# Vulcan: Classifying Vulnerabilities in Solidity Smart Contracts Using Dependency-Based Deep Program Representations

by

Shashank Srikant

B.Tech, National Institute of Technology Kurukshetra, India (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020



Author .....
Department of Electrical Engineering and Computer Science
May 15, 2020

Certified by .....
Una-May O'Reilly
Principal Research Scientist, Computer Science & AI Laboratory
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

# Vulcan: Classifying Vulnerabilities in Solidity Smart Contracts Using Dependency-Based Deep Program Representations

by

Shashank Srikant

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science

## Abstract

As domain specific languages such as `Solidity` for *Ethereum* have emerged to program blockchain distributed ledgers, domain-specific bugs and vulnerabilities have reciprocally arisen. In this thesis, I propose a machine learning approach to designing classifiers which can flag specific lines of programs containing such vulnerabilities. This classification task calls for reasoning beyond *what tokens are in a line* to reasoning about *how each token lies within a control context (e.g. loop) and how its meaning depends on its preceding definition.* We present a neural architecture, Vulcan, which employs a distributed representation in a latent feature space to express these properties, while ensuring that lines of similar meaning have similar features. Using paths of the program's abstract syntax tree (AST), Vulcan inputs contextual information about tokens in a line to a bi-directional LSTM with an attention mechanism. It concurrently represents the meaning of a token in a line by recursively embedding all preceding lines where it is defined. In our experiments, Vulcan compares favorably with a state-of-the-art classifier that requires significant preprocessing of programs, suggesting the utility of using deep learning to model program dependence information.

Thesis Supervisor: Una-May O'Reilly
Title: Principal Research Scientist, Computer Science & AI Laboratory

# Acknowledgments

Thank you Una-May, for providing me an opportunity to pursue ideas at the intersection of PL/SE, ML, and Neuroscience. This is a unique area of research which few can boast of being involved in. I am grateful.

Thanks Erik, Abdullah, Jamal for the fun group members you are. Thank you Nicole for all your help in ensuring I am productive.

Thank you Ev Fedorenko, for accepting me so readily to collaborate with your group on the exciting neuroscience project.

I'm privileged to be related to some wonderful women who set the bar for unconditional kindness and perseverance. Thank you aunts Aparna and Meenakshi. You are an inspiration.

Thanks Murthy and Ranjitha for making my move to Cambridge as smooth as I could have hoped for. I will make that visit to the west coast happen. Soon.

I also got a chance to meet and make some wonderful friends during my stay here. Meanwhile, older friends continued to keep in touch and engaged in stimulating conversations. Thanks folks.

Gratitude to my parents, family, and teachers who helped me get here.

Thank you Kencha and Julie for your weekly conversations over video calls.

This note is a work in progress. I will have more to write about in my subsequent thesis.

# Contents

# List of Figures

5-1 **Overview of Vulcan.** It uses the example of Figure 3-1. The inputs are the line of interest, i.e. `L4`, and the AST of the program. A representation for `L4` is computed, which is assigned to be variable `x`'s representation for its subsequent uses, and 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 3-1, 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 . . . . . 31

# List of Tables

# Chapter 1

# Introduction

Vulnerabilities and bugs that escape recognition persist and remain a concern for developers and enterprises. Preemptive detection of vulnerabilities before program execution is a challenging problem because it is hard to statically characterize a program. Although a vulnerability (such as a buffer overflow) is easy to describe, identifying a causal path leading up to it is hard. A pathological combination of control paths and data transformations has to be anticipated.

In this thesis, I study programs written in `Solidity`, a domain specific language (DSL) designed to implement *Ethereum*, a popular decentralized, distributed ledger. `Solidity` programs support high-stakes transactions but recent concurrency and over/underflow bugs and vulnerabilities exploited in them have resulted in multi-million dollar losses [16]. The combination of the `Solidity` programming model and requirements of a distributed ledger drive unique versions of these bugs and vulnerabilities, likely to be common to other (but not wider) distributed ledger and DSL settings. Unfortunately, without highly expert repurposing, existing analysis tools and techniques are inadequate for detection and a pool of active developers who can contribute to this task is usually too small to help. The method we investigate in this work aims to reduce the expertise and effort required to repurpose traditional analysis tools.

A recent direction of program analysis research infers properties of programs by learning statistical models of them with 'Big Code' architectures [23]. Typically, a

'BigCode' architecture relies upon a corpus of programs and has two key stages. The first stage models the program space represented by the corpus. The representation space is a high-dimensional latent space in which the features of programs have comparative value i.e. the representations of similar program components are close in this space while dissimilar components are far apart. The input to the stage is a program and the output is the program's *distributed representation*. This distributed representation is passed to the second stage where another machine learning model (classifier) is trained in a supervised manner with the distributed representations and their corresponding labels. A variety of applications are well served by this approach including renaming poorly named variables to meaningful ones [7], detecting clones [30], and detecting bugs in functions [22]. See Allamanis et al. [3] for a review of relevant literature.

In this work, we investigate whether such a 'Big Code' architecture can classify line-level vulnerabilities in `Solidity` programs. Applications reaping improvements from the 'Big Code' approach often, in their first stage, represent programs merely by sequences of their source-code tokens. The majority of examples listed in Table 1 in the survey of Allamanis et al. [3] use tokens. Tokens are convenient to use, however whether they are ideal for the task of vulnerability classification is open to question. Programs have rich structural and contextual information which a sequence of tokens does not explicitly capture. Further, programmers need to debug or to reason about vulnerabilities. For this, they need line-level detection. To date, popular applications such as CodeSummarize - predicting a semantically meaningful name to a method when provided its body [5], VarRename - predicting a semantically meaningful name to a variable [2], VarMisuse - predicting whether a variable is 'out of place', given the context it appears in [2] either reason at the granularity of a whole class or method, or reason about specific tokens (variables) appearing in the program. Our goal is line-level vulnerability classification.

We present Vulcan (Vulnerability Classification Network), a neural network architecture, which balances the twin goals of capturing rich semantic information while having the contents of a line of program as the unit of representation. We demon-

strate Vulcan on vulnerability classification at the line level in `Solidity` programs. Vulcan takes a much more nuanced approach to forming a distributed representation than tokenization. Using paths of the program's abstract syntax tree (AST), Vulcan inputs contextual information about tokens in a line to a bi-directional LSTM with an attention mechanism. It concurrently represents the meaning of token in a line by recursively embedding all preceding lines where it is defined. We train Vulcan on labels provided by `Mythril`, a state of the art static analysis tool for `Solidity` [19]. While `Mythril`, like Vulcan, can also detect vulnerabilities, it relies upon expertly hand-crafted assertions related to erroneous program states in `Solidity`. Our goal is to establish that a purely data-driven approach, without specialized proficiency and manual effort, can match the performance of an expert-designed tool like `Mythril`. We foresee architectures like Vulcan being central to performing similar detection for other nascent DSLs, through transfer learning and domain adaptation. We evaluate Vulcan's performance against VULDEEPECKER, another state-of-the-art, data-driven, vulnerability detection system. We also evaluate whether the distributed representation produced by our system defines a latent feature space where lines of similar meaning have similar features.

We proceed as follows: Chapter 2 describes the `Solidity`-specific vulnerabilities Vulcan classifies. Chapter 3 presents the representation design. Chapter 4 covers related work, Chapter 5 conveys Vulcan's algorithm and method and Chapter 6 experimental setup. Chapter 7 presents experimental results and Chapter 8 conclusions and future work.

# Chapter 2

# Motivation

We describe briefly the *Ethereum* environment, vulnerabilities encountered in `Solidity` programs, and highlight why extant tools can be served well by our current approach.

## 2.1 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).

## 2.2 Vulnerabilities in `Solidity` programs

`Solidity` programs control operations occurring in a blockchain environment. A combination of various design flaws in `Solidity` and the unique nature of operations on *Ethereum* have resulted in attackers exploiting very specific vulnerabilities. Two which are very unique to the blockchain model of operations are:

- **Transaction order dependency (`TOD`)** In `Solidity`'s programming model, a

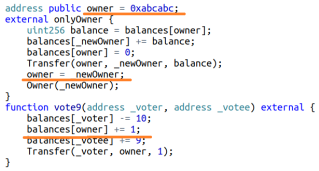



Figure 2-1: **`TOD`** - The value of `owner` in `vote9` can be affected by a call to `onlyOwner`

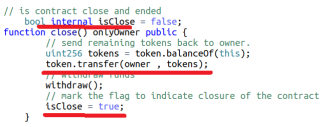



Figure 2-2: **`StateChange`** - `isClose` will not be set to `true` if `transfer` hangs

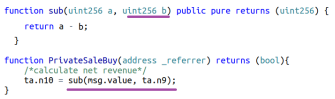



Figure 2-3: **`IntUnOv`** - The subtraction operation can underflow if $b > a$

caller calling a function can witness different program states depending on the order in which the function was called with respect to other functions in a contract. Important transactions, whose order of executions determines the consistency of operations, can fail as a consequence. See Figure 2-1 for an example.

- **State change after execution (`StateChange`)** `Solidity` and the EVM does not support concurrent programming in that, when an external contract is called from within a contract, control-flow switches to the callee. As a consequence, if the callee code does not execute as expected, the function call gets stuck forever. If critical logic is coded *after* such a function call in the calling function, those lines of code may never get executed. Unaware of the implications of such a design constraint, programmers write critical sections of their codes (say, updates to an account balance) after such calls to external functions without proper exception handling. Figure 2-2 shows an example. A number of attacks have exploited this vulnerability.

Vulcan also addresses **Integer Overflows, Underflows (`IntUnOv`)**. While not unique to `Solidity` or blockchain and DSLs, these are vulnerabilities that arise when the result of an arithmetic operation is larger than the word-size assigned by EVM. See Figure 2-3 for an example. See Luu et al. [16] for detailed examples of how each of these language defects have been exploited in vulnerabilities leading to substantial financial losses.

## 2.3 Existing methods of vulnerability detection for `Solidity`

Three tools of considerable sophistication exist for detecting vulnerabilities in `Solidity` - `Manticore` [17], `Mythril` [19], and `Oyente` [16]. We evaluated all three and found only `Mythril` to be maintained by an active community. It uses symbolic analysis, which uses a SAT-solver to find erroneous program states [8]. While it is easily to use this tool, it is non-trivial to design and develop it. It requires experts who intimately understand the nuances of `Solidity` and are able to encode erroneous states as assertions, and requires sophisticated software design that explores simulations of different program states. It took a core group of expert developers multiple months to put together `Mythril`. Using `Mythril` as our standard, we evaluate whether the methods we introduce in this work can achieve similar detection capabilities.

# Chapter 3

# 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.

We design 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 3-1 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 variable `x` on line `L4`: $x = y/r$, thus representing `L4` itself? We know that `L4` updates variable `x` and this new value of `x` depends on variables `y,r` which are operands of a division operation. To represent this division expression, we need the values of `y` and `r`. What, at `L4`, are the values

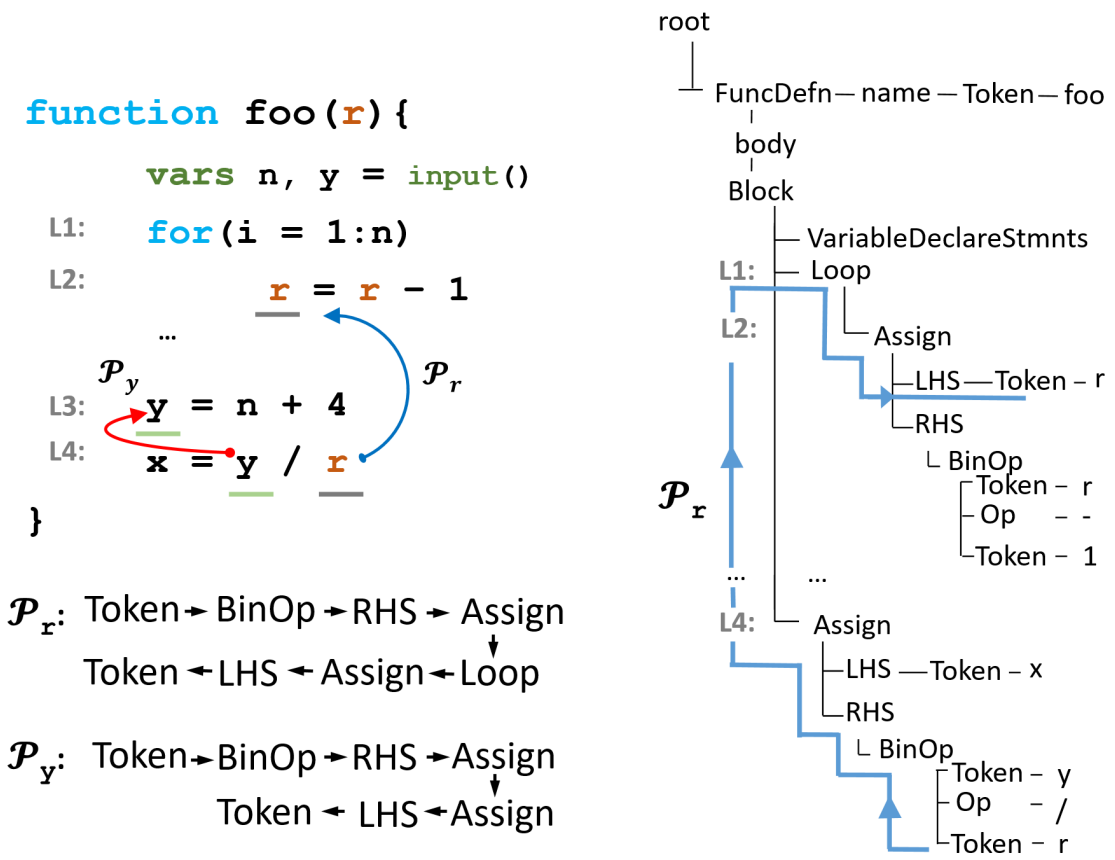



Figure 3-1: An example code snippet. We show how different tokens and their dependencies can be represented. Paths comprising the nodes of a program AST represent both control and data information. Here, paths $\mathcal{P}_r, \mathcal{P}_y$ connect the usages of variables `r` and `y` on line `L4` to their respective updates on lines `L2` and `L3`.

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 `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`. The more involved example is backtracking `r`. It is updated on `L2` where `r`'s assignment is in the control context of a loop. We have one more detail to consider: the right hand side of expressions which assign a value to `y` and `r` have prior definitions and contexts. Thus, we have to recursively represent these until we finally recurse to the base case of their first definition, which we can express directly.

In Chapter 5 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. In the Methods chapter (Chapter 5), 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 $v$.

# Chapter 4

# Related Work

We present works related to our approach. There are two aspects we cover - works which have explored different representations for programs, and works which have dealt with detecting vulnerabilities. We summarize these works in Table 4.1.

## 4.1 Program Representations

Vulcan is related to other "Big Code" approaches that use AST-based representations to reason about programs. These works, like Vulcan, process a static view of programs while ignoring any run-time related phenomena like aliasing and dynamic dispatching.

Bielik et al. [7] correct improper variables names using probabilistic graphical models (PGM) of features that express AST edge information. Vulcan, in contrast, employs a neural architecture to represent AST edge information thus avoiding learning and scaling challenges of graphical models.

Both Hsiao et al. [10] and Srikant et al. [27] use program dependence graphs to reason about code-clone detection and bug finding, or automated assessments, respectively. They represent an entire program as counts of edge information in the dependence graphs. They then build $n$-gram models based on these counts. Vulcan instead builds a distributed representation. Given its simplicity and effectiveness, we re-implemented this approach and use it as a baseline in our work.

Alon et al [6] introduce the notion of paths - a sequence of AST nodes capturing

the dependencies between different occurrences of a variable appearing in a program. They use these paths as model inputs and demonstrate the resulting program representations are suitable for modeling a variety of tasks. Vulcan also uses this notion, positioning paths as building blocks in a larger representation scheme.

Allamanis et al and Zhou et al. [4, 32] incorporate AST edge information in the graph neural networks they use to model programs. The bi-LSTM of Vulcan has same inductive bias as a graph neural network because both specifically model the recurrent structure of the input. Graph modeling approaches do not naturally support inferring program properties at the granularity of lines because their atomic unit is a token which leads to averaging the representations for each token on a line. Vulcan is unique in its attention to how tokens and lines are defined by preceding code and code providing control context. We defer using graph networks in lieu of bi-LSTMs to future work in order to focus on a comparison with a state-of-the-art vulnerability classification method which, like Vulcan, reasons on lines of programs.

| **Reference** | **Input Representation** | **Learning Algorithm** | **Prediction Site** | **Task** | **Language** | **N** |
|---|---|---|---|---|---|---|
| Allamanis et al. [4] | AST edges | Gated Graph Neural Net (GGNN) | Variables | Rename, find misuse of variables | Java | 2.9M |
| Alon et al. [6] | AST paths | Bi-LSTM with Attn | Function | Predict function & variable names | Java, Py | 1.7M |
| *DeepBugs* [22] | Word2Vec on tokens | LSTM | Function | Predicting bugs | Java | 1M |
| *Devign* [32] | AST edges | Graph Neural Network (GNN) | Functions | Vulnerability prediction | C | 590k |
| VulDeePecker [14] | *Gadgets* extracted from AST edges | Bi-LSTM | Line | Vulnerability prediction | C | 200K |
| Russell et al. [24] | Tokens | CNN | Function | Vulnerability prediction | C, C++ | 1.1M |
| Vulcan (This work) | AST paths | Bi-LSTM with Attn | Line | Vulnerability prediction | Solidity | 100K |

Table 4.1: A summary of relevant research on data-driven program analysis and vulnerability prediction. *Input Representation* refers to how the input programs were represented and fed to the *Learning Algorithm*. *Prediction Site* refers to the granularity of the program on which the model makes its prediction. *N* refers to the number of programs trained on.

## 4.2 Vulnerability Detection

Some recent works have focused on detecting and classifying vulnerabilities through traditional program analysis techniques [31, 20, 26]. They use static analysis and fuzzing to detect vulnerabilities.

In works employing machine learning, VulDeePecker [14], DeepBugs [22], and Russell et al. [24] are closest to the design we propose. We discuss them in detail.

### 4.2.1 *VulDeePecker*

VulDeePecker [14] 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 [13], 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.

### 4.2.2 *Russell et al.*

This work [24] 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.

### 4.2.3 *DeepBugs*

The representation used in this work [22] 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.

# Chapter 5

# Method

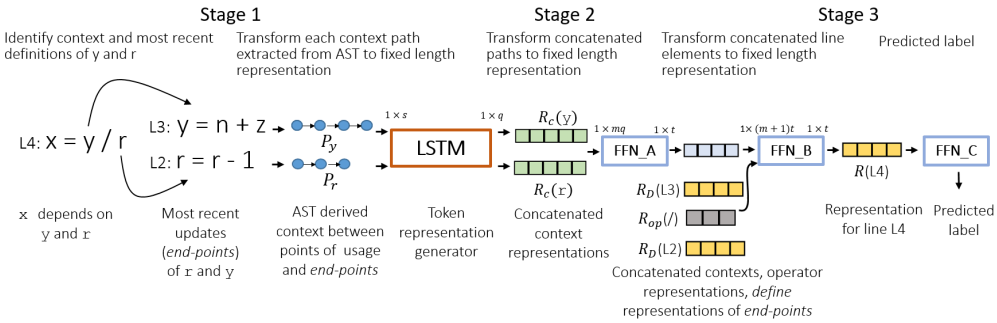



Figure 5-1: **Overview of Vulcan.** It uses the example of Figure 3-1. The inputs are the line of interest, i.e. `L4`, and the AST of the program. A representation for `L4` is computed, which is assigned to be variable `x`'s representation for its subsequent uses, and is used by a classifier to predict 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 3-1, 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

## 5.1 Overview

We describe our neural network architecture in this section. It consists of three stages. The input to the system is a line of the program in which a value is assigned, and its output is a distributed representation for the line which is used to predict a label for the line. When training the model, this line is accompanied by a label. Each line, in the order it appears in the program, is provided to the system one after the other. We provide dimensions for intermediate and final outputs of the architecture in Figure 5-1. This architecture is sketched in Algorithm 1.

## 5.2 Stage 1

The input to Stage 1 is a tokenized line of code 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 an inter-procedure computation, and the base case is the variable's very first definition in the file. This representation 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 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 3-1 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.

## 5.3 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 by means of a recurrent neural network. We select a bi-directional LSTM network to account for possible long range dependencies in the sequence [11] (Network `LSTM` in Figure 5-1). The LSTM network includes a dot-product attention mechanism [15] because 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 5-1).

## 5.4 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 5-1). It is this final representation of the line of interest we feed into a classifier for our downstream inference task of

vulnerability detection. 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 5-1). 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.

**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 ($R_D(\cdot)$, Fig 5-1)
        **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 ($R_C(\cdot)$, Fig 5-1)
        ▷ See Fig 5-1 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

# Chapter 6

# Experiment Setup

## 6.1 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 lines of code. Further, we removed programs which were two standard deviations away from the corpus median in program length, and removed duplicates, reducing the total set to 194,988 lines of code.

## 6.2 Labeling

Given the aim of this work is to evaluate a deep learning approach to program representation and vulnerability detection, we used `Mythril` [19], 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.

## 6.3 Implementation Details

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.[15] 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 5-1) 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.

## 6.4 Error Metrics

We use five error metrics to measure how well our classifier does, the same used by [14] - 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.

# Chapter 7

# Experiments & Results

We investigate Vulcan's performance as a vulnerability classifier using the metrics described in Chapter 6.4, and understand its components' contribution to its performance.

## 7.1 Research Question 1.

The first research question we investigate is -

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

Per Table 7.1, 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 [14, 13] 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 bet-

[1] We did not communicate with the authors.

ter by 15% when compared to VULD-DeepLrn's, whereas the recall of both models is roughly equivalent.

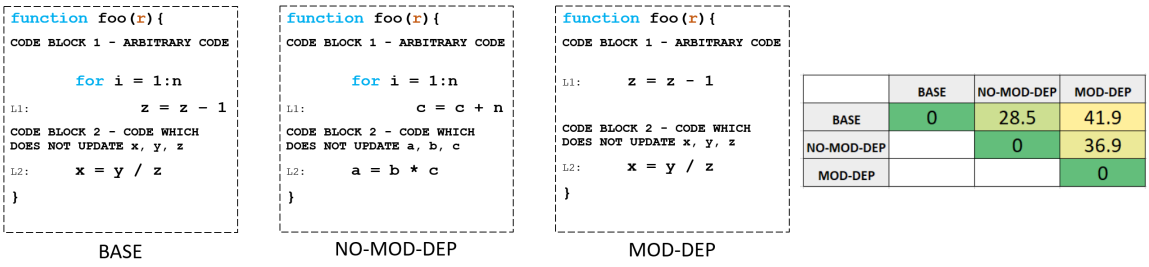
| Model | F1 | P | R | FPR | FNR |
|---|---|---|---|---|---|
| Vulcan (This work) | 60 (3) | 59 | 60 | 2 | 40 |
| VULD-DeepLrn | 51 (2) | 44 | 63 | 1 | 37 |
| Tok-as-BOW | 5 (0) | 3 | 36 | 46 | 64 |
| Only-AST-Nodes | 18 (0) | 10 | 70 | 22 | 29 |
| Only-AST-Paths | 30 (0) | 61 | 20 | 0 | 80 |
| VULD-LogRegr | 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 7.1: 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 $(\cdot)$ 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. Vulcan, in contrast, requires far fewer design decisions. For instance, Vulcan does not need the manual effort required 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 as evidenced by Table 7.1's results overall. They further indicate that it demands a representation which, at least, accounts for the dependence information of the constituent tokens. Tok-as-BOW, a naïve baseline model of a bag of words of just the tokens appearing in a line, does not discriminate presence of vulnerabilities (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.




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

Figure 7-1: **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 Chapter 7.2.4). 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.

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

Related to model performance improvement, very recent contributions in NLP [21, 18] have shown that high performing models similar to Vulcan's end up learning spurious correlations. The NLP community is currently engaged in asking how to effectively probe such models and how to prevent this 'memorization'[29]. Along with a push for performance improvement, our community should be wary of similar outcomes. The community should pursue equivalent probing and design of program modeling tasks that specifically answer whether a model truly comprehends, rather than spuriously correlates, its training corpus.

### 7.1.1 Predictions missed by Mythril

In this work, we consider detections made by Mythril as the gold-standard and use them to train Vulcan. However, given we found that Mythril and the two other tools we examined had label disagreement, one might consider that Mythril's labels are also prone to error. It could, therefore, be the case that Vulcan learns to distinguish vulnerabilities which Mythril does not identify. To investigate this, we manually evaluated 60 lines of programs out of the 2314 lines that were flagged by Mythril as being benign but were flagged by Vulcan as containing a vulnerability. These were out of a total of 49928 lines of programs in the test set. Note - this is not the same metric as False Negative Rate (FNR) reported in Table 7.1. We found nine lines out of 60 ($\sim$ 10%) to indeed have a vulnerability, while the rest were correctly labeled by Mythril. All these programs had the *state change after execution* (StateChange) vulnerability in them. This issue had been raised and eventually fixed by the developers of Mythril in a version ahead of the one we initially used in our experiments. See https://github.com/ConsenSys/mythril/issues/633 for details. This is an important, though preliminary, result because it shows how, despite Mythril being designed with handcrafted conditions, does not recognize all erroneous conditions. It suggests a need for complementary techniques, such as Vulcan, to augment manual approaches. Importantly, it indicates that a purely data-driven method, as demonstrated by Vulcan, is able to learn even with noisy labels.

## 7.2 Research Question 2.

The second research question we investigate is -

> **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.

### 7.2.1 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. This is closest to the representation used by Hsiao et al. [10] and Srikant et al. [27]. See Chapter 4 for details. 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.

### 7.2.2 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.

### 7.2.3 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 [28, 25]. We defer investigating the interpretability provided by attention to future work.

### 7.2.4 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 -

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 7-1, 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 7-1, 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 7-1, 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 ($20 \times 3$) 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.

# Chapter 8

# 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. A promising direction of research having demonstrated a system like Vulcan is to serve other nascent languages with with insufficient detection tools and a developer community. Models trained using domain adaptation techniques on Vulcan can aid in inferring properties of the language while circumventing the need to have a large corpus of labeled data. 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.

# Bibliography


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

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

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

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

[5] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100, 2016.

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

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

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

[9] James V Haxby, M Ida Gobbini, Maura L Furey, Alumit Ishai, Jennifer L Schouten, and Pietro Pietrini. Distributed and overlapping representations of faces and objects in ventral temporal cortex. *Science*, 293(5539):2425–2430, 2001.

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

[11] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

[12] Anna A Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'reilly, Marina U Bers, and Evelina Fedorenko. Comprehension of computer code relies primarily on domain-general executive resources. *BioRxiv*, 2020.

[13] Zhen Li, Jialai Wang, and et al. Sysevr: A framework for using deep learning to detect software vulnerabilities. abs/1807.06756, 2018.

[14] Zhen Li, Yuyi Zhong, and et al. 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*, 2018.

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

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

[17] Manticore. `https://github.com/trailofbits/manticore`, 2018.

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

[19] Mythril. `https://github.com/ConsenSys/mythril`, 2017.

[20] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

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

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

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

[24] Rebecca L Russell, Louis Kim, Lei H Hamilton, Tomo Lazovich, Jacob A Harer, Onur Ozdemir, Paul M Ellingwood, and Marc W McConley. Automated vulnerability detection in source code using deep representation learning. *arXiv preprint arXiv:1807.04320*, 2018.

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

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

[27] Shashank Srikant and Varun Aggarwal. 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*, pages 1887–1896, 2014.

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

[29] Elena Voita and Ivan Titov. Information-theoretic probing with minimum description length. *arXiv preprint arXiv:2003.12298*, 2020.

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

[31] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 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*, pages 605–609. IEEE Computer Society, 2009.

[32] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207, 2019.