For example, in Figure 3, the line of interest is L2, where variable z is updated in a loop before L1.
2. **MOD-DEP.** To set up programs in this category we first replicate the programs in BASE. Then each program is modified in a way which retains its overall structure but which changes variables by renaming them in the line of interest, operators by substitution and the quantity of arbitrary code by insertion or deletion. For instance, in the program in MOD-DEP in Figure 3, variables are renamed in the line of interest, the choice of specific arithmetic operators on the lines are changed, and the amount of arbitrary code (in blocks 1, 2) varies.
3. **NO-MOD-DEP.** To set up programs in this category we again first replicate the programs in BASE. Then each program in NO-MOD-DEP is left to be identical to its counterpart in BASE except that we modify the control context in which the token is last updated. For example, in Figure 3, the only difference is that variable z is not updated in a loop anymore (line L1).

We seed category BASE with 20 unique programs, with randomly inserted contexts and lines of interest. These then have one corresponding modified program each in categories MOD-DEP and NO-MOD-DEP. The lines of interest from each of these 60 (20×3) programs are the inputs to Vulcan after training. We extract line representations from our trained Vulcan and compute the L^2-distance between corresponding lines of corresponding programs across BASE, MOD-DEP and NO-MOD-DEP. We tabulate the average L^2-distance across the data and we observe the distance between programs in categories BASE vs. MOD-DEP, to be much less than in categories BASE vs. NO-MOD-DEP, and MOD-DEP vs. NO-MOD-DEP. Corresponding lines in programs in BASE vs. NO-MOD-DEP and MOD-DEP vs. NO-MOD-DEP should indeed have the farthest representations since the contexts of the tokens appearing in the lines of interest are vastly different, despite the lines themselves looking identical. Additionally, the difference between the averages of BASE vs. NO-MOD-DEP and MOD-DEP vs. NO-MOD-DEP is not significant, further suggesting that representations of the lines of interest of programs in BASE and MOD-DEP are similar. This shows that the representations our models generates capture the contexts of the tokens appearing in it.

7 Conclusion and Future work

We introduce Vulcan, a novel neural architecture to construct distributed representations for lines of programs. We use these to classify whether a line of a `Solidity` program has a vulnerability in it or not. We show that Vulcan compares favorably with a state-of-the-art line-level classifier but which involves significant pre-processing steps. Further, we show, through ablations, that the different components which make up our architecture contribute to the model's performance and are necessary. We also show experimentally that Vulcan generates similar representations for lines of similar meaning. Our work opens up interesting areas of future work, where we can compare this architecture with other modeling approaches like graph neural networks and compare their performance on different applications. We also provide one possible answer to the larger question of what the right representation ought to be when reasoning about programs statistically. Understanding these alternatives will lead us to truly leverage and scale a data-driven approach to analyzing and generating programs.

8 Acknowledgement

We thank the members of the ALFA lab, CSAIL, MIT for helpful discussions on drafts of this work. This work was funded by the *Fintech@CSAIL* initiative. Nicolas was funded by Ecole Polytechnique Fédérale de Lausanne (EPFL) to carry out his master's thesis at ALFA lab.

References

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

[5] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: probabilistic model for code. In *International Conference on Machine Learning*. 2933–2942.

[6] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[7] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. 2014. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 49–65.

[8] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF models for sequence tagging. *arXiv preprint arXiv:1508.01991* (2015).

[9] Zhen Li, Jialai Wang, and et al. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. abs/1807.06756 (2018).

[10] Zhen Li, Yuyi Zhong, and et al. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[11] Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. 2016. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics* 4 (2016), 521–535.

[12] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).

[13] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[14] Manticore. 2018. https://github.com/trailofbits/manticore.

[15] R Thomas McCoy, Ellie Pavlick, and Tal Linzen. 2019. Right for the wrong reasons: Diagnosing syntactic heuristics in natural language inference. *arXiv preprint arXiv:1902.01007* (2019).

[16] Mythril. 2017. https://github.com/ConsenSys/mythril.

[17] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *NDSS*, Vol. 5. Citeseer, 3–4.

[18] Timothy Niven and Hung-Yu Kao. 2019. Probing neural network comprehension of natural language arguments. *arXiv preprint arXiv:1907.07355* (2019).

[19] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[20] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[21] Rebecca L Russell, Louis Kim, Lei H Hamilton, Tomo Lazovich, Jacob A Harer, Onur Ozdemir, Paul M Ellingwood, and Marc W McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. *arXiv preprint arXiv:1807.04320* (2018).

[22] Tao Shen, Tianyi Zhou, and et al. 2018. Disan: Directional self-attention network for rnn/cnn-free language understanding. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[23] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[24] Shashank Srikant and Varun Aggarwal. 2014. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1887–1896.

[25] Gongbo Tang, Mathias Müller, and et al. 2018. Why self-attention? a targeted evaluation of neural machine translation architectures. *arXiv preprint arXiv:1808.08946* (2018).

[26] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[27] Ethan Wilcox, Roger Levy, Takashi Morita, and Richard Futrell. 2018. What do RNN Language Models Learn about Filler-Gap Dependencies? *arXiv preprint arXiv:1809.00042* (2018).

[28] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2009. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proceedings of the 2009 IEEE/ACM International Conference on automated software engineering*. IEEE Computer Society, 605–609.