

XFL: Naming Functions in Binaries with Extreme Multi-label Learning

James Patrick-Evans^{*†}, Moritz Dannehl^{*} and Johannes Kinder^{*}

^{*}Research Institute CODE, Bundeswehr University Munich, Germany

[†]Information Security Group, Royal Holloway, University of London, United Kingdom
james.patrick-evans.2015@rhul.ac.uk, {moritz.dannehl,johannes.kinder}@unibw.de

Abstract—Reverse engineers benefit from the presence of identifiers such as function names in a binary, but usually these are removed for release. Training a machine learning model to predict function names automatically is promising but fundamentally hard: unlike words in natural language, most function names occur only once. In this paper, we address this problem by introducing eXtreme Function Labeling (XFL), an extreme multi-label learning approach to selecting appropriate labels for binary functions. XFL splits function names into tokens, treating each as an informative label akin to the problem of tagging texts in natural language. We relate the semantics of binary code to labels through DEXTER, a novel function embedding that combines static analysis-based features with local context from the call graph and global context from the entire binary. We demonstrate that XFL/DEXTER outperforms the state of the art in function labeling on a dataset of 10,047 binaries from the Debian project, achieving a precision of 83.5%. We also study combinations of XFL with alternative binary embeddings from the literature and show that DEXTER consistently performs best for this task. As a result, we demonstrate that binary function labeling can be effectively phrased in terms of multi-label learning, and that binary function embeddings benefit from including explicit semantic features.

I. INTRODUCTION

Software reverse engineering is the process of understanding the inner workings of a software system [1]. In a computer security context, reverse engineering is typically performed on a binary without access to source code. The goals of binary reverse engineering include security audits or vulnerability discovery [2], interoperability with systems where no source code is available [3], or forensic analysis of malicious code [4].

Binary reverse engineering is a labor-intensive task, despite existing tooling to automate disassembly and function discovery. A particular challenge is the lack of readily available information about code. When binaries are released, helpful debugging information such as function and variable names are typically removed, or stripped, from the binary. This is done not only to reduce the size of the binary, but also to actively discourage reverse engineering of closed-source code and to protect intellectual property.

In an observational study, Votipka et al. [5] identify three phases in reverse engineering: overview, sub-component scanning, and focused experimentation. The first two of these make heavy use of the available textual information in a binary, such as the names of called API functions or the contents of string constants. This suggests that additional textual information hinting at functionality in the binary would assist the reverse

engineer at least during these initial phases that focus on mapping out the binary. Indeed, the study reports that most reverse engineers focus on improving readability of the code by adding their own annotations that essentially reconstruct debugging information like variable and function names or data structure types. Another recent study by Montavani et al. [6] empirically confirms that especially experts frequently assign their own names to functions during reverse engineering.

Disassemblers such as IDA Pro, Ghidra, Radare2, and Binary Ninja have long identified this need and provide some automation to recognize and name common functions in frequently used software components. In addition to naming arguments to common API functions, they also perform binary pattern matching to name local functions from statically linked libraries and compiler runtimes. While this helpful functionality is widely-used to make reverse engineering more efficient, it is inherently limited. Signatures are designed to exactly match known functions, with only some flexibility to account for minor changes in compilation settings.

Machine learning promises a new generation of more powerful tools for function identification, and initial academic work appears to confirm that it is possible to classify binary code into function names [7, 8]. These systems learn models of the contents and structure of functions and their most likely name. However, much of the success of these systems can be attributed to the identification of highly similar, repeated functions across multiple binaries (e.g., static library functions).

This type of approach faces two fundamental problems that limit its applicability: it can only generate function names that have been seen in the training set; and each such function name represents a separate output class, with the number of possible function names being essentially unbounded. Even worse, the classes are heavily imbalanced, with the majority of classes having a single sample and a minority of classes being over-represented (e.g., `main`). Normalizing function names to remove differences in coding style such as `CamelCase` vs. `snake_case` can alleviate the problem to some extent [8], but still no such approaches can accurately predict function names that remain unseen after normalization.

The drastic class imbalance is visible in Figure 1, which plots the frequencies of function names observed in a dataset of functions derived from 10,047 binaries in Debian packages. Six function names occur in at least 95% of binaries (standardized names like `main`, `libc_csu_init`, etc.). Over

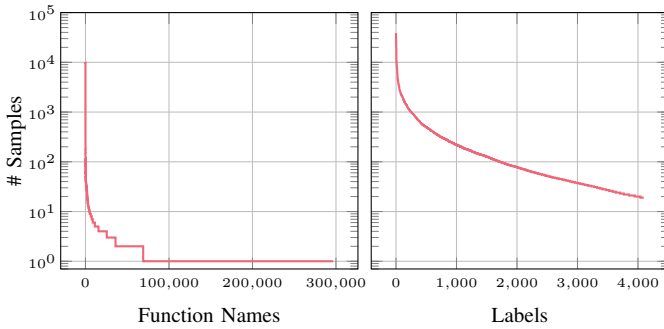


Figure 1: Semi-log plots showing the number of samples in each class when learning function names and labels, respectively. With whole function names, the vast majority of classes have only a single sample.

98% of function names occur fewer than 10 times, and 73% of function names occur only once. This long tail of single-sample classes indicates that any model for whole function names is doomed to mispredict the vast majority of functions.

Our solution to this problem is to split function names into meaningful tokens. For instance, the `xscreensaver` function `make_smooth_colormap` would correspond to the set of labels `{make, smooth, color, map}`. Figure 1 shows that the imbalance in the label distribution is less pronounced. The total number of labels can be controlled such that each function has at least one descriptive label but there are also sufficiently many samples available per label. We therefore arrive at the problem of assigning a set of labels to each function.

A similar problem is that of tagging text with a set of relevant labels, which motivates multi-label learning [9, 10, 11] and extreme multi-label learning (XML) [12, 13], where the number of labels is very large. Based on this insight, we show how to leverage state-of-the-art algorithms from XML for labeling functions in stripped binaries with XFL (*eXtreme Function Labeling*). XFL scales to millions of data points and labels. XFL is parameterized by a given *function embedding*, which maps each binary function to a vector representation. XFL is compatible with state-of-the-art general-purpose binary code embeddings such as PalmTree [14], SAFE [15] and Asm2Vec [16]. In addition, we designed and implemented the novel function embedding DEXTER, which particularly emphasizes semantic properties of the binary code. To this end, DEXTER is trained from a vector of per-function features combined with vectors capturing the context of the local call graph and of the whole binary. While this partly manual feature engineering runs counter to current trends in machine learning, we demonstrate that it is highly effective for the XFL task, providing further evidence that semantic preprocessing of code can improve over syntactic language models [17].

In summary, we make the following main contributions:

- With XFL, we introduce extreme multi-label learning as a solution to the problem of labeling binary functions (§V). XFL solves the problems of sparsity and class imbalance in binary function labeling and provides information-

theoretic metrics for meaningful evaluation. In an extensive evaluation on a dataset with 741,724 functions from 10,047 binaries, we demonstrate that our implementation significantly outperforms the state of the art.

- We introduce DEXTER (§IV), a new vector representation of binary functions using static analysis-based feature engineering and deep learning. We demonstrate that DEXTER outperforms state-of-the-art function embeddings on the task of function labeling. This suggests that increasing the level of abstraction from assembly tokens to static analysis results improves the embedding quality.
- We present an end-to-end function name generation pipeline for stripped binaries based on DEXTER, XFL, and a language model for synthesizing plausible function names from assigned label sets (§VI).

II. EXTREME MULTI-LABEL LEARNING

We now introduce multi-label classification (§II-A), review classic (II-B) and information gain-based (§II-C) metrics, and introduce PfastreXML (§II-D), the state-of-the-art approach to extreme multi-label learning used in XFL.

A. Multi-Label Classification

Multi-label classification is the problem of predicting a variably-sized set of labels per data point [9, 10, 11]. This is different from multi-class classification, where each data point belongs to exactly one class. Because some labels can be more relevant to a data point than others, one also usually wishes to rank the labels by relevance. In a small label space one can get away with a 1-vs-all approach and train an independent classifier for each label [18].

With larger label spaces, both training and prediction become too computationally expensive, however. *Extreme multi-label learning (XML)* deals with very large label spaces: the canonical XML problem is that of predicting a set of suitable categories for a new Wikipedia page from the millions of categories of Wikipedia [12]. To make the problem tractable, both *embedding-based* and *tree-based* methods have been proposed in the literature.

Embedding-based approaches aim to compress the label space by exploiting the fact that most data points will have only few labels, and that labels are highly correlated [13, 19, 20, 21]. While recent deep learning-based approaches appear promising [22, 23, 24], embedding-based methods have traditionally been less precise than tree-based ones, on which we focus in this work.

Tree-based approaches combine classifiers in a hierarchy to reduce the number of classes in each individual problem. This can be done by partitioning either the label space [25, 26] or the feature space. When partitioning the feature space [12, 27, 28], the feature space is split into regions such that in each such region only a small set of labels is active, i.e., has at least one training sample. Then a precise multi-label classifier can be trained for this reduced problem.

Finding the partitioning is part of the training process; and as usual, the training process requires a loss function

that should be minimized over the training samples. For the training process to be practical, the loss function should also be efficiently computable. In most XML settings, a given sample will have vastly more negative (irrelevant) labels than positive (relevant) ones. Therefore, the loss function should give more weight to correct positive labels than to correct negative labels. This rules out loss functions for smaller multi-label problems like Hamming distance [29], which gives equal weight to positive and negative labels. The key idea behind FastXML [12], a precursor of the algorithm we use in this paper, is to directly optimize a loss function based on the *rank* of labels, as in a *learning to rank* problem, where labels are ranked by their relevance to the datapoint. FastXML uses the normalized discounted cumulative gain (nDCG), a metric introduced in information retrieval for calculating the usefulness of search results [30] (see §II-C).

B. Multi-label Metrics

Traditional metrics used in multi-label classification problems are precision, recall, and F-measure, in either their macro- or micro-average forms. *Macro*-averaging computes a metric independently for each label and then averages across labels. This treats all labels equally, leading to skewed results when classes are highly imbalanced. In that situation, *micro*-averaging is preferred: *micro*-averaging adds up true and false positives and negatives of all labels to compute an aggregate metric. Because of the class-imbalance in function labels, we use micro-average scores throughout, in particular micro-average precision P_μ , micro-average recall R_μ , and micro-average $F_{1\mu}$ as defined below:

$$P_\mu = \frac{\sum_{\ell \in L} TP_\ell}{\sum_{\ell \in L} TP_\ell + FP_\ell} \quad R_\mu = \frac{\sum_{\ell \in L} TP_\ell}{\sum_{\ell \in L} TP_\ell + FN_\ell}$$

$$F_{1\mu} = \frac{2 \cdot P_\mu \cdot R_\mu}{P_\mu + R_\mu}$$

Here, L is the set of labels in the corresponding label space. TP_ℓ , FP_ℓ , FN_ℓ , correspond to the number of true positives, false positives, and false negatives for each label defined in the standard way.

Because they are widely-used and easy to interpret, we use micro-average precision, recall, and F_1 to evaluate XFL on the task of predicting the relevant subset of correct labels assigned to each function name (§VII-D). However, these metrics do not take the ranking information into account, which is why they are less suited for training an XML classifier. Therefore, FastXML and PfastreXML use the cumulative gain-based metrics below; we also use these metrics to give more insight into the effectiveness of DEXTER and XFL in our evaluation.

C. Cumulative Gain-based Metrics

Cumulative gain-based metrics are widely used in information retrieval [12, 13, 21, 27, 28, 31, 32]. Intuitively, they sum up the gain in useful information from looking at the first k results of a search query [30]. This makes them ideal for evaluating the quality of a ranked list of results, or equivalently, the relevance of the ranked labels for a datapoint.

Definition 1 (Cumulative Gain): The cumulative gain $CG_{@k}$ measures the information gain in the top k labels of a ranked list of labels and is defined as

$$CG_{@k} = \sum_{i=1}^k rel_i$$

where rel_i is the relevance of the label at position i .

In this work, we use a binary relevance of either 1 or 0, depending on whether the label is contained in the set of true labels for each data point or not. Note that the cumulative gain ignores the ordering of labels within the top k elements, so for our purposes it simply counts the number of correct labels.

The *discounted cumulative gain* introduces a logarithmic discounting factor to the relevance of each label such that labels appearing earlier are given more weight:

Definition 2 (Discounted Cumulative Gain):

$$DCG_{@k} = \sum_{i=1}^k \frac{rel_i}{\log_2(i+1)}$$

Neither cumulative gain nor discounted cumulative gain account for the difference in the true relevance of labels for each data point or query: some may have many relevant labels, others only few. To be comparable across data points, the *normalized discounted cumulative gain* (nDCG) multiplies the DCG with a normalization factor Z_k that is the inverse of the maximum DCG achievable by returning the k truly most relevant labels. In our case of binary relevance, we thus have

$$Z_k = \left(\sum_{i=1}^{\min(k,n)} \frac{1}{\log_2(i+1)} \right)^{-1}$$

for a data point with n true labels, as all other labels will have a relevance of 0. We can now define:

Definition 3 (Normalized Discounted Cumulative Gain):

$$nDCG_{@k} = Z_k \cdot \sum_{i=1}^k \frac{rel_i}{\log_2(i+1)}$$

The nDCG produces a metric for evaluating multi-label classification in the range $[0, 1]$ whereby a perfect ranking would achieve a score of 1.0. To evaluate over multiple points in a dataset, we compute the mean of all nDCG scores.

In our evaluation, we consistently use $k = 5$ as most functions will tokenize to five or fewer labels in the label-space sizes we considered (see also the dataset analysis in Table II).

D. Propensity-based Scoring in PfastreXML

A common issue in the very large datasets used for training XML models is that data points in the observed ground truth often do not carry all truly relevant labels (the complete ground truth). In datasets like Wikipedia, authors and editors may not simply not be aware of all the categories that would in principle apply to their article. It is much less common to have labels assigned to data points that are plain wrong, however. The noise in the ground truth therefore mostly goes into the direction of data points missing labels.

A key insight behind *PfastreXML* [33] is that successful predictions of labels which are rarely assigned although they are relevant, should be specifically rewarded during training. To that end, *PfastreXML* improves *FastXML* by using label *propensities* [34] as part of the loss function. Intuitively, for a specific data point the propensity of a label is the probability that it has actually been assigned in the ground truth if it is truly relevant to the data point. The propensity-scored loss function then estimates the true loss that would be observed on the complete ground truth without any missing labels.

Based on empirical evidence on large datasets that permit reliable estimates of the true ground truth, Jain et al. [33] propose that the propensity p_ℓ of label ℓ can be modeled by a sigmoidal function of $\log N_\ell$, where N_ℓ is the number of data points annotated with label ℓ in the observed ground truth dataset of size N :

$$p_\ell = \frac{1}{1 + C e^{-A \log(N_\ell + B)}}$$

Here, A and B are hyper-parameters, and C is derived from them as $C = (\log N - 1)(B + 1)^A$. The hyper-parameters should be tuned for each dataset such that plotting the propensities of all labels against their respective $\log N_\ell$ fits a sigmoid.

PfastreXML uses propensities by defining a propensity-scored variant PSDCG of the DCG for computing its loss function, in particular it multiplies the discount with the corresponding label propensity such that

$$\text{PSDCG}@k = \sum_{i=1}^k \frac{rel_i}{p_{\ell_i} \log_2(i + 1)}$$

where ℓ_i is the label predicted at rank i .

Finally, *PfastreXML* also adjusts the final classification step in the leaf nodes of the tree learned by *FastXML* by reranking results using classifiers for rare labels, which introduce additional hyperparameters α (for re-ranking) and γ (for predicting rare labels). Through propensity-scoring and reranking, *PfastreXML* improves the multi-label classification accuracy of *FastXML* especially when dealing with infrequent labels in large datasets.

III. OVERVIEW

We now give a brief overview of our proposed end-to-end architecture for function labeling, including the processes for training (§III-A) and prediction (§III-B). The architecture is shown in Figure 2. When given a stripped binary, our system is trained to predict a ranked set of labels for each function, corresponding to tokens found in the names of similar functions. These labels can inform a reverse engineer directly in their ranked form, or they can be used to automatically synthesize a likely function name containing them.

A. Training

There are several components that require training. Training data consists of unstripped binary executables, such that we know the developer-assigned names of functions. In a practical deployment, one would collect as many binaries as possible;

for evaluating performance in our paper, we split the available data into training, validation, and test sets.

1) *Generating a Label Space*: We need to define a label space from which we will draw the labels to predict. To that end, we tokenize all function names in the training set according to a set of syntactic rules. The rules take into account (combinations of) multiple naming conventions and substitute common abbreviations. The union of all tokens becomes the label space. We bound the size of the label space to between 512 and 4096 labels in our experiments, which excludes extremely rare and unique tokens such as typos, highly program-specific or non-English words.

2) *Training the Function Embeddings*: Binary function embeddings are a fundamental building block for our approach. They map the code of a function within a binary to a vector representation. The training process for function embeddings attempts to optimize the representation such that similar functions have low distance in the vector space, whereas dissimilar functions are further apart, for some notion of similarity. We can either use (partially) pre-trained embeddings from the literature, or our own embedding, DEXTER.

3) *Training XFL*: We train the extreme multi-label classifier to predict sets of labels for each embedding vector, based on the tokenized name of its corresponding binary function. *PfastreXML* maximizes the nDCG for the training data, i.e., it aims to ensure that the labels in a function’s name will be ranked highest for that data point among all the labels in the label space. The observed probabilities of labels for training samples also imply a threshold value for the probabilities of true labels in a ranking of all labels. To improve performance, one can perform hyper-parameter tuning as part of this step, in particular for parameters A , B , α , and γ .

4) *Training the Language Model*: To generate actual function names, we train a classical trigram language model from the tokenization of function names in the training set. From a set of labels, the language model is then able to predict their most likely order in a real function name.

B. Prediction

Our resulting system will predict labels for stripped binaries, as a reverse engineer would encounter them. We convert each target binary function into a vector using the trained embeddings model. We feed that vector into the XFL model, which will produce a ranking for all labels in the label space. All labels with probabilities above the threshold value observed during training are returned as the predicted set. From the predicted set of labels, the language model can then synthesize the most likely function name containing all these labels according to a coding convention like `snake_case`.

IV. FUNCTION EMBEDDING

We now present our approach to function embeddings in detail. First, we briefly describe existing embeddings for binary functions (§IV-A), before introducing the features of our new DEXTER embedding (§IV-B), its representation of context (§IV-C), and the training process (§IV-D).

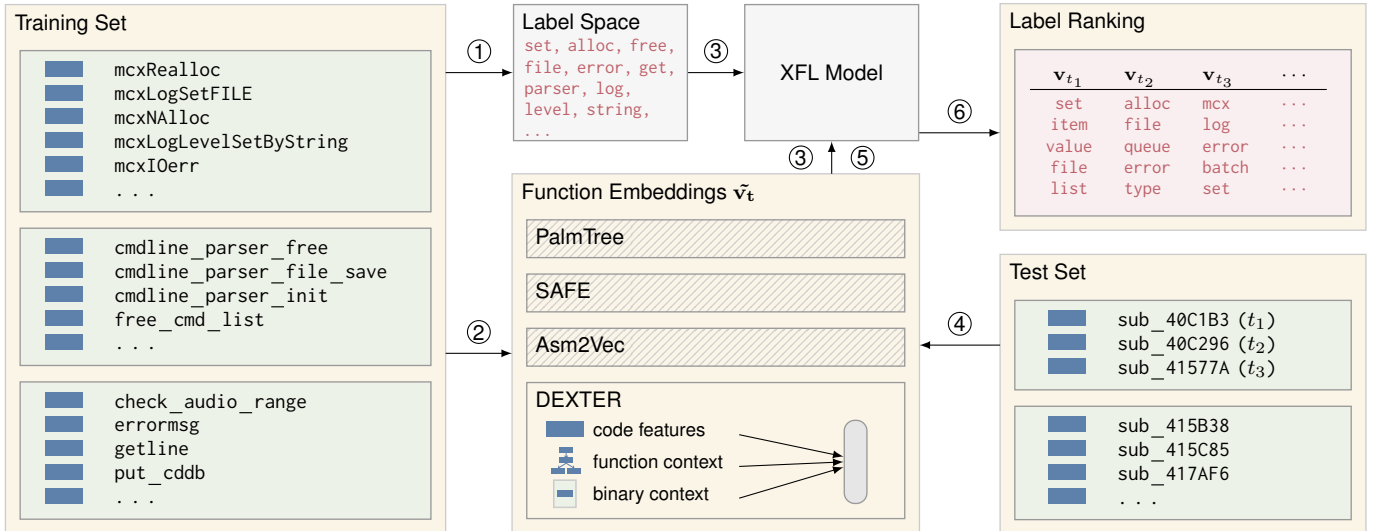


Figure 2: Overview of the XFL training and inference process. Function names in the training dataset are preprocessed to create the label space ①. The function bodies are used to train the embeddings ②. Function embeddings and the label space serve as input for training an XFL model ③. To infer labels for functions in a binary, an embedding v_{t_i} is calculated for each unknown function t_i ④ and fed into the XFL model ⑤, which then produces a ranked list of labels per function ⑥.

A. Existing Binary Function Embeddings

Function embeddings represent binary code with a real-valued vector, capturing similarities and arranging the vector space such that (syntactically or semantically) similar functions have a smaller distance between them relative to other dissimilar functions. We discuss three existing embeddings for binary code, `Asm2Vec` [16], `SAFE` [15], and `PalmTree` [14], all of which can be used with XFL. All three are inspired by natural language processing, treating instructions sequences and possibly additional information as sentences.

`Asm2Vec` [16] adopts a `Word2Vec`-like approach [35] for assembly language. For every assembly function, the model generates execution traces and applies a Paragraph Vector Distributed Memory (PV-DM) model on it, generating a distributed representation for opcodes and operands of assembly instructions. Along the way, a vector representation for assembly functions is learned, similar to paragraph vectors [36].

`SAFE` [15] uses a self-attentive neural network architecture and models sequences of assembly code. `SAFE` first models each instruction using an adapted skip-gram method. The sequence of instruction embeddings is then used to compute local summaries using a self-attentive neural network, and the summaries are combined in a weighted sum.

`PalmTree` [14] is a BERT-based [37] model generating instruction embeddings from assembly code. It is trained via self-supervised learning using multiple training objectives on assembly code, such as Masked Language Model, Next Sentence Prediction, and Def-Use Prediction. To obtain function embeddings from `PalmTree`, the authors use the Gemini Siamese architecture [38].

Both `Asm2Vec` and `SAFE` are syntactic in nature, although

they use static analysis for constructing the CFG and retrieving feasible traces of assembly instructions. `PalmTree` adds def-use information, which makes results of static analysis available for training the embedding.

B. DEXTER Feature Engineering

Our hypothesis behind the `DEXTER` embeddings is that making code semantics explicit will help the training process derive meaningful embeddings with less training data. Where existing embeddings largely follow a modern natural language processing pipeline that does only minimal preprocessing [14, 15, 16], we consciously go against current trends in deep learning and follow an approach that gives more weight to classical feature engineering.

Our features build upon those derived by the analysis framework of Patrick-Evans et al. [8], which lifts each function to an intermediate representation (VEX) and performs a symbolic analysis of each function in isolation. We analyze live memory addresses and register values used in each function to determine the number of input arguments. Finally, we taint each input argument recovered and calculate flows to the individual arguments of callee functions. The full list of features is shown in Table I. We use high-level intra-procedural and inter-procedural features in order to mitigate large differences in machine code from different compilation environments. Our intuition for using these features comes from Kim et al.’s overview of binary similarity [39] which analyzes the most prevalent features for matching similar binary functions.

We add two graph based vector representations of each function. The first uses the Local Degree Profile (LDP) [40] graph embedding technique to map each function’s intra-

Table I: Binary code features used by DEXTER embeddings.

| <i>Quantitative Features: \mathbf{q}</i> | |
|--|--|
| Size of the symbol in bytes | |
| Number of IR instructions | |
| Sum of one-hot-encoded vectors of branch types | |
| Number of temporary variables in the IR | |
| Sum of one-hot-encoded vectors of IR elements | |
| Number of callers | |
| Number of callees | |
| Number of transitively reachable functions | |
| Vector representation of the function CFG | |
| Vector representation of the function node in the binary callgraph | |
| Number of bytes referenced on the stack | |
| Number of bytes referenced on the heap | |
| Number of bytes referenced in Thread Local Storage | |
| Number of function arguments | |
| Number of bytes used for local variables on the stack | |
| One-hot encoded vector of tainted register types | |
| Number of tainted bytes of the heap | |
| Number of tainted bytes of the stack | |
| Number of tainted bytes in arguments to other functions | |
| Number of conditional jumps that depend on a tainted variable | |
| Number of tainted flows to other functions | |
| <i>Categorical Features: \mathbf{c}</i> | |
| Common SHA-256 hashes of assembly opcodes | |
| Common MinHash hashes of assembly opcodes | |
| Common constants referenced | |
| Names of dynamically linked callees | |
| Known function names reachable from this function | |
| References to known data objects in dynamically linked libraries | |
| Names of dynamic functions and argument registers tainted | |

procedural CFG into a vector representation. The second uses the BoostNE [41] node embedding algorithm to embed the function’s location in the callgraph.

Our features also include two hashes of assembly opcode sequences to aid the identification of common functions: a SHA-256 hash matches exact opcode sequences, and the locality-sensitive hashing algorithm MinHash [42] matches very similar opcode sequences.

The feature vector for a binary function b is built from concatenating two vectors of quantitative and categorical data:

$$\mathbf{f}_b = [\mathbf{q}_b, \mathbf{c}_b]$$

Here, \mathbf{q} represents dense, quantitative information describing features of functions in executables and \mathbf{c} represents sparse, categorical data of features that have no numerical basis, in a one-hot-encoded form. Each quantitative feature vector contains 512 elements, and each categorical feature vector contains 3 million elements.

C. Function and Binary Context

In addition to features of the function itself, we include information from its local and global context in DEXTER. To represent the calling context, we build a function context vector \mathbf{f}_C as the mean of the feature vectors of its callers and callees. This captures information from surrounding functions where the body of the target function does not contain enough unique information to correctly predict its name. For example, overloaded functions, which can be called with a different number of parameters, will typically have small stub functions

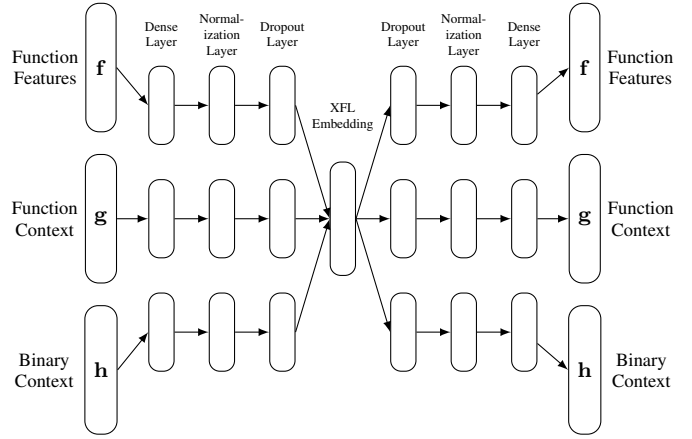


Figure 3: Overview of the autoencoder design used to generate DEXTER embeddings.

that initialize parameters and then jump to the larger function implementing the actual functionality. We define the function context of a function b as

$$\mathbf{g}_b = \frac{1}{|\mathcal{C}_b|} \sum_{k \in \mathcal{C}_b} \mathbf{f}_k$$

where \mathcal{C}_b is the set of callers and callees of function b .

We also wish to include context from the entire binary, as especially the naming of functions can be affected by the context in which it is found (e.g., some libraries use common prefixes to all their functions). To this end, we average the feature vector of every function in the whole binary and define the binary context as

$$\mathbf{h}_b = \frac{1}{|\mathcal{B}_b|} \sum_{k \in \mathcal{B}_b} \mathbf{f}_k$$

where \mathcal{B}_b is the set of all functions in the binary that b is in.

Finally, we build a modified feature vector $\hat{\mathbf{f}}_b$ for each function b in our dataset by concatenating the function feature vector with its context:

$$\hat{\mathbf{f}}_b = [\mathbf{f}_b, \mathbf{g}_b, \mathbf{h}_b]$$

This modified feature vector of each function is then used as input to train an autoencoder.

D. Autoencoder Training

A deep autoencoder is then trained on our modified feature vectors for each function to create a dense, distributed embedding of functions. The autoencoder architecture is depicted in Figure 3. The model first creates dense representations for the function and its contexts before combining them into a single embedding. Our methodology captures structural information not present when using features from each function in isolation. To enforce generalization we first connect the input layer into three dense sub-layers, each with 768 nodes, before performing batch normalization and creating a dropout layer. These layers, along with L_1 and L_2 regularization

on each dense sub-layer, aim to prevent our model from over-fitting and force our model to learn an embedding that generalizes well. Finally, we connect the output from all three dropout layers into a 512 node dense layer that we use as our embedded representation. The model architecture is reversed for predicting the output feature vector, with the exception that, during training we apply Gaussian noise immediately after the embedding layer using standard deviation of 0.1.

Our model is created using Tensorflow [43] and aims to minimize the loss between an input feature vector and a corresponding output feature vector. When feeding our model pairs of input and output vectors, the output feature vector is randomly sampled from the set of all functions with the same name as the corresponding input feature vector. To effectively train useful embeddings, we optimize our model using max margin contrastive loss with Adagrad. Using standard loss metrics such as the Binary Cross Entropy or Hamming loss is ineffective at training models with very sparse data. In using the contrastive loss, we aim to produce an embedding in which functions with similar features appear close together in the embedded space.

V. XML FOR FUNCTION BINARIES

XFL uses PfastreXML and a binary function embedding to efficiently perform multi-label classification for tokenized labels of function names. We first detail how XFL splits function names into tokens (§V-A) and generates a label space (§V-B), before describing how XFL uses PfastreXML to rank associated labels (§V-C) for functions.

A. Tokenizing Function Names

For XFL to predict labels in function names we first need a well defined label space consisting of string tokens found in the names of functions. Labels should be informative, take into account programming styles and be mutually exclusive where possible; for example, the tokens `str`, `string`, `String`, and `__xStr__` should all have the common denominator token `string`. XFL generates a *canonical token set* for each function name, a set of string tokens that canonically describe it. We generate a well-defined label space of a fixed size by analyzing the union of the resulting canonical token sets from the corpus of function names in the training set.

To generate the canonical token set L_c from a function name, XFL uses the following procedure:

1) *Strip Library Decorations*: Regular expressions remove common symbol annotations added by compilers, e.g., `.*\.constp$`, `.\.avx\d+`. Regular expressions for Radare2, IDA Pro, and Ghidra annotations are also applied depending on the analysis platform.

2) *Split Alphanumerical*: The function name is split into character sequences along non-alphanumeric characters. Numeric and alpha characters are further split into separate groups. For example, `__libxyz_init` \mapsto `{libxyz,init}`.

3) *Split Camel Case*: We recognize common naming conventions in C binaries and split a continuous character sequence into sets if we detect the use of camel case. For example, `IsWindowOpen` \mapsto `{is,window,open}`.

4) *Abbreviation Expansion*: We expand a predefined list of common programming abbreviations such as `fd` for `file descriptor` and `init` for `initialization`. For example, `makedirs` \mapsto `{make,directories}`.

5) *Best Split of the Rod*: We use a dynamic programming algorithm to split character sequences into the largest possible non-overlapping sequences. We check all permutations of sub-sequences to find the largest collection of English words. Our algorithm scores longer words higher over two or more same-length sub-sequences, e.g., `{background}` $>$ `{back,ground}`. However, a longer length of total characters scores higher, e.g., `foreach` \mapsto `{for,each}` and not `{reach}`.

To confirm that our method produces tokens representative of the developer intent, we conducted a manual validation experiment. Two of the authors manually processed a list of function names in the dataset and the corresponding split from our tokenizer. The task was to evaluate for each split whether it is (1) perfectly correct, (2) reasonable, or (3) wrong. To estimate the accuracy of our tokenizer, we used a subset of 385 randomly selected samples so as to obtain a confidence interval of 95% with an error margin of 5%. The experiment resulted in 80% perfect and 95% at least reasonable splits, averaged between both authors, which confirmed the tokenization to work well in practice. As a reasonable split, we counted meaning-preserving but slightly off splits such as `ndelay_on` \mapsto `{delay,on}`; wrong splits change the meaning, as in `fstarpu_matrix` \mapsto `{fs,tarp,matrix}`.

B. Label Space

After generating all canonical token sets for the training set we take the union of all string tokens found and count the occurrences of each label. The union over all canonical token sets defines our complete label space L . However, we define label spaces of varying size to ensure a minimum number of data points per label and explore the impact of an increasing number of labels. To this end, XFL generates a new *label space* L_n of size n , by taking the top n most frequently used labels used from the complete label space such that $L_n \subseteq L$.

To obtain the ground truth of labels expressed for each function name, we project each name’s canonical set L_c onto those labels that exist in the generated label space L_n as the intersection $L_c \cap L_n$.

C. PfastreXML for Function Labels

The assignment of function labels, i.e., function name tokens, to functions has many parallels to XML problems such as assigning categories to Wikipedia articles: compared to the total size of the label space, each single function name has only very few positive labels. The frequency distribution of labels is extremely skewed (see Figure 1), as labels like `get` or `set` are extremely common (over 36,000 and 20,000 occurrences, respectively), while there is a long tail of infrequent labels (L_{4096} has 2,364 labels with fewer than 100 samples). The re-ranking mechanism of PfastreXML specifically ensures that such infrequent labels do not get lost in the tree-based hierarchy of FastXML.

In addition, there may be more labels that would have been suitable for a given function than just the ones that part of its name in the ground truth. There are often alternative but equivalent ways to name a function because of synonyms among tokens such as delete and erase. As a result, we see “noisy” data where relevant labels are missing from the ground truth. The propensity-scored loss function in PfastreXML can model the probability of labels to be missing, with appropriate hyper-parameter training.

To remove popularity bias and assign rare labels we weight each label inversely to its popularity. XFL optimizes hyper-parameters A and B for the propensity calculation (§II-D) such that the log distribution of propensities matches the sigmoid function. Following the recommendation of the PfastreXML authors, we perform a grid search on hyper-parameters α and γ close to known good parameter values that optimizes the nDCG calculated on our models validation dataset when trained on a training dataset as discussed in §VII-C.

The PfastreXML model predicts the probability distribution of labels for each embedding of a binary function, which induces a ranking over the labels. To produce a concrete set of predicted labels for multi-label classification and synthesizing a function name, we define a threshold p_t such that XFL outputs those labels with probability greater than p_t .

VI. FUNCTION NAME GENERATION

The subset of labels chosen by thresholding is ordered by relevance. We can either present this ranked list of labels directly to the reverse engineer, or we can synthesize a plausible function name containing the labels. Although generating an actual function name loses the ranking information, it may be more easily understood by reverse engineers and can be directly integrated into the reverse engineering process. We now introduce our approach to generating function names (§VI-A) and discuss its accuracy (§VI-B).

A. Language Model

We use a language model for synthesizing plausible function name strings. Our hypothesis is that developers order words in function names following certain rules, defining a form of language where function names are sentences made up of labels. We face the following problem: given a set of labels, which ordering of the labels would be the most likely in a real function name? To solve this, we train a language model that can compute a probability score for a given sequence of labels and then pick the order with the highest probability.

The corpus for training our model consists of all function names split into their constituent labels. We train a language model with modified Kneser-Ney smoothing [44, 45] that interpolates trigram frequencies over function names with bigrams and unigrams to account for unseen trigrams at test time. We use an efficient implementation by Heafield [46], which trains a model from 330k function names in under two seconds on a laptop.

Finding the ordering of labels with maximum probability is essentially the NP-hard maximum Hamiltonian path problem.

We use a version of branch and bound to rule out unlikely combinations of labels early and compute scores of orderings using the language model. As a seed solution, we use a simple greedy strategy that picks the most likely trigrams from left to right. Branch and bound is guaranteed to eventually find the optimal ordering, but we use a timeout of 1 million steps to terminate long-running queries with many labels, after which we accept the currently highest-scoring ordering.

B. Accuracy

Using a training/testing split of 9:1 over 330k unique function names, the language model predicts the correct order of labels for 70.0% of the function names (averaged over 10 runs). On average, each function name is processed in under 2ms. This shows that indeed a classical language model is able to predict useful label orders in most cases, including seemingly hard names such as `tp_svc_channel_type_streamed_media_emit_stream_state_changed` (from `telepathy-glib`).

The cases where it fails to produce the right order roughly correspond to three groups: First, often the model produces an order that is simply an alternative to the original one with the same meaning. For instance, it predicts `init_tree` where `tree_init` would have been correct, or `alloc_obj` for `obj_alloc`. Second, there are cases with multiple equally plausible orderings with different semantics, e.g., `is_type_array` vs. `is_array_type` or `index_to_dir_list` vs. `dir_list_to_index`. Third, some labels are just too rare for the language model to meaningfully generalize. For instance, `evict_user_connection` is ordered as `evict_connection_user` because the label `evict` occurred only once in that testing split, as part of `evict_connection`.

VII. EVALUATION

We now present our evaluation of XFL and DEXTER. We begin by explaining the makeup of our dataset (§VII-A) and the computational resources used (§VII-B). The main evaluation is split into two parts: §VII-C focuses on comparing different function embeddings on a given task, and §VII-D focuses on comparing different tools for end-to-end function name prediction. This corresponds to two research questions that we answer in the evaluation:

RQ1: *Which binary function embedding is most suited for the task of ranking function labels?*

To answer this question, we compare DEXTER against the `Asm2Vec` [16], `SAFE` [15], and `Palmtree` [14] embeddings on the task of function labeling using XFL and demonstrate that DEXTER outperforms the state of the art on this task (§VII-C).

RQ2: *Does XFL generate more suitable function labels than state-of-the-art approaches?*

To answer this question, we compare XFL (using DEXTER embeddings) against the state of the art in function name prediction (§VII-D). We compare tools both in a ranking task with information-theoretic metrics (§VII-D1) and using traditional metrics for multi-class classification (§VII-D2). This

Table II: Comparison of the embeddings and the datasets generated for training XFL. Differences are due to limitations in pre-processing. Average labels per point and points per label were calculated for a label space of size 4096.

| Property | DEXTER | Asm2Vec | PalmTree | SAFE |
|-----------------------|----------|---------|----------|-----------|
| Model Params. | 377.0 M | - | 3.2 M | 57.2 M |
| Size | 512 | 50 | 128 | 100 |
| Model Type | Autoenc. | PV-DM | BERT | Self-Att. |
| Train Samples | 400,357 | 396,796 | 386,205 | 342,610 |
| Test Samples | 22,422 | 22,224 | 21,405 | 19,035 |
| Avg. Labels per Point | 2.93 | 2.93 | 2.85 | 2.75 |
| Avg. Points per Label | 320.45 | 317.44 | 298.83 | 255.24 |

demonstrates that XFL picks out the most relevant labels for functions and is therefore able to generate the most accurate names. We also specifically investigate the problem of dealing with names a model has never seen before (§VII-D3).

DEXTER, XFL, and the language model are implemented in Python and TensorFlow in about 30 KLOC. Our models and data are available on GitHub¹.

A. Dataset

Training and testing require a set of ELF binaries with ground truth symbol information that is sufficiently large for generalizing semantics of binary functions for each label. We use the Punstrip dataset [8], which contains 741,724 functions from 10,047 C binaries taken from pre-compiled Debian packages. These binaries have been compiled with a mixture of compilers and compiler versions from individual package maintainers.

We use the global symbol bindings in the ELF symbol table as ground truth for obtaining function boundaries and the corresponding names. We exclude pseudo functions of size zero, overlapping functions, and locally bound symbols, which do not clearly correspond to a well-defined function. Note that in training, the symbol table is available for reading function boundaries. When predicting labels in an unknown binary, symbols have been stripped and function boundaries would have to be obtained using function boundary prediction, for which a number of mature tools exist [47, 48, 49]. XFL supports reading function boundaries from the disassemblers Ghidra and Radare2, and the academic tool Nucleus [49]. However, to factor out the performance of function boundary prediction from our evaluation, we equip all tools with function boundaries from the symbol table also for test data.

The hyper-parameters A and B of PfastreXML for computing label propensities (see §II-D) are dataset specific. We compute them for the dataset such that the log distribution of propensities matches the sigmoid function and obtain $A = 0.5$ and $B = 0.425$ for the Debian dataset.

B. Computational Cost

All of our experiments were carried out on a machine with an AMD EPYC 2 64-Core CPU, a single 16 GB NVIDIA Tesla T4 GPU, and 1 TB of RAM.

¹<https://github.com/unibw-patch/xfl>

All embedding approaches rely on pre-processing a given binary using disassemblers like Radare2 or IDA Pro, which is required for practically any form of reverse engineering. Training the embeddings is then the step that is most taxing on GPUs. Once a pretrained embedding is available, producing an embedding from an already preprocessed binary is very fast.

For the evaluation of DEXTER and XFL, we cache our binary analysis results and experimentation configurations using PostgreSQL and Redis databases. We parallelized the binary analysis and were able to finish the preprocessing and feature extraction for the full Debian dataset in 120 hours.

Training the DEXTER embeddings used about 850 GB of main memory and took 72 hours. Once complete, generating embeddings for a pre-extracted feature vector takes an average of 10ms per function. Training the Asm2Vec embeddings took about 30 hours. For SAFE and PalmTree, we used pretrained embeddings so we cannot directly compare cost.

Complete training and testing for an XFL model took 13–15 minutes and 73–85 GB of memory depending on the size of the embedding used, with DEXTER taking most and Asm2Vec least time and memory. Table II compares the embeddings by their embedding sizes, their model parameters, and their type.

C. Comparing DEXTER against SotA Binary Embeddings

We evaluate DEXTER against the state-of-the-art embeddings Asm2Vec [16], SAFE [15], and PalmTree [14]. As we are interested in using the embeddings for an XML task, we compare the performance of each embedding evaluated using cumulative gain-based metrics. SAFE and PalmTree both rely on contrastive learning approaches that require the same source code to be compiled under different compilation settings. As the binaries in our dataset were pre-compiled, we could not use it to create new models and therefore use pretrained models for SAFE² and PalmTree³ to generate embeddings for the binaries in our dataset. For Asm2vec, we used the published code⁴ and applied it to the binaries in our dataset to train a custom model. To use PalmTree embeddings, we trained a Gemini model to convert the instruction-level embeddings provided into function-level embeddings. With advice from the PalmTree authors, we replicated the setting as described in section 4.4.1 from the original paper [14] and obtained a model which achieved similar performance⁵. We then used DEXTER, Asm2Vec, SAFE, and PalmTree representations of each function to train corresponding XFL models.

Using our dataset we generate embeddings for all functions and randomly split them into a training, validation, and test set using a 90:5:5 ratio. After training a PfastreXML model for each approach on the training set, we performed a grid search to fine-tune the model hyper-parameters over our validation dataset, and report the evaluation results on the test set. Using values close to known good parameters estimated from the

²<https://github.com/gadiluna/SAFE>

³<https://github.com/palmtreeemod/PalmTree>

⁴<https://github.com/McGill-DMaS/Kam1n0-Community>

⁵Our PalmTree Gemini model achieved a testing AUC of 0.939 using a dataset made up from 518 binaries from binutils and coreutils.

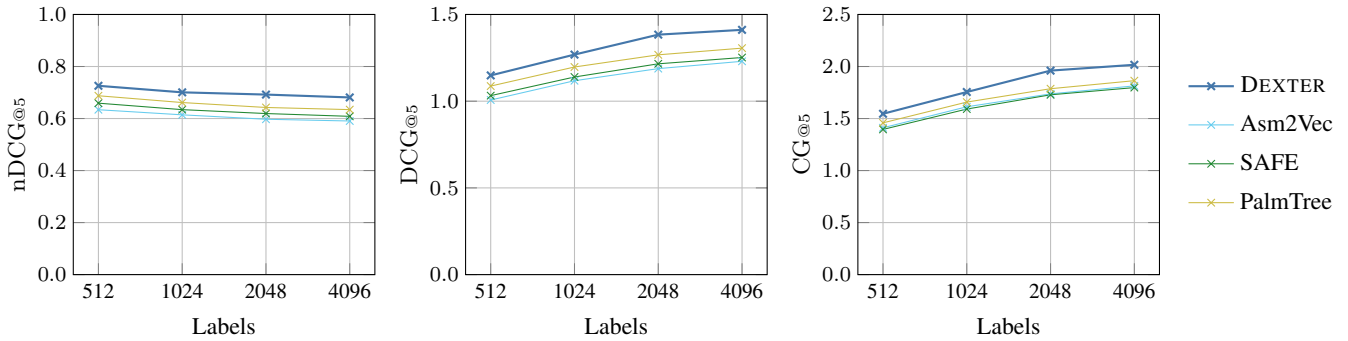


Figure 4: A comparison of the normalized discounted cumulative gain (nDCG@5), the discounted cumulative gain (DCG@5), and the cumulative gain (CG@5) achieved by XFL between DEXTER, Asm2Vec, SAFE, and PalmTree.

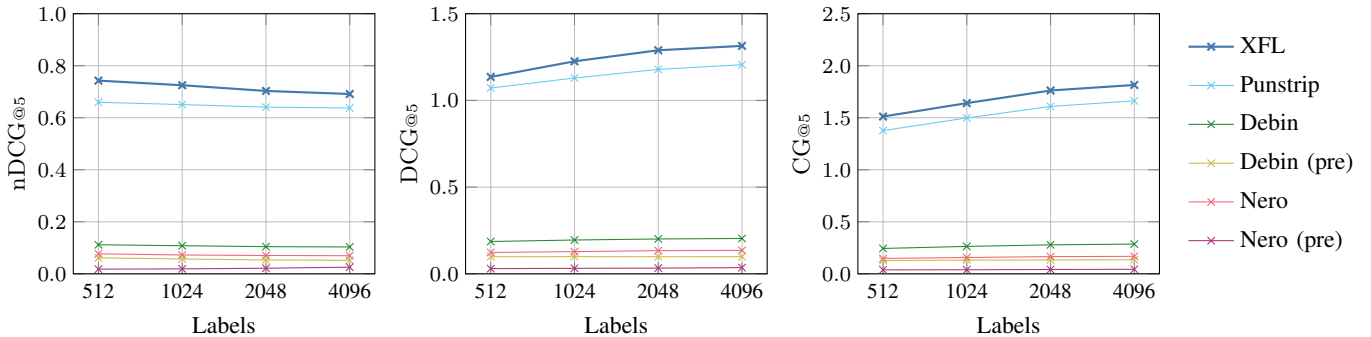


Figure 5: An information theoretic comparison between Debin, XFL, Nero, and Punstrip with increasing label space sizes. All metrics were taken @5 and a default order was added where non-ranking tools predicted fewer than five labels.

PfasterXML paper, we optimize the α and γ hyper-parameters to minimize our nDCG loss (see §II-D). We run experiments with four label spaces L_n with $n \in \{512, 1024, 2048, 4096\}$ to see any relative differences in information gain caused by label space size.

Figure 4 shows the corresponding mean nDCG@5, DCG@5, and CG@5 for all embeddings. We can see that DEXTER outperforms Asm2Vec, SAFE, and PalmTree embeddings across all label space sizes. Among the other three, PalmTree clearly dominates; in their paper, Li et al. [14] demonstrate that the semantics-related components are important factors. Using pre-processing and manual feature engineering DEXTER includes even more code semantics and thus can outperform pure representation learning on the binary code domain.

D. Comparing XFL against SotA in Function Naming

We evaluate XFL against the three state-of-the-art function name prediction tools Debin [7], Nero [50], and Punstrip [8]. We also include a comparison to published results of a customized version of Dire [51] taken from the Nero paper [50] (the Dire authors recommended us not to use their tool for function name prediction). Nero combines heavy static analysis to obtain an Augmented Control Flow Graph and feeds this representation into state-of-the-art neural networks. In their work, a variant based on a Graph Neural Network (GNN) performs best, so we compare against this. Debin [7]

and Punstrip [8] both make predictions for function names based on Conditional Random Fields to compute a maximal joint probability of function name assignments in a binary. For Debin and Nero, we compare against both a pretrained model released by the authors and a custom model trained on our dataset. For Punstrip, no pretrained model is available.

We first randomly split our dataset obtained in §VII-A into training, validation, and testing sets with a respective 90:5:5 ratio. For each of the tools, we train a model on binaries in the training set, tune model hyper-parameters on the validation set, and show our results against the model’s predictions on the test dataset. The list of binaries in all three sets are kept consistent when evaluating all of the tools. We use the same four label space sizes as before.

We evaluate all tools on two problems, both related to predicting relevant labels contained in function names. The first problem (§VII-D1) evaluates the measure of rank, whereby we produce a ranked list of labels in the label space for each data point. We assign each correct/relevant label an equal weight. The second problem (§VII-D2) evaluates each tool in a multi-label classification setting predicting a variably-sized set of relevant labels. Note that Debin and Punstrip are designed to predict whole function names. Therefore, we use our canonicalization step to project the predicted function names onto our generated label space. Nero predicts labels itself, but we report here the results of using Nero on our label

Table III: Multi-label classification evaluation against state-of-the-art tools for the Debian and Nero datasets, highest values in bold. Results marked with † are taken directly from [50] without us rerunning the experiment.

| Tool | Debian Dataset | | | Nero Dataset | | |
|----------|----------------|---------------|----------------|---------------|---------------|----------------|
| | Prec. | Recall | F ₁ | Prec. | Recall | F ₁ |
| XFL | 0.8345 | 0.5750 | 0.6809 | 0.8664 | 0.4383 | 0.5821 |
| Nero | 0.1600 | 0.0622 | 0.0896 | 0.4861 | 0.4282 | 0.4553 |
| Debin | 0.1564 | 0.1081 | 0.1279 | 0.3486† | 0.3254† | 0.3366† |
| Punstrip | 0.6336 | 0.6350 | 0.6343 | - | - | - |
| Dire | - | - | - | 0.3802† | 0.3333† | 0.3552† |

spaces. We confirmed experimentally that Nero’s performance evaluated in this way is in line with the values reported by their own pipeline.

1) *Measure of Rank*: We evaluate this experiment using the information gain metrics nDCG@5, DCG@5, and CG@5 as depicted in Figure 5. As the size of the label space increases, every tool evaluated performed worse in terms of nDCG but better in terms of DCG and CG. With a larger label space, there are more labels per function to be learned. Accordingly, the growing CG and DCG show that more information is learned by the models; still, the normalization against the total number of labels per data point leads to a lower nDCG.

Our results show that XFL consistently outperforms other state-of-the-art tools in terms of information gain irrespective of label space size; this provides evidence for our hypothesis that predicting labels outperforms predicting whole function names. Surprisingly, while Nero outperforms Debin on the Nero dataset for function name prediction (§VII-D2), it performs worse than Debin on the Debian dataset. We investigated this anomaly to rule out mistakes in our experimental setup, but we were able to reproduce the published values on the Nero dataset, and correspondence with the Nero authors confirmed our results. We believe this to underline the risks of dataset bias in contemporary approaches to machine learning on binary code, which we attempt to counter with an increased dataset size and an embedding-based approach.

Note that the results for XFL do not match those for DEXTER in §VII-C due to using different dataset splits: for the embeddings experiment, we randomly split functions independently into training, validation, and testing, whereas in the multi-label classification experiment, we ensured that functions from the same binary are always in the same set.

2) *Multi-label Classification*: To evaluate XFL’s performance in a multi-label classification task on standard metrics, we modify the experiment used in the measure of rank setup to include a linear threshold p_t to decide if the label is relevant. This results in a variably sized subset of predicted relevant labels for each data point. We can then define true positives as the number of correct labels predicted with a threshold greater than p_t . False positives are defined by labels which are predicted but not present in the ground truth, and false negatives are correct labels that are missed by our prediction.

During validation, we set p_t to a value that maximizes the F₁ score on the validation dataset.

Our results as shown in Table III display the micro-averaged multi-label precision, recall, and F₁ score on both the Debian and Nero datasets using a label space of size 1024. XFL outperforms other state-of-the-art tools in precision and F₁, with Punstrip coming closest and winning on recall with the current p_t . By varying the threshold parameter p_t , one would be able to adjust the precision and recall trade off.

We also include an evaluation on the Nero dataset, given the relative difference in performance of Debin and Nero on those binaries. While XFL achieves comparable, if slightly worse results on this smaller dataset, Debin and Nero both perform significantly better. This shows that XFL is able to benefit from a larger dataset to achieve more generalized representations, whereas Debin and Nero both significantly drop in performance when scaling up. Apart from the dataset size, another factor may be that in Nero’s test set of 13 binaries, many are from the same source repository and are all compiled with the same settings. In contrast, our Debian dataset contains 503 binaries in the test set, which are compiled by individual package maintainers where the compiler, version, and optimization levels are not fixed.

By ordering relevant labels using the language model and joining them by an underscore or in camel case, we can synthesize whole function names. Table IV shows complete examples of such names, including typical errors that can be introduced through the function name generation process. These examples show that our unsupervised tokenization process was automatically able to deduce the importance of the labels *HID* for *human input device*, *lkf* for *locked file*, and *mcx* as a term used in the Markov Clustering set of algorithms.

3) *Generalization*: Although function *bodies* are never shared between training and testing sets, many function *names* do occur in both. Function names like *hash*, *get_line*, or *usage* are used frequently by developers, although the corresponding function bodies will be quite different. In practice, we expect such cases to occur frequently, and predicting such names correctly would be useful for a reverse engineer.

However, multi-label learning for function name generation has the advantage of being able to generalize to new function names that have never been seen before, which is impossible for multi-class approaches. Therefore, we now specifically investigate how well all approaches are able to identify combinations of labels when no function names are shared between the training and testing sets. To this end, we repeated our experiments as in §VII-D2, but restricted the test set to function names not present in training.

Table V shows that performance drops for every tool but also confirms that XFL is able to predict labels in unseen names better than other approaches. Note that while Punstrip and Debin cannot predict unseen function names by design, applying our tokenization to their predictions still yielded some correct labels. Common labels recovered correctly as part of unknown function names include *get*, *set*, *new*, *free*, and the OCaml-specific tokens *caml* and *fun*.

Table IV: Example data points of the function name generation process. Ground truth names are first split into known labels which we aim to predict. The predicted labels are then used to generate a function name according to the language model. Labels from the canonicalization process that did not make it into the label space are crossed out.

| Original Name | Ground Truth Tokens | Predicted Labels | Output Name |
|-------------------------|----------------------------|-------------------------------------|----------------------------|
| grub_crypto_cbc_decrypt | {grub, cbc, de, crypt} | {de, crypt, grub, crypt, cb, odisk} | grub_crypto_odisk_de_cb |
| make_smooth_colormap | {make, color, map, smooth} | {make, color, map, random} | make_random_color_map |
| mcxRealloc | {realloc, mcx} | {realloc, mcx} | mcx_realloc |
| check_audio_range | {range, audio, check} | {audio, range, check} | audio_range_check |
| HIDSetItemValue | {set, item, hid, value} | {set, item, hid, value, is} | is_item_hid_set_value |
| mcxTingRelease | {release, mcx, ting} | {release, mcx} | mcx_release |
| chirp_recursive_put | {put, recursive, chirp} | {put, recursive, chirp, ticket} | ticket_chirp_recursive_put |
| lkfopendata | {lkf, data, open} | {lkf, open, switches, data} | switches_lkf_open_data |
| cmdline_parser_init | {line, init, cmd, parser} | {parser, init, cmd, line, csu} | csu_cmd_line_parser_init |

Table V: Comparison of the ability to predict completely unseen function names (Debian).

| Tool | Prec. | Recall | F ₁ |
|----------|---------------|---------------|----------------|
| XFL | 0.4342 | 0.1739 | 0.2416 |
| Nero | 0.0494 | 0.0238 | 0.0321 |
| Debin | 0.0453 | 0.0292 | 0.0380 |
| Punstrip | 0.1170 | 0.1190 | 0.1180 |

VIII. DISCUSSION

We now discuss our results and current limitations of our approach, and point out directions for future work.

A. Generalization and Dataset Diversity

Our results for multi-label classification show that Nero is outperformed by Debin on the Debian dataset; a surprising outcome given that Nero outperforms Debin on the Nero dataset. We contacted the authors of Nero to investigate further and they suggested the following possible causes: (i) Nero’s dataset limited the maximum size of binaries to 1MB. This may influence the inference of common function names available in most *libc* binaries. (ii) The vocabulary size in our dataset is significantly bigger and as a result Nero predicts empty labels 43% of the time. (iii) Nero’s configuration and implementation had not been fine-tuned to our dataset. This highlights an interesting issue. Even when taking care to avoid overfitting and ensuring proper train-test splits, there are aspects of the model that are inherently dataset-specific. We can see that even our own model decreases in performance when testing on the Nero dataset, which uses different compilers and build settings.

Although our preprocessing and analysis should reduce the amount of data needed, even the Debian dataset is still much smaller than the corpuses used for training state-of-the-art models for natural language processing. We believe that public, standardized datasets are the way forward, both for source and binary code-based tasks.

B. Application Domains

Using machine learning for predicting function names inherently requires a training dataset whose distribution resembles the target dataset. That is, we cannot hope to predict

function names in Windows binaries with a model trained on GNU/Linux. As it stands, the model we trained for XFL can identify labels in binaries compiled for GNU/Linux systems only, but the target binaries can be closed source. Because of our multi-label approach, XFL can construct suitable names from tokens for function names it has never seen in the training set (see also §VII-D3). This can make XFL useful in practice today, for reverse engineering closed-source applications or device drivers, or for forensic analysis of GNU/Linux malware. However, as no ground truth is available for closed source software, evaluating this aspect will require a study with human participants. We believe this to be an interesting avenue for future work, in line with recent pioneering work into observational studies on reverse engineers [5, 6].

In principle, a Windows-specific model for XFL could be trained, too, as long as we can construct a sufficiently large ground truth dataset. A dataset the size of the Debian dataset would be challenging to obtain, but not impossible. For instance, Github contains the source code of over 230,000 C files with a `WinMain` function defined. Another possibility are to use public servers with debug symbols for ground truth, or the HexRays Lumina server, which stores function names for reversed closed-source binaries. Similarly, our model is specific to the `x86_64` architecture. Because our analysis is built on the VEX IR used by `angr`, one could use the same tool chain for analyzing other instruction set architectures.

Our entire toolchain and setup is currently geared towards binaries compiled from C, only. Apart from the differences in distribution that would come with code compiled from other languages, the different naming conventions, namespaces and the resulting name mangling would require changes in our preprocessing infrastructure.

C. Impact of Function Boundaries

Our evaluation assumes that precise function boundaries are available. At training time, this is a safe assumption as we require debug symbols regardless. At prediction time, tools such as Nucleus [49] are highly accurate but can detect more or fewer functions than are actually present. XFL will predict labels for whatever function body it is given, so it would attempt to predict labels for parts of a function or for the combination of multiple functions. When used interactively,

a reverse engineer would have to query XFL again for any functions they change boundaries for.

D. Concept Drift

Even on the same architecture, the testing data distribution can change over time as source code and compilers change. This *concept drift* is a known phenomenon in many application domains. The performance differences between different datasets suggest that it is present in the problem of function labeling. It is possible to counteract concept drift through periodic re-training with labeled data. Recent work by Yang et al. [52] uses contrastive learning to detect the and explain concept drift in new data samples. Such an approach may also be combined with XFL to mitigate the impact of changing test data over time.

E. Adversarial Settings

In this paper, we rule out an adversarial setting where the developers actively try to prevent recognition of functions by automated means. We explicitly assume the code of functions to be generated by a regular compiler, with only the debug and symbol information to be removed as is typically done for release builds. Obfuscation methods like runtime packing or opaque predicates that prevent successful disassembly would break our pipeline without explicit countermeasures [53]. Similarly, more subtle obfuscation methods would substantially change the character of functions and thus likely sufficiently affect the features to increase mispredictions.

Beyond obfuscation, one could consider adversarial attacks against the multi-label classifier model in XFL [54, 55]. Using code transformations as adversarial perturbations [56], an attacker could intentionally cause mislabeling of functions with specific labels. Adversarial robustness is an active area of research, and most proposed countermeasures have quickly been shown to be ineffective [57].

IX. RELATED WORK

We now review related work on the problem of binary function labeling, binary code similarity, and of learning source-level code representations.

XFL predicts labels for functions and is thus related to similar projects which produce labels or names for functions. Punstrip [8] uses Conditional Random Fields (CRFs) to capture the dependencies between generated fingerprints of functions and their callers and callees. Debin [7] attempts to recover function names and other debug information such as program variables and their mapping to registers and memory offsets. In Debin, CRFs represent relationships between code and data and are used to predict properties of extracted memory cells. Nero [50] builds an *augmented control flow graph* which extends a CFG with call sites, specifically engineered for procedure name prediction. The representation is fed into GNNs, LSTMs and Transformer architectures. Dire [51] only supports variable name prediction; however, the authors of Nero modified the project to support the prediction of procedure names. Dire uses an encoder-decoder neural network, taking as input both

tokenized code and the AST from a decompiler and generates embeddings for each identifier which are used by the decoder to predict names.

Much recent work on vector embeddings for binary code focuses on the task of *binary code similarity*. DeepBinDiff [58] defines the task as trying to find the best match between similar basic blocks based upon their control flow dependency. A different approach is to use the *same source policy* which defines two binaries or functions to be similar if they are compiled from the same source code but for different target architectures [38], different source code versions [59] or different compilers and compiler settings [16].

Many approaches borrow from natural language processing. We already discussed SAFE [15], Asm2Vec [16], and PalmTree [14] in §IV-A. Zuo et. al [60] rely on Word2Vec, but adapt *Neural Machine Translation* to handle single instructions as words and basic blocks as sentences. Sun et al. [61] use approaches found in bioinformatics such as longest common sub-sequence algorithms in conjunction with Word2Vec embeddings to measure binary semantic similarity.

While XFL uses debug information from the compilation process as ground truth, our model relies only on information found in stripped binaries, i.e. *without* access to the source code. Nevertheless, we review source code based approaches to function similarity. Source code function similarity has been used in program comprehension, function name suggestion, and source code completion. Models for source code utilize syntax information from Abstract Syntax Trees [62] [63]. Program representations may be constructed using generative models [64], graph neural networks [65], graph models enriched with sequence encoders [66], or attention-based models [67]. A convolutional network is used to summarize source code to tokens [68].

Another related area is code authorship attribution, i.e., identifying, verifying, or clustering code authors; we refer to Kalgutkar et al. [69] for a recent survey.

X. CONCLUSION

We present DEXTER and XFL to solve the function naming problem, addressing limitations in earlier methods.

DEXTER creates a distributed representation of binary code that concisely captures the semantics of functions. Our embeddings condense millions of features drawn from the whole binary, the function’s calling context, and the function itself. We show that it outperforms state-of-the-art binary code embeddings when used for predicting labels in function names. This provides evidence that using static analysis results for learning embeddings improves performance in comparison to relying on just learning on raw syntax.

XFL uses DEXTER to perform multi-label classification and learn an XML model to predict common tokens found in the names of functions from C binaries in Debian. We show that our approach outperforms existing approaches to function name prediction. In particular, XFL is able to predict names for functions even when no function of that name is contained in the training set.

REFERENCES

- [1] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, 1990.
- [2] I. Arce, "Bug hunting: The seven ways of the security samurai (supplement to computer magazine)," *IEEE Computer*, vol. 35, no. 04, pp. 11–15, 2002.
- [3] C. Cifuentes, "The impact of copyright on the development of cutting edge binary reverse engineering technology," in *Working Conf. Reverse Engineering (WCRE)*. IEEE Computer Society, 1999, pp. 66–76.
- [4] M. Siksorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco: No Starch Press, 2012.
- [5] D. Votipka, S. M. Rabin, K. K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *29th USENIX Security Symposium, USENIX Security 2020*. USENIX Association, 2020, pp. 1875–1892.
- [6] A. Mantovani, S. Aonzo, Y. Fratantonio, and D. Balzarotti, "Re-mind: a first look inside the mind of a reverse engineer," in *Proc. 31st USENIX Security Symposium (USENIX Security)*. USENIX Association, 2022.
- [7] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev, "Statistical deobfuscation of android applications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 343–355.
- [8] J. Patrick-Evans, L. Cavallaro, and J. Kinder, "Probabilistic naming of functions in stripped binaries," in *Proc. 35th Annu. Computer Security Applications Conference (ACSAC)*. ACM, 2020, pp. 373–385.
- [9] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *European Conf. Machine Learning (ECML)*, ser. LNCS, vol. 1398. Springer, 1998, pp. 137–142.
- [10] R. E. Schapire and Y. Singer, "BoosTexter: A boosting-based system for text categorization," *Mach. Learn.*, vol. 39, no. 2/3, pp. 135–168, 2000.
- [11] K. Nigam, A. McCallum, S. Thrun, and T. M. Mitchell, "Text classification from labeled and unlabeled documents using EM," *Mach. Learn.*, vol. 39, no. 2/3, pp. 103–134, 2000.
- [12] Y. Prabhu and M. Varma, "FastXML: a fast, accurate and stable tree-classifier for extreme multi-label learning," in *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*. ACM, 2014, pp. 263–272.
- [13] K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain, "Sparse local embeddings for extreme multi-label classification," in *Annu. Conf. Neural Information Processing Systems (NIPS)*, 2015, pp. 730–738.
- [14] X. Li, Y. Qu, and H. Yin, "PalmTree: Learning an assembly language model for instruction embedding," in *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*. ACM, 2021, pp. 3236–3251.
- [15] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni, "SAFE: self-attentive function embeddings for binary similarity," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 11543. Springer, 2019, pp. 309–329.
- [16] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 472–489.
- [17] R. Mukherjee, Y. Wen, D. Chaudhari, T. W. Reps, S. Chaudhuri, and C. M. Jermaine, "Neural program generation modulo static analysis," in *Annu. Conf. Neural Information Processing Systems (NeurIPS)*, 2021, pp. 18984–18996.
- [18] E. L. Allwein, R. E. Schapire, and Y. Singer, "Reducing multiclass to binary: A unifying approach for margin classifiers," *J. Mach. Learn. Res.*, vol. 1, pp. 113–141, 2000.
- [19] F. Tai and H. Lin, "Multilabel classification with principal label space transformation," *Neural Comput.*, vol. 24, no. 9, pp. 2508–2542, 2012.
- [20] M. Cissé, N. Usunier, T. Artières, and P. Gallinari, "Robust bloom filters for large multilabel classification tasks," in *Advances in Neural Information Processing Systems 26*, 2013, pp. 1851–1859.
- [21] H. Yu, P. Jain, P. Kar, and I. S. Dhillon, "Large-scale multi-label learning with missing labels," in *Proc. 31st Int. Conf. Machine Learning (ICML)*, ser. JMLR Workshop and Conference Proceedings, vol. 32, 2014, pp. 593–601.
- [22] K. Dahiya, D. Saini, A. Mittal, A. Shaw, K. Dave, A. Soni, H. Jain, S. Agarwal, and M. Varma, "Deepxml: A deep extreme multi-label learning framework applied to short text documents," in *Proceedings of the ACM International Conference on Web Search and Data Mining*, March 2021.
- [23] H. Jain, V. Balasubramanian, B. Chunduri, and M. Varma, "Slice: Scalable linear extreme classifiers trained on 100 million labels for related searches," in *Proc. ACM Int. Conf. Web Search and Data Mining (WSDM)*. ACM, 2019, pp. 528–536.
- [24] A. Mittal, K. Dahiya, S. Agrawal, D. Saini, S. Agarwal, P. Kar, and M. Varma, "Decaf: Deep extreme classification with label features," in *Proceedings of the ACM International Conference on Web Search and Data Mining*, March 2021.
- [25] S. Bengio, J. Weston, and D. Grangier, "Label embedding trees for large multi-class tasks," in *Annu. Conf. Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2010, pp. 163–171.
- [26] J. Deng, S. Satheesh, A. C. Berg, and L. Fei-Fei, "Fast and balanced: Efficient label tree learning for large scale object recognition," in *Annu. Conf. Neural Information Processing Systems (NIPS)*, 2011, pp. 567–575.

- [27] R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma, “Multi-label learning with millions of labels: recommending advertiser bid phrases for web pages,” in *Proc. World Wide Web Conf. (WWW)*. ACM, 2013, pp. 13–24.
- [28] J. Weston, A. Makadia, and H. Yee, “Label partitioning for sublinear ranking,” in *Proc. 30th Int. Conf. Machine Learning (ICML)*, ser. JMLR Workshop and Conference Proceedings, vol. 28. JMLR.org, 2013, pp. 181–189.
- [29] K. Balasubramanian and G. Lebanon, “The landmark selection method for multiple output prediction,” in *Proc. 29th Int. Conf. Machine Learning (ICML)*. icml.cc / Omnipress, 2012.
- [30] K. Järvelin and J. Kekäläinen, “Cumulated gain-based evaluation of IR techniques,” *ACM Trans. Inf. Syst.*, vol. 20, no. 4, pp. 422–446, 2002.
- [31] D. J. Hsu, S. M. Kakade, J. Langford, and T. Zhang, “Multi-label prediction via compressed sensing,” in *Annu. Conf. Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2009, pp. 772–780.
- [32] J. Weston, S. Bengio, and N. Usunier, “WSABIE: scaling up to large vocabulary image annotation,” in *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI)*. IJ-CAI/AAAI, 2011, pp. 2764–2770.
- [33] H. Jain, Y. Prabhu, and M. Varma, “Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications,” in *SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 935–944.
- [34] P. R. Rosenbaum and D. B. Rubin, “The central role of the propensity score in observational studies for causal effects,” *Biometrika*, vol. 70, no. 1, pp. 41–55, 1983.
- [35] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Annu. Conf. Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2013, p. 3111–3119.
- [36] Q. V. Le and T. Mikolov, “Distributed representations of sentences and documents,” in *Proc. 31st Int. Conf. Machine Learning (ICML)*, ser. JMLR Workshop and Conference Proceedings, vol. 32. JMLR.org, 2014, pp. 1188–1196. [Online]. Available: <http://proceedings.mlr.press/v32/le14.html>
- [37] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: pre-training of deep bidirectional transformers for language understanding,” in *Proc. Conf. North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*. Association for Computational Linguistics, 2019, pp. 4171–4186.
- [38] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in *Proc. 2017 ACM SIGSAC Conf. Computer and Communications Security (CCS)*. ACM, 2017, p. 363–376.
- [39] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, “Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned,” *IEEE Trans. Software Eng.*, 2022.
- [40] C. Cai and Y. Wang, “A simple yet effective baseline for non-attribute graph classification,” in *ICLR ’19: International Conference on Learning Representations*, 2019.
- [41] J. Li, L. Wu, R. Guo, C. Liu, and H. Liu, “Multi-level network embedding with boosted low-rank matrix approximation,” in *ASONAM ’19: International Conference on Advances in Social Networks Analysis and Mining*. ACM, 2019, pp. 49–56.
- [42] A. Z. Broder, “On the resemblance and containment of documents,” in *Proc. Compression and Complexity of SEQUENCES 1997*. IEEE, 1997, pp. 21–29.
- [43] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [44] R. Kneser and H. Ney, “Improved backing-off for m-gram language modeling,” in *Int. Conf. Acoustics, Speech, and Signal Processing, (ICASSP)*. IEEE Computer Society, 1995, pp. 181–184.
- [45] S. F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” in *Proc. Annu. Meeting Association for Computational Linguistics (ACL)*. Morgan Kaufmann Publishers / ACL, 1996, pp. 310–318.
- [46] K. Heafield, “KenLM: Faster and smaller language model queries,” in *Proc. Workshop on Statistical Machine Translation (WMT@EMNLP)*. Association for Computational Linguistics, 2011, pp. 187–197.
- [47] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “BYTEWEIGHT: learning to recognize functions in binary code,” in *Proc. 23rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2014, pp. 845–860.
- [48] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 611–626.
- [49] D. Andriessse, A. Slowinska, and H. Bos, “Compiler-agnostic function detection in binaries,” in *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017, pp. 177–189.
- [50] Y. David, U. Alon, and E. Yahav, “Neural reverse engineering of stripped binaries using augmented control flow graphs,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 225:1–225:28, 2020.

- [51] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig, and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.
- [52] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang, "CADE: detecting and explaining concept drift samples for security applications," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 2021, pp. 2327–2344.
- [53] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys*, vol. 49, no. 1, 2016.
- [54] S. Hu, L. Ke, X. Wang, and S. Lyu, "Tkml-ap: Adversarial attacks to top-k multi-label learning," in *Proc. IEEE/CVF Int. Conf. on Computer Vision (ICCV)*, 2021, pp. 7649–7657.
- [55] Q. Song, H. Jin, X. Huang, and X. Hu, "Multi-label adversarial perturbations," in *IEEE Int. Conf. Data Mining (ICDM)*. IEEE Computer Society, 2018, pp. 1242–1247.
- [56] R. L. Castro, L. Muñoz-González, F. Pendlebury, G. D. Rodosek, F. Pierazzi, and L. Cavallaro, "Universal adversarial perturbations for malware," *CoRR*, vol. abs/2102.06747, 2021. [Online]. Available: <https://arxiv.org/abs/2102.06747>
- [57] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *CoRR*, vol. abs/1902.06705, 2019.
- [58] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," in *Annu. Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2020.
- [59] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, " α diff: cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Software Engineering (ASE)*. ACM, 2018, pp. 667–678.
- [60] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Annu. Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2019.
- [61] W. Sun, Y. Chen, Z. Shan, Q. Wang *et al.*, "Binary semantic similarity comparison based on software gene," in *Journal of Physics: Conference Series*, vol. 1325, no. 1. IOP Publishing, 2019, p. 012109.
- [62] P. Bielik, V. Raychev, and M. T. Vechev, "PHOG: probabilistic model for code," in *Proc. 33rd Int. Conf. Machine Learning (ICML)*, ser. JMLR Workshop and Conference Proceedings, vol. 48. JMLR.org, 2016, pp. 2933–2942.
- [63] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proc. 39th ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2018, pp. 404–419.
- [64] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," in *7th Int. Conf. Learning Representations (ICLR)*. OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=Bke4KsA5FX>
- [65] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.
- [66] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *7th Int. Conf. Learning Representations (ICLR)*. OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=H1ersoRqtm>
- [67] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th Int. Conf. Learning Representations (ICLR)*. OpenReview.net, 2019. [Online]. Available: <https://openreview.net/forum?id=H1gKYo09tX>
- [68] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. 33rd Int. Conf. Machine Learning (ICML)*, ser. JMLR Workshop and Conference Proceedings, vol. 48. JMLR.org, 2016, pp. 2091–2100.
- [69] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina, "Code authorship attribution: Methods and challenges," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–36, 2019.