On using distributed representations of source code for the detection of C security vulnerabilities

David Coimbra✔, Sofia Reis✔, Rui Abreu, Corina Păsăreanu, Hakan Erdogmus
✔INESC-ID & IST/U.Lisbon, Portugal
{david.coimbra, sofia.o.reis}@tecnico.ulisboa.pt
INESC-ID & FEUP, Portugal
rui@computer.org
Carnegie Mellon University, USA
corina.pasareanu@west.cmu.edu, hakane@andrew.cmu.edu

Abstract

This paper presents an evaluation of the code representation model Code2vec when trained on the task of detecting security vulnerabilities in C source code. We leverage the open-source library astminer to extract path-contexts from the abstract syntax trees of a corpus of labeled C functions. Code2vec is trained on the resulting path-contexts with the task of classifying a function as vulnerable or non-vulnerable. Using the CodeXGLUE benchmark, we show that the accuracy of Code2vec for this task is comparable to simple transformer-based methods such as pre-trained RoBERTa, and outperforms more naive NLP-based methods. We achieved an accuracy of 61.43% while maintaining low computational requirements relative to larger models, compared to an accuracy of 61.05% achieved by RoBERTa on the same benchmark.

1 Introduction

Security vulnerabilities are a major concern in software development, as even the simplest mistakes can be turned into attack vectors by a malicious party. In continuous integration (CI), it is common to introduce static analysers into the build pipeline to verify code against known patterns [1–3]. For example, Brakeman[1] and SonarQube[2] are static analysers capable of detecting software vulnerabilities in source code that can be used for this purpose.

Static analyzers can reduce security concerns in the software development life-cycle when introduced in the implementation phase, to assist developers to produce safer software. These analyzers are also useful tools for code review. Nowadays, one common practice to provide feedback is using GitHub pull requests. Warnings are usually added automatically to pull requests as comments, which can be reviewed manually before merging. Such pipelines streamline the quality assurance process and increase productivity. However, current techniques in static analysis are limited: they are prone to false positives (wasting developer effort), and false negatives (making the analysis less reliable) [4, 5].

In recent years, there has been an increase in the application of statistical models, namely neural networks, to a variety of code intelligence tasks, including vulnerability detection [6, 7]. This research has mainly focused on the application of pre-trained models that capture knowledge particular to a specific programming language, and has been inspired by transformer-based models [8] such as `BERT` [9] and `GPT` [10], both developed in the context of natural language processing (NLP). For example, models such as `CodeBERT` [11], `C-BERT` [12] and `PLBART` [13] produce distributed representations from source code that have been applied to many code tasks, such as code search, code translation and vulnerability detection. The recent `CodeXGLUE` benchmark [14] aims to facilitate the comparison and evaluation of these recent models in a large variety of tasks. This benchmark is publicly available and open for for further contributions [3].

Although these models are demonstrably effective, they also have their limitations: mainly, large models with hundreds of millions of parameters need a relatively large amount of computational resources, including both memory and CPU time [15]. The requirement for large amounts of computational resources is a significant limitation for researchers and developers, and it can make the usage of large pre-trained models impractical. There is a noticeable trade-off between model representativeness and convenience of use. As a result, often smaller models are preferred for detection performance.

In this work, we investigated the applicability of a code representation model, `Code2vec` [16], to the vulnerability detection task. `Code2vec` is built on a simple attention-based feed-forward neural network that learns and combines semantic knowledge extracted from syntactic paths (*path-contexts*) in an abstract syntax tree; a *bag of path-contexts* serves as a representation of a particular code snippet [17].

To the best of our knowledge, `Code2vec` has not been evaluated for vulnerability detection. We evaluated `Code2vec` on a labeled corpus of `C` functions. We found that `Code2vec` outperformed traditional NLP-based approaches for vulnerability detection at an accuracy of

[1]Brakeman Tool: `https://brakemanscanner.org/` (Accessed August 11, 2021)

[2]SonarQube GitHub Integration Latest Details Webpage: `https://docs.sonarqube.org/latest/analysis/github-integration/` (Accessed August 11, 2021)

[3]CodeXGLUE GitHub Repository: `https://github.com/microsoft/CodeXGLUE` (Accessed August 11, 2021)

```
void scsi_req_abort(SCSIRequest *req, int status) {
    if (!req->enqueued) {
        return;
    }
    scsi_req_ref(req);
    scsi_req_dequeue(req);
    req->io_canceled = true;
    if (req->ops->cancel_io) {
        req->ops->cancel_io(req);
    }
    scsi_req_complete(req, status);
    scsi_req_unref(req);
}
```

```
req,ef0f501e85af13c3f3106da4bf2906e0,scsirequest
status,ef0f501e85af13c3f3106da4bf2906e0,int
scsirequest,be3b43d7f4a43891e0385f2eca2cf07f,req
scsirequest,d5ca1a6cadb392e083136474a8cabc30,enqueued
req,4f27c208767e095fd30422fc396d9a65,enqueued
scsirequest,8c1b9d254efaff732c5b9f31dc9b2a4d,return
req,08ce61e16e85e647cf23073a8ca791b5,return
enqueued,e8e7e89c85a8b6a64d78437b0cec31b7,return
req,06f1bd4eabc1f73da38b772edea29dca,scsirequest
scsireqref,111f072b31664dc52f36bbafdf999c3a,req
scsireqref,1dbfd33caf135e6c4c08fc7a55af5682,scsirequest
req,06f1bd4eabc1f73da38b772edea29dca,scsirequest
scsireqdequeue,111f072b31664dc52f36bbafdf999c3a,req
```

Figure 1: Example of a non-vulnerable function and the first 13 path-contexts extracted by `astminer`. For each function, `astminer` extracts a maximum of 200 path-contexts. This example only presents 13 path-contexts due to space limitations. The syntactic paths between tokens are encoded using `MD5`.

```
uint32_t HELPER(shr_cc)(CPUM68KState *env, uint32_t val,
                        uint32_t shift) {
    uint64_t temp;
    uint32_t result;
    shift &= 63;
    temp = (uint64_t)val << 32 >> shift;
    result = temp >> 32;
    env->cc_c = (temp >> 31) & 1;
    env->cc_n = result;
    env->cc_z = result;
    env->cc_v = 0;
    env->cc_x = shift ? env->cc_c : env->cc_x;
    return result;
}
```

```
cpumkstate,c4b44d57c510ae50e3f1f4c368b9e232,env
uintt,c4b44d57c510ae50e3f1f4c368b9e232,val
uintt,c4b44d57c510ae50e3f1f4c368b9e232,shift
uintt,8a2b9543a7267ced6831ca9e368e2149,temp
uintt,8a2b9543a7267ced6831ca9e368e2149,result
uintt,be3b43d7f4a43891e0385f2eca2cf07f,shift
uintt,7da512f0ba552d363c04f7a91579d848,63
shift,e8b80f42a57ff812a6cf44feaef78935,63
uintt,be3b43d7f4a43891e0385f2eca2cf07f,temp
uintt,be3b43d7f4a43891e0385f2eca2cf07f,val
uintt,3c106a8b913930580b4bacdb0fa176ee,uintt
uintt,157679f80996bc7a15c5fad0dedbf11d,val
shift,06f1bd4eabc1f73da38b772edea29dca,uintt
```

Figure 2: Example of a vulnerable function and the first 13 path-contexts extracted by `astminer`. For each function, `astminer` extracts a maximum of 200 path-contexts. This example only presents 13 path-contexts due to space limitations. The syntactic paths between tokens are encoded using `MD5`.

61.43%, comparable to simple transformer models such as pre-trained `RoBERTa` [18]. We show that these results are achievable at a fraction of the computational resources and training time necessary for transformer-based models; `Code2vec` ran a full training session in approximately 5 minutes on a consumer-grade GPU, handling 1024 samples of data at each training step without exhausting its memory. We submitted our results to the `CodeXGLUE` leaderboard[4] for comparison with the state-of-the-art in the defect detection task.

Our contributions are as follows:

- An evaluation of `Code2vec` on the task of vulnerability detection using a dataset of labeled `C` functions both regarding accuracy and computational requirements (training time and memory).
- A replication package with the scripts and data to train and test the model, for reproducibility. Available online: `https://github.com/dcoimbra/dx2021`.

The paper is organized as follows: in section 2, we present our approach for applying `Code2vec` to vulnerability detection. In section 3, we describe the implementation details related to our extraction of path-contexts and `Code2vec` configuration. In section 4, we describe the evaluation metrics employed and present our results alongside previous studies, and discuss them in section 5. In section 6, we give a brief summary of the related work in deep learning for vulnerability detection. Finally, section 7 presents our conclusions and lays discusses future work.

[4]Microsoft's CodeXGLUE Leaderboard Website: `https://microsoft.github.io/CodeXGLUE/`. Our results for the `code2vec` model are presented in the Defect Detection (Code-Code) table. (Accessed August 11, 2021)

	Vulnerable	Non-Vulnerable
Train	10018	11836
Test	1255	1477
Validation	1187	1545

Table 1: Distribution of vulnerable and non-vulnerable functions in `Devign`.

2 Approach

In this section, we describe our approach for using path-context representations for the detection of security vulnerabilities using `Code2vec`. We describe our procedure for the extraction of path-contexts from a corpus of labeled functions and provide a summary of `Code2vec` and how we adapted it for the vulnerability detection task.

2.1 Dataset

We used the public dataset `Devign`[5] [19], provided as part of the `CodeXGLUE` benchmark. `Devign` includes 27318 manually-labeled functions collected from `QEMU` and `FFmpeg`, two large `C` open-source projects. These functions were extracted by collecting security-related commits and selecting vulnerable and non-vulnerable versions of functions from the labeled commits. Each function was manually labeled by a group of three professional security experts. Functions are labeled as "'vuln"' and "'safe"', with

[5]Devign Dataset Website: `https://sites.google.com/view/devign` (Accessed August 11, 2021)

no distinction made with regard to the type of vulnerability. Examples of a non-vulnerable and vulnerable functions extracted from `Devign` are displayed in Figures 1 and 2 respectively, along with the first 13 path-contexts extracted from each function using `astminer`. For each function, `astminer` extracts a maximum of 200 path-contexts per function. We only present 13 of the path-contexts due to space limitations. We used the dataset splits as prepared by `CodeXGLUE`[6]: 80%/10%/10% for training, validation, and testing, respectively. The distribution of vulnerable and non-vulnerable functions for the dataset is presented in Table 1.

2.2 Representing code snippets as bags of path-contexts

The path-attention model on which `Code2vec` is built takes a source code function as represented by a set of *path-contexts*, extracted from its abstract syntax tree (AST). We describe the definitions of AST, AST path and path-context as presented in the `Code2vec` paper:

Definition 2.1 (Abstract Syntax Tree)**.** An abstract syntax tree (AST) for a code snippet $\mathcal{C}$ is a tuple $< N, T, X, s, \delta, \phi >$ where N is a set of non-terminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)^*$ is a function that maps a non-terminal node to a list of its children, and $\phi : T \rightarrow X$ is a function that maps a terminal node to its associated value. Every node except the root appears exactly once in the lists of children; that is, each node has exactly one parent.

Definition 2.2 (AST path)**.** An AST path of length k is a sequence of the form: $n_1 d_1 \ldots n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminal nodes, $\forall i \in [2..k] : n_i \in N$ are non-terminal nodes and $\forall i \in [1..k] : d_i \in \{\uparrow, \downarrow\}$ are movement directions (up or down in the tree). If $d_i = \uparrow$, then $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then $n_{i+1} \in \delta(n_i)$.

Definition 2.3 (Path-context)**.** Given an AST path p, a path-context is a triplet $< x_s, p, x_t >$ where p is a syntactic path in the AST and x_s and x_t correspond to the values associated with the start and end terminals of p, respectively. A possible path-context for the statement `x = 7` would be:

$$<\texttt{x},(NameExpr\uparrow AssignExpr\downarrow IntegerLiteralExpr),\texttt{7}>$$

It is possible to limit the paths to a maximum length and a maximum width. The maximum width of a path-context is the maximum distance between sibling nodes that are part of the same path. A code snippet $\mathcal{C}$ is represented as a *bag of path-contexts* consisting of path-contexts extracted from the AST for $\mathcal{C}$. We kept the `Code2vec` defaults of maximum length 8 and maximum width 3 and, for each function in the corpus, extracted a bag of at most 200 path-contexts. These values were empirically determined by the `Code2vec` authors [16].

2.3 The Code2vec model

`Code2vec` learns embedding matrices for paths, values and labels ($\boldsymbol{path\_vocab}$, $\boldsymbol{value\_vocab}$, $\boldsymbol{tags\_vocab}$, respectively), a fully-connected layer, and an attention vector $\boldsymbol{a}$. An illustration of the `Code2vec` architecture is shown in Figure 3. An embedding for a single path-context $b_i =< x_s, p_j, x_t >$ is a *context vector* $\boldsymbol{c_i}$ which corresponds to the concatenation of the embeddings of the start and end tokens and of their connecting paths:

$$\boldsymbol{c_i} = \text{embedding}(< x_s, p_j, x_t >) = [\boldsymbol{value\_vocab}_s ; \boldsymbol{path\_vocab}_j ; \boldsymbol{value\_vocab}_t] \in \mathbb{R}^{3d} \quad (1)$$

In the previous equation, the operator ; is the concatenation operator and d is an empirically-determined hyperparameter defining the length of the internal representation.

A fully connected layer of `Code2vec` learns to combine each component of the embedding of a path-context, through a simple linear combination with a learned weights matrix $\boldsymbol{W}$ passed through a hyperbolic tangent function:

$$\tilde{\boldsymbol{c_i}} = \tanh(\boldsymbol{W} \cdot \boldsymbol{c_i}) \in \mathbb{R}^d \quad (2)$$

In the previous equation, $\boldsymbol{W} \in \mathbb{R}^{3d \times d}$. Finally, `Code2vec`'s attention mechanism aggregates all combined context vectors $\{\tilde{\boldsymbol{c_1}}, \ldots, \tilde{\boldsymbol{c_n}}\}$ into a single representation. The attention weight α_i of each $\tilde{\boldsymbol{c_i}}$ is obtained through the normalized inner product between $\tilde{\boldsymbol{c_i}}$ and the global attention vector $\boldsymbol{a}$.

$$\alpha_i = \frac{\exp(\tilde{\boldsymbol{c_i}}^\intercal \cdot \boldsymbol{a})}{\sum_{j=1}^{n} \exp(\tilde{\boldsymbol{c_j}}^\intercal \cdot \boldsymbol{a})} \quad (3)$$

The final code vector $\boldsymbol{v}$ is a weighted average of the combined context vectors factored by their attention weights:

$$\boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \cdot \tilde{\boldsymbol{c_i}} \quad (4)$$

For prediction, the probability that a specific label y_i is assigned to a code snippet $\mathcal{C}$ is the normalized inner product between the embedding for y_i and the code vector $\boldsymbol{v}$:

$$\forall y_i \in Y : q(y_i) = \frac{\exp(\boldsymbol{v}^\intercal \cdot \boldsymbol{tags\_vocab}_i)}{\sum_{y_j \in Y} \exp(\boldsymbol{v}^\intercal \cdot \boldsymbol{tags\_vocab}_j)} \quad (5)$$

In the previous equation, Y is the set of label values found in the training corpus. Training is performed by minimizing cross-entropy loss using the Adam optimization algorithm. For inference, we take the target label to which `Code2vec` assigned the highest probability.

3 Implementation

In this section, we describe the implementation details for our approach. We describe the open-source library `astminer`, which we leveraged for extracting bags of path-contexts, as well as the parameters used for extraction and training. Our final pipeline is illustrated in Figure 4.

3.1 Extracting bags of path-contexts

To extract a set of path-contexts for each code snippet in `Devign`, we use the open-source library astminer[7] [20]. We wrote a custom script that visits each function in the corpus, and computes its path-contexts. Following `Code2vec` defaults, we limit the maximum length and width of each path-context to 8 and 3 respectively, and extract at most 200 path-contexts per function.

[6]Microsoft's Devign Splits for Defect Detection (CodeCode): `https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection` (Accessed August 11, 2021)

[7]Astminer GitHub Repository: `https://github.com/JetBrains-Research/astminer` (Accessed August 11, 2021)

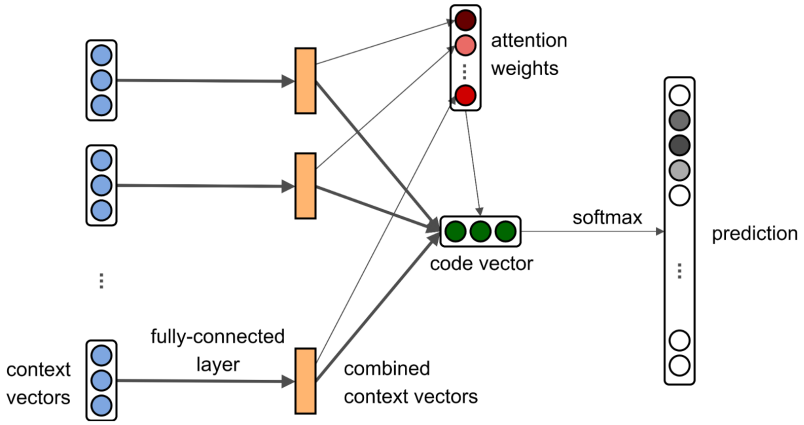

Figure 3: `Code2vec` architecture. Adapted from original image by [16].

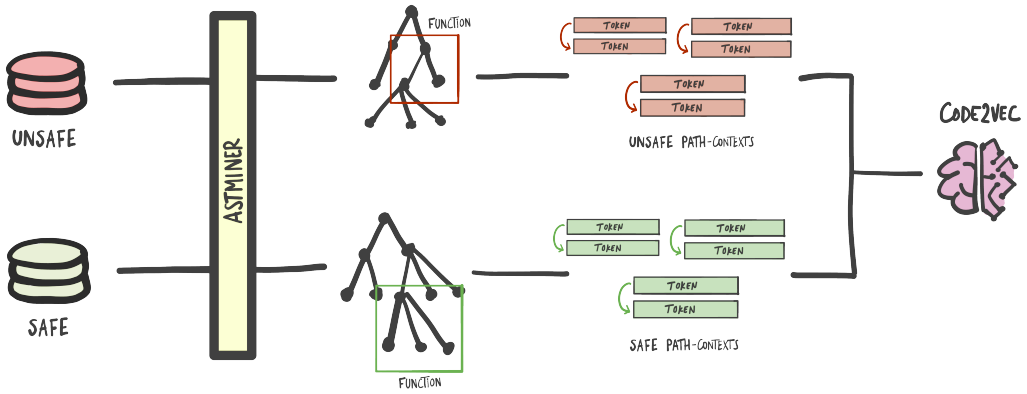

Figure 4: Experiments pipeline. The vulnerable and non-vulnerable raw source code functions are independently passed to `astminer`. For each function, `astminer` extracts the respective AST, from which it computes an appropriate bag of path-contexts. After labeling each bag of path-contexts corresponding to each function, the result is passed to `Code2vec` for training. The trained model is then used for inference.

	Vulnerable	Non-Vulnerable
Train	9987	11809
Test	1253	1472
Validation	1185	1541

Table 2: Distribution of vulnerable and non-vulnerable functions in `Devign` after applying `astminer`.

By default, `astminer` replaces each value and path in a path-context with a corresponding ID number in order to reduce memory consumption and training time when passing the samples to `Code2vec`. This is implemented by maintaining tables of $<$ id, value $>$ pairs for tokens, node types, and paths throughout the entire run. However, this was considerably memory-intensive on the machine we used. As such, we bypassed this feature and directly extracted path-contexts in their original format. We computed the `MD5` hash of the string representation of the path component of each path-context instead of extracting the entire path as is. This process is identical to the one used in the `Code2vec` paper for function name prediction [16].

The `astminer` library was unable to extract path-contexts from 71 functions of the `Devign` dataset. Therefore, these samples were not included in our study. The final distribution of functions after applying `astminer` is described in Table 2: 9987 vulnerable functions and 11809 non-vulnerable functions for the training dataset; 1253 vulnerable functions and 1473 non-vulnerable functions for the testing dataset; and, finally, 1185 vulnerable functions and 1541 non-vulnerable functions for the validation dataset.

3.2 Code2vec

We used the official implementation of `Code2vec`[8]. We trained for 20 epochs and performed inference on the epoch

[8]Code2vec GitHub Repository: `https://github.com/tech-srl/code2vec` (Accessed August 11, 2021)

Model	Accuracy	Precision	Recall	F1
CoTexT	66.62	-	-	-
C-BERT	65.45	-	-	-
PLBART	63.18	-	-	-
CodeBERT	62.08	-	-	-
Code2vec	**61.43**	**57.50**	**61.77**	**59.56**
RoBERTa	61.05	-	-	-
TextCNN	60.69	-	-	-
BiLSTM	59.37	-	-	-

Table 3: Results for the `Devign` dataset alongside the `CodeXGLUE` leaderboard. Our contributions are in bold.

Model	Train Time	#Epochs	Hardware
Code2vec	**5 minutes**	**20**	**1050Ti x1**
CodeBERT	**7 hours**	**5**	**1050Ti x1**
CodeBERT	1 hour	5	Tesla P100 x2

Table 4: Training time on the `Devign` dataset alongside the `CodeXGLUE` measurement, for a complete training session. Our contributions are in bold.

with the highest F1-score on the validation dataset. The performance measures of the model were adapted to the vulnerability detection task: we considered a prediction of "'safe"' to be a negative prediction, while a prediction of "'vuln"' to be a positive prediction. The training hyperparameters followed `Code2vec` defaults: batch size of 1024, embedding size of 128, and dropout rate of 0.25.

4 Evaluation

The `CodeXGLUE` benchmark for defect detection reports only accuracy as an evaluation metric. As `Devign` is a balanced dataset, accuracy is an appropriate metric for assessing performance. Nevertheless, we also computed precision, recall and F1-score in addition to accuracy. These metrics help us assess the model's ability to distinguish between vulnerable and non-vulnerable samples. Our results are shown in Table 3 alongside the current entries in the `CodeXGLUE` leaderboard. The results reported in this paper are different from the ones reported on the leaderboard due to the way the CodeXGLUE evaluated the model. We do not consider functions that do not generate path-contexts while CodeXGLUE does. In addition to these scores, we compared the training time and memory consumption for `Code2vec` and `CodeBERT` on this task on our hardware. We computed the training time for a complete training session for each model, which corresponds to 20 epochs on `Code2vec` and 5 epochs on `CodeBERT`. We chose 5 epochs for `CodeBERT` as this is the value used in `CodeXGLUE` to obtain the original results. Training times for each model are presented in Table 4 while memory consumption is shown in Table 5.

5 Results and Discussion

Based on accuracy alone, `Code2vec` outperformed the traditional NLP-based methods `BiLSTM` [21] ($61.43 > 59.37$) and `TextCNN` [22] ($61.43 > 60.69$). The obtained accuracy score for `Code2vec` was slightly higher than pre-trained `RoBERTa` ($61.43 > 61.05$): the two methods have similar performance. This is to be expected as these models were not designed for code intelligence tasks, nor pre-trained on source code data. In the same vein, `Code2vec` was outperformed by `PLBART` ($61.43 < 63.18$), `C-BERT` ($61.43 < 65.45$) and `CodeBERT` ($61.43 < 62.08$), which uses the transformer architecture to learn source-code features through pre-training on large amounts of source code data. As an advantage, transformer-based models do not require an intermediate representation and can be fine-tuned on the source code directly.

Model	#Params	Embedding size	Memory (MB)
Code2vec	31M	128	600
CodeBERT	125M	400	2484

Table 5: Parameter count and memory consumption for each model. Memory consumption is defined as the amount occupied in RAM by the model and a single sample of data during a training step, with all gradients loaded.

Regarding training time, as shown in Table 4, `Code2vec` completed a 20-epoch training session in approximately 5 minutes on our hardware. Conversely, on the same hardware, `CodeBERT` completes a 5-epoch training session in approximately 7 hours. This is to be expected, as transformer-based models need a larger number of parameters, in turn requiring much more computational resources to process the large amounts of data. Additionally, transformer-based models create internal representations that occupy large amounts of memory, which typically are not available on consumer-grade hardware: therefore, training has to be carried out in very small batches of data, increasing the training time. A large advantage of `Code2vec` compared with transformer-based models is its relatively low memory footprint: as shown in Table 5, a single training step with `CodeBERT` requires approximately 2.5GB of memory to complete one training step, an amount mostly represented by saved gradients during back-propagation. Conversely, a single training step with `Code2vec` was performed with just 600MB of GPU memory. This is due to `Code2vec`'s simpler architecture, allowing for a lower amount of gradients to be saved during training, as well as a lower number of parameters and smaller embedding sizes. Additionally, `Code2vec`'s lower memory footprint allows us to load larger batches of data into memory at each training step, increasing training performance.

6 Related Work

Recent research on machine learning for security vulnerability detection has used both token-based and graph-based approaches [23–25].

Natural Language Processing: Token-based models consider the code as a sequence of tokens. Several models have been proposed using different neural network architectures such as Bidirectional Long-Short-Term Memory (BSLTM) [23], Convolutional Neural Networks (CNN) [25], and Recurrent Neural Networks (RNN). However, simple token-based models struggle to reason about the long sequences produced from transforming source code into token sequences. To help address this problem, newer approaches using code slices instead of the entire code sequences were proposed, see for example `VulDeePecker` [23] and `SySeVR` [24]. The hypothesis behind slicing is that different parts of the code are not

equally important for the model to learn vulnerability patterns. Therefore, these newer approaches consider only slices extracted from *interesting points* (e.g., API calls)—points considered important for vulnerability prediction. The rest of the code elements are ignored. Token-based approaches usually fail to maintain the dependencies between the tokens that are the root of the problem. Thus, learning those dependencies (or semantic relationships) is at best difficult and at worst impossible.

Program Analysis: Graph-based models incorporate syntactic and semantic dependencies between different code elements. Source code can be transformed to syntactic graphs (Abstract Syntax Trees) and semantic graphs (Control Flow Graphs, Data Flow Graphs, Program Dependency Graphs, and so on). `Devign` [19] uses Code Property Graphs (CPGs) to build a vulnerability detection model as proposed by Yamaguchi et al. [26]. Chakraborty et al. [27] also generate code property graphs from source code to consider the syntax and the semantics of the code. CPGs consider succinct information from the control-flow and data-flow graphs in addition to the AST and the program dependency graphs. Each of these elements offer additional context about the semantic structure of the code that may be relevant for vulnerability detection.

Source Code Representations: Both token and graph-based models suffer from vocabulary explosion—the number of possible identifiers (e.g., variables and function names) in the code can be infinite. Some approaches replace tokens with abstract names [23, 24]. Other techniques use word embedding tools (e.g., `word2vec`) to create vector representations of every token. For instance, `VulDeePecker` [23] and `SySeVR` [24] use `word2vec` to transform symbolic tokens into vectors. In contrast, `Devign` [19] uses `word2vec` to transform pure code tokens to real vectors. Alon et al. [16] proposed continuous distributed vector (or code embeddings) to represent code. `Code2vec` aggregates an arbitrary sized snippet of code into a fixed-size vector in a way that captures its semantics. Code functions are transformed in groups of path-contexts, where each path-context represents a semantic relationship between two code elements in a function.

Transformers for Source Code: `CodeBERT` [11] is a transformer-based model that represents snippets of source code in a distributed representation vector [8]. The non-sequential nature of the architecture of the transformer encoder, being based on a simple attention mechanism, is designed to address the problem of reasoning about long sequences: each token is processed in parallel throughout the model. `CodeBERT` was pre-trained on pairs of programming language and natural language data. A pre-trained model produces distributed representations that can be used in a variety of downstream tasks, on which the model itself can be further fine-tuned.

This study evaluated how `Code2vec`—a model that considers syntactic and semantic relationships in the code—fairs compared to other non-token-based and token-based models (specifically, `CodeBERT`) for vulnerability detection in `C/C++`.

7 Conclusions

We applied `Code2vec`, a model for distributed code representations using AST path-contexts, to the task of binary vulnerability detection. We evaluated `Code2vec` on the `Devign` dataset as part of the `CodeXGLUE` benchmark. Our experiments achieved an accuracy score that outperformed naive NLP-based approaches and was equivalent to a simple transformer-based model that had not been pre-trained on source code data. However, as expected, it was outperformed by more expressive models that directly leverage features unique to the source code. Additionally, we showed that smaller models such as `Code2vec` have modest computational resource requirements compared to other alternatives: when computational resources are scarce, the reduced training time requirement and memory consumption of `Code2vec` on a labeled dataset of source code functions may outweigh the loss in accuracy that results from its lower expressiveness.

The `Devign` dataset provides a balanced distribution between safe and unsafe samples, which is not representative of a real-world application, where the data would be heavily imbalanced due to the lack of unsafe samples. This study would benefit from applying these experiments on a more realistic sample distribution. Additionally, the performance of `Code2vec` for vulnerability detection could potentially be improved through hyperparameter tuning, for example, by optimally choosing the maximum length and width of path-contexts as well as the learning rate and batch size. We plan to investigate these improvements in future work.

Although `Code2vec` is limited compared to state-of-the-art, it remains an attractive choice for developers for code intelligence tasks such as vulnerability detection, as it demonstrates comparable performance and can be more easily integrated in CI pipelines than static analyzers or larger models.

Acknowledgments

This work was supported in part by Fundação para a Ciência e a Tecnologia (FCT) under Grants CMU/TIC/0064/2019 (a project funded by the Carnegie Mellon Portugal Program) and UIDB/50021/2020.

References

[1] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE Symposium on Security and Privacy*, pages 332–345, 2010.

[2] Juha Kuusela. Security testing in continuous integration processes. Master's thesis, Aalto University. School of Science, 2017.

[3] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 334–344, 2017.

[4] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, 2006.

[5] Paul Anderson. The use and limitations of static-analysis tools to improve software quality. *CrossTalk: The Journal of Defense Software Engineering*, 21(6):18–21, 2008.

[6] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani,

Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, and Paul M. Ellingwood. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[7] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10):1825–1848, 2020.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[10] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, and Daxin Jiang. CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[12] Luca Buratti, Saurabh Pujar, Mihaela Bornea, Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, and Yufan Zhuang. Exploring software naturalness through neural language models. *arXiv preprint arXiv:2006.12641*, 2020.

[13] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

[14] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, and Duyu Tang. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

[15] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. Low-memory neural network training: A technical report. *arXiv preprint arXiv:1904.10631*, 2019.

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

[17] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *SIGPLAN Not.*, 53(4):404–419, June 2018.

[18] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[19] 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*, volume 32. Curran Associates, Inc., 2019.

[20] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. PathMiner: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[21] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005. IJCNN 2005.

[22] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[23] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[24] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, page 1, 2021.

[25] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018.

[26] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.

[27] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *arXiv preprint arXiv:2009.07235*, 2020.