Path-Based Function Embedding and Its Application to Error-Handling Specification Mining

Daniel DeFreez
University of California, Davis
United States of America
dcdefreez@ucdavis.edu

Aditya V. Thakur
University of California, Davis
United States of America
avthakur@ucdavis.edu

Cindy Rubio-González
University of California, Davis
United States of America
crubio@ucdavis.edu

ABSTRACT

Identifying relationships among program elements is useful for program understanding, debugging, and analysis. One such kind of relationship is synonymy. Function synonyms are functions that play a similar role in code; examples include functions that perform initialization for different device drivers, and functions that implement different symmetric-key encryption schemes. Function synonyms are not necessarily semantically equivalent and can be syntactically dissimilar; consequently, approaches for identifying code clones or functional equivalence cannot be used to identify them. This paper presents Func2vec, a technique that learns an embedding mapping each function to a vector in a continuous vector space such that vectors for function synonyms are in close proximity. We compute the function embedding by training a neural network on sentences generated using random walks over the interprocedural control-flow graph. We show the effectiveness of Func2vec at identifying function synonyms in the Linux kernel. Finally, we apply Func2vec to the problem of mining error-handling specifications in Linux file systems and drivers. We show that the function synonyms identified by Func2vec result in error-handling specifications with high support.

CCS CONCEPTS

• **Computing methodologies → Unsupervised learning**; • **Software and its engineering → Automated static analysis**; **Error handling and recovery**;

KEYWORDS

program analysis, program embeddings, error handling, program comprehension, specification mining

ACM Reference Format:
Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-Based Function Embedding and Its Application to Error-Handling Specification Mining. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236059

1 INTRODUCTION

Apart from writing new code, a software engineer spends a substantial amount of time understanding, evolving, and verifying existing software. *Program comprehension* [16] entails inferring a mental model of the relationships among various program elements [15]. When available, documentation can aid program comprehension [10]. For instance, documentation about high-level API functions often contains a "See Also" section enabling the reader to navigate to relevant functions. However, such documentation is either unavailable for low-level code, or it is difficult to maintain as the code evolves [14]. Furthermore, languages such as C lack features such as polymorphism and encapsulation that make explicit the relationships between functions.

This paper deals with the problem of identifying *function synonyms*: functions that serve the same purpose or play a similar role in code. Identifying function synonyms is especially challenging in low-level code written in C, such as the Linux kernel, because (i) function synonyms are often semantically different and syntactically dissimilar; e.g., `snd_atiixp_free` and `snd_intel8x0_free` in the Linux PCI sound drivers `atiixp` and `intel8x0`, respectively; (ii) function synonyms need not always have similar names; e.g., `acpi_video_get_brightness` and `intel_panel_get_backlight` each return the brightness level of the backlight; and (iii) functions with similar names are not necessarily synonyms; e.g., `rcu_seq_start` adjusts the current sequence number, while `kprobe_seq_start` returns the current sequence number. Thus, techniques that identify syntactic and semantic code clones, check semantic equivalence, or rely on naming conventions [7, 11, 31] cannot be used to identify function synonyms.

This paper presents Func2vec, a technique that learns an embedding mapping each function to a vector in a continuous vector space such that vectors for function synonyms are in close proximity. Figure 1 illustrates the output of Func2vec for the PCI sound drivers; in particular, Func2vec maps each function to a vector in $\mathbb{R}^{300}$, which is then projected onto 2-dimensions using t-SNE [17]. Functions are grouped based on functionality (*probe*, *open*, *free*, etc.); e.g., `snd_atiixp_free` and `snd_intel8x0_free` both belong to the group labeled *free*. Functions within each group are synonymous, and are clustered together in the Func2vec embedding.

Func2vec is based on the insight that two functions are synonyms if the contexts in which they are called along program paths are similar. Specifically, Func2vec encodes the program as a labeled pushdown system (ℓ-PDS), where labels represent program elements such as function calls, instructions, types, and error codes. Random walks generated over the ℓ-PDS are used to learn a vector

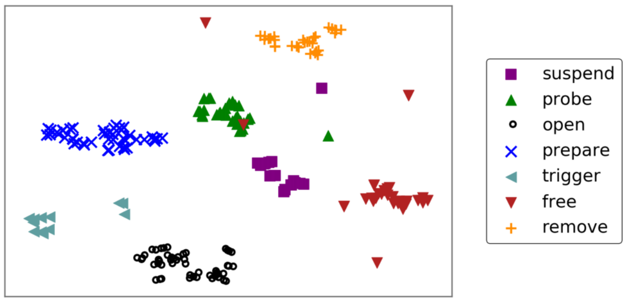


Figure 1: Function Synonym Clusters

embedding using a neural network [19]. *Func2vec is the first technique to use static interprocedural program traces to learn a function embedding that captures the hierarchical structure of programs.*

Using two data sets, we show that Func2vec is capable of accurately identifying function synonyms for a runnable Linux kernel (2 million LOC). *Func2vec is the first to learn a function embedding for large-scale low-level code such as the Linux kernel.*

As a case study, we apply Func2vec to the problem of *mining error-handling specifications* in the Linux kernel. An error-handling specification lists the functions that should be executed when an error occurs based on the previous functions called. Specification mining is a popular technique with many applications [1, 8, 30, 32]. The number of times a specification occurs in the code is called its *support*. To avoid reporting false specifications, only specifications above a large support threshold are reported. However, even for a code base as large as Linux, many true specifications have a low support [30]. Furthermore, error-handling specifications have an even lower support because error-handling code is less common.

The key insight behind mining high-support error-handling specifications is that several low-support specifications can be merged if they involve function synonyms. Our specification miner leverages Func2vec as well as multiple implementations of Linux file systems and drivers to obtain *merged* error-handling specifications that have high support; Section 2 presents an example.

The contributions of this paper are as follows:

- Func2vec, a technique for learning a function embedding that captures the hierarchical structure of programs (§ 3).
- An evaluation of the effectiveness of Func2vec for finding function synonyms in the Linux kernel (§ 4).
- A formulation of mining error-handling specifications for low-level systems code that uses a function embedding (§ 5).
- An evaluation of the usefulness of Func2vec for mining merged error-handling specifications across 10 Linux file systems, and 48 Linux device drivers (§ 6).

We describe related work in § 7, and conclude in § 8.

2 MOTIVATING EXAMPLE

This section illustrates how identifying function synonyms can be used to mine error-handling specifications in Linux PCI sound drivers. Figure 2 shows excerpts from the function `snd_atiixp_create` in driver `atiixp`, and `snd_intel8x0_create` in driver `intel8x0`. Note that the functions look almost identical only because we have not shown dissimilar code (36 LOC in `snd_atiixp_create` and 193 LOC in `snd_intel8x0_create`).

An error handler is code that is executed if an error occurs; in Linux, it corresponds to a conditional statement that checks for an error. Figure 2 shows error handlers for `snd_atiixp_create` and `snd_intel8x0_create` (marked `H1` through `H5`).

An error-handling specification imposes requirements on an error handler based on the set of functions that have been previously called. An error-handling specification is defined as $C \overset{e}{\Rightarrow} R$, where C is the *context* and R is the *response*. This specification states that if the set of functions in the context are successfully called and an error occurs, then the functions in the response must be called (usually to release resources acquired by functions in the context).

The specification for error handler `H2` in `snd_atiixp_create` (Figure 2a) is {`pci_enable_device`} $\overset{e}{\Rightarrow}$ {`pci_disable_device`}. The same specification is true for error handler `H2` in `snd_intel8x0_create` (Figure 2b). The *support* of the specification is 61 across *all* 48 Linux device drivers; i.e., there are 61 occurrences that follow this specification in the code.

The specification for error handler `H4` in `snd_atiixp_create` is {`pci_enable_device,kzalloc,pci_request_regions`} $\overset{e}{\Rightarrow}$ {`snd_atiixp_free`}. The support of the specification is only 4. The specification for the error handler `H4` in `snd_intel8x0_create` is {`pci_enable_device,kzalloc,pci_request_regions`} $\overset{e}{\Rightarrow}$ {`snd_intel8x0_free`}, which has the same context as the specification found in `atiixp` but has a different response. This specification has a support of 7. However, assuming a support threshold of 10, these two (true) specifications would not be reported by the specification miner due to their low support.

Func2vec reports that `snd_atiixp_free` and `snd_intel8x0_free` are function synonyms. Thus, the above specifications can be combined to form a *merged* error-handling specification. An error handler that supports any one of the specifications in the merge supports all of the specifications because the functions involved are synonyms. Consequently, merged specifications have support higher than those of the corresponding individual specifications. For example, when considering 48 device drivers, Func2vec finds 16 function synonyms for `snd_atiixp_free` and `snd_intel8x0_free`. Using this information, we find a merged specification with a support of 80. In Linux, merged specifications often result from combining multiple, similar implementations such as multiple device drivers or multiple file systems. As can be seen, function synonyms are crucial for finding merged specifications with high support.

3 FUNC2VEC: PATH-BASED FUNCTION EMBEDDING

Func2vec maps a discrete set of functions to a continuous vector space; that is, given a vocabulary L of program functions, each program function $\ell \in L$ is mapped to a d-dimensional vector in $\mathbb{R}^d$. To accomplish this, Func2vec generates a linearized representation of programs, viz. "sentences" over a given vocabulary. Func2vec is the first to use static interprocedural program paths for this purpose. Intuitively, if many program paths have a call to function `f2` after a call to function `f1`, and a call to `f3` after a call to `f1`, then `f2` and `f3` should be embedded close to each other.

```
int snd_atiixp_create(struct snd_card *card){

 if ((err=pci_enable_device(pci)) < 0) //H1
   return err;

 chip = kzalloc(sizeof(*chip),...);
 if (chip == NULL) { //H2
   pci_disable_device(pci);
   return -ENOMEM;
  }

 if((err=pci_request_regions(pci)) < 0){ //H3
   pci_disable_device(pci);
   kfree(chip);
   return err;
 }
 chip->addr = pci_resource_start(pci, 0);

 if (request_irq(pci->irq,...)) { //H4
   dev_err(card->dev, "IRQ␣%d", pci->irq);
   snd_atiixp_free(chip);
   return -EBUSY;
 }

 if((err=snd_device_new(card,...) < 0)) { //H5
   snd_atiixp_free(chip);
   return err;
 }

}
```

(a) Function snd_atiixp_create in driver atiixp

```
int snd_intel8x0_create(struct snd_card *card){

 if ((err=pci_enable_device(pci)) < 0) //H1
   return err;

 chip = kzalloc(sizeof(*chip),...);
 if (chip == NULL) { //H2
   pci_disable_device(pci);
   return -ENOMEM;
 }

 if ((err=pci_request_regions(pci)) < 0){ //H3
   kfree(chip);
   pci_disable_device(pci);
   return err;
 }

 if ((err = snd_intel8x0_chip_init()) < 0) { //H4
   snd_intel8x0_free(chip);
   return err;
 }

 if (request_irq(pci->irq,...)) { //H5
   dev_err(card->dev, "IRQ␣%d", pci->irq);
   snd_intel8x0_free(chip);
   return -EBUSY;
 }

}
```

(b) Function snd_intel8x0_create in driver intel8x0

Figure 2: (a) An excerpt from the function `snd_atiixp_create` in the driver `atiixp` (sound/pci/atiixp.c). The function contains *three* error-handling specifications. Each specification consists of a context set (function calls highlighted in gray) and a response set (function calls in a box). The first specification is associated with error handler `H2` and has a 1-element context (highlighted in gray) and a 1-element response (in a box). The second specification is associated with error handler `H3` and has a 2-element context (highlighted in gray) and a 2-element response (in a box). The third specification is associated with handlers `H4` and `H5`. It has a 3-element context (highlighted in gray) and a 1-element context (in a box); (b) an excerpt from the function `snd_intel8x0_create` in the driver `intel8x0` (sound/pci/intel8x0.c) in which *three* error-handling specifications are also found. The specifications for the two functions differ only in the response for the third specification.

A naive approach for linearizing a program is to generate a sentence using the instructions along every valid interprocedural path in the program. Such an approach 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. Func2vec abstracts each program instruction to reduce the vocabulary. To address the path explosion problem, Func2vec performs a random walk of the program restricted to generate γ paths of length at most k starting at a call to each function. Lastly, on encountering a function call, the random walk either outputs the function name

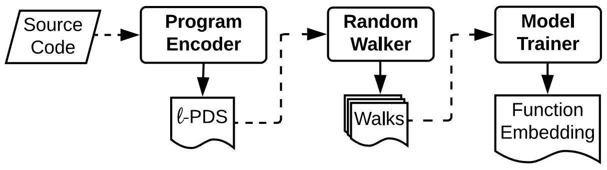


Figure 3: Func2vec Architecture

itself, or decides to step into the function definition. This strategy of the random walk 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 3 shows the three main components of Func2vec.

3.1 Program Encoder

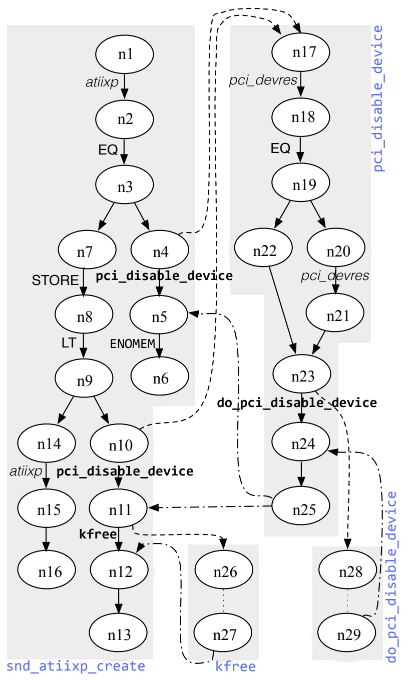
A pushdown system (PDS) is used to model the set of valid interprocedural paths in the program [26]. A PDS is defined as follows:

Definition 3.1. A *pushdown system* is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P and Γ are finite sets, *control locations* and *stack alphabet*, respectively. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ and $w \in \Gamma^*$. Δ contains a finite number of rules $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $w \in \Gamma^*$, which define a transition relation $\Rightarrow$ between configurations of $\mathcal{P}$ such that if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, then $\langle p, \gamma w' \rangle \Rightarrow \langle p', ww' \rangle$ for all $w' \in \Gamma^*$. □

$c \overset{r}{\Rightarrow} c'$ denotes that the rule $r \in \Delta$ was used to transition from configuration c to c' of $\mathcal{P}$. To model control flow of a program a

```
1   void pci_disable_device(struct
2       pci_dev *dev) {
3
4     struct pci_devres *dr = ...;
5     if (dr == NULL) {
6       dr->enabled = 0;
7     }
8     do_pci_disable_device(dev);
9   }
10
11  int snd_atiixp_create(...,
12      struct pci_dev *dev,...) {
13
14    struct atiixp *chip = ...;
15    if (chip == NULL) {
16      pci_disable_device(dev);
17      return -ENOMEM;
18    }
19    int err =...;
20    if (err < 0){
21      pci_disable_device(dev);
22      kfree(chip);
23      return err;
24    }
25    chip->addr = ...;
26    ...
27  }
```

(a) Simplified code from Figure 2a



(b) Graphical representation of ℓ-PDS

Rules for snd_atiixp_create:
(1) $\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle$: *atiixp*
(2) $\langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle$: EQ
(3) $\langle p, n_3\rangle \hookrightarrow \langle p, n_4\rangle$
(4) $\langle p, n_3\rangle \hookrightarrow \langle p, n_7\rangle$
(5) $\langle p, n_4\rangle \hookrightarrow \langle p, n_{17}\ n_5\rangle$
(6) $\langle p, n_4\rangle \hookrightarrow \langle p, n_5\rangle$: **pci_disable_device**
(7) $\langle p, n_5\rangle \hookrightarrow \langle p, n_6\rangle$: ENOMEM
(8) $\langle p, n_6\rangle \hookrightarrow \langle p, \epsilon\rangle$
(9) $\langle p, n_7\rangle \hookrightarrow \langle p, n_8\rangle$: STORE
(10) $\langle p, n_8\rangle \hookrightarrow \langle p, n_9\rangle$: LT
(11) $\langle p, n_9\rangle \hookrightarrow \langle p, n_{10}\rangle$
(12) $\langle p, n_9\rangle \hookrightarrow \langle p, n_{14}\rangle$
(13) $\langle p, n_{10}\rangle \hookrightarrow \langle p, n_{17}\ n_{11}\rangle$
(14) $\langle p, n_{10}\rangle \hookrightarrow \langle p, n_{11}\rangle$: **pci_disable_device**
(15) $\langle p, n_{11}\rangle \hookrightarrow \langle p, n_{26}\ n_{12}\rangle$
(16) $\langle p, n_{11}\rangle \hookrightarrow \langle p, n_{12}\rangle$: **kfree**
(17) $\langle p, n_{12}\rangle \hookrightarrow \langle p, n_{13}\rangle$
(18) $\langle p, n_{13}\rangle \hookrightarrow \langle p, \epsilon\rangle$
(19) $\langle p, n_{14}\rangle \hookrightarrow \langle p, n_{15}\rangle$: *atiixp*
(20) $\langle p, n_{15}\rangle \hookrightarrow \langle p, n_{16}\rangle$
(21) $\langle p, n_{16}\rangle \hookrightarrow \langle p, \epsilon\rangle$

Rules for pci_disable_device:
(22) $\langle p, n_{17}\rangle \hookrightarrow \langle p, n_{18}\rangle$: *pci_devres*
(23) $\langle p, n_{18}\rangle \hookrightarrow \langle p, n_{19}\rangle$: EQ
(24) $\langle p, n_{19}\rangle \hookrightarrow \langle p, n_{20}\rangle$
(25) $\langle p, n_{19}\rangle \hookrightarrow \langle p, n_{22}\rangle$
(26) $\langle p, n_{20}\rangle \hookrightarrow \langle p, n_{21}\rangle$: *pci_devres*
(27) $\langle p, n_{21}\rangle \hookrightarrow \langle p, n_{23}\rangle$
(28) $\langle p, n_{22}\rangle \hookrightarrow \langle p, n_{23}\rangle$
(29) $\langle p, n_{23}\rangle \hookrightarrow \langle p, n_{28}\ n_{24}\rangle$
(30) $\langle p, n_{23}\rangle \hookrightarrow \langle p, n_{24}\rangle$: **do_pci_disable_device**
(31) $\langle p, n_{24}\rangle \hookrightarrow \langle p, n_{25}\rangle$
(32) $\langle p, n_{25}\rangle \hookrightarrow \langle p, \epsilon\rangle$

(c) ℓ-PDS rules

Figure 4: Example of ℓ-PDS Encoding

single control location p, and the following three types of rules $r = \langle p, \gamma\rangle \hookrightarrow \langle p, w\rangle$ are sufficient: (i) *internal* rules with $|w| = 1$ that model intraprocedural flow; (ii) *push* rules with $|w| = 2$ that model function calls, and (iii) *pop* rules with $|w| = 0$ that model function returns.

A mostly standard way of encoding an interprocedural control-flow graph (ICFG) of a program as a PDS is used. The main difference is when a function call is encountered: given a call to function f whose entry node is e_f on the ICFG edge $n_1 \rightarrow n_2$, the PDS contains both the standard call rule $\langle p, n_1\rangle \hookrightarrow \langle p, e_f\ n_2\rangle$ as well as an internal rule $\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle$. This new internal rule is akin to a summary edge for the called procedure. The addition of this internal rule allows the random walk used by Func2vec (Algorithm 1) to either step over or step into the function call.

A labeled PDS (ℓ-PDS) is a PDS in which each rule is associated with a sequence of labels, and these labels are concatenated as the ℓ-PDS makes its transitions. More formally:

Definition 3.2. A *labeled pushdown system* (ℓ-PDS) is a triple $\mathcal{L} = (\mathcal{P}, L, f)$, where $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, L is a finite set of labels, and $f : \Delta \rightarrow L^*$ is a map that assigns a sequence of labels to each rule of $\mathcal{P}$. A *configuration* of $\mathcal{L}$ is a pair (c, l), where c is a configuration of the PDS $\mathcal{P}$ and $l \in L^*$. Δ and f define the transition relation $\Rightarrow_l$ between configurations of $\mathcal{L}$ such that if $c \overset{r}{\Rightarrow} c'$, then $(c, l) \Rightarrow_l (c', ll')$, where $f(r) = l'$. □

$c \overset{r}{\Rightarrow}_l c'$ denotes that the rule $r \in \Delta$ was used to transition from configuration c to c' of $\mathcal{L}$. In practice, labels are only attached to internal rules of the ℓ-PDS. A unique label is assigned to each instruction category, error code, `struct` type, and function. Each instruction is abstracted to a list of such labels as follows:

- Instructions are classified into categories such as LOAD, STORE, EQ, etc. The internal rule associated with a particular instruction is labeled with the corresponding instruction category.
- Systems code defines specific constants that are used as error codes; see §5. If such an error is used in the instruction, then we add the error-code label to the corresponding internal rule.
- If the instruction loads or stores to a `struct` variable, then we add the `struct`-type label to the corresponding internal rule.
- If the instruction is a function call, then the function label to the corresponding internal rule is added.

Example 3.3. Figure 4a shows simplified functions `pci_disable_device` and `snd_atiixp_create` (from § 2). Figure 4b shows a graphical representation of the corresponding ℓ-PDS. Figure 4c lists the ℓ-PDS rules; we use the notation $r : l$ to mean that $f(r) = l$ in the ℓ-PDS. The instruction-category labels used in this example are {EQ, STORE, LT} (for simplicity, we do not include labels LOAD and RET); the `struct`-type labels are {*atiixp*, *pci_devres*}, the error-code labels are {ENOMEM}, and the function labels are {**`pci_disable_device`**, **`kfree`**, **`do_pci_disable_device`**}.

We describe the first 7 rules for the function `snd_atiixp_create` in Figure 4c. The internal rule (1) corresponds to line 14 in Figure 4a, where the variable `chip` of type `struct atiixp` is assigned. Thus, the rule is labeled with `struct`-type label *atiixp*. The internal rule (2) corresponds to the equality expression on line 15, and is attached the instruction label EQ. Unlabeled rules (3)

Algorithm 1: RandomWalk($\mathcal{L}$, ℓ, k)

```
Input: ℓ-PDS $\mathcal{L} = (P, L, f)$, start label $\ell$, walk length $k$
Output: walk = $\{\ell_1, \ldots, \ell_n\}$
1 $\langle p, n\rangle \hookrightarrow \langle p, n'\rangle : l \leftarrow$ Random($\{r : l | r \in \Delta$ and $\ell \in f(r)\}$)
2 $c \leftarrow (\langle p, n'\rangle, l)$
3 for $n \leftarrow 0$ to $k$ do
4     $c \leftarrow$ Random($\{c' | c \overset{r}{\Rightarrow}_l c'$ for some $r \in \Delta\}$)
5 return $Labels(c)$
```

Algorithm 2: Func2vec ($\mathcal{L}$, w, d, γ, k)

```
Input: ℓ-PDS $\mathcal{L} = (P, L, f)$, window size $w$, embedding size $d$,
       walks per label $\gamma$, walk length $k$
Output: Vector representation for labels $\Phi : L \rightarrow \mathbb{R}^d$
1 $W \leftarrow \emptyset$
2 for $i \leftarrow 0$ to $\gamma$ do
3     foreach $\ell \in L$ do
4         $W \leftarrow W \cup$ RandomWalk($\mathcal{L}, \ell, k$)
5 $\Phi \leftarrow$ TrainModel($W, d, w$)
```

and (4) correspond to the `true` and `false` branches of the conditional on line 15. Call rule (5) and internal rule (6) are associated with the function call `pci_disable_device` on line 16. Note that the call rule is not labeled; the internal rule has a function label **`pci_disable_device`**. Finally, rule (7) is given the error-code label `ENOMEM`, and corresponds to the return statement on line 17. ∎

3.2 Random Walker

Algorithm 1 shows how to generate a random walk of an ℓ-PDS. Given a set S, `Random`(S) returns an element $s \in S$ that is chosen uniformly at random. $Labels(\cdot)$ returns the sequence of labels associated with an ℓ-PDS configuration. Given an ℓ-PDS $\mathcal{L}$, a start label ℓ, and a walk length k, a random walk is generated as follows. A rule associated with label ℓ is chosen at random (line 1), and the configuration c is initialized (line 2). Then, in the loop at line 4 the current configuration c is updated by choosing uniformly at random a next configuration in the ℓ-PDS. Note that in Definition 3.2, labels are concatenated when the configuration is updated.

Example 3.4. Consider label ℓ = *atiixp*, and walk length $k = 10$. There are two rules associated with *atiixp*: rules (1) and (19) from Figure 4c. Assume rule (1) was chosen and the random walk starts at rule (1) with label *atiixp*. Two possible random walks would be:
$W_1 \overset{\text{def}}{=}$ *atiixp* EQ **`pci_disable_device`** `ENOMEM`
$W_2 \overset{\text{def}}{=}$ *atiixp* EQ *pci_devres* EQ *pci_devres* **`do_pci_disable_device`**.
During walk W_1 internal rule (6) was chosen, while walk W_2 stepped into the function call by choosing the call rule (5). ∎

3.3 Model Trainer

Given an ℓ-PDS $\mathcal{L}$, a window w, a dimension d, a number of walks per label γ, and a walk length k, Func2vec (Algorithm 2) generates γ walks for each label in L, and uses them to train the model. The result is a vector representation for labels $\Phi : L \rightarrow \mathbb{R}^d$.

`TrainModel` on line 5 uses a neural network to learn Φ. 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. $\Pr\left(\ell_i | \ell_1, \ell_2, \ldots, \ell_{i-1}\right)$. Apart from learning the probability distribution of label co-occurrences, we also want to learn the embedding: $\Phi : L \rightarrow \mathbb{R}^d$. 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. [19] introduce a technique that uses a single-layer fully-connected neural network to approximate this likelihood. It uses a context of size w both before and after the given word, and considers the context as a set ignoring the ordering constraint. This results in the the following optimization problem for computing Φ: $\text{maximize}_\Phi \log \Pr \left(\ell_i | \{\Phi(\ell_{i-w}), \ldots, \Phi(\ell_{i-1}), \Phi(\ell_{i+1}), \ldots, \Phi(\ell_{i+w})\}\right)$. The implementation of Func2vec uses the implementation of Mikolov et al. [19] provided in Gensim [25].

4 FUNC2VEC EVALUATION

The experiments in this section are designed to answer the following research questions:

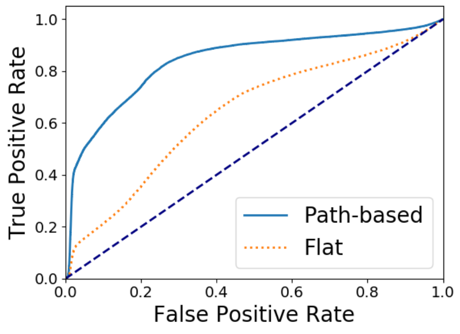
RQ1. How effective is Func2vec at identifying function synonyms?
RQ2. What impact do path-based sentences have?

Func2vec is evaluated against a runnable Linux kernel with all file systems and PCI sound drivers included, roughly 2 million LOC. The resulting ℓ-PDS consists of 5,407,483 stack locations (nodes), 6,083,632 PDS rules, and 77,194 labels. For this experiment the number of walks generated per label γ was 100, the walk length k was 100, the vector dimension d was 300, and the window size w was 10. Setting the window size w to 10 indicates a window of 10+10, where 10 labels in either the forward or backward direction are within the sliding window. These parameters were chosen via grid search over a small training set that is disjoint from the gold standard below. To process the Linux kernel, Func2vec requires approximately 24G of memory and two hours of compute time on Amazon EC2 R4 instances. These instances use Intel Xeon E5-2686 v4 (Broadwell) Processors and DDR4 Memory.

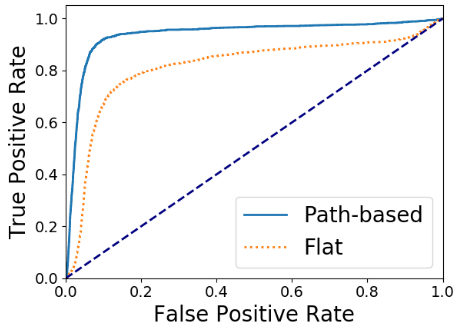
Gold Standard. We constructed two gold standard data sets to evaluate Func2vec. The first data set, GS1, consists of 2,652 unique functions that were identified by hand and reviewed with the help of a Linux kernel developer. These functions form 265 equivalence classes of function synonyms, ranging in size from two to 94. Note that the functions in GS1 cross components, can violate naming conventions, and are generally the hardest class of synonyms to find. Specifically, GS1 includes 6,079 function pairs that are synonyms, but do not follow any known naming scheme, e.g., `nlmsg_trim` and `genlmsg_cancel`, and 2,279 functions pairs that are *not* synonyms, but do follow a naming scheme, e.g., `sr_release` and `sd_release`.

We constructed a second data set GS2 by relying on a single strong naming convention. In Linux it is often the case that two different functions are created that perform essentially the same task, and one of the functions has underscores prepended to the name. GS2 consists of the 2,017 pairs of functions of this form. As a sanity check, a random sample of 50 pairs was selected and all 50 were judged to be function synonyms.

Evaluation Metrics. Two different metrics are used to evaluate Func2vec. The first metric is the area under the Receiver Operating



(a) Gold Standard GS1



(b) Gold Standard GS2

Figure 5: Receiver Operating Characteristic (ROC) Curves

Characteristic (ROC) curve for pairs of functions in our gold standard, when those pairs are sorted by their cosine similarity. This evaluates the claim that *function synonyms are more similar to each other than to unrelated functions.* For the second metric, the output of Func2vec is clustered using K-Means clustering, and the results are compared with the gold standard by computing the F-score.

Path-Based vs. Flat Comparison. For each metric, the performance of the path-based approach used by Func2vec described in § 3 is compared with a "flat" walker. The flat walker uses an identical ℓ-PDS, but lists the labels in the order that they appear in a function instead of generating path-based sentences. All other parameters and inputs are identical to the path-based version.

Metric 1: ROC Curves. Receiver Operating Characteristic (ROC) curves plot the true positive rate against the false positive rate as a parameter varies. They are commonly used to measure the accuracy of a signal detector, as they are well suited for needle-in-a-haystack scenarios where true positives are expected to be relatively rare [18]. That is the case here, where any two function pairs are unlikely to be synonyms of each other. For the evaluation, a much larger number of not-synonym pairs are used than synonym pairs, to mimic this expected distribution. The accuracy of a detector can be visually gauged by how bowed the curve is. Note that small changes in the area under the curve can hide large differences.

The ROC curves in Figures 5a and 5b are generated by sorting the similarities in the two gold standard data sets, respectively, and then varying the rank cutoff, an approach used in Lau and Baldwin [13]. In essence, function pairs in the gold standard data sets are sorted and each pair is counted as either a true positive or a false positive based on its classification in the gold standard, with the true positive and false positive rates recalculated after each pair.

Table 1: Func2vec Metrics

Gold Standard Test	Metric	Path-based	Flat
GS1: Manually reviewed	AUROC	0.72	0.56
GS2: Underscore names	AUROC	0.81	0.67
GS1: Manually reviewed	Cluster F1	0.73	0.56
GS2: Underscore names	Cluster F1	0.59	0.52

Results. Figure 5a shows the ROC curve produced when Func2vec is evaluated against our gold standard data set of manually identified function synonyms (GS1). Figure 5b shows the ROC curve produced when Func2vec is evaluated against our gold standard data set of function pairs that have identical names, one with prepended underscores (GS2). The area under each ROC curve (AUROC) is reported in Table 1. The overall accuracy of Func2vec is high, as evidenced by the sharp rise of the ROC curves, with Func2vec performing better on the easier underscore names than the manually inspected test set. This answers RQ1. For each of our test data sets, the path-based walker *significantly* outperforms the flat walker. This answers RQ2.

Metric 2: K-Means F-score. For the second metric, the Func2vec vectors are clustered with K-Means and the results are compared with the equivalence classes in the gold standard. For each cluster C_i and gold standard class L_j, the precision, recall, and F-score are defined as follows:

$$\text{Precision}(C_i, L_j) = \frac{|C_i \cap L_j|}{|C_i|} \qquad \text{Recall}(C_i, L_j) = \frac{|L_j \cap C_i|}{|L_j|}$$

$$F(C_i, L_j) = \frac{2 \times \text{Recall}(C_i, L_j) \times \text{Precision}(C_i, L_j)}{\text{Recall}(C_i, L_j) + \text{Precision}(C_i, L_j)}$$

To get an overall score that combines precision and recall, the F-score over all gold standard classes is used [3]. Since an imperfect cluster may partially overlap multiple gold standard classes we compute precision and recall scores for the product of gold standard classes and K-Means clusters. The precision and recall matrices are combined into a single F-score matrix, penalizing a cluster for either including extra functions or missing functions. The maximum F-score for each set of synonyms in the gold standard is used, creating a mapping between K-Means clusters and gold standard classes. The average F-score over all classes in the gold standard is reported here, weighted by the size of each gold standard class.

$$F = \sum_i \frac{L_i}{N} \max\{F(C_i, L_j)\}$$

Results. Our evaluation of Func2vec shows that it is capable of clustering function synonyms between functions in the Linux kernel with both high precision and recall. The F-score numbers are reported in Table 1. The F-score for the path-based walker is higher than that for the flat walker. The F-score for GS1 comports with the AUROC. However, the functions in GS2 do not cluster as well as

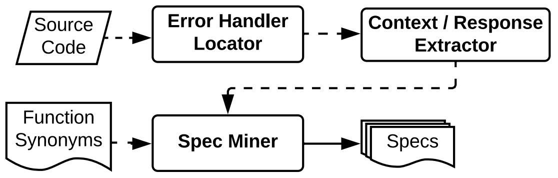


Figure 6: Specification Mining Architecture

those in GS1. After extensive testing, we believe this is a challenge of applying K-Means clustering, and that the AUROC is a more reliable metric here. This provides additional evidence for the answers to RQ1 and RQ2.

Threats to Validity. Identifying function synonyms is an inherently subjective task. We tried to mitigate this as much as possible by enlisting the assistance of a Linux kernel developer when constructing our gold standard. Multiple people performed overlapping classification independently, and any conflicting classifications were flagged and discussed. Nonetheless, it is not feasible to construct exhaustive ground truth for the Linux kernel, and therefore it is possible that Func2vec will perform differently for classes of function synonyms we are unaware of.

5 ERROR-HANDLING SPECIFICATIONS

To show the practical utility of identifying synonymous functions with Func2vec, we present a detailed case study exploring their effect on mining error-handling specifications. The major phases of the mining process are locating error handlers, extracting error handler contexts and responses, and frequent itemset mining. These phases are shown in Figure 6 and described in this section.

5.1 Defining Error-Handling Specifications

Error-handling specifications are mined based on the observation that the actions performed after an error occurs (the error-handler response) frequently depend on the actions carried out before the error occurred (the error-handler context). Such a context and response pair define an error-handling specification.

Error Handlers. An error handler is a piece of code that is executed upon detection of an error. Our evaluation targets the Linux kernel, which is written in C. Without explicit error-handling language constructs such as try/catch, locating error handlers in C code must rely on some amount of domain knowledge. We know that Linux defines a specific set of integer error constants, referred to as error codes. When returned from a function, these error codes denote that an error has occurred. This is known as the return-code idiom.

The *context* and *response* of an error handler consist of the functions called before and after an error is detected, respectively. In many cases, the order in which these functions are called does not matter. Once the predicate associated with the error handler is identified, the context and response are computed via traversing the code paths in the backward and forward directions, respectively.

Example 5.1. The function `snd_atiixp_create` in Figure 2a contains the five error handlers H1-H5; which are listed in Table 2 along with their corresponding contexts and responses. ■

```
1  static struct inode *btrfs_new_inode(...) {
2    // Most of the function is omitted
3    path = btrfs_alloc_path();
4    ...
5    btrfs_free_path(path);
6    ret = ...
7    if (ret) {
8      // btrfs_free_path not required
9    }
10 }
```

Figure 7: Code snippet from the `btrfs` file system. The path is freed prior to the error handler on line 7 is reached.

Error-Handling Specifications. An *error-handling specification* is defined as an association rule whose antecedent is the *specification context* and consequent is the *specification response*. This rule simply means that the set of function calls in the context implies that the set of function calls in the response are required to happen once an error is detected.

Definition 5.2. An *error-handling specification* S is defined as $C_S \overset{e}{\Rightarrow} R_S$, where $C_S = \{c_1, c_2, \ldots, c_m\}$ is the context set of function calls for the specification S, and $R_S = \{r_1, r_2, \ldots, r_m\}$ is the response set of function calls for the specification S. □

Example 5.3. The following error-handling specifications can be inferred for the atiixp sound driver:
$S_1 \overset{\text{def}}{=}$ {`pci_enable_device`} $\overset{e}{\Rightarrow}$ {`pci_disable_device`}
$S_2 \overset{\text{def}}{=}$ {`pci_enable_device, kzalloc`} $\overset{e}{\Rightarrow}$ {`pci_disable_device, kfree`}
$S_3 \overset{\text{def}}{=}$ {`pci_enable_device, kzalloc, pci_request_regions`} $\overset{e}{\Rightarrow}$ {`snd_atiixp_free`} ■

Definition 5.4. An error-handling specification $S \overset{\text{def}}{=} C_S \overset{e}{\Rightarrow} R_S$ is *applicable to* an error-handler H, denoted by $S \triangleright H$, iff $C_S \subseteq C_H$ and $C_S \cup R_S \not\subseteq C_H$, where C_H and R_H are the context and response of the error handler H, respectively. □

Example 5.5. Consider the error-handlers `H1`–`H5` in Ex. 5.1 and the specifications in Ex. 5.3; then $S_1 \triangleright H2$, $S_2 \triangleright H3$, $S_3 \triangleright H4$, and $S_3 \triangleright H5$. ■

The first term $C_S \subseteq C_H$ in Definition 5.4 says that the entire specification context must apply to the handler context. The second term $C_S \cup R_S \not\subseteq C_H$ is added for cases where the required response actions have already happened prior to a particular error handler being reached, as illustrated by the following example.

Example 5.6. Consider the specification $S \overset{\text{def}}{=}$ {`btrfs_alloc_path`} $\overset{e}{\Rightarrow}$ {`btrfs_free_path`}, and the code snippet from the `btrfs` file system, shown in Figure 7. The context C_H for the error handler H at line 7 is {`btrfs_alloc_path, btrfs_free_path`}. If Definition 5.4 is restricted to only contain the term $C_S \subseteq C_H$, then specification S is applicable to the error handler H. Clearly this is incorrect, as the path has been allocated and then freed prior to the error handler being reached. Consequently, without the second term in Definition 5.4, the error handler H would be flagged as a violation, even though the path has already been freed. ■

Definition 5.7. An error-handling specification $S \overset{\text{def}}{=} C_S \overset{e}{\Rightarrow} R_S$ is *satisfied by* an error-handler H, denoted by $S \blacktriangleright H$, iff $S \triangleright H$ and $R_S \subseteq R_H$, where R_H is the response of H. □

Table 2: Error Handlers in `snd_atiixp_create`

Error Handler	Lines	Function Generating Error	Context	Response
H1	1596	pci_enable_device	∅	∅
H2	1600-1601	kzalloc	{pci_enable_device}	{pci_disable_device}
H3	1610-1612	pci_request_regions	{pci_enable_device,kzalloc}	{pci_disable_device,kfree}
H4	1623-1625	request_irq	{pci_enable_device,kzalloc,pci_request_regions}	{dev_err,snd_atiixp_free}
H5	1633-1634	snd_device_new	{pci_enable_device,kzalloc,pci_request_regions,request_irq}	{snd_atiixp_free}

Given an error-handling specification and an error handler, the miner is able to report a violation of the specification using Definition 5.7.

Example 5.8. Consider the error-handlers in Ex. 5.1 and the specifications in Ex. 5.3; then $S_1 \blacktriangleright H2$, $S_2 \blacktriangleright H3$, $S_3 \blacktriangleright H4$, and $S_3 \blacktriangleright H5$. If the call to function `snd_atiixp_free` on line 1633 in Figure 2a was missing, then $S_3 \not\blacktriangleright H5$; i.e., S_3 would not be satisfied by H5. ■

The above definitions of error-handling specifications and satisfiability can be extended to handle *synonymous functions*. Let F be the set of functions in the program that is being mined. $\Pi : F \rightarrow F$ is a said to be a *partition function* iff $\Pi(f_1) = \Pi(f_2)$ for all functions f_1 and f_2 that are identical or synonyms. We abuse notation slightly by extending the partition function that applies to a single function to a set of functions: $\Pi(\{f_1, f_2, \ldots, f_n\}) = \{\Pi(f_1), \Pi(f_2), \ldots, \Pi(f_n)\}$. Similarly, given an error-handling specification $S \overset{\text{def}}{=} C_S \overset{e}{\Rightarrow} R_S$, we use $\Pi(S)$ to mean $\Pi(C_S) \overset{e}{\Rightarrow} \Pi(R_S)$.

Definition 5.9. Given a partition function Π, a set X of error-handling specifications is said to be a *merged* error-handling specification with respect to Π iff $\Pi(S) = \Pi(S')$ for all $S, S' \in X$. □

Example 5.10. Let the partition function Π be such that $\Pi($`snd_atiixp_free`$) = \Pi($`snd_intel8x0_free`$)$. Then {{`pci_enable_device, kzalloc, pci_request_regions`} $\overset{e}{\Rightarrow}$ {`snd_attixp_free`}, {`pci_enable_device, kzalloc, pci_request_regions`} $\overset{e}{\Rightarrow}$ {`snd_intel8x0_free`}} is a merged error-handling specification with respect to Π. ■

Using this notation, Definitions 5.4 and 5.7 can be naturally extended to merged error-handling specifications.

Definition 5.11. A merged error-handling specification X *is applicable to* an error-handler H, denoted by $X \triangleright H$, iff there exists $S \in X$ such that $S \triangleright H$. □

Definition 5.12. A merged error-handling specification X *is satisfied by* an error-handler H, denoted by $X \blacktriangleright H$, iff there exists $S \in X$ such that $S \blacktriangleright H$. □

5.2 Mining Error-Handling Specifications

Given a set of error handlers with their respective contexts and responses, *frequent itemset mining* is used to infer likely error-handling specifications. Frequent itemset mining algorithms take the minimum support threshold as a parameter, and return all sets of items that have a support greater than or equal to this threshold.

Let $\mathcal{T}$ be a set of transactions, where each transaction $T \in \mathcal{T}$ is a set of items. A frequent itemset mining algorithm returns the sets of items that frequently co-occur in the same transaction in $\mathcal{T}$.

Definition 5.13. The *support* of a set of items I given a set of transactions $\mathcal{T}$ is defined as $\text{supp}(I) \overset{\text{def}}{=} |\{T \in \mathcal{T} \mid I \subseteq T\}|$. □

To construct the merged specifications, frequent itemsets are mined at a low support threshold. After frequent itemset mining is performed, specifications that use function synonyms are merged together. All pairs of specifications with functions in either the context or response that exceed a similarity threshold are merged. Set intersection is performed over the remaining items. The transitive closure over these merged specification pairs is taken, yielding equivalence classes that form the final set of merged specifications.

The support for an unmerged specification is identical to its support as an itemset. However, the support of a merged specification is not simply the sum of the support of corresponding unmerged specifications, but the number of *unique* transactions (error handlers) that support the merged specification, which may be lower if the unmerged specifications share supporting error handlers.

6 SPECIFICATION MINING EVALUATION

This section is designed to answer the following question: what impact do function synonyms have on the quality of mined error-handling specifications?

We mine error-handling specifications in 48 drivers and 10 Linux file systems (`btrfs`, `ecryptfs`, `ext2`, `ext4`, `GFS2`, `HFS`, `HFS+`, `nilfs2`, `OCFS2`, and `UFS`). We chose Linux file systems and drivers because of the dire consequences of error handling defects, but the mining approach is general and can be applied to other parts of Linux, or to any C program that uses the return-code idiom for error handling. Frequent itemset mining is used to infer specifications in this evaluation, but function synonyms could be used to enhance other mining techniques.

Implementation Details. As shown in Figure 6, our miner takes as input (1) error handler context and response sets, and (2) synonymous function information. Our implementation relies extensively on the LLVM compiler infrastructure [12]. To locate error handlers, we use an existing LLVM-based error-propagation analysis [27–29], augmented with an LLVM pass that identifies handlers based on a list of functions that are known to be called only on error paths. Table 3 shows the total number of error handlers found. Function synonyms are identified by Func2vec. Eclat/LCM [4] is used to compute frequent itemsets.

Table 3: Number of Error Handlers

Impl.	# Handlers	Impl.	# Handlers
btrfs	1990	nilfs2	160
ecryptfs	230	OCFS2	2714
ext2	110	UFS	85
ext4	826	Shared (VFS)	887
GFS2	589	48 sound drivers	3173
HFS / HFS+	104	**Total**	**10868**

Table 4: Specification support improvement when using synonyms. The "Un." column lists the support for each specification without merging, and the "Mrg." column lists the support for each specification after merging.

Specification	Un.	Mrg.
{pci_enable_device, kzalloc, pci_request_regions} $\overset{e}{\Rightarrow}$ {snd_intel8x0_free}	7	80
{pci_enable_device, kzalloc, pci_request_regions} $\overset{e}{\Rightarrow}$ {snd_intel8x0m_free}	6	80
{pci_enable_device, kzalloc, pci_request_regions} $\overset{e}{\Rightarrow}$ {snd_atiixp_free}	4	80
{pci_enable_device, kzalloc, pci_request_regions} $\overset{e}{\Rightarrow}$ {snd_ice1712_free}	4	80
{pci_enable_device, kzalloc, pci_request_regions} $\overset{e}{\Rightarrow}$ {snd_via82xx_free}	6	80
{pci_enable_device, kzalloc} $\overset{e}{\Rightarrow}$ {snd_nm256_free}	3	124
{pci_enable_device, kzalloc} $\overset{e}{\Rightarrow}$ {azx_free}	2	124
{pci_enable_device, kzalloc} $\overset{e}{\Rightarrow}$ {snd_ali_free}	2	124
{pci_enable_device, kzalloc} $\overset{e}{\Rightarrow}$ {snd_emu10k1x_free}	3	124
{pci_enable_device, kzalloc} $\overset{e}{\Rightarrow}$ {snd_ca0106_free}	6	124
{btrfs_alloc_path} $\overset{e}{\Rightarrow}$ {btrfs_free_path}	487	541
{btrfs_alloc_path} $\overset{e}{\Rightarrow}$ {btrfs_release_path}	54	541
{gfs2_holder_init} $\overset{e}{\Rightarrow}$ {gfs2_holder_uninit}	33	74
{gfs2_glock_nq} $\overset{e}{\Rightarrow}$ {gfs2_glock_dq}	23	74
{gfs2_glock_nq_init} $\overset{e}{\Rightarrow}$ {gfs2_glock_dq_uninit}	28	74
{hfs_find_init} $\overset{e}{\Rightarrow}$ {hfs_find_exit}	17	49
{hfsplus_find_init} $\overset{e}{\Rightarrow}$ {hfsplus_find_exit}	32	49
{__kmalloc} $\overset{e}{\Rightarrow}$ {kfree}	187	353
{kzalloc} $\overset{e}{\Rightarrow}$ {kfree}	166	353

Results. We inspected all specifications with at least support 50 for the btrfs, ext4, GFS2, and OCFS2 file systems, support 20 for the ext2, ecryptfs, HFS, HFS+, nilfs2, and UFS file systems, support 10 for sound drivers, and support 200 for code shared by all implementations. Before merging, the sound driver results contained only 10 error-handling specifications above the support threshold, all of which were deemed correct by manual inspection. After merging, there were 31 specifications above the threshold, all 31 being correct. Before merging, the file system results contained 11 error-handling specifications above the support thresholds. After merging, there were 21 specifications above the thresholds, all correct.

```
1  static int gfs2_get_flags(...) {
2    ...
3    gfs2_holder_init(ip->i_gl, ...);
4    error = gfs2_glock_nq(&gh);
5    if (error)
6      // missing call to gfs2_holder_uninit
7      return error;
8    ...
9    gfs2_glock_dq(&gh);
10   gfs2_holder_uninit(&gh);
11   return error;
12 }
```

Figure 8: Bug found in GFS2. The function should not exit on line 7 without calling gfs2_holder_uninit.

Examples of Specifications. To further illustrate the impact on specification quality, Table 4 shows six examples of merged error-handling specifications. The first example shows a specification found across device drivers (see § 2). All 18 functions found in the response sets (e.g., snd_intel8x0_free, snd_via82xx_free, etc.) are function synonyms reported by Func2vec (for brevity, we only list 5 individual specifications). This results in a merged specification with a support of 80. Without synonyms, these individual specifications would not be reported at all because they fall below the support threshold of 10.

Not all of the sound driver specifications in Table 4 have the same form. The five specifications in the second row, starting with snd_nm256_free, are taken from a group of 8 merged specifications. Unlike the first example, these do not include pci_request_regions in the context, even though all of these free functions are synonyms and close together in the Func2vec embedding. The free functions in the second row do not call pci_release_regions, and therefore pci_request_regions does not appear in any of the contexts. Here, specification mining and Func2vec have worked in concert to arrive at the correct specifications.

The btrfs example in Table 4 illustrates merging specifications within a single implementation. btrfs_free_path is a synonym of the function btrfs_release_path, as the primary purpose of each is to drop references to a path in the file system. The difference is that btrfs_free_path also frees the memory, while callers of btrfs_release_path handle the memory operation separately. Merging these related specifications brings up the btrfs_release_path version so that they appear together in the mining report.

The specifications shown in the fourth row of Table 4 led to the discovery of two previously unknown bugs in the GFS2 file system. The patch we submitted to fix these bugs was accepted by Red Hat and merged into Linux version 4.7. Figure 8 shows one of the two bugs. The function first calls gfs2_holder_init, which acquires a reference to a glock. The function then attempts to enqueue this holder structure. On the normal path where gfs2_glock_nq succeeds, there is no problem. If gfs2_glock_nq fails on line 4, however, gfs2_holder_uninit is never called even though gfs2_holder_init completed successfully. As is common with error handling bugs, only in rare circumstances will this problem be encountered because it requires gfs2_glock_nq to fail. But when the bug is triggered the consequences are severe, resulting in an inaccurate reference count for the glock.

The fifth row in Table 4 shows a specification that crosses two related file systems, HFS and HFS+. The HFS and HFS+ file systems are implemented by Linux to interoperate with Mac OS. Other related file systems such as `ext2` and `UFS` also contain specifications that can be improved by merging with function synonyms.

Finally, the sixth row illustrates how function synonyms can be useful even for specifications that already have high support. The function `kzalloc` is similar to `__kmalloc` because they both allocate memory, but `kzalloc` zeroes the memory first. From an error-handling perspective they are equivalent.

7 RELATED WORK

Distributed Representations. Distributed representations have been extensively studied in natural language processing and cognition [9]. Recent advances have resulted in scalable approaches to computing such distributed representations (or vector embeddings) given a corpus of sentences; for instance, `word2vec` [19], and Glove [22]. DeepWalk [23] computes vector embeddings of *nodes* in a graph. DeepWalk is similar to Func2vec in that they both use random walks to generate a corpus of sentences. However, DeepWalk generates walks consisting of *nodes*, while Func2vec generates walks consisting of *labels* along edges. Func2vec also abstracts the program code into an ℓ-PDS. Ye et al. [36] apply vector representations to information retrieval in software engineering by using `word2vec` on documentation associated with code.

Nguyen et al. [21] computes distributed representations of API functions using `word2vec`. They generated sentences using the program AST, as opposed to interprocedural paths, and used their technique to migrate API usages from Java to C#. Similarly, Alon et al. [2] presents an approach for learning program representations using paths in the AST, which is then used to predict names for variables and methods in JavaScript, Java, Python, and C# programs. Wang et al. [33] learns program embeddings from program execution traces, and use the embeddings to classify errors in student programming assignments written in C#. DeepBugs [24] learns a `word2vec`-based embedding for each identifier in a JavaScript program using the sequence of tokens around the identifier. This embedding is used for name-based bug detection. In contrast to these approaches, we propose a technique to learn a function embedding from interprocedural static paths in the program, and use the embedding in mining error-handling specifications in Linux, which is written in C.

Error-Handling Specification Mining. One of the key developments in the error-handling specification mining literature has been the use of normal paths to mine specifications for error-handling paths. This line of thought was first mentioned in [35], and was subsequently used in several other papers [1, 8, 30, 32, 35].

Weimer and Necula [35] find association rules of the form $FC_a \Rightarrow FC_e$, where function call FC_a should be followed by call FC_e, and FC_e is found at least once in exception-handling code. Improving on [35], Thummalapenta and Xie [32] mine *conditional* association rules of the form $(FC_c^1 ... FC_c^n) \wedge FC_a \Rightarrow FC_e^1 ... FC_e^n$, which denotes a sequence of function calls prior to the target function FC_a that throws an exception, and then a sequence of recovery function calls. Acharya and Xie [1] mine error-handling specifications from interprocedural traces. Cleanup functions are identified from error traces, which are then used along with normal traces to find specifications. Goues and Weimer [8] broaden this notion of trace reliability to include a number of other features (e.g., execution frequency, cloning, code age, density, etc.), and significantly improve the false positive rate reported in [35]. Collectively, these approaches have been successful at finding defects in error-handling code that shares function calls with normal paths. But there exist functions that are only called on error paths, and that are only meaningful to Linux. Thus the correct use of these functions cannot be deduced from normal paths or outside programs, and they would be missed by the above approaches.

Much of the work on error-handling specifications has focused on languages with exception-handling support, such as Java or C++. Several approaches [5, 32, 34, 35] find error-handling specifications in Java programs using static analysis. Buse and Weimer [5] infer and characterize exception-causing conditions, which are then used as documentation. Weimer and Necula [34] use dataflow analysis to locate resource management mistakes in error-handling code and propose a language extension to improve reliability. A common mistake found might be the failure to release resources or to clean up properly along all paths. Identifying blocks of error-handling code in these languages is comparatively easy, but as we have shown, it is more challenging to distinguish between normal and error-handling paths in Linux.

Implementation Inconsistencies. Engler et al. [6] use the notion of internal consistency to find programming errors. One of their techniques for finding related pieces of code relied on the idiomatic use of function pointers to define multiple implementations of a single interface. Min et al. [20] compare multiple file systems by leveraging the VFS interface to identify implementations of the same functionality. These are complementary to our work.

8 CONCLUSION

We introduced the notion of *function synonyms*, which are functions that play a similar role in code. Synonymous functions might be syntactically dissimilar and might not be semantically equivalent. We presented Func2vec, a technique that learns an embedding mapping each function to a vector in a continuous vector space such that vectors for function synonyms are in close proximity. Specifically, Func2vec computes a function embedding by training a neural network on sentences generated using random walks of the interprocedural control-flow graph of the program. We showed the effectiveness and scalability of Func2vec by evaluating it against two gold standards of function synonyms in the Linux kernel.

We showed how Func2vec can improve the quality of error-handling specifications for 10 Linux file systems and 48 Linux device drivers. A challenge in mining error-handling specifications is that their support is often too low. To overcome this challenge, Func2vec was used to identify function synonyms across multiple implementations of file systems and device drivers. These function synonyms were then used to merge individual error-handling specifications, resulting in merged specifications with high support.

ACKNOWLEDGMENTS

This work was supported by NSF grant CCF-1464439, and AWS Cloud Credits for Research.

REFERENCES

[1] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009 (Lecture Notes in Computer Science)*, Marsha Chechik and Martin Wirsing (Eds.), Vol. 5503. Springer, 370–384. DOI:http://dx.doi.org/10.1007/978-3-642-00593-0_25

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 404–419. DOI:http://dx.doi.org/10.1145/3192366.3192412

[3] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. 2009. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Inf. Retr.* 12, 4 (2009), 461–486. DOI:http://dx.doi.org/10.1007/s10791-008-9066-8

[4] Christian Borgelt. 2016. Eclat/LCM - Frequent Item Set Mining. http://www.borgelt.net/eclat.html. (2016). Accessed: 2016-04-29.

[5] Raymond P. L. Buse and Westley Weimer. 2008. Automatic documentation inference for exceptions. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, Barbara G. Ryder and Andreas Zeller (Eds.). ACM, 273–282. DOI:http://dx.doi.org/10.1145/1390630.1390664

[6] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles, SOSP 2001, Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, Keith Marzullo and M. Satyanarayanan (Eds.). ACM, 57–72. DOI:http://dx.doi.org/10.1145/502034.502041

[7] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 321–330. DOI:http://dx.doi.org/10.1145/1368088.1368132

[8] Claire Le Goues and Westley Weimer. 2009. Specification Mining with Few False Positives. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009 (Lecture Notes in Computer Science)*, Stefan Kowalewski and Anna Philippou (Eds.), Vol. 5505. Springer, 292–306. DOI:http://dx.doi.org/10.1007/978-3-642-00768-2_26

[9] Geoffrey E Hinton, James L Mcclelland, and David E Rumelhart. 1986. Distributed representations, Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations. (1986).

[10] Mira Kajko-Mattsson. 2005. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55. http://www.springerlink.com/index/10.1023/B:LIDA.0000048322.42751.ca

[11] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings (Lecture Notes in Computer Science)*, Patrick Cousot (Ed.), Vol. 2126. Springer, 40–56. DOI:http://dx.doi.org/10.1007/3-540-47764-0_3

[12] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. DOI:http://dx.doi.org/10.1109/CGO.2004.1281665

[13] Jey Han Lau and Timothy Baldwin. 2016. An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation. *CoRR* abs/1607.05368 (2016). arXiv:1607.05368 http://arxiv.org/abs/1607.05368

[14] Timothy C Lethbridge, Janice Singer, and Andrew Forward. 2003. How software engineers use documentation: The state of the practice. *IEEE software* 20, 6 (2003), 35–39.

[15] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and software* 7, 4 (1987), 325–339.

[16] 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.

[17] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, Nov (2008), 2579–2605.

[18] Roy A Maxion and Rachel R Roberts. 2004. *Proper use of ROC curves in Intrusion/Anomaly Detection.*

[19] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. (2013). http://arxiv.org/abs/1301.3781

[20] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: the case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 361–377. DOI:http://dx.doi.org/10.1145/2815400.2815422

[21] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 438–449. DOI:http://dx.doi.org/10.1109/ICSE.2017.47

[22] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. http://aclweb.org/anthology/D/D14/D14-1162.pdf

[23] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani (Eds.). ACM, 701–710. DOI:http://dx.doi.org/10.1145/2623330.2623732

[24] Michael Pradel and Koushik Sen. 2018. Deep Learning to Find Bugs. (2018). To appear in Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA.

[25] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[26] 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. DOI:http://dx.doi.org/10.1016/j.scico.2005.02.009

[27] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 270–280. DOI:http://dx.doi.org/10.1145/1542476.1542506

[28] Cindy Rubio-González and Ben Liblit. 2010. Expect the unexpected: error code mismatches between documentation and the real world. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010*, Sorin Lerner and Atanas Rountev (Eds.). ACM, 73–80. DOI:http://dx.doi.org/10.1145/1806672.1806687

[29] Cindy Rubio-González and Ben Liblit. 2011. Defective error/pointer interactions in the Linux kernel. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, Matthew B. Dwyer and Frank Tip (Eds.). ACM, 111–121. DOI:http://dx.doi.org/10.1145/2001420.2001434

[30] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*. IEEE Computer Society, 1–12. DOI:http://dx.doi.org/10.1109/DSN.2013.6575307

[31] Saul Schleimer, Daniel Shawcross Wilkerson, and Alexander Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, Alon Y. Halevy, Zachary G. Ives, and AnHai Doan (Eds.). ACM, 76–85. DOI:http://dx.doi.org/10.1145/872757.872770

[32] Suresh Thummalapenta and Tao Xie. 2009. Mining exception-handling rules as sequence association rules. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 496–506. DOI:http://dx.doi.org/10.1109/ICSE.2009.5070548

[33] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair. In *International Conference on Learning Representations (ICLR'18)*.

[34] Westley Weimer and George C. Necula. 2004. Finding and preventing runtime error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 419–431. DOI:http://dx.doi.org/10.1145/1028976.1029011

[35] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005 (Lecture Notes in Computer Science)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.), Vol. 3440. Springer, 461–476. DOI:http://dx.doi.org/10.1007/978-3-540-31980-1_30

[36] Xin Ye, Hui Shen, Xiao Ma, Razvan C. Bunescu, and Chang Liu. 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 404–415. DOI:http://dx.doi.org/10.1145/2884781.2884862