Poster: Path-Based Function Embeddings

Daniel DeFreez, Aditya V. Thakur, Cindy Rubio-González
University of California, Davis
{dcdefreez,avthakur,crubio}@ucdavis.edu

ABSTRACT

Identifying relationships among program elements, such as functions, is useful for program understanding, debugging, and analysis. We present `func2vec`, an algorithm that uses static traces to embed functions in a vector space such that related functions are close together, even if they are semantically and syntactically dissimilar. We present applications of `func2vec` that aid program comprehension.

CCS CONCEPTS

• **Software and its engineering → Software post-development issues**;

KEYWORDS

Embeddings; Program Comprehension; Systems Software

ACM Reference Format:
Daniel DeFreez, Aditya V. Thakur, Cindy Rubio-González. 2018. Poster: Path-Based Function Embeddings. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195042

1 INTRODUCTION

Apart from writing new code, software engineers spend a substantial amount of time understanding, evolving, and verifying existing software. *Program comprehension* [2] entails inferring a mental model of the relationships among various program elements. When available, documentation can aid comprehension, such as "See Also" sections that list other related functions, but this documentation is rarely available for low-level functions. For these functions, programming language features such as polymorphism and encapsulation can make explicit the relationships between functions, but in languages such as C that are lacking these features, such relationships remain implicit.

Identifying relationships among functions is challenging because related functions are often semantically different and syntactically dissimilar. For example, the functions `snd_atiixp_free` and `snd_intel8x0_free` in the `atiixp` and `intel8x0` Linux device drivers, respectively, are semantically different, but serve the same purpose in these device drivers. For this reason, techniques that identify syntactic and semantic code clones cannot be used [1, 7].

This paper presents `func2vec`, a technique that embeds each function in a vector space such related functions are in close proximity. Recent work computes such an embedding using abstract syntax trees or dynamic traces [4, 9]. We are the first to use static traces to learn an embedding that captures the hierarchical structure of programs, and the first to apply such a technique to large-scale, low-level code such as the Linux kernel.

2 TECHNICAL APPROACH

For a given vocabulary L of program functions, `func2vec` computes an embedding $\Phi : L \to \mathbb{R}^d$ that maps each program function $\ell \in L$ to a d-dimensional vector in $\mathbb{R}^d$. Distributed representations are often used in natural-language processing (NLP); an embedding is learned from a corpus of sentences so that words sharing common contexts are embedded near each other. To reuse NLP algorithms requires `func2vec` to generate a linearized representation of programs, viz. "sentences" over a given vocabulary, for which `func2vec` is the first to use static program paths for this purpose. Intuitively, if we see many program paths with a call to function `f2` after a call to function `f1`, and paths with a call to `f3` after a call to `f1`, then `f2` and `f3` should be embedded close to each other.

A naive approach of linearizing a program by generating a sentence using the instructions along every valid interprocedural path has the following disadvantages: using the entire instruction set would generate sentences with a very large vocabulary; there are too many program paths for this approach to be practical; and it does not capture the hierarchical structure of programs.

The design of `func2vec` addresses each of these disadvantages. The vocabulary size is reduced by `func2vec` through abstracting each program instruction. To address the path explosion problem, `func2vec` performs a random walk over the program. On encountering a function call, the random walk either outputs the function name itself, or decides to step into the function definition. This strategy is able to capture the hierarchical structure of programs: the context preceding the function call can be linked to either the function call itself or to the context in the body of the function being called. Figure 1 shows the three main components of `func2vec`.

Program Encoder. `func2vec` uses a labeled pushdown system (ℓ-PDS) to model the set of valid interprocedural paths in the program. An ℓ-PDS is a PDS [6] in which each rule is associated with a sequence of labels, and these labels are concatenated as the ℓ-PDS makes its transitions. We associate a unique label for each instruction category (e.g., `load`, `store`), `struct` type (the instruction loads

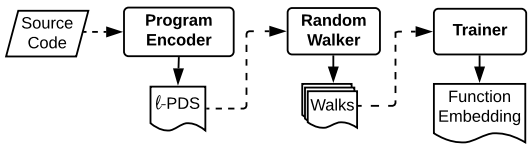
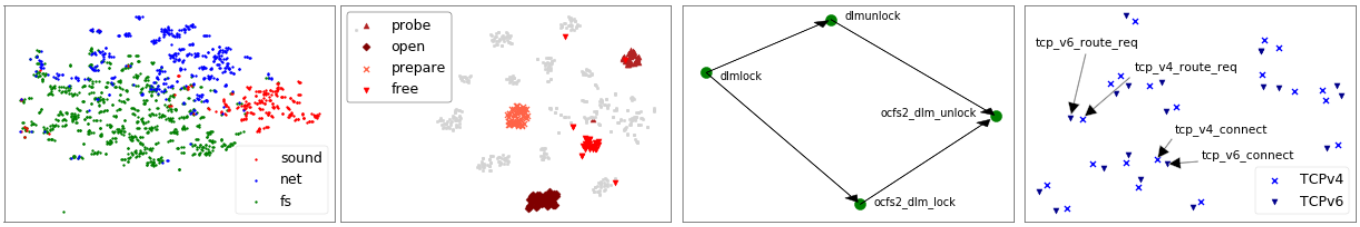


Figure 1: `func2vec` Architecture

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195042



(a) Three Linux Subsystems **(b) Sound Operation Clusters** **(c) DLM Analogy** **(d) TCPv4 / TCPv6 Alignment**

Figure 2: Visualization of `func2vec` relationships

or stores to a `struct` variable), and function (the instruction is a function call). We use a mostly standard way of encoding an interprocedural control-flow graph (ICFG) of a program as a PDS with one main difference: given a call to function `f` whose entry node is e_f on the ICFG edge $n_1 \to n_2$, we not only add the standard call rule $\langle p, n_1 \rangle \hookrightarrow \langle p, e_f\ n_2 \rangle$, but also an internal rule $\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$. This internal rule allows the random walker to either step over or step into the function call.

Random Walker. Given the ℓ-PDS for the program, the random walker generates γ paths of length at most k, starting at a call to each function ℓ. For a given start label ℓ, a walk is generated by randomly selecting a rule associated with label ℓ, and initializing the PDS configuration c. The current configuration c is updated by picking uniformly at random a next configuration in the ℓ-PDS. Note that labels are concatenated when the configuration is updated.

Model Trainer. Given the set of walks of the ℓ-PDS, `func2vec` uses a neural network to learn a vector representation for labels $\Phi : L \to \mathbb{R}^d$. Traditional language models try to estimate the probability of seeing a label ℓ_i given the context of the previous labels in the random walk; viz. $\Pr\left(\ell_i | \ell_1, \ell_2, \ldots, \ell_{i-1}\right)$. However, we also want to learn the distributed representation in the form of an embedding: $\Phi : L \to \mathbb{R}^d$. Thus, our problem is to estimate the likelihood: $\Pr\left(\ell_i | \Phi(\ell_1), \Phi(\ell_2), \ldots, \Phi(\ell_{i-1})\right)$. Mikolov et al. [3] introduce an unsupervised-learning technique that uses a single-layer fully-connected neural network to approximate this likelihood.

Implementation. The ℓ-PDS is constructed from LLVM IR. The implementation of `func2vec` uses the implementation of Mikolov et al. [3] provided by Gensim [5].

3 APPLICATIONS

The embedding computed by `func2vec` has many applications in program comprehension. We applied `func2vec` to a runnable Linux kernel with approximately 2 million LOC. To process the Linux kernel, `func2vec` requires approximately 20G of memory and two hours of compute time on Amazon EC2 R4 instances.

Subsystem Identification. Functions within subsystems are embedded closer to each other than functions between subsystems. Figure 2a shows a t-SNE projection of functions in three major subsystems: sound, networking, and file systems. File systems such as GFS2 that rely on networking are closer to the networking component than local-only file systems.

Identifying Function Synonyms. We define function *synonyms* to be functions that play the same role in different components. Function synonyms are close together in the `func2vec` embedding, forming clusters by role. Figure 2b shows such clusters of function synonyms in the PCI sound drivers. Function synonyms in Linux often follow a naming convention, such as `snd_via82xx_free` and `snd_cmipci_free` for the `via82xx` and `cmipci` sound drivers. However, function synonyms do have different names; `acpi_video_get_brightness` and `intel_panel_get_backlight` each return the brightness level of the backlight. Conversely, functions with similar names are not synonyms; `rcu_seq_start` adjusts the current sequence number, while `kprobe_seq_start` merely returns the current sequence number.

Analogical Reasoning. The relationship between OCFS2 Distributed Lock Manager (DLM) locking and unlocking is captured by the analogy `dlmlock : dlmunlock :: ocfs2_dlm_lock : ?`. Figure 2c shows a PCA plot of four functions belonging to the DLM. Similar analogies can be answered for other DLM locking and unlocking pairs, such as `dlmlock_remote` and `dlmunlock_remote`.

Alignment. The `func2vec` embedding can be used to match functions between related components. Figure 2d shows a t-SNE projection of TCPv4 and TCPv6 function pairs that have been matched via Procrustes alignment [8].

Specification Mining. Function embeddings can be used to improve the quality of mined specifications. Often there are not enough supporting examples for a specification within a single implementation, such as a single driver or file system. By merging specifications across multiple implementations, using function synonyms identified by `func2vec`, the support for a specification can be dramatically increased.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant CCF-1464439, and Amazon Web Services Cloud Credits for Research.

REFERENCES

[1] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable Detection of Semantic Clones. In *ICSE*. 321–330.
[2] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014), 31:1–31:37.
[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013).
[4] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *ICSE 2017*. 438–449.
[5] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Workshop on New Challenges for NLP Frameworks*. 45–50.
[6] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.* 58, 1-2 (2005), 206–263.
[7] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *SIGMOD*. 76–85.
[8] Peter H. Schönemann. 1966. A generalized solution of the orthogonal procrustes problem. *Psychometrika* 31, 1 (1966), 1–10.
[9] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair. In *ICLR*. To appear.