# Software Analysis Through Binary Function Identification

James Patrick-Evans

Submitted in fulfilment for the degree of
Doctor of Philosophy

Information Security Group
Royal Holloway, University of London

# Declaration of Authorship

The work presented in this thesis is the result of original research carried out by myself, in collaboration with others, whilst enrolled at the Centre for Doctoral Training in Cyber Security in the Department of Information Security as candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

James Patrick-Evans
December, 2021

# Acknowledgements

I wish to start by expressing my gratitude to my supervisor Johannes Kinder whose guidance and mentorship has been essential to the completion of my PhD. I would like to thank him not only for years of in-depth technical discussions but for being joyous and willingness to debate solutions to difficult research problems. I am especially grateful to both Johannes Kinder and my advisor Lorenzo Cavallaro for their wisdom, time, and good company they gave to me. It has been a wonderful journey and without their expert knowledge in very specific fields of research, this PhD would not have been achieved.

I would like to thank my lab colleagues for their continuous insightful input into our research group. In particular, I would like to thank Duncan Mitchell, Claudio Rizzo, and Blake Loring for their support and cheerful banter throughout this time. I would like to thank Hernán Ponce for proof reading sections of this thesis and Moritz Dannehl for his discourse and collaboration in our work on XFL.

I am grateful to everyone at the Centre for Doctoral Training (CDT) in Cyber Security at Royal Holloway who have supported me and gave me a comprehension into a vast number of intricate research topics. I'm especially thankful of the CDT for supporting my excursions around globe to present and discuss my research with world leading experts and visit spectacular locations. Most importantly, I thank all CDT members for their companionship and conviviality who made this period so enjoyable. Ap-

preciation also needs to be given to my colleagues at Kings College London and Universität der Bündeswehr München for their continuous support and good spirits.

Finally, I would like to thank my mother for all the sacrifices she made during my childhood in striving for the betterment of my education, without which, I would most certainly never have embarked on a PhD.

# Abstract

Executable binaries are made up of functional components interacting with each other and the operating system they run on. When high-level source code is compiled into executable binaries, information on the name, size, location, and type of these functional components is included in the executable through the use of *symbols*. Most software distributed today that is compiled into machine code is released without this *symbol* information i.e., they are *stripped*. This makes understanding and analysing binary software very difficult due to the lack of recognisable information in a structured and ordered manner.

In this thesis, we propose new techniques used to recover the names of functions in *stripped* binaries. We explore problems inherent in recovering textual information in the large *label space* associated with naming functions and develop deep-learning embeddings for both binary functions and their names. Furthermore, we demonstrate how symbol name information can be used to aid the exposure of previously undiscovered software bugs by injecting faults in the high-level logic of client USB kernel drivers.

We design a scalable approach for symbol recovery that uses static and symbolic program analysis to extract high-level features from machine code. These features are then used to learn the structure of how binary *code* and *data* interact with each other to infer name information from functions in executables. We build a toolkit, *DESYL* (DEbug Symbol Learning), that

is able to modify stripped executable binaries and add symbol information using machine learning models learnt over a very large dataset. Finally, we develop *USBDT* (USB Driver Testing), our tool for hooking known kernel functions and using selective symbolic execution to analyse Linux USB kernel drivers. Our work extends QEMU to build a software defined virtual USB device used to analyse the Linux USB stack and helped develop two previously unreported mainline Linux kernel zero-day exploits.

# Contents

# List of Figures

# List of Listings

# List of Tables

# 1 Introduction

Modern software has adapted to accommodate the demand for better solutions to more problems across an ever-increasing set of devices. The exponential increase in computational performance from Moore's Law and the prevalent access of computing devices has driven the need for more advanced software to fill the performance and storage capacity now available in Common-Off-The-Shelf (COTS) devices. Ubiquitous access to small, cheap, and powerful systems has compelled manufacturers to fill new simple devices with complex functionality in an attempt to be more user friendly and altogether *smarter*. This ever growing number and type of devices has resulted in a marked increase in the security risk posed from their deployment. The analysis and security of software that drives these devices is therefore of paramount importance.

Indeed, much has been done in the history of security for computational devices, however it is a constant arms race between defences and contemporary attack methods. The race has driven changes in the design of hardware to enforce separation of memory regions and in software to create well established principles to safely allow multiple users running different processes in different security domains. Despite an abundance of defence mechanisms built into modern hardware and software, the majority of consumer grade devices are routinely exploited. Techniques that provide low-level programming languages strong or complete guarantees for

spatial and temporal memory safety [111, 110], fault isolation, and the separation of information flow [159, 52], incur too high a performance overhead for them to be used on consumer devices in the real-world. Alternatively, applications may present vulnerabilities not only through memory corruption, but through external environmental factors and the mishandling of data. In mishandling data, sensitive information may leak from a high security context to that of a lesser privilege and cross the bounds of a security model. Similarly, data may be constructed by an attacker to manipulate a program to carry out unintended actions in a *data-only* attack; a style of software attack that will succeed even in spatial and temporal memory correct software. These methods of insecurity drive a need for the security analysis of modern-day software that is relevant now more than ever.

Even on modern devices, software is still written in low-level programming languages such as C/C++ or built upon abstractions of them. Whilst much work has been done to move programs into higher-level programming languages that offer certain security guarantees such as Rust, Python, JavaScript, Lua, or Go, all computers execute Instruction Set Architecture (ISA) specific machine code and are blissfully unaware of any higher-level constructs available in their respective languages. Security failures are still possible in the instrumentation of these higher-level languages and until hardware moves away from processing low-level code, security failures may still flow from the interpretation or compiled execution of them. WebAssembly, for example, is an established compilation target designed to be widely deployable and offers security guarantees on memory safety that has recently been shown [92] to be vulnerable to primitive stack-based buffer overflow attacks.

The typical methods for finding security issues in modern software is through performing static and dynamic analysis, source code auditing, reverse engineering, and fuzzing. The approach taken depends on the level

of information presented to the person analysing it; source code auditing and white-box fuzzing cannot be done without access to the source code. In light of the large number and large scale of analyses needed to be carried out, new deep learning approaches that combine static analysis and machine learning have attempted to develop Artificial Intelligence (AI) to automate software analysis. Modern static analysis tools have been used to detect critical, previously unseen bugs in source code, but they cannot reliably detect carefully crafted malicious code or complex application-specific vulnerabilities. When carried out by a human, source code auditing is tediously time-consuming and error prone. Where the source code of software is unavailable, an analyst must audit the software by reverse engineering the corresponding executable binary. This process is even more time-consuming and is often exceptionally more difficult.

As a result, one of the most common methods for discovering new bugs is through dynamic analysis and fuzzing. Fuzzing approaches may be white-box, grey-box, or black-box, depending on the level of information with regards to the software's source code that is available. Black-box fuzzing is the most difficult of the three, as no information can be used by an analyst to steer or control the program prior to execution other than random or generated inputs. White and grey box fuzzing allow full or partial information with regards to the software's source code and allows a security analyst to focus their fuzzing efforts on relevant pieces of the software's components or generate inputs that explore new execution paths. Furthermore, many different types of fuzzers exist, with the most common being mutational, generational, grammar-based, and symbolic fuzzers. Each of these categories have their own strengths and weaknesses, with some modern approaches combing multiple types to incorporate the advantages of each individual type. For one to reduce the overall security risk, we need to develop new automated methods of software testing that can easily scale to meet the demands of the marketplace.

Although there has been much development in source code based static analysers that are capable of detecting bugs in large software repositories, automated binary-only analyses are limited in their effectiveness without additional user input.

## 1.1 Reverse Engineering

Reverse engineering is useful not only for finding vulnerabilities in software, but is used extensively in malware analysis, commercial software audits, software debugging, and exploit development. It is a process of understanding how software components interact and conform to carry out the desired functionality of a program. Aside from security analysis, reverse engineering has been used for detecting patent infringements in released code that violate GPL licensed software and no doubt has also been used for copying commercially sensitive methods. Typically, reverse engineering methods are used to understand how, potentially obfuscated, executable binaries operate. When software engineers develop and debug programs, *symbols* are used to relate objects in memory back to the software's source code. This allows the developer to get a better understanding into the current execution properties by reading the source code in a higher-level language.

When software is released for distribution, *symbols* are usually removed to reduce the file size and speed up execution, as well as impeding reverse engineering if the software is released for commercial or malicious purposes. Many software security products that provide spatial memory safety through dynamic binary rewriting or intraprocedural security analyses rely on known function boundaries through *symbol* definitions or their boundaries being recovered for *stripped* binaries. Products often

rely on IDA Pro[1] to recover function boundaries before performing their own additional analysis [30][73][117]. For Windows binaries, *symbols* may be downloaded in Program Database (PDB) files through Windows Symbol Servers, however, only public symbol information is included, with a substantial set of private symbols omitted.

To impede reverse engineering, commercial software and malware are typically obfuscated to further hide their functionality from users. Therefore, it is often extremely difficult to reverse engineer and understand a program's intended purpose when it is presented as a flat region of memory with little distinguishable features. Anti-reverse engineering techniques are also commonplace in Digital Rights Management (DRM) software to protect its distribution or the content it delivers. Similar to the arms race between software attacks and defences, new reverse engineering tools aim to overcome the latest approaches to hiding the true content from the user.

## 1.2 The Problem

The problem in the current security landscape is that vulnerabilities exist in the majority of software and advanced defence mechanisms are too costly to implement, not sufficient, or not applicable. This has led to large collections of vulnerable software running on the majority of computer systems in the world.

Current methods for the security analysis of binary code are slow or inefficient, and sometimes completely ineffective. When considering the analysis of kernel drivers, of which many are written in C/C++, most methods lack the ability to model the full spectrum of interactions between hardware, the kernel, and the user. Most fuzzing tools today focus

[1] `https://hex-rays.com`

on analysing user space applications where the bounds and interactions of each program can be more easily defined. This has led to a large set of software that exhibits more complex interactions, executes in the highest privilege level, and is installed in the majority of devices. This software set is a ripe target for exploitation as it is less thoroughly analysed, and its binaries are compiled from unsafe, low-level programming languages.

When performing static or dynamic software analysis, the number of feasible execution paths typically grows exponentially in relation to its size; however, unbounded loops may lead to infinitely many paths. This fundamental problem, so called the *path explosion* problem, impedes the security analysis of executables with the increasing number of instructions executed. Modern techniques used in white-box fuzzing focus on small, interesting sections of programs in order to target specific code regions that would not otherwise be possible to reach or too high an overhead to analyse effectively. The same approach cannot be applied to black-box fuzzing and thus without reverse engineering the interesting components, one cannot effectively analyse closed source proprietary software.

The goal of this thesis is to build automated systems that analyse compiled binaries and recover symbol information for use in binary analysis. This information includes the name and size of function names and data structures present in the binary, and the relevant source code file associated with them. While it would be most useful to recover all symbol information, this thesis focusses on recovering the names of functions. Automatically inferring textual name information of components in *stripped* binaries may enhance existing binary analysis techniques and thus improve the contemporary security landscape. It is useful, but not limited to, reverse engineering, patch code analysis, code clone detection, vulnerability detection, software hardening through binary rewriting, exploit development, fuzzing, binary only software debugging, and malware analysis. Any tools or techniques developed must be able to scale to large quantities

of programs used in real-world scenarios. This thesis develops techniques that provide the ability to quickly pre-process a stripped binary to identify known similar components learnt from a large collection of open source or previously reverse engineered binaries.

## 1.3 Challenges

One major problem when reverse engineering software is the lack of portability in the work done to transfer the knowledge to another binary. A field of research exists for the special use case whereby sequential updates to the same software exist with minor changes. In this case, an analyst looks at the difference between the binaries using tools such as BinDiff [162, 60] or BinNavi[2] to port information across to the new binary. The same technique is also used in *patch code analysis* to identify changes in updated versions which may reveal fixed vulnerabilities in older versions. This may be a difficult problem as there are large differences between compilations of the same source code with different optimisations or compilers.

The problem of automatically naming components in stripped binaries is exacerbated by compilers and their myriad of optimisation combinations which can manipulate the compiled code in unexpected ways. In the following paragraphs we briefly discuss the core components of this challenge.

*Each compilation is different*. Machine code generated from different compilations of the same source code is vastly different between different compilers. Large differences also commonly occur with different optimisations enabled when using the same compiler. Early works in function identification used simple heuristics and common function prologue and epilogue signatures stored per compiler. DynInst [27], a dynamic binary re-writer,

[2] `https://github.com/google/binnavi`

uses this method to identify function boundaries in its instrumentation of several run-time security measures.

*Non-contiguous functions*. Procedures declared in high-level source code are typically defined by a contiguous set of alphanumerical characters. During compilation, gaps of null bytes, random data, or machine code instructions with no effect on execution, may be inserted as padding to ensure code aligns along pages in memory or CPU cache lines. In doing so the compiler optimises low-level ISA specific memory loads for frequently occurring code and reduces run-time overhead. When analysing a binary function given the predetermined set of bounds, *start* and *end*, not every byte belongs to the function. Gaps may also be created by large data structures, jump tables, or instructions from completely different functions. Techniques that minimise the size of compiled binaries [69] can create the scenario of functions sharing code, and as a result, create non-contiguous functions.

*Unreachable functions*. Binary decompilation is usually carried out by lifting machine code into assembly code starting at an entry point and recursively following control flow paths, or through a linear sweep. When decompiling machine code using the recursive method, some functions may be present in the binary that are never called and are thus *non-reachable*. The presence of unreachable functions may influence their analysis, or they may be missed entirely during the detection of function boundaries. Therefore, it may be impossible to predict the names of functions one cannot detect. Such *non-reachability* may be changed depending on the level of optimisation employed by the compiler, for example, Figure 1.1 shows the function `fac` that computes a number's factorial. When the program is compiled with `clang` in a highly optimised manner (`-O3`) the result is precomputed and placed inline with the function `fac` still present in the binary, even though it is never called. Determining the reachability of functions is necessary in black-box security mechanisms, for example, en-

```c
uint64_t fac(uint64_t x)
{
    if (x==1) return 1;
    return x * fac(x-1);
}
int main(int argc, char
    **argv)
{
    printf("%llu", fac(5));
    return 0;
}
```

(a) C source code of a program computing 5 factorial.

```
<0x19b0.fac>:
  ...
<0x1040.main>:
  sub rsp, 8
  mov esi, 0x78
  lea rdi, str._llu
  xor eax, eax
  call sym.imp.printf
  xor eax, eax
  add rsp, 8
  ret
```

(b) Assembly generated by compiling the source code in Figure 1.1a with `clang -O3`.

Figure 1.1: Source code of the `fac` function and its compiled assembly code. `fac` is never called in the assembly code as its result is precompiled with a value of `0x78` or 120.

forcing CFI [1], XFI [54], or memory safety through binary rewriting.

*Multi-entry functions.* Source code for high-level languages typically define a single entry point to a procedure, with the possibility of having multiple return locations. However, when compiled, functions may have multiple entry points due to optimisations made by the compiler and may correspond to multiple symbol definitions. An example of this behaviour occurs when passing specific arguments that perform a subset of the procedure's logic compared to the full procedure. This implements slight variations of a complex function using a simplified interface.

*Functions may be removed.* Compilers perform function inlining to reduce the size of the compiled binary and speed up execution by removing the need to create and destroy a stack frame. This is typically done for small functions and completely removes the functions existence in the compiled binary. When comparing to source code, completely inlined

functions may be impossible to detect.

*Naming Functions*. When training machine learning models to learn the names of functions, one needs to overcome differences in the label space as different authors pick *subjectively* different names for the same function. This may be due to programmer labelling style, naming conventions, or project specific requirements. When treated as a multi-class classification problem with each unique function name as a separate class, one faces a large number of classes and little training samples per class. These "tail labels" prove difficult for model generalisation and accurate prediction on previously unseen data.

## 1.4 Our Approach

This thesis develops a novel tool and binary analysis framework, *DESYL*, for analysing stripped binaries and inferring function names. Our analysis framework aims to give coherent results across different compilations of the same or similar source code and engineers a feature vector for use in machine learning pipelines. We primarily explore the state-of-the-art methodologies in representing and learning the structure of binary code. *DESYL* does this by learning the importance of pairwise and multivariate relationships between symbols in binary code expressed as feature functions. We design feature function types and apply it to millions of unique features to learn the interactions between code and data on a large real-world dataset. With plugins for major reverse engineering tools such as Ghidra and Radare2, *DESYL* can quickly pre-processes *stripped* binaries to predict the names of functions along with their boundaries.

We extend our approach to aiding the reverse engineering process by creating a distributed representation of binary code and use it to label functions. Our machine learning pipeline is able to learn key character-

istics and assign a human readable set of labels to functions that are useful to reverse engineers and security analysts. Our embedded representation of binary code is built using a deep autoencoder and captures significantly more information than what is possible using traditional machine learning techniques. Our dense, vector representation of binary code is created in an unsupervised approach such that similarly labelled functions appear close in the new embedded representation.

Finally, we investigate new fuzzing methods that make use of *known* or *recovered* symbols and can exercise deep paths in the operating system model. We build a new system capable of emulating hardware USB devices and use it with selective symbolic execution to precisely fuzz high-level USB device drivers in the Linux kernel. Our approach is significant due to the difficulty of analysing kernel drivers that depend on hardware for interactions as existing approaches are limited in their effectiveness of targeting the core logic in driver code. A simple and common approach is to fuzz input from the hardware by using devices such as the Facedancer [63] board. Such a device allows an analyst full control over a particular *bus* to imitate, mutate, and inject faults into operations. The limiting factor with this approach is that the fuzzing device is operating at a level too low to effectively stimulate code paths in higher level modular drivers. For example, a USB over Ethernet driver imports common functionality from core USB and networking kernel subsystems and is not concerned with bit-flips present in raw meta-packets. The drivers imported interface simply presents a subsystem that exports a passed or failed state, and can be easily handled by even novice programmers.

We structure the remainder of this thesis into four chapters. Chapter 2 looks at the relevant background material on program analysis, machine learning, the USB protocol, and the Linux kernel's implementation of USB. Chapter 3 brings forward a model that uses program analysis and machine learning to identify function names in stripped binaries. Chapter 4 intro-

duces new deep learning vector representations of binary code that may be used for generic purposes in other machine learning pipelines. Chapter 5 looks into the fundamental problems of identifying function names in stripped binaries and describes theoretical limitations if applied in practise. A function labelling approach is developed that is more general and able to create new function names not seen during training for unseen stripped binaries. Chapter 6 then presents a new model of analysing USB kernel drivers that depends on symbol information for hooking relevant kernel subsystems. Such a model would be applicable to closed source kernel drivers where symbol information may be recovered or inferred using techniques presented in prior chapters.

## 1.5 Contributions

In this thesis, we first discuss the relevant background (Chapter 2) and then make the following contributions:

- We present a new approach for identifying function names in stripped binaries (Chapter 3). We discuss our method of recovering symbol information based on the structure of executable binaries to assist in the reverse engineering process. This chapter comprises an extended version of work first published in the Annual Computer Security Applications Conference (ACSAC) 2020 [119].

- We present an approach for generating embedded representations of machine code (Chapter 4) and prove that it is superior to prior approaches. This chapter provides an extended look at work first published in the *ACSAC 2020* and work in submission to IEEE Symposium on Security and Privacy (S&P) 2022 [118, 120].

- We present a design to labelling symbol information in stripped binaries (Chapter 5) and prove that it is superior to prior approaches. This chapter provides an extended look at work in submission to *S&P 2022* [120].
- We present a novel approach for finding software flaws in USB drivers targeting the Linux kernel (Chapter 6). This chapter looks the hardware and software security model presented by the Linux kernel, identifies shortcomings in existing approaches, and reports two new zero-day vulnerabilities (CVE-2016-5400 and CVE-2017-15102) as discussed in the IEEE Workshop on Offensive Technologies (WOOT) 2017 [118].

# 2 Background

This chapter gives the necessary background information required to understand the remainder of this thesis. We start by discussing the program analysis technique symbolic execution (Section 2.1) and explore the individual strengths and weaknesses for different variants. Secondly, we give an overview of the machine learning techniques used to create distributed representations (Section 2.2) on which material in this thesis builds upon. Thirdly, we explain the fundamentals required for understanding Conditional Random Fields (Section 2.3) that is relevant to structured prediction of *symbols* in executable binaries. Then we cover software abstractions necessary to understand the USB protocol and the USB stack as implemented in the Linux kernel (Section 2.6). Finally, we give an overview of security mechanisms employed by the Linux kernel (Section 2.7) that have to be overcome by modern attacks and briefly discuss the powerful SystemTap kernel debugging infrastructure.

## 2.1 Symbolic Execution

Symbolic Execution (SE) is a program analysis technique based on the idea of interpreting a program on *symbolic* instead of concrete data and having instructions manipulate symbolic expressions instead of concrete values. This allows one to explore all control flows through a program contingent

on symbolic inputs at any point during execution of the program. This is typically done by replacing some program inputs with symbolic values and using a special interpreter to execute the program. Conditional instructions are interpreted by computing the symbolic expressions for the condition and forking the program state if both outcomes are feasible. The feasibility of symbolic expressions are computed with a *constraint solver* by checking the satisfiablity of the branch conditions from the start of execution to the current contional instruction. Symbolic execution suffers from *path explosion*, the problem of exponential growth in the number of paths through a program. The complete exploration of paths through a program is therefore generally infeasible for real-world programs.

Dynamic Symbolic Execution (DSE) (also called concolic execution) extends this idea by also executing the program concretely at the same time. It does this by maintaining a *symbolic state* which maps program variables to symbolic expressions. When executing conditional instructions dependent on symbolic inputs, new conditions are added in the form of *path conditions*. At the end of the program execution, the *symbolic state* can be used to negate path conditions along branches in the previous execution trace to generate new constraints. The advantage of DSE over SE lies in the fact that where symbolic conditions cannot be satisfied or functions external to the program under test need to be analysed, DSE is able to concretise symbolic variables whilst maintaining its correctness.

Selective symbolic execution, a technique introduced in $S^2E$ [35], reduces path explosion by switching between concrete and symbolic execution modes at module boundaries. It improves over DSE by reducing the amount of code that needs to be executed symbolically, thus reducing path explosion. $S^2E$ uses QEMU [16] and KLEE [31] to provide a symbolic virtual machine that allows one to inject symbolic data into arbitrary memory addresses for full-system emulation. Figure 2.1, for example, depicts the analysis of a library on top of an operating system (OS) kernel un-

der S$^2$E. The function `libFn` is called from the application function `appFn`. Upon doing so the execution domain changes and the library function is executed symbolically. During `libFn`'s execution it invokes the system call `sysFn`, which is outside of the symbolic domain, and concretises function arguments and data that pass the boundary. In S$^2$E, each code block (translation blocks from QEMU's Tiny Code Generator) that manipulates symbolic data is compiled to LLVM instructions and symbolically executed by the KLEE symbolic execution engine. S$^2$E minimises the path explosion problem inherent with symbolic execution by switching execution domains on the entries and exits of a target module's boundaries, latently converting symbolic expressions to concrete data and vice-versa.

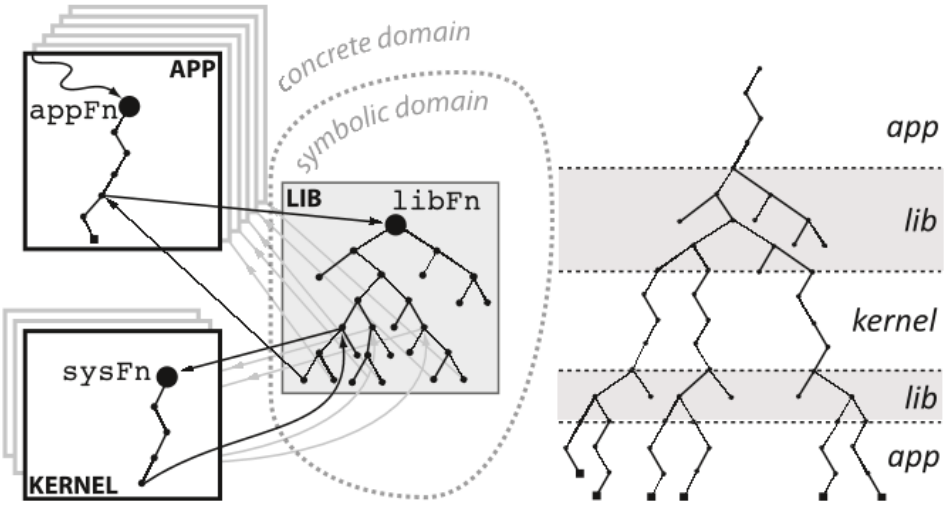



Figure 2.1: Execution diagram showing the multi-path/single-path execution through three different modules (left) when using S$^2$E and the resultant execution tree (right). Shaded areas in the execution tree represent the symbolic execution domain from the library under test, while white areas execute in the concrete domain. The diagram is taken directly from S$^2$E documentation and is used in the original paper [35].

A plugin system allows for interactions with the virtual machine and provides an interface to processes running in the guest OS. Typically, the stock-plugins either collect analysis information about execution traces or provide a search strategy for exploring the target module.

## 2.2 Distributed Representations

*Local representations* of data points used for computational tasks is the most simplistic rendition of them whereby each data point is represented by one computational element, e.g., one-hot encoding. *Distributed representations* however, are encodings of data points (typically vectors) that are distributed over many computing elements, and each element is involved in representing different data points. Their use in contemporary research has helped machine learning algorithms achieve better performance than local representations for their corresponding tasks. Distributed representations of words, for example, drastically increase performance of natural language processing tasks by providing a vector space that captures their semantic meaning and groups similar words. These techniques may be used to create a numerical representation of data that inherently lacks a numerical basis and are incredibly powerful methods leveraged in modern deep learning approaches. Specifically, categorical data that would previously have been represented using one-hot-encoding may be represented by dense vectors whose relative distances in the embedded vector space are meaningful relative to other categories.

This section covers the foundations of distributed representations starting from the original Word2Vec paper by Mikolov et al. [104] and describes contemporary advancements in the field leading to distributed representations of assembly and binary code.

### 2.2.1 Word2Vec

Historical methods used for computing continuous representations of words and Natural Language Processing (NLP) tasks focused around Latent Dirichlet Analysis (LDA) and Latent Semantic Analysis (LSA) [158]. Both techniques were used extensively in topic modelling and sentiment analysis; however, recent work has shown that neural networks perform significantly better [105, 160] when preserving linear regularities among words. Furthermore, modern training methods for neural networks allow them to express millions of words whereas LDA becomes very computationally expensive on large real-world datasets.

Limitations inherent in these classical approaches and the advancement in Neural Network based techniques drove the design of Neural Network Language Models (NNLMs). Initial NNLMs [18, 103, 154] consisted of a simple design where one-hot-encoded vectors that represented words from a fixed corpus were used to predict sentence sentiment or the next word with a Recurrent Neural Network (RNN) architecture. Mikolov et al. [102] identified that most of the complexity in contemporary Neural Network Language Models (NNLMs) of the time was caused by the non-linear hidden layer in each model. Therefore, they proposed two new log-linear models in their work on Word2Vec [104] that would minimise the computational complexity and could be trained on data more efficiently: Continuous Bag of Words (CBOW) and the Continuous Skip-gram model.

*Continuous Bag of Words (CBOW).* The CBOW model is similar to the feed-forward neural network language model architecture in that a projection layer is shared between all words and the order of words does not influence the projection. However, unlike the standard bag-of-words model, it uses a distributed representation of the context by using four previous words and four future words to correctly classify the current (middle) word.

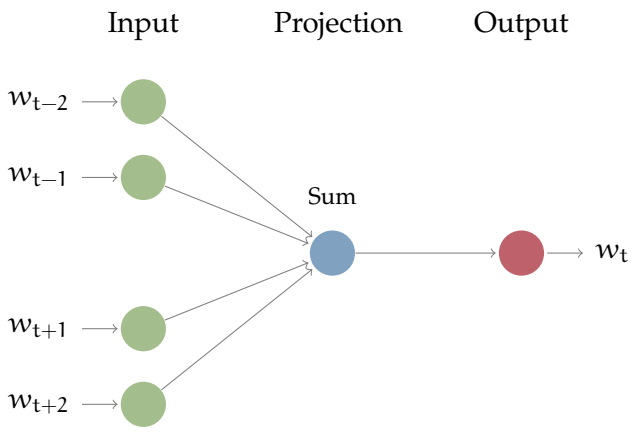



Figure 2.2: The CBOW model architecture predicts the current word based on the context words. The diagram shows how a sliding window of 2 is used to sum the context words in the projection layer.

*Skip-gram Model*. The Continuous Skip-gram model aims to find word representations that are useful for predicting surrounding words in a sentence or a document. It reverses the CBOW setup and trains a feed-forward neural network to predict context words from a given target word. Both the CBOW and Skip-gram model aim to minimise the training complexity and perform differently on different NLP tasks. Their authors report that the Skip-gram model outperforms CBOW and NNLM models at semantic tasks but is outperformed by CBOW in syntactic related tasks.

Furthermore, Mikolov et al. [104] describe *negative sampling* as an alternative training method for the Skip-gram model that results in a significant speed up and better representations. The technique replaces the objective function in the Skip-gram model with a new negative sampling function, NEG, that trains a model to distinguish a target word from a random sample of k negative examples. This technique allows a significant speedup in training NNLMs and gives more regular word representations

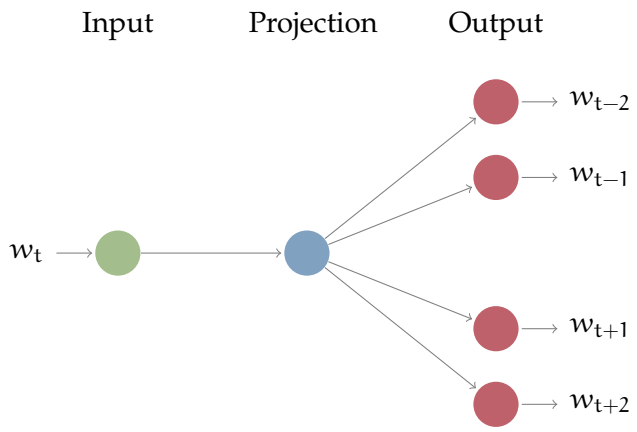



Figure 2.3: The Skip-gram model architecture predicts surrounding context words given the current word.

across repeated experiments.

A major result of Word2Vec brought continuous word vectors that captured semantic information of words that allowed its vector representations to express abstract relationships. The pinnacle result being that the $\mathrm{vector}(\textit{king}) - \mathrm{vector}(\textit{man}) + \mathrm{vector}(\textit{woman}) \approx \mathrm{vector}(\textit{queen})$.

### 2.2.2 Paragraph2Vec

Despite the popularity of models based on bag-of-words such as *Word2Vec*, these techniques ignore the ordering of words and their semantics. For example, Le et al. [91] report that in their Word2Vec model, the vectors for *strong*, *powerful*, and *Paris* are equally distant. Paragraph2Vec [91] proposed an unsupervised algorithm that learns a fixed-length vector representation of variable length text, such as paragraphs or whole documents. The rationale behind including paragraphs (or documents) as input nodes is based upon considering them as another context. The model has the potential to overcome weaknesses of bag-of-words models and their re-

sults on text classification and sentiment analysis out-perform prior approaches.

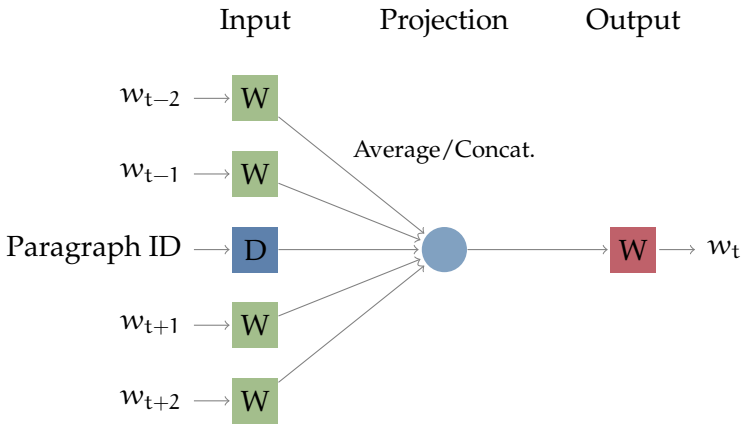



Figure 2.4: Overview of the Paragraph Vector Distributed Memory (PV-DM) model presented by Paragraph2Vec. Paragraph IDs are used to look up paragraph vectors in **D**. Word vectors are looked up by their associated word in **W**.

Paragraph2Vec uses the Paragraph Vector Distributed Memory (PV-DM) model as depicted in Figure 2.4 that associates each paragraph in a dataset with a unique identifier. The identifier indexes the row of the matrix **D** that is used to store paragraph vectors. Similarly, the matrix **W** is used as a look up table to store the word vector for the word $w_i$. Paragraph and word vectors are initialised to random values and learnt as the model is trained. Stochastic Gradient Descent (SGD) is used to update the corresponding vectors on the error calculated through back-propagation.

The authors report that the PV-DM model consistently out-performs CBOW models and concatenation of vectors is superior to using the sum method.

### 2.2.3 Asm2Vec

Asm2Vec [48] describes an approach to producing vector representations of binary code based on Paragraph2Vec [91]. The model learns vector representations for assembly code that captures lexical semantic relationships between assembly operations, operands, and functions. The technique models binary functions as sequences of abstract tokens, iteratively learning vectors for each assembly operation, assembly operand, and function, using back-propagation in a similar design to paragraph vectors and abstract word tokens.

To describe binary functions as sequences of assembly code, Asm2Vec first assumes the function boundaries are known, and then generates sequences of code executions based on the recovered Intraprocedural Control Flow Graph (ICFG). In doing so, the corresponding vector representations will be learnt from data that more accurately describes the functionality of each function. To generate sequences of assembly code, Asm2Vec performs the following techniques on the recovered ICFG:

*Selective Callee Expansion*. To mitigate function inlining, small callee function's assembly code is selectively inlined into the caller function's assembly sequence. Procedures are selectively inlined based on the ratio of size between the caller and callee to include code from callees without being over-represented by them.

*Edge Coverage*. Random edges on the ICFG are sampled and their assembly instructions of each basic block are added to the sequence. This method may include control flow paths that are never taken under a real execution.

*Random Walk*. Random walks from a function's entry point are calculated to simulate real execution paths through each function. Sequences of assembly instructions from the executed basic block are adjoined to form a sequence.

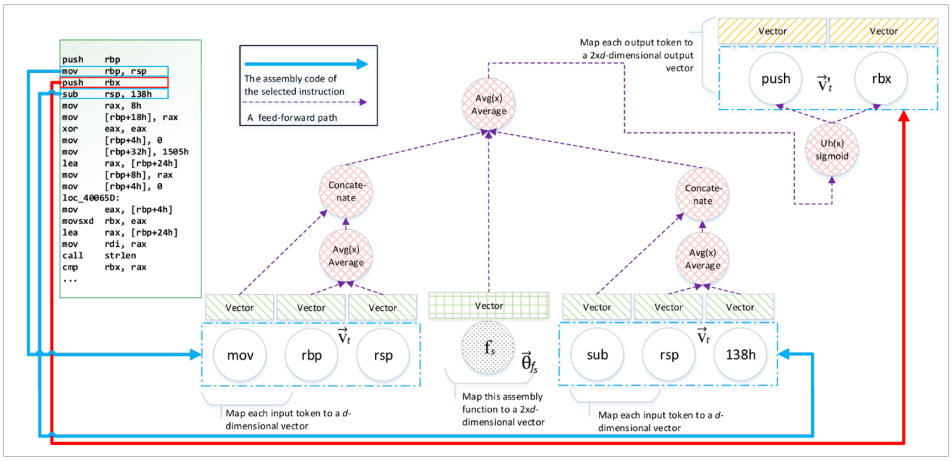



Figure 2.5: Depiction of the Asm2Vec neural network model for assembly code. Sequences of assembly instructions are mapped to vectors that are used to predict a target instruction from the surrounding context instructions. The diagram is directly taken from the original author's publication [48].

Once a sequence of assembly instructions is generated for each function, the model in Figure 2.5 is used to generate vectors for each assembly token and function vector. A sliding window is used to predict a target assembly instruction from the surrounding context instructions. Separate vectors to represent assembly operations, assembly operands, and function vectors are initially randomised and learnt as the model is trained. To support assembly instructions with arbitrary operands the operand vectors are averaged before being concatenated with the operation vector. The vectors for the context instructions are then averaged with the vector of the current function and used to predict the target instruction. By iterating over the sequence of instructions, backpropagating the relative errors between the predicted target instruction and the true vector for the target instruction, function and instruction vectors are learned. Function vectors for previ-

ously unseen functions may then be generated by fixing the vectors for assembly instructions and only propagating errors to the function vector.

Similar to the PV-DM model, Asm2Vec's function vectors are used as a *distributed memory* to help the model predict the target instruction from its context. The embedding produced out-performed many existing techniques for function identification and vulnerability detection.

### 2.2.4 Graph2Vec

Graph2Vec [112] is a contemporary approach to learning a fixed size distributed representation of arbitrarily sized graphs. The corresponding embeddings can be used for downstream tasks such as graph classification and clustering. The intuition behind graph2vec is to view an entire graph as a document with rooted subgraphs around every node in the graph as words that the compose the document. A graph kernel is then chosen to identify and compare rooted subgraphs; the authors use the labelled Weisfeiler-Lehman (WL) [136] kernel. The WL kernel is used to compare rooted subgraphs for every node on each graph in a corpus. A skip-gram model is then used to learn neural embeddings for each graph as depicted in Figure 2.6. Graph2Vec significantly out-performed other graph-based embeddings [156] and substructure representations such as node2vec [65] and sub2vec [2].

## 2.3 Conditional Random Fields

Structured prediction tasks deal with inferring large numbers of variables that depend on observed inputs and at the same time are inter-dependent on each other. These tasks combine multivariate classification methods and graphical models to perform prediction on large sets of structured input features. Conditional Random Fields (CRFs), first described in Laf-

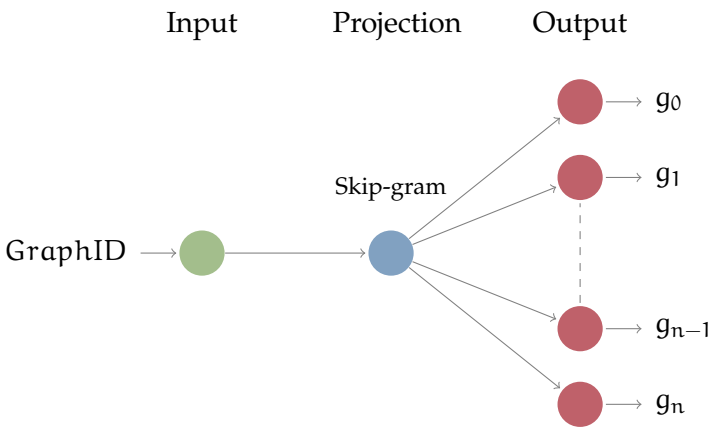


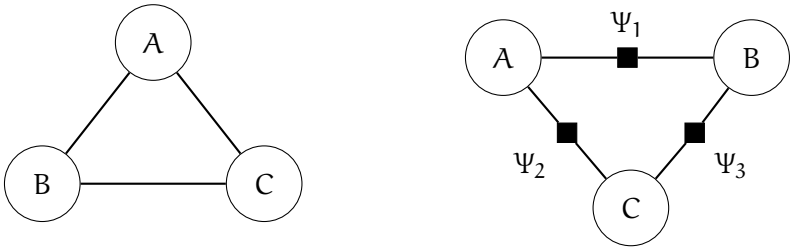

Figure 2.6: Overview of the Graph2Vec model. Graph IDs are used to look up graph vectors for unique graphs. Subgraph vectors are looked up by their associated rooted subgraph for the nodes present in the graph corresponding to GraphID. Rooted subgraphs are identified using the labelled Weisfeiler-Lehman graph kernel.

ferty et al.[89], are a form of probabilistic graphical model widely used in computer vision [86, 15], natural language processing [135, 134, 123], and bioinformatics [131, 150] to learn the structure of random variables based on observed data. We structure this section by briefly giving an overview of probabilistic graphical models before focussing on Conditional Random Fields and their inference algorithms.

### 2.3.1 Probabilistic Graphical Models

Probabilistic graphical models infer a *set* of *unknown random variables* $\mathbf{y} = y_0, y_1, ..., y_n$ from a *set* of *observed variables* $\mathbf{x} = x_0, x_1, ..., x_m$. The models are graph-based representations of probability distributions. Figure 2.7 depicts a graphical model in which nodes correspond to random variables and the edges represent statistical dependencies. The structure of the

graphical model represents the factorisation of the probability distribution which can be defined as the product of fully connected node cliques. In that way, any distribution can be described with a graphical model; a distribution with completely independent variables being represented by a graph with no edges. Graphical models' main advantage is in describing complex distributions with many inter-dependencies as they provide inference algorithms for computing marginals and conditional probabilities.




(a) Graphical model depicting 3 variables with inter-dependencies.

(b) A valid factor graph for the graphical model described in (a).

Figure 2.7: An undirected graphical model with three variables. An edge between two vertices represents a conditional dependence. The corresponding probability distribution defined by this graph factorises $p(A, B, C) = \frac{1}{Z}\Psi_1(A, B)\Psi_2(A, C)\Psi_3(B, C)$ where $Z$ is the normalisation constant and $\Psi$ are functions over the variables.

Formally, a graphical model is a collection of vertices and edges $G = (V, E)$, where vertices are the variables $X \cup Y$. Given a set of cliques $A \subset V$, the distribution over the variables is written as:

$$p(Y, X) = \frac{1}{Z} \prod_{A} \Psi_A(X_A, Y_A) \tag{2.1}$$

where $\{\Psi_A\}$ are factors mapping variables in a clique to $\mathbb{R}^+$, and $Z$ is the normalisation constant:

$$Z = \sum_{X,Y} \prod_{A} \Psi_A(X_A, Y_A) \tag{2.2}$$

The insight of graphical models is to represent a distribution over many variables as a product of local functions that each depend on a smaller subset of variables. They can be either *undirected* or *directed*, *single variable*, *sequence-based*, or *graph-based*, *discriminative* or *conditional*, and *dynamic* or *relational*. The simplest form of graphical model is Naive Bayes in the generative case or Logistic Regression in the conditional case. Linear-chain CRFs can be intuitively thought of as conditional Hidden Markov Models and are an application of Logistic Regression applied to sequential data. In the same way General CRFs are an application of Linear Chain CRFs to general graph-based data. Furthermore, Markov Models are a form of probabilistic graphical model used to model graph-based data with unknown variables only, rather than factors and variable nodes. All these models follow the general equations as defined in Equation 2.1 and Equation 2.2.

Much of the existing work with graphical models has focused on *generative* models, such as Hidden Markov Models, that explicitly model the joint probability distribution $p(y, x)$. Whilst these approaches have their advantages, they do not scale to large number of variables with complex dependencies. CRFs on the other hand, are a *discriminative* approach and model the conditional distribution $p(y|x)$. The advantages of such approaches are that dependencies that involve only variables in $\mathbf{x}$ may be ignored, thus an accurate conditional model may have a simpler structure than a joint one. In practice, we find that parameter estimation and inference on graphs with a large number of variables is only possible with approximate methods. Therefore, the complexity required by generative directed models over graphs may lead them to be intractable on large real-world datasets.

### 2.3.2 General Conditional Random Fields

A general Conditional Random Field can be represented by a bipartite factor graph, G, with the factors $\Psi_A$. Each factor is defined in terms of a set of *feature functions* $f_A$, and parameters $\theta_A$, that act as model weights associated with each *feature function* such that:

$$\Psi_A(X_A, Y_A) = e^{\theta_A f_A(X_A, Y_A)} \tag{2.3}$$

As with other probabilistic graphical models, the log-linear modelling of *feature functions* allows for useful parameter estimation and inference properties. The resultant condition probability distribution models unknown variables given a set of observed variables as:

$$p(Y|X) = \frac{1}{Z(x)} \prod_{\Psi_A \in G} e^{\theta_A f_A(X_A, Y_A)} \tag{2.4}$$

where the normalisation term is conditioned on the evidence defined by:

$$Z(x) = \sum_{Y} \prod_{A} \Psi_A(x_A, Y_A) \tag{2.5}$$

CRFs are useful for structure learning because they allow one to add arbitrary feature functions that describe properties of data. There main use has been in applications with sequences of data in a form called Linear-chain CRFs. Such models are common for Natural Language Processing, biometric, and gene classification tasks. The other, more complex, form is a general graph-based CRF, which can model arbitrarily structured data and relationships. Algorithms for exact parameter estimation and inference have been developed for sequential, tree-based, and certain graph structured data, however these algorithms are not always applicable to large, complex graphs. For example, the exact junction tree algorithm re-

peatedly groups variables until the graph becomes a tree and then uses exact inference algorithms specific to trees. For large graphs, the junction tree algorithm requires exponential time in the worst case [143, 85]. For this reason, inference on large and complex graphs with many loops is typically done using approximate inference methods such as Markov Chain Monte Carlo or Belief Propagation.

When implementing CRFs, one has many design decisions to incorporate into their final application. A pertinent design decision is how feature functions are applied to training and testing data and how the structure of the CRF is formed. In the most common application of image processing, parameters are tied to repeating structures (e.g., a 3x3 grid) that have global feature function weightings. This structure is then replicated across the pixels of the 2D image to learn the optimal relationship weightings. Where the structure is fixed for both training and testing datasets, the best implementation uses feature functions that are unique to each maximal clique over the graph. In cases where the structure is unknown ahead of time, a templating or dynamic mechanism of assigning feature functions is used.

### 2.3.3 Loop Belief Propagation

Loopy Belief Propagation (LBP) is a variant of Belief Propagation (BP) applied to graphs that may contain loops. In BP, messages are passed between nodes in a factor graph that update a node's beliefs about another node. There are two types of messages, variable to factor messages $m$, and factor to variable messages $\hat{m}$. The general idea is that a node can send a message to its neighbours only when it has received messages from all of its other neighbours. In this way, node $i$ may update its belief about node $j$ only when it has received all incoming messages that are not from $j$.

In sum-product belief propagation, messages from factors connected to

a target variable are summed whereby messages from external variables nodes have a multiplicative weighting. The recursion is defined for a variable node $\mathbf{x}_i$ in a factor graph, and the time period $t$ as follows:

$$\hat{m}_{a\to i}^{(t)}(\mathbf{x}_i) \propto \sum_{\mathbf{x}_a \setminus i} \Psi_a(\mathbf{x}_a) \prod_{j \in \mathcal{N}(a)\setminus i} m_{j\to a}^{(t)}(\mathbf{x}_j) \tag{2.6}$$

$$m_{i\to a}^{(t+1)}(\mathbf{x}_i) \propto \prod_{b \in \mathcal{N}(i)\setminus a} \hat{m}_{b\to i}^{(t)}(\mathbf{x}_i) \tag{2.7}$$

Where $\propto$ is shorthand for normalisation and $\mathcal{N}(a)$ is the set of indices for all variables of factor $a$. Similarly, $\mathcal{N}(i)$ contains the set of all indices for all factors connected to variable $i$. The marginals $\mu^{(t)}(\mathbf{x}_i)$ are then given by:

$$\mu^{(t)}(\mathbf{x}_i) \propto \prod_{a \in \mathcal{N}(a)} \hat{m}_{a\to i}^{(t)}(\mathbf{x}_i) \tag{2.8}$$

## 2.4 eXtreme Multi-Label Learning

We now introduce the problem of multi-label classification with eXtreme Multi-Label learning (XML) and review the metrics for information gain used in the XML community. Finally, we provide the necessary background and motivation for PfastreXML, a state-of-the-art XML approach used in Chapter 5.

### 2.4.1 XML Problem

Multi-label classification is the problem of training a model to predict the assignment of one or more labels to each input instance. Typically, this is more difficult than multi-class classification in which each data point belongs to exactly one class. In the multi-label setting, a data point can have

any number of associated labels, and some labels may be more strongly representative of the data point than others.

A common solution when dealing with a small label space is to use the 1-vs-all technique to learn an independent classifier per label. This technique quickly becomes intractable due to the large training and prediction costs for a modestly sized set of labels. The problem is exacerbated by the quality of labels associated with the data. Given a large set of labels to tag an input with, multiple authors may choose different labels and may tag an image of an orange, for example, with *fruit*, *the colour orange*, *spherical*, *citrus*, *food*, etc.

Due to the large number of possible labels, it is comparatively easy to identify irrelevant labels, but much more difficult to identify the relevant ones. In multi-label classification, this imbalance must be taken into account during training and in our evaluation metrics.

XML addresses the problem of learning a classifier that can tag a data point with the most relevant subset of labels from a very large label set. Common real-world datasets such as the Wikipedia dataset contain millions of distinct labels that each article may be tagged with. With the possibility for millions of labels with unknown inter-dependencies, it is very difficult to obtain the amount of training data required for modern deep learning approaches.

Many approaches circumvent the large label space problem by assuming that the training label matrix has low rank and build a label space embedding to project the high dimensional label space into a low dimensional subspace. Techniques such as SVD [144] and Bloom filters [37] have been applied to compress the number of labels and then decompress a condensed output vector back to the original label space. While methods such as label partitioning by sub-linear ranking (LPSR) [152] and LEML [157] help to make the XML problem tractable they provide low prediction accuracies compared to recent XML approaches. XML solutions must be

able to handle large class imbalances with many label instances occurring only a handful of times; for example, the WikiLSHTC[1] data set contains hundreds of thousands of "tail" labels which are present in fewer than five documents.

### 2.4.2 XML Metrics

Problems inherent with XML learning lead to errors in the ground truth, a large class imbalance in the dataset, and sparsity in the label space. Traditional metrics such as Hamming distance or Binary Cross Entropy are poor indicators of performance for multi-labelled problems. The performance metrics used in the XML community transform the problem of identifying the subset of relevant labels into ranking the complete set of labels and measuring information gain for each data point; with the highest ranked label being the most relevant and the lowest ranked the least relevant label. These metrics are typically used for evaluating search engine responses to queries and allow different labels to take on different weightings of importance, with rare tail labels being more important than common ones. Finally, these metrics consider the ordering of predicted labels and allows us to evaluate performance within the top $n$ predictions.

In almost all multi-label classification settings that exhibit a large number of labels, the number of relevant positive labels for a data point is orders of magnitude smaller than the number of irrelevant negative ones. Therefore, it does not make sense to use traditional metrics such as $F_1$ score or the Hamming loss [13], which give equal weighting to all positive and negative labels. To provide a better understanding of the relevance of assigned labels, we draw metrics from information retrieval theory and focus on the problem of measuring each label's rank. For each data point we wish to attribute labels with, we assign an ordered rank over all possible

[1] `https://www.kaggle.com/c/lshtc`

labels that is indicative of each label's relevance to the data point depending on its position in the ordered set. With an ordered set of labels for each data point, we can define the following metrics:

**Definition 1** (Cumulative Gain)**.** The cumulative gain $CG_p$ or $CG@p$ measures the information gain in the top $p$ labels and is defined as:

$$CG_p = \sum_{i=1}^{p} rel_i$$

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

The cumulative gain ignores the ordering of labels within the top $p$ elements. Traditionally, the relevance of each label takes a value in $\{0, 1\}$ depending on if the label is contained in the set of true labels for each data point; other schemes may be used in which the relevance of each label is not binary.

The discounted cumulative gain introduces a term that applies a weighting to the order of relevant labels within the top $p$ results. If a relevant label appears lower in rank, the discounted cumulative gain is weaker.

**Definition 2** (Discounted Cumulative Gain)**.**

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$$

Both prior metrics fail to account for a variance in the total number of true labels for each data point. To overcome this, the normalised discounted cumulative gain normalises the predicted label set to the size of the true set of relevant labels. This metric provides an intuitive indication of how many labels were predicted correctly out of the total labels assigned per data point.

**Definition 3** (Normalised Discounted Cumulative Gain)**.**

$$nDCG_p = \frac{DCG_p}{IDCG_p}$$

where the $IDCG_p$ is the ideal $DCG$ defined by:

$$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{rel_i}{\log_2(i+1)}$$

where in turn $REL_p$ is the true set of relevant labels in the corpus up to position $p$. The nDCG produces a metric for evaluating multi-label classification in the range $[0, 1]$ whereby a perfect ranking algorithm would achieve a score of $1.0$.

### 2.4.3 PfastreXML

PfastreXML [77] is an XML approach based on FastXML [127] that improves prediction accuracy by using a propensity-scored loss function. FastXML is a tree-based machine learning algorithm similar to Multi-Label Random Forests (MLRFs) that ranks labels depending on their relevance to an associated data point. The approach aims to learn a hierarchy by using a collection of trees to subdivide the feature space until a multi-label classifier can be used to predict the associated labels in this smaller subspace. For each data point, PfastreXML and FastXML output a ranked list containing every label from a fixed size label space and their associated probabilities.

Consider an extremely large set of labels $L$ in an extreme multi-label problem. When determining the ground truth labels for each data point, it is often not possible to determine the exact subset of relevant labels through either manual or automated annotation. In the context of function labelling, as explored in this thesis, this would correspond to each

function name containing all words that are relevant to its behaviour. We denote with $\mathbf{y}^* \in \{0,1\}^{|L|}$ the complete ground truth label vector, which cannot be obtained in practice. With $\mathbf{y} \in \{0,1\}^{|L|}$ we denote the actually observed ground truth label vector, where certain labels are missing. We then have that $\mathbf{y}_i^* = \mathbf{y}_i = 1$ for observed relevant labels, $\mathbf{y}_i^* = 1$ and $\mathbf{y}_i = 0$ for unobserved relevant labels; and $\mathbf{y}_i^* = \mathbf{y}_i = 0$ for irrelevant labels. This assumes that truly irrelevant labels are never labelled as relevant, but that relevant labels may be missing.

In the design of FastXML it was assumed the marginal propensities of labels were known; however, in the XML setting ground truth labels may be incomplete or missing. PfastreXML aims to mitigate this problem by using a propensity-scored loss function that weights the loss characteristics for each label $l \in L$ that matches a predetermined distribution. The propensity $p_{il} \equiv P(y_{il} = 1 | y_{il}^* = 1)$ corresponds to the marginal probability of observing a relevant label $l$ for the data point $i$. Unfortunately, the propensities are unknown as the complete ground truth is unobservable. Therefore, propensities are modelled as a sigmoid function of $\log N_l$ as given in Equation 2.9:

$$p_l \equiv P(y_l = 1 | y_l^* = 1) = \frac{1}{1 + Ce^{-A \log(N_l + B)}} \tag{2.9}$$

Here, $N_l$ is the number of data points annotated with label $l$ in the observed ground truth dataset of size $N$. $A$ and $B$ are problem specific hyper-parameters and $C = (\log N - 1)(B + 1)^A$. These hyper-parameters should be tuned for each dataset such that log of the number of instances of each label fit a sigmoidal function.

## 2.5 Function Boundary Detection

In this section we briefly review state of the art research methods for function detection that underpin several assumptions made in this thesis. Function detection in stripped binaries is a problem facing all major reverse engineering and binary analysis tools. HexRays, Binary Ninja, Radare2, Ghidra, DynInst, and BAP [26], for example, all implement their own heuristic-based methods for detecting function boundaries. In academic research, function detection in stripped binaries consists of two different problems: *function start detection* and *function boundary detection*. In function start detection, the aim is to determine a set of start addresses in a stripped binary that correspond to the first line (entry point) of functions in the source code. In function boundary detection, the goal is to determine the set of virtual address pairs (start, end) in a stripped binary that correspond to the function entry point in the source code and the last address of code that constitutes to the source function. The research literature usually splits these distinct problems into two oracles in which function boundary detection is dependent on function start detection.

Jima [10] is the current state of the art research tool for function boundary detection and uses static analysis and limited behavioural analysis to detect function starts and bounds. It uses a combination of jump pointer analysis, exception handler analysis, and terminal function detection to overcome the problem of identifying jump tables, exception handlers, and terminal functions. In the authors evaluation of Jima, they report their tools F1 scores in binary function detection over three datasets; Jima achieves an F1 score of 0.9959 on UNIX utilities, 0.9927 for the SPEC CPU 2017 dataset, and 0.9904 for the Chrome web browser. The impact of these results is enhanced when one considers problems with finding the ground truth for binaries function boundaries. When function boundary detection tools extract ground truth information they must take into considera-

tion zero-length functions, multiple function aliases, compiler added *new functions* with multiple entry points, and inconsistent function boundaries (junk suffixes).

Byteweight [14] is a function boundary detection technique that builds a prefix tree of assembly instructions from known function start sequences. The authors learn a weighted tree of normalised assembly instructions to perform binary classification. Using their tool, the authors are able to achieve an average F1 score of 0.97 for function boundary detection.

Nucleus [11] is another state-of-the-art method for function boundary detection that uses recurrent neural networks and connected component analysis of inter-procedural control flow graphs to cluster a binary's basic blocks into functional boundaries. The approach achieves a drastic reduction in computation time whilst still maintaining high accuracy (approx. 98%).

These methods prove that function boundary detection in stripped binaries can be done to high accuracy and underpin our assumptions of known function boundaries in the remainder of this thesis. Higher accuracy may be achieved when limiting the set of inferred functions to those that are "well defined", e.g., non-alias functions that have a global binding in ELF binaries.

## 2.6 Universal Serial Bus

USB devices are ubiquitous in the modern world with most hardware manufacturers choosing it as their inter-connect of choice. The flexibility in electronic requirements from the USB specification allows for high power delivery to end-user devices, very high transfer speeds, and reversible master-slave setups. The prevalent nature of USB devices impels operating system developers into supporting as many USB device drivers

as possible in an effort to meet market demands. Unfortunately, writing error free kernel device drivers is a complicated task; the author must anticipate the context of execution, potential race conditions, hardware delays and user space interactions.

In this section we provide an overview of the USB protocol (Section 2.6.1) and then focus on its implementation in the Linux kernel (Section 2.6.2).

### 2.6.1 USB Specification

The USB specification supports a vast number of features, such as hot-plugging, generic class drivers, multi-master USB On-The-Go, and data and power transfer modes. The protocol follows a master-slave design in which a Host Controller Interface (HCI) communicates with USB client devices—'gadgets' in the USB specification terminology. USB HCIs implement the low-level protocols of timings, packet scheduling, and signalling for each version of the specification. This is abstracted to Universal Request Blocks (URBs), which form the logical basis of communication with gadgets. URBs contain a *device address* to signify the device connected on the bus and an *endpoint* to specify a channel to a device. They can be one of four types:

- **Control Transfers**: device configuration and signal control information;
- **Bulk Transfers**: large quantities of time-insensitive data;
- **Interrupt Transfers**: small quantities of time-sensitive data;
- **Isochronous Transfers**: real-time data at predictable bit-rates.

An endpoint provides an address and direction for URB transfers from the perspective of the host. Control transfers to endpoint 0 are special; all

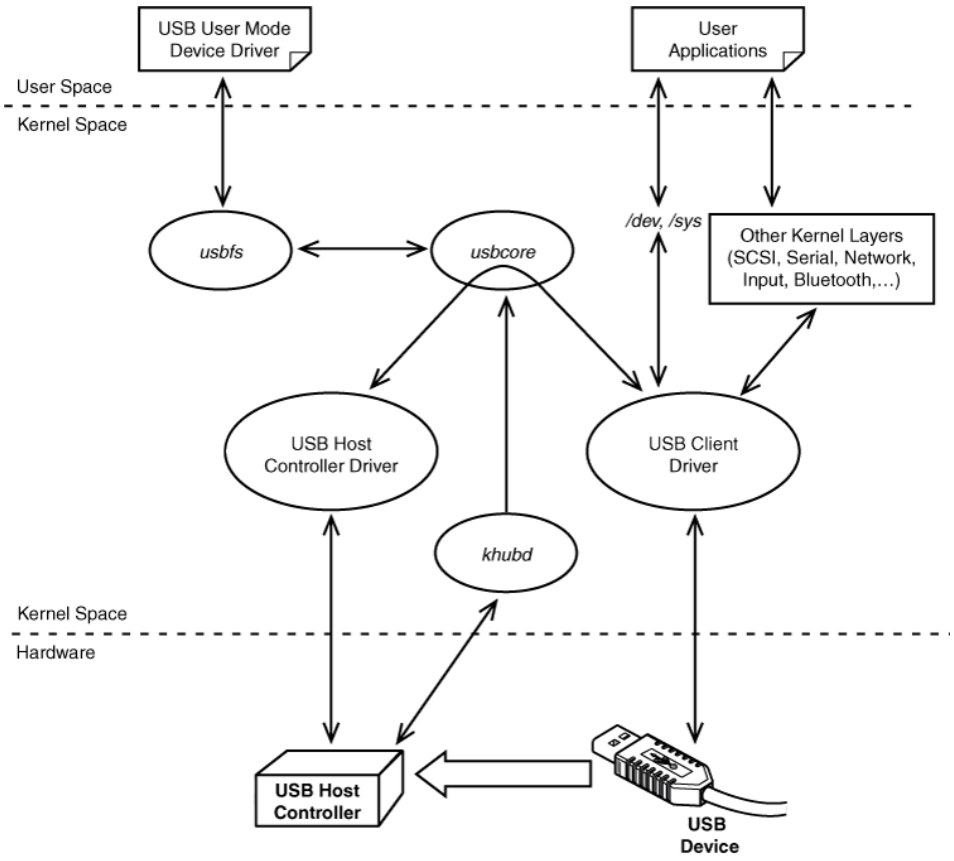



Figure 2.8: Design of the Linux USB stack (diagram taken from Linux Device Drivers 3rd Edition [39] and released under the Creative Commons license). Layers are separated between user space, kernel space, and hardware. Arrows represent a direction of communication between different modules in the stack.

USB devices must implement bi-directional (IN and OUT) communication to this endpoint as it is used during device initialisation.

All USB devices present descriptions of their identify and functionality through the use of *descriptors*. The USB specification contains five types of

descriptors: Device descriptors, Configuration descriptors, Interface descriptors, Endpoint descriptors, and String descriptors. Each descriptor type presents data in a pre-determined structure that identifies fields relevant to communication such as: vendor ID, product ID, manufacturer, USB protocol version, transfer speeds, etc. Typically, the vendor and product ID received from the USB device are used to select a relevant driver, however, USB devices can also present generic device classes and subclasses that trigger generic USB drivers to be selected, e.g., mass storage device.

### 2.6.2 Linux USB Stack

The Linux kernel's USB subsystem has a modular design allowing new client device drivers to easily interface with hardware [39]. The Linux kernel maintains a kernel daemon thread called *khubd*, which is responsible for monitoring any USB hubs by communicating with the on-board HCI and configuring USB devices. When a USB device is plugged into a machine, the kernel enumerates the device's capabilities and configurations in a process called *USB enumeration*. In this process, *khubd* is awoken by the main kernel thread upon a new USB event being triggered via a Host Controller's hardware interrupts. The purpose of USB enumeration is to iterate over its configuration descriptors to determine power output, associated endpoint addresses, transfer types, speed, device class, etc. and follows the following procedure:

1. The root hub detects a current change due to device being attached and invokes *khubd*.

2. *khubd* deciphers the identity of the USB port that caused the change.

3. *khubd* chooses a device address between 1-127 and assigns it to the

USB gadget via standard USB device requests. It associates the gadgets endpoint using a control URB to endpoint 0.

4. *khubd* uses the endpoint address 0 to obtain the device descriptor from the gadget, requests the devices configuration descriptors and selects a suitable descriptor.

5. *khubd* requests *usbcore* to bind a matching device client driver to the inserted device. The new kernel driver is then initialised by calling its *probe* function with initialised USB device structures as arguments.

The *usbcore* Linux subsystem matches USB devices with the corresponding client driver by matching USB descriptors exported by client drivers when the Linux kernel was compiled. A client USB Linux driver must declare these by creating a `usb_device_id` struct array corresponding to all of the vendor and product IDs it wishes to be associated with or the relevant device driver class, e.g., USB Mass Storage. The driver then registers the `usb_device_ids` with the `MODULE_DEVICE_TABLE` macro as seen in Listing 1; with the `usb` variable corresponding to a `usb_driver` driver struct listing core function callbacks, e.g., *probe*, *disconnect* and *suspend*. The macro is used by the kernel to generate global map files, which are then used to reference the relevant driver that should be loaded upon a new USB device being plugged in and is used by *usbcore*.

Once enumeration is complete, the associated client driver's probe function is invoked with a reference to the USB device and a USB client device driver can then communicate with the device through library functions in *usbcore* such as `usb_submit_urb`. Such communication requests are sent as URBs to a HCI (typically encapsulated over the PCI bus) and forwarded on to the gadget.

Most Linux device drivers allow user space processes to access hard-

```
struct usb_device_id {
  /* ... */
  __u16 idVendor; /* Vendor ID */
  __u16 idProduct;  /* Product ID */
  /* ... */
  __u8  bDeviceClass;  /* Device class */
  __u8  bDeviceSubClass; /* Device subclass */
  __u8  bDeviceProtocol; /* Device protocol */
  /* ... */
}
...
static struct usb_device_id ids[] = {
    { USB_DEVICE( VENDOR_ID, PRODUCT_ID) },
    {} /* Terminate */
}
...
MODULE_DEVICE_TABLE( usb, ids)
```

Listing 1: USB device driver association under Linux.

ware through special files, e.g., a character device, block device, socket, or named pipe. Typically, this involves system calls on files located in *devfs* or *sysfs*, which in turn call functions in kernel drivers. For example, the driver for USB Mass Storage Devices wraps a SCSI device around a USB device; file operations to its node under `/dev/sd[a-zA-Z]{1-3}` trigger URB transfers to the corresponding USB device.

## 2.7 Linux Kernel Security

In this section we discuss contemporary software defences employed in the Linux kernel necessary to understand potential exploitation mechanisms. We describe kernel protection mechanisms, advanced kernel debugging systems, and run-time memory integrity protections that may be compiled into the kernel.

### 2.7.1 Linux Kernel Defences

Modern GNU/Linux operating systems come with a variety of software defence mechanisms compiled into the kernel. Here, we briefly explain the most prevalent security techniques that are commonly applied to a *hardened* Linux kernel.

#### Kernel Address Space Layout Randomisation (KASLR)

KASLR is an inexpensive probabilistic defence mechanism that applies the user space concept of Address Space Layout Randomisation (ALSR) to the Linux kernel. It does this by randomising the base address of the kernel *code* section at boot. This technique prevents common attacks that insert malicious code into the kernel address space, or jumping to known offsets, and is easy to implement. The simplest *privilege escalation* attack in the Linux kernel executes the following code:

```
commit_creds(prepare_creds());
```

which prepares an empty set of credentials that corresponds to the *root* user in `prepare_creds` and applies them to the current process with `commit_creds`. If an attacker has the capacity to execute arbitrary kernel code the only unknown is the address of these two functions. With KASLR employed, successful exploitation now depends on an initial information leakage to calculate the relative offset from the randomised base address of the code section to these function addresses.

#### Return Address Protection (RAP)

RAP is a kernel defence mechanism developed by GrSecurity [66] that adds forward and return edge CFI into the Linux kernel. It is developed

as a plugin to the GCC[2] compiler and aims to stop Return Orientated Programming (ROP) attacks. ROP attacks are a leading method of overcoming protections provided by No eXecute (NX) (Data Execution Prevention (DEP) in Windows) in which existing executable code in a binary is reused to craft an unintended or malicious execution.

To defend against these attacks, RAP implements forward edge and return edge Control Flow Integrity (CFI) in two parts. First, the GCC plugin performs a deterministic analysis limiting the set of functions that may be called from each code location, and what locations may be returned from each function. Latest research proves that weak CFI is still exploitable [44] thus secondly, RAP implements a probabilistic defence to ensure that within each determined set of functions, the correct location is always returned.

#### Supervisor Mode Access Prevention (SMAP)

SMAP is a hardware feature on some CPU implementations that allows access to user space memory to be restricted when executing supervisor mode programs. This security feature is designed to stop arbitrary data from being read from user space, a method commonly used by kernel exploits to deliver payloads. The feature may be disabled on `x86` based CPUs by setting bit 21 to `0` in the CR4 control register. This operation, and all modifications to the CR4 register, may only be performed when the CPU is executing in supervisor mode.

#### Supervisor Mode Execution Prevention (SMEP)

SMEP is a hardware feature on some CPU implementations that restricts executing code residing in user space memory while the CPU is operat-

[2] `https://gcc.gnu.org`

ing in supervisor mode. This security feature is designed to stop exploits jumping to arbitrary code residing in user space and makes vulnerabilities where a user can influence a function pointer more difficult to exploit. It is enabled on `x86` CPUs by setting bit 20 to 1 in the CR4 control register.

#### Kernel Address Sanitiser (KASAN)

KASAN is an implementation of the user space dynamic error detector Address Sanitizer (ASAN) [133] for the Linux kernel. It uses compile-time instrumentation to add run time checks for detecting out-of-bounds accesses to both stack and global variables. It can be enabled by compiling the Linux kernel with `CONFIG_KASAN=1` and provides memory reports to the standard logging facility in the event of an out-of-bounds write or use-after-free is detected. It maintains a shadow memory section separated from main memory that tracks objects and their references. Gaps between objects in main memory are inserted to create poisoned *red zones* which cause a trap to be executed on access. When a kernel module is compiled with KASAN, every memory access is instrumented to perform a safety check on a modified pointer value. The pointer is used to check meta-data in the shadow memory region to detect out-of-bounds accesses.

### 2.7.2 System Tap

System Tap [128] is a tool to support system-wide instrumentation in Linux-based operating systems. It allows developers to analyse a live system without modifying an original program's source code by executing arbitrary routines with associated *hook* points. Developers create *handlers* written in the *stap* language which are assigned to *probe* points in existing software. The *stap* routines have the capacity to run arbitrary C code and modify internal programs memory. This makes them an incredibly powerful and versatile tool for analysing kernel modules as well as user space

```
probe begin{
    printf("init systemtap kernel module\n")
}

function gtod() {
    return gettimeofday()
}

probe syscall.*.return {
    errno   = $return   #return value
    if (errno < 0) {
        elapsed = gtod() - @entry(gtod())
        printf("errno %d %s after %d\n", errno,
            errno_str(errno), elapsed)
    }
}
```

Listing 2: An example SystemTap program that prints out the elapsed time and error message from all system calls that return error values.

programs.

*Stap* modules must be compiled into a Linux kernel module (PIE ELF `.ko` file) and inserted into the running kernel ahead of time using *staprun*. When compiled using *Return probes* instead of *Kprobes*, return addresses are replaced with *trampoline addresses* which mitigates the need for using break points. This allows *stap* modules to be used for debugging multi-process applications at near full speed. The code snippet in Listing 2 shows an example of a *stap* program. It first declares an entry point to be run when the kernel module is inserted defined by the `begin` syntax after a `probe` statement. It then defines a function called `gtod` which returns the in-built `gettimeofday` stap function. Finally, probes are inserted on the return of all syscalls as expressed by `syscall.*.return` and the elapsed time between the syscall being called and returning is calculated if the return value is less than `0`.

When specifying *stap* probe locations, debug information such as symbols are required when specifying named locations, however, limited *stap* functionality can be implemented using *statements* that specify absolute virtual addresses.

# Probabilistic Naming of Binary Functions

# 3

In this chapter, we discuss our approach to naming functions in stripped binaries and describe the theoretical foundations of our tool, *Punstrip*, that is used as a binary analysis platform through the remainder of this dissertation. *Punstrip* builds a probabilistic model that learns how developers use and name functions across a set of existing open-source projects; using this model, we infer meaningful symbol information based on similarities in program structure and semantics in previously unseen, stripped binaries. It is not necessary to discover exactly the same identifiers that the developers used in the original program: for reverse engineering, we are interested in discovering symbol names that are helpful to an analyst. With *Punstrip*, reverse engineers can pre-process an unknown binary to automatically annotate it with symbol information, saving them time and preventing mistakes in doing further manual analysis. We make *Punstrip* available as open source[1] software.

This chapter describes the following:

- We present a novel approach to function identification and signature recognition for stripped binaries that uses features in a higher-level intermediate representation. This approach can scale to real world software and seeks to be agnostic to both compiler architectures, bi-

[1] `https://github.com/punstrip/punstrip`

nary formats, and optimisations.

- We introduce a probabilistic graphical model for inferring function names in stripped binaries that compares the joint probability of all unknown symbols simultaneously rather than treating each function in isolation. The model builds on our probabilistic fingerprint and analysis between symbols in binaries.

We evaluate *Punstrip* against the current state of the art in function name detection against all C binaries found in open-source Debian repositories with a 10-fold cross validated evaluation. Furthermore, we evaluate our probabilistic fingerprint against leading tools using binaries with a large common code base that were compiled in different environments. In the remainder of this chapter, we give an overview of *Punstrip*'s pipeline (Section 3.1), introduce our technical approach to the problem of function fingerprinting (Section 3.2), and describe the abstract graphical structure for learning (Section 3.3). We then present our method for relating function names (Section 3.4) before evaluating *Punstrip* against previous work (Section 3.5). Finally, we discuss limitations of our approach (Section 3.6) and contrast with related work (Section 3.7).

## 3.1 Overview

Figure 3.1 shows an architectural overview of the *Punstrip* pipeline. *Punstrip* takes as input a set of ELF binaries, which for training should be unstripped. In the initial analysis stage, *Punstrip* extracts function symbols and their boundaries as defined in the symbol table, disassembles them, and lifts the instructions to the VEX intermediate representation. From this representation and the inter-procedural control flow information among functions, *Punstrip* extracts a set of features that are stored in a database.

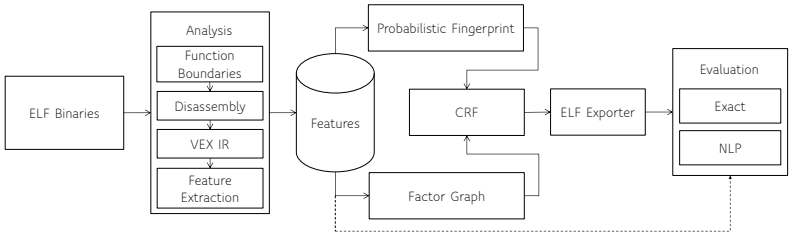



Figure 3.1: A block diagram overview of the components in *Punstrip*.

These features are used to build a per-function fingerprint, as well as a factor graph representing the relationships between functions and feature values for each executable. Our probabilistic fingerprint and factor graph are used to construct a General Conditional Random Field (CRF) that learns how individual functions interact with other *code* and *data*.

After training a model on a large corpus of programs that include symbol information, we are able to use the learned parameters to infer the most likely function names in *stripped* binaries. The inferred function symbol names can then be added to the *stripped* binary and used for debugging and reverse engineering purposes. In this chapter, we focus exclusively on the problem of naming functions; for detecting function boundaries in stripped binaries, we refer to recent approaches from the literature [11, 14].

We evaluate the accuracy of function name prediction using two metrics: (1) exact matches of function names, and (2) normalised matches of function names. In the remainder of this section, we detail the individual stages of the pipeline referring to examples in Figure 3.2, in which we separate known from unknown functions, that we aim to label.

### 3.1.1 Probabilistic Fingerprint

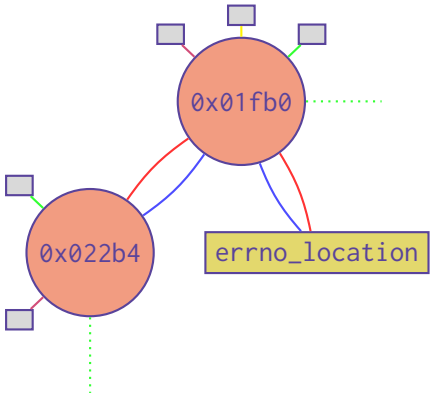
Figure 3.2a shows the disassembly of a function from a stripped binary. Static analysis is able to detect that the unknown function is called from

```
(fcn) 0x01fb0
;CALL XREF from 0x0162d 0x0162d
0x01fb0      test rdi, rdi
0x01fb4      je 0x2027
0x01fb6      mov esi, 0x2f
0x01fbb      mov rbx, rdi
0x01fbe      call imp.errno_location
0x01fc3      test rax, rax
0x01fc6      je 0x2017
0x01fc8      lea rdx, [rax + 1]
0x01fcc      mov rcx, rdx
0x01fcf      sub rcx, rbx
0x01fd2      cmp rcx, 6
0x01fd6      jle 0x2017
0x01fdf      mov ecx, 7
0x01fe6      jne 0x2017
0x01fed      call 0x022b4
```

(a) Disassembly of a function in a stripped executable.

```
IRSB {
   t1:Ity_I64 t2:Ity_I64
      t3:Ity_I64 t6:Ity_I64
      t7:Ity_I1
   00| --IMark( 0x1fb0, 3, 0)--
   01| t2 = GET:I64(rdi)
   02| PUT(cc_op) = 0x00000014
   03| PUT(cc_dep1) = t2
   04| PUT(pc) = 0x01fb3
   05| t15 = CmpEQ64(t2,0x00)
   06| t14 = 1Uto64(t15)
   ...
   19| if (t7)
     { PUT(pc)=0x2027;
       Ijk_Boring }
   NEXT:
    PUT(rip)=0x01fb6; Ijk_Boring
}
```

(b) VEX IR of the function's first basic block.




(c) Graph-based CRF based on relationships between features, knowns, and unknowns.

```
(fcn) set_program_name
; CALL XREF from 0x0162d main
0x01fb0  test rdi, rdi
0x01fb4  je 0x2027
0x01fb6  mov esi, 0x2f
0x01fbb  mov rbx, rdi
0x01fbe  call imp.errno_location
0x01fc3  test rax, rax
0x01fc6  je 0x2017
 ...
0x01fe6  jne 0x2017
0x01fed  call strchr
```

(d) Disassembly of the same function with inferred function names added to the binary's symbol table.

Figure 3.2: Stages in *Punstrip*'s pipeline for learning and evaluating function name inference on stripped binaries.

the unnamed function at address 0x0162d and calls the dynamically linked function errno_location (which gets the error identifier from the last executed system call) before calling the function at 0x022b4.

As the binary sequence of machine code can differ even for the same source code depending on compilation settings, we opt for using an intermediate representation that abstracts some implementation detail. We lift machine code to the VEX intermediate representation, as shown in Figure 3.2b, which offers an appropriate level of detail and abstraction for our analysis. From this representation, we extract a collection of features that help identify names of functions designed to be agnostic to changes in compilers and optimisations. While the binary code of a function may change due to compiler differences, and hence values of our features, our intuition is that even if a compiler modifies the order, number of, and type of instructions, changes in feature values based on optimised VEX IR will still be similar.

After extracting a set of features we first convert each into a vectorised form before stacking them and using it as the input to a multiclass classifier. The output of multiclass classification gives a probability mass function over the set of all unknown functions in our training data. Thus, for each function we learn a probability distribution over all function names given input features derived from VEX.

### 3.1.2 Probabilistic Structural Inference

After extracting high-level features and relationships between functions, we build a probabilistic graphical model in the form of a Conditional Random Field (CRF) [85, 143] as depicted in Figure 3.2c. The CRF operates on a factor graph which factorises inter-dependencies between functions and separates unknown and known information. Figure 3.2c shows that the unknown function 0x01fb0 has a factor-based on calling the known func-

tion errno_location, factors based on statically derived known features such as its probabilistic fingerprint, and a factor-based on its relationship with 0x022b4.

Our CRF models pairwise and generic factor-based relationships among *code* and *data* for all functions in an executable. We first train the graphical model and learn the weightings of relationships on a large set of prior unstripped binaries before building a new CRF for each stripped binary and applying the learned model parameters. Function names are then inferred by maximising the conditional probability of unknown function names, given the known information and our models parameters. This enables *Punstrip* to take into consideration known information from the whole binary simultaneously rather than considering each function in isolation; this may help identify functions that are weakly identifiable in and of themselves, but have strong connections to other more easily recognisable functions.

### 3.1.3 Function Name Matching

Finally, after we have inferred function names for the set of unknown symbols, we modify the ELF executable's symbol table and insert entries into the `strtab` and `symtab` sections. As a result, subsequent disassembly will yield the newly predicted function names (Figure 3.2d).

The names developers give to functions can vary wildly based on personal preferences and project styles. Therefore, relying on exact lexical matching only to evaluate the accuracy of name predictions would miss cases where predicted names are semantically correct but syntactically different. We therefore implement an additional metric for evaluating name similarity. We propose a method based on natural language processing, which uses lexical techniques to determine the similarity of two names. The metric tries to mitigate grammatical differences in language used, ex-

pands common naming conventions used by programmers, and takes into account similar words, such as `start` and `begin`, within each name.

## 3.2 Probabilistic Fingerprint

We now explain how *Punstrip* creates a probabilistic fingerprint for each function in a (potentially stripped) binary. We start by giving an overview of the VEX IR (Section 3.2.1) before describing how we extract features from it using static (Section 3.2.2) and symbolic (Section 3.2.3) analysis. Finally, we detail how extracted features are combined into a probabilistic identifier (Section 3.2.4).

### 3.2.1 VEX Intermediate Representation

Whilst binary sequences of machine code are subject to vast differences across small compilation changes, in the design of *Punstrip* we aim to build a set of feature functions that expose abstract relationships between segments of binary code that survive multiple compilations in different ISAs and compiler technologies. Therefore, we first lift the raw machine code into a higher level Intermediate Representation (IR) that enables us to abstract away differences in register names, memory accesses, memory segmentation and instruction side effects across ISAs.

There are many IR languages available with the most common being LLVM IR [90], REIL [51], GCC's GENERIC [145] and GIMPLE [146], however, *Punstrip*'s analysis is built using the VEX Intermediate Representation (IR) that was designed by the Valgrind project [114] [149]. VEX aims to be an architecture agnostic representation and supports lifting from x86, AMD64, ARM, ARM64, PowerPC32, PowerPC64, MIPS32, MIPS64, TILEGX and S390X ISAs. It therefore matches our needs better than other intermediate languages. The representation itself abstracts machine code

in a unified way which explicitly models all instruction side-effects and abstracts away differences in register names, memory accesses, memory segmentation, and machine state such as the EFLAGS register [84] in `x86`. We choose to use python bindings for *libVEX* as it has strong community support and development first being used in firmalice [139] before being adopted by angr [149] and Driller [141].

The VEX IR only supports lifting machine code per basic block. Therefore, *Punstrip* first needs to disassemble a function's machine code and recover the Intra-procedural Control Flow Graph (ICFG) of basic blocks. Starting with the Function Entry Point (FEP), *Punstrip* uses the *capstone* [116] disassembly framework and the leaders algorithm to sequentially separate assembly code into straight line execution sequences. We use capstone as it supports multiple ISAs, multiple language implementations, has an architecture neutral API and has vast community development and support. Capstone can determine constant jump targets of disassembled code. We use this information to sequentially set the addresses of jump targets as well as the address immediately following the jump instruction as leaders. At this stage, *Punstrip*, is unable to fully recover the ICFG if jump targets are non-constant; however, through a more detailed static analysis at a later stage, a more accurate ICFG may be recovered.

Once the basic block boundaries for each function are known, each is then lifted into the VEX IR language that consists of five main classes of objects:

- **Expressions** represent a constant or calculated value including the results of arithmetic operations, memory loads and register reads.

- **Operations** describe a modification of Expressions including bit operations, integer arithmetic, floating point arithmetic, etc. An Operation applied to an Expression yields an Expression as a result.

- **Statements** model changes in the state of the current machine including memory stores and register writes.
- **Temporary Variables** model internal registers and are used to store Expressions in a strongly typed manner.
- **Blocks** describes a collection of Statements representing an extended basic block. A Block may have multiple exits but only a single entry point.

An example of the first basic block of the function used in Figure 3.2a lifted to VEX IR can be seen in Figure 3.2b which shows the number and type of temporary variables used and instructions in the block. The first basic block only consists of the first two decoded assembly instructions before a potential control flow change; however it describes all modifications to the state such as updating the *program counter* if the 64 bit integer comparison of the temporary variable `t2` is `0`. Finally, the basic block is terminated with a labelled intraprocedural jump (indicated by `Ijk_Boring`). The VEX IR basic block jump may be labelled as: call, return, fall-through, termination, or a *boring jump*.

### 3.2.2 Static Analysis

All features extracted from each function are listed in Table 3.1. We include two low-level features that help to find exact matches: a hash of the machine code and a hash of the opcodes in the disassembly. The opcode hash is included to recognise exact patterns of generated machine code with different parameters or relative offsets, which would not be matched by an exact binary hash. All other features are extracted from VEX IR.

Symbols contained in an ELF binary's symbol table detail a component's address, size, and string description in the target executable. This information is first extracted along with the raw bytes corresponding to the

Table 3.1: Features extracted from functions and their representation in the probabilistic fingerprint.

| Feature | Type | Description |
|---|---|---|
| | *Static features* | |
| Size | Scalar | Size of the symbol in bytes. |
| Hash | Binary | SHA-256 hash of the binary data. |
| Opcode Hash | Binary | SHA-256 hash of the opcodes. |
| VEX instructions | Scalar | Number of VEX IR instructions. |
| VEX jumpkinds | Vector(8) | VEX IR jumps inside a function e.g., *fall-through*, *call*, *ret* and *jump* |
| VEX temporary variables | Scalar | Number of temporary variables used in the VEX IR. |
| VEX IR Statements, Expressions and Operations | Vector(54) | Categorised VEX IR Statements, Expressions and Operations. |
| Callers | Vector(N) | Vector one-hot encoding representation of symbol callers. |
| Callees | Vector(N) | Vector one-hot encoding representation of symbol callees. |
| Transitive Closure | Vector(N) | Symbols reachable under this function. |
| Basic Block ICFG | Vector(300) | Graph2Vec vector representation of labelled ICFG. |
| VEX IR constants types and values | Dict | Number of type of VEX IR constants used. |
| | *Symbolic features* | |

*Continued on the next page*

Table 3.1: Features extracted from functions and their representation in the probabilistic fingerprint (cont.).

| Feature | Type | Description |
|---|---|---|
| Stack bytes | Scalar | Number of bytes referenced on the stack. |
| Heap bytes | Scalar | Number of bytes referenced on the heap. |
| Arguments | Scalar | Total number of function arguments. |
| Stack locals | Scalar | Number of bytes used for local variables on the stack. |
| Thread Local Storage (TLS) bytes | Scalar | Number of bytes referenced from TLS. |
| Tainted register classes | Vector(5) | One-hot encoded vector of tainted register types, e.g., stack pointer, floating point. |
| Tainted heap | Scalar | Number of tainted bytes of the heap. |
| Tainted stack | Scalar | Number of tainted bytes of the stack. |
| Tainted stack arguments | Scalar | Number of tainted bytes that are function arguments to other functions |
| Tainted jumps | Scalar | Number of conditional jumps that depend on a tainted variable. |
| Tainted flows | Vector(N) | Vector of tainted flows to known functions. |

function. Using *pyvex* and each function's boundaries as specified in the binary's symbol table [17], we lift each basic block into its optimised VEX IR and build a labelled Intraprocedural Control Flow Graph (ICFG) for each function. We then resolve dynamically linked objects and build a call

graph for each statically linked function in the binary.

Table 3.2: Rules for VEX IR categorisation.

| Regex | Description |
|---|---|
| | *VEX IR Operations* |
| `Iop_Add(.*)` | Addition |
| `Iop_Sub(.*)` | Subtraction |
| `Iop_Mul(.*)` | Multiplication |
| `Iop_Div(.*)` | Division |
| `Iop_S(h\|a)(.*)` | Arithmetic and logical shifts |
| `Iop_Neg(.*)` | Negation |
| `Iop_Not(.*)` | Logical NOT |
| `Iop_And(.*)` | Logical AND |
| `Iop_Or(.*)` | Logical OR |
| `Iop_Xor(.*)` | Logical XOR |
| `Iop_Perm(.*)` | Permute bytes |
| `Iop_(.*)to(.*)` | Type conversion |
| `Iop_Reinterp(.*)as(.*)` | Reinterpretation |
| `Iop_(Cmp\|CasCmp)(.*)` | String comparison |
| `Iop_Get(M\|L)SB(.*)` | Get significant bit |
| `Iop_Interleave(.*)` | Bit interleaving |
| `Iop_(Min\|Max)(.*)` | Min/max operations |
| | *Statements* |
| `Ist_Exit` | Exit |
| `Ist_IMark` | Instruction marker |
| `Ist_MBE` | Exit |
| `Ist_Put_(.*)` | Put |
| `Ist_(Store\|WrTmp)` | Write |

We track all features given in Table 3.1 and convert this structure of features into a form that can be represented by a single stacked vector to be used as a fingerprint for each function. When labelling the ICFG, VEX

basic blocks are distinguished by their terminators (jumps, calls, returns, and fall-throughs). We only store numeric integer constants, greater than $2^8$, that are not operands of jump instructions to focus on infrequent and distinctive values.

After machine code is lifted into the VEX IR we categorise each instruction into a one-hot encoded vector according to the regular expressions defined in Table 3.2. The one-hot encoded vectors are then summed to produce an impression of functions operations.

To convert generic graphical structures to a vector representation we utilise the feature embedding technique *graph2vec* [122]. We compare the similarity of all ICFGs by using an implementation of the Weisfeiler-Lehman graph kernel [136]. By training a *graph2vec* model, each ICFG is converted into a vector space in which similar graphical structures are numerically similar. We store each vector in an Annoy Database[2] that allows us to quickly find the nearest vectors for each graph-based on the Euclidean distance from our model's embeddings (and hence the most similar graphs). Training the *graph2vec* model is computationally expensive; however, it allows us to avoid comparing pairs of graphs with a graph kernel over the testing set for every element in the training set. Using *graph2vec* we can efficiently compare the similarity of abstract graphical structures without a quadratic query time after training the model.

### 3.2.3 Symbolic Analysis

We extract additional semantic features using our own symbolic analysis built on top of the VEX IR. We write our own execution engine over the existing angr [140] implementation to provide a lightweight and more consistent analysis across multiple platforms. Upon loading the binary, we

[2]Annoy: Approximate Nearest Neighbours Oh Yeah: `https://github.com/spotify/annoy`

fill a virtual memory space with concrete data from the ELF binaries' data (`.data` and `.bss`), strings (`.rodata`), and code (`.text`) sections. After the boundaries of each basic block are known from the initial static analysis, we are able to lift each block into VEX in Single Static Assignment (SSA) form. Within our model, reads from registers and memory locations with undefined contents return symbolic values for the size of data requested.

*Function Argument Extraction*. To identify the number of function arguments we carry out live variable analysis on argument registers and memory references to pointers above the current stack pointer. Our implementation of live variable analysis determines the set of function arguments that are live at the start of each function. According to the System V AMD64 ABI [101] followed by GNU/Linux for `x86_64`, arguments may be passed to each function using *rdi*, *rsi*, *rdx*, *rcx*, *r8*, *r9*, *xmm{0-7}*, and then additional arguments are passed on the stack. As we need to track the value of the stack pointer, we perform a fixed-point iteration algorithm to determine the base and stack pointer values on each use. Finally, we build a model to track memory references between basic blocks as the VEX IR's SSA form is only consistent per basic block.

*Heap and Stack Analysis*.

We implement a stack of 2048 bytes starting at `0x7FFFFFFFFF0000` and model the stack registers, segment registers, and heap accordingly. For each function, we track the total number of bytes referenced on both the heap and stack, local variables and function arguments placed on the stack, thread local storage accesses, and perform taint analysis to calculate data flows from each input argument to arguments of other functions. Finally, we compute the transitive closure for each function under the binary's call graph.

*Symbolic Execution*. After identifying the number of input arguments to a function we symbolically execute the function using our own execution engine that uses Claripy [140, 38] to formulate symbolic values and ex-

pressions. This allows us to create symbolic expressions for return values from each function, e.g., $\texttt{ret} \models \texttt{SymbVec(ARG1)+BitVec(0x2)}$. Where our symbolic execution engine cannot easily determine the result of an operation, e.g., the `x86_64` instruction `AESENC`, we inject symbolic values. This may lead to incorrect analyses, however, this optimisation is infrequently used and allows our lightweight approach to scale to analyse functions on *big code*.

After identifying symbolic variables we are able to extract call sites to other functions that are control-dependent on a symbolic variable. We then track the number of call sites that are control-dependent on each input argument and use it as a feature in our fingerprint. We run our analysis for every recovered function argument to extract per input-dependent taints. Finally, we also include an analysis pass that taints all input arguments to mitigate against reordering of function arguments producing different results.

*Register Classification.*

We classify registers referenced during execution into five generic classes: general purpose, floating point, stack and base pointer, segment register, and control register. For each input argument we produce a vector of tainted register classes from the set of tainted registers after taint propagation. This allows *Punstrip* to capture the types of behaviour performed by functions for individual arguments. Finally, we produce a final vector of tainted register classes irrespective of taint.

### 3.2.4 Probabilistic Classification

We aim to convert features extracted in Table 3.1 to a probability distribution over a corpus of symbol names $s \in \mathbf{S}$ whereby $\mathbf{S}$ is the set of all function names seen in our training dataset. For *binary* and *dictionary* typed features we emit a normalised vector of size $|\mathbf{S}|$ that counts if the feature

has been seen during training for each function name. To clarify, if we find a binary hash collision or match VEX IR constants, we output a vector in which we sum the number of times the feature has been seen for each function name $s \in \mathbf{S}$. We stack the resultant scalars and vectors into a single feature vector to be used as the input to a machine learning classifier. As the feature vector is sparse, we reduce its dimensionality by performing Principal Component Analysis (PCA) and scale the transformed principal components such that each column has 0 mean and a unit variance.

Finally, we train a Gaussian Naive Bayes[3] model to predict the probability of each input function belonging to $s \in \mathbf{S}$. Our model is implemented using ScikitLearn [122].

## 3.3 Probabilistic Structural Inference

In this section we explain how we combine our probabilistic fingerprint with a third order general graph-based CRF for symbol inference. First, we explain how we generate the CRF (Section 3.3.1) using relationships between multiple symbols and features of individual functions. We then explain how *Punstrip* performs parameter estimation (Section 3.3.2) and model inference (Section 3.3.3).

### 3.3.1 CRF Generation

We refer to the process of symbol inference as predicting the most likely symbol names using a probabilistic graphical model that utilises unary potentials from our probabilistic fingerprint and known nodes, pairwise potentials between unknown functions, and generic factor potentials between sets of unknowns and knowns.

[3]We found Gaussian Naive Bayes out-performed Random Forests, Logistic Regression, and Neural Networks.

Table 3.3: Feature functions used in the CRF. *label-node* relationships relate known features $x \in \mathbf{x}$ to the current node $y_u$. *label-label* relationships relate unknown nodes $y_u \rightarrow y_v \in \mathbf{y}$.

| **Relationship** | **Description** |
|---|---|
| | *label-node relationships* |
| Probab. fingerprint | The probability of function $y_u$ given its extracted features in Table 3.1. |
| | *label-label relationships* |
| $d^{th}$ pairwise callers | The probability of function $y_u$ calling $y_v$ through $d-1$ other nodes. |
| $d^{th}$ pairwise callees | The probability of function $y_u$ being called by $y_v$ through $d-1$ other nodes. |
| Pairwise data xrefs | The probability of function $y_u$ referencing object $x_v$. |
| Generic factor callers | The probability of function $y_u$ calling the set of known functions $\mathbf{x}$. |
| Generic factor callees | The probability of function $y_u$ being called by the set of known functions $\mathbf{x}$. |

In general, CRFs are used to predict an output vector $\mathbf{y} = \{y_0, y_1, \ldots, y_N\}$ of random variables that may have dependencies on each other given an observed input vector $\mathbf{x}$. Our goal is that of structured prediction or learning high-level relationships between symbols. Modelling the dependencies between all symbols in binary executables would most likely lead to a computationally intractable graphical model, therefore we use a discriminative approach and model the conditional distribution $p(\mathbf{y} \mid \mathbf{x})$ directly without needing to model $p(\mathbf{x} \mid \mathbf{y})$.

As depicted in Figure 3.3, the CRF built is of the general graph form with relationships between known functions, known features (known features of unknown functions), and unknown functions. In our model,

known vertices represent feature values or known function names, e.g., $\mathit{Size} = 5, \mathit{name} = \mathit{read}$; unknown vertices represent unknown symbol names. Edges between nodes represent relationships between feature values of which we define two types: label-observation and label-label. Label-observation edges represent relationships connecting known nodes in $\mathbf{x}$ to unknown nodes in $\mathbf{y}$ and label-label edges represent relationships between unknown nodes in $\mathbf{y}$. Each feature function is replicated for each symbol name $s \in \mathbf{S}$. This is implemented as a vector $\mathbf{N}$ of size $|\mathbf{S}|$ with each element $n \in \mathbf{N} \rightarrow [0, 1]$. Our implementation exploits the sparsity between connected functions across millions of unique function names by storing each vector in a sparse matrix.

The feature functions used in building the CRF are listed in Table 3.3. For pairwise feature functions, we track dependencies to the $d^{\text{th}}$ degree for $d \in \{1, 2, 3\}$. To clarify, under the $\mathit{callee}_d$ feature function, each edge potential is a probability distribution (probability mass function) over all known symbol names $\mathbf{S}$ which describes the probability of the symbol name transition $s_u^d \rightarrow s_v^d$. This represents the probability of a symbol $s_u$ being $d$ calls away from $s_v$.

The CRF aims to predict the conditional probability over all unknown nodes $\mathbf{y}$ simultaneously given the set of known nodes $\mathbf{x}$. Let $G$ be a factor graph of relationships over all known symbols $\mathbf{x}$ and all unknown symbols $\mathbf{y}$, then $(\mathbf{x}, \mathbf{y})$ is a conditional random field if for any value $x \in \mathbf{x}$, the distribution $p(\mathbf{y} \mid \mathbf{x})$ factorises according to $G$. If we partition the graph $G$ into maximal cliques $\mathcal{C} = \{C_1, C_2, ..., C_P\}$ and into a set of factors $F = \{\Psi_c\}$, then the conditional distribution for the CRF is given by:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{1}{\mathcal{Z}(\mathbf{x})} \prod_{C_p \in \mathcal{C}} \prod_{\Psi_c \in C_p} \Psi_c(\mathbf{y}_c, \mathbf{x}_c; \theta_p) \tag{3.1}$$

where $\mathcal{Z}(\mathbf{x})$ is a normalising constant.

All of our label-label feature functions are discrete and return 0 or 1 for each function name depending on if the relationship exists in the training set. The weightings for pairwise feature functions are repeated for each clique and can be thought of as a global matrix between all function names $\mathbf{N} \times \mathbf{N}$. Then there exist $|\mathbf{N}|$ pairwise feature functions between a known and unknown node per relationship, e.g., for the first callee relationship, the probability of a known function being called by every other function in $\mathbf{S}$. We set $\Psi_c$ to be log linear for efficient inference and define it in the usual way as follows:

$$\Psi_c(\mathbf{y}_c, \mathbf{x}_c; \theta_p) = \exp\left(\sum_{k=1}^{K(p)} \theta_{pk} f_{pk}(\mathbf{y}_c, \mathbf{x}_c)\right) \tag{3.2}$$

whereby $K(p)$ returns the feature functions connected to vertex $p$. Both weightings $\theta_{pk}$ and feature functions $f_{pk}$ are indexed by vertex $k$ and factor $p$ implying that each factor has its own set of weights. As the graphical structure of binaries is not fixed, and hence the structure of our CRF, our implementation replicates the weightings of each feature function globally. The normalisation constant $\mathcal{Z}(\mathbf{x})$ is defined as:

$$\mathcal{Z}(\mathbf{x}) = \sum_{\mathbf{y}} \prod_{c_p \in \mathcal{C}} \prod_{\Psi_c \in C_p} \Psi_c(\mathbf{y}_c, \mathbf{x}_c; \theta_p) \tag{3.3}$$

### 3.3.2 Parameter Estimation

To estimate the weightings associated with the CRF we use a maximum likelihood approach, i.e., $\theta$ is chosen such that the training data has the highest probability under the model. We achieve this by maximising the pseudo log-likelihood given by Equation 3.4 over all our training set graphs $g \in \mathcal{G}$.

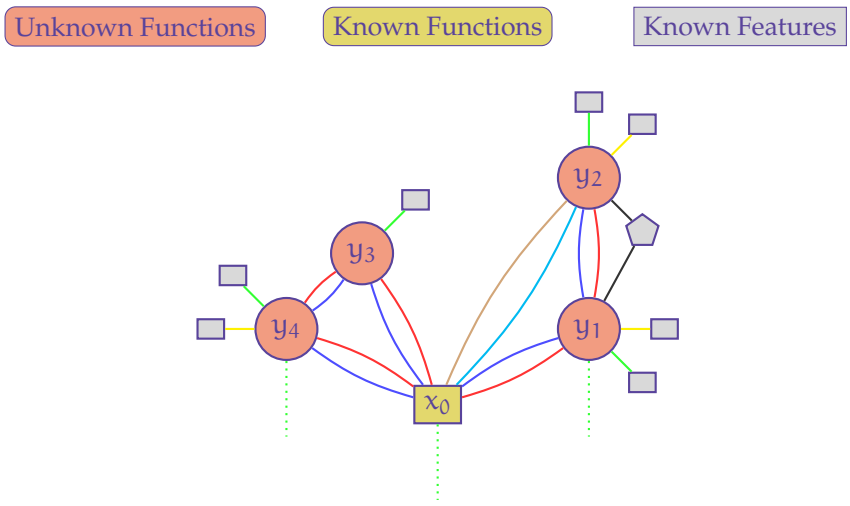



Figure 3.3: A visualisation of a snapshot of the general graph-based Condition Random Field showing known and unknown nodes and the relationships between them. Different types of relationships are represented by separate colours. Pairwise and generic factor-based feature functions are represented by rectangles and polygons.

$$\ell(\theta) = \sum_{g}^{\mathcal{G}} \sum_{p}^{\mathcal{C}} \sum_{k=1}^{K(p)} \theta_{pk} f_{pk}(\mathbf{y}_c, \mathbf{x}_c) - \frac{\theta_{pk}^2}{2\sigma^2} \tag{3.4}$$

As we aim to expect changes in structure and features, we regularise the log likelihood with Tikhonov regularisation [147] so that we do not overfit our model to the training dataset. We combine L-BFGS-B [29] on subgraphs in the training set with Stochastic Gradient Descent (SGD) to iteratively learn the optimal weightings for $\theta$. L-BFGS-B[4] is used to op-

[4]Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) is an optimisation algorithm that minimises a target function $f(\mathbf{x})$ over a real-valued vector $\mathbf{x}$. The function is assumed to be differentiable.

timise Equation 3.4 per-subgraph and SGD is used to update our global weights from the local maxima.

As we assume a large collection of independent and identically distributed samples in the training data, using a numerical approach to maximising the likelihood in a batch setting is unwarranted and needlessly slow. We suspect that different items in the training data from disconnected graphs provide similar information about relationship parameters; therefore, we opt to using a stochastic method for optimising the likelihood. While such an approach is sub-optimal, we believe the trade-off for training the CRF on big code is acceptable.

### 3.3.3 CRF Inference

Whilst exact inference algorithms exist for linear-chain and tree-based models, in the general case the problem has been shown to be NP-hard. For inferring symbol names from the CRF we employ Maximum a Posteriori (MAP) estimation using the approximate inference method Loopy Belief Propagation [108] combined with an optimised greedy algorithm based on stochastic gradient descent. We use an approximate inference method to model large and complex graphical structures with the possibility for many loops whilst still being a tractable model for convergence. Loopy Belief Propagation allows for an efficient approximation of marginal distributions for individual unknown node assignments. Our greedy algorithm then optimises global beliefs by making small changes to a subset of the most confident nodes in the model. This reduces the label space of possible assignments over $\mathbf{y}$ to a tractable amount that we can explicitly compute. Finally, we pick assignments that maximise Equation 3.1. In each run of Loopy Belief Propagation, we use a random permutation of message updates in an attempt to avoid remaining in local minima. By combining the two approaches it is hoped the model falls into a global

minimum rather than a weak local minimum. The use of the CRF gives the possibility of inferring functions that have large machine code differences to previous instances based on the interactions with other known and unknown functions that are more easily recognisable.

## 3.4 Lexical Analysis of Function Names

When inferring symbol names based on heuristics of the underlying code, it is difficult to know if the inferred name is correct. As previously mentioned in Section 3.2.4, multiple symbol names have exactly the same machine code, e.g., `xstrtol`, `strtol` and `__strtol` have the same byte sequence for the same compiler settings and come from different software packages. For this reason, we perform a series of measures adopted from NLP to compare the differences between the inferred symbol name and the ground truth.

We first pre-process all function names to remove common character sequences such as capital letters surrounded by underscores used to signify library versions, CPU extension named functions such as `function.avx512`, added compiler notation such as `function.constprop` and `function.part`, and ISA-specific naming of functions. This significantly reduces the number of unique symbol names stored in our database. Upon comparing the names of functions for a possible match, we first calculate the Levenshtein [93] distance between the symbol names to detect small changes similar to appending a suffix or prepending a prefix. Secondly, we perform canonicalisation and tokenization on both the inferred and ground truth before lemmatising and word stemming [62] each token in order to match words of different tenses and cases. Our implementation uses the Porter Stemming algorithm [153] and the WordNet lemmatiser provided by the Python Natural Language

Toolkit[5] library. This enables us to match the symbol `wd_compare` and `wd_comparator` based on the stemmed word `compar`. In name canonicalisation we maintain a list of common programming abbreviations such as `fd` for 'file descriptor' or `dir` for 'directory' and then use the dynamic programming rod cutting algorithm[6] to match sub-sequences with a scoring function that prefers longer word lengths to produce a set of word descriptions for each function name. For example, the real function name `hexCharToInt` after symbol canonicalisation is represented by the set $\{\mathtt{hexadecimal}, \mathtt{character}, \mathtt{to}, \mathtt{integer}\}$. We then use synonym sets from the Wordnet [106] lexical database for the English language to compare the synonyms of individual descriptions. Using synonym sets allows us to match the function name `float_stream` with `watercourse_drift` as *watercourse* and *stream*, and *drift* and *float* match in the Wordnet database. A naming similarity score is produced based on the Jaccard distance between the matching canonicalisation sets $x_c$ and $s_c$ as given by:

$$d_j(x_c, s_c) = \frac{|x_c \cup s_c| - |x_c \cap s_c|}{|x_c \cup s_c|} \tag{3.5}$$

The Jaccard distance gives a measure of the overlapping similarity in the synonyms of the canonical names for each function name. When the distance falls below a given threshold, we deem the function descriptions to be similar.

This method aims to implement a subjective match on the similarity between function names but may introduce false positives into our results. However, our thresholds and techniques were derived from manual analysis to align function name similarity close to the decisions of a human analyst.

[5] `https://www.nltk.org`

[6] The rod cutting algorithm is more precisely defined in Algorithm 1.

## 3.5 Evaluation

We evaluate *Punstrip* in two ways. First, we evaluate our probabilistic fingerprint (Section 3.5.1); for this we require a dataset that has source code compiled under different optimisation levels from different compilers. Second, we evaluate the combination of our probabilistic fingerprint with *Punstrip*'s probabilistic structural inference (Section 3.5.2) on a large scale.

### 3.5.1 Probabilistic Fingerprint

To evaluate our approach to inferring function names in previously unseen binaries we constructed a dataset of programs that have moderate code reuse between them. We built a corpus of binaries from *coreutils*, *moreutils*, *findutils*, *x11-utils* and *x11-xserver-utils*. This resulted in 149 unique binaries and 1,362,379 symbols. Our criteria for choosing these binaries were open-source software packages containing a large number of ELF executables. All of the binaries were then compiled under all combinations of $\{\texttt{og}, \texttt{o1}, \texttt{o2}\} \times \{\texttt{static}, \texttt{dynamic}\}$ for both *clang* and *gcc* resulting in 2132 distinct binaries[7]. We randomly split the 149 programs into 134 in the training set and 15 in the test set. All binaries with debugging information included were replicated to a stripped set of binaries before running the `strip` utility on them; this removed all symbols where possible (dynamic binaries still have dynamic symbols for linking purposes). By having two copies of the binaries, one stripped, the other with debugging information, we can obtain the ground truth for the results of our experiments. Previous work [11, 14] has shown that function boundaries can be identified in stripped binaries with an average F1 score of 0.95 across compiler

[7] We felt the difference in optimisation levels was sufficient for our experiment. At the time of experimentation, *o3* provided little difference over *o2* when using *clang*. *Clang og* is equivalent to *clang o1*.

optimisations O0-O3.

Throughout all our experiments, the same program name, and hence exactly the same source code was never in both the training and testing set. Thus 100% accuracy may be impossible as there are many functions only contained within the testing dataset; however, common pieces of source code may exist between binaries from the same package. We perform this experiment to evaluate how our probabilistic fingerprint recognises functions compiled into binaries under different compilation settings. By training a model on binary names for a given configuration of compilation options we then inferred function names for a different configuration of compilation options in the testing set. Our results can be seen in Table 3.4 for which we compare our fingerprint against leading industry tools IDA FLIRT and Radare 2 Zignatures. A comparison against BinDiff[8] proved impossible since it aims to perform differential comparisons between similar binaries rather than inferring function names in completely new binaries. We were unable draw a comparison against existing state-of-the-art research projects that build searchable code fingerprints such as BinGold [9] and Genius [58] because they were not fully available. We provide a larger evaluation of the entirety of *Punstrip* against the leading state-of-the-art research tool Debin [70] in Section 3.5.2 that combines program features and structural inference.

All our experiments were run under Debian Sid with dual Intel Xeon CPU E5-2640 and 128GB of RAM. On average the computation of feature functions after training was carried out in the order of seconds. We make our full dataset available online[9].

*Explanation of Results*.

In evaluating all schemes, we calculate Precision (P), Recall (R) and F1 score for as per Equation 3.6, 3.7, and 3.8. The number of true positives

[8] `https://zydnamics.com/software/bindiff.html`

[9] `https://github.com/punstrip/cross-compile-dataset`

TP, is given by the number of correctly named functions[10]. The number of false positives FP, is given by the number of functions that were named incorrectly. The number of false negatives FN, is given by the number of functions in which we did not predict a name (lack of prediction confidence), but valid names existed. We define the correctness of an inferred function name to be result of our NLP matching scheme (Section 3.4) between the inferred function and the ground truth.

$$P = \frac{TP}{TP + FP} \qquad (3.6) \qquad R = \frac{TP}{TP + FN} \qquad (3.7)$$

$$F_1 = \frac{2 \times P \times R}{P + R} \qquad (3.8)$$

All approaches performed worst in cross compiler, cross optimisation inference on dynamic binaries. From manual analysis, there are large differences in both the structure and interactions between functions and also in the number and name of functions. For example, *clang* always produces the symbol `c_isalnum` which is never present in binaries compiled by *gcc*. It's also worth noting that in general, the number of symbols in a binary decreased with higher levels of optimisation, with the *coreutils* binary *who* ranging between 80–130 functions for the dynamically linked case across optimisations `og`–`o3` for `x86_64`. The same program compiled under *clang* with *og* produced 129 symbols in its `.text` section whereas under *gcc* with *og* produced 106 symbols with 35 symbols that were not shared between the two binaries.

In the cases of very low recall, Zignatures's and FLIRT's precision rises. We attribute this to domain knowledge of ELF binaries with Radare2 always finding the symbol `__libc_csu_init`, a function with a size of 0 which without structure prediction, our fingerprint does not.

[10]For structural inference, *Punstrip* makes the assumption that *libc* initialisation (e.g., `libc_csu_init`) and deinitialisation (e.g., `fini`) functions can be found based on static analysis and the ELF header. This assumption was also applied when evaluating Debin.

Table 3.4: Evaluation on the accuracy of symbol inference of different corpora and the different compilation settings used.

| Experiment | IDA FLIRT | | | R2 Zignatures | | | *Punstrip* | | |
|---|---|---|---|---|---|---|---|---|---|
| | P | R | $F_1$ | P | R | $F_1$ | P | R | $F_1$ |
| `gcc,og,dynamic -> gcc,og,dynamic` | 0.94 | 0.38 | 0.47 | 0.51 | 0.72 | 0.60 | 0.99 | 0.85 | 0.91 |
| `gcc,og,dynamic -> gcc,o1,dynamic` | 0.95 | 0.14 | 0.24 | 0.36 | 0.37 | 0.37 | 0.84 | 0.61 | 0.70 |
| `gcc,og,dynamic -> gcc,o2,dynamic` | 0.30 | $< 0.01$ | $< 0.01$ | 0.29 | 0.04 | 0.07 | 0.52 | 0.37 | 0.44 |
| `clang,o1,dynamic -> clang,o1,dynamic` | 0.78 | 0.38 | 0.51 | 0.40 | 0.49 | 0.43 | 0.97 | 0.87 | 0.92 |
| `clang,og,static -> clang,og,static` | 0.61 | 0.18 | 0.29 | 0.13 | 0.16 | 0.14 | 0.997 | 0.90 | 0.94 |
| `clang,og,static -> clang,o2,static` | 0.60 | 0.17 | 0.27 | 0.12 | 0.14 | 0.13 | 0.98 | 0.87 | 0.92 |
| `clang,og,static -> gcc,og,static` | 0.61 | 0.16 | 0.26 | 0.11 | 0.12 | 0.11 | 0.96 | 0.82 | 0.82 |
| `clang,og,static -> gcc,o2,static` | 0.61 | 0.16 | 0.26 | 0.11 | 0.12 | 0.11 | 0.98 | 0.83 | 0.84 |

Table 3.5: 10-fold cross validation against Debin and *Punstrip*.

| Metric | Debin | | | *Punstrip* | | |
|---|---|---|---|---|---|---|
| | P | R | $F_1$ | P | R | $F_1$ |
| Exact | 0.63 | 0.66 | 0.51 | 0.65 | 0.92 | 0.73 |
| NLP | 0.66 | 0.67 | 0.55 | 0.68 | 0.92 | 0.75 |

### 3.5.2 Probabilistic Structural Inference

To test if we can learn abstract relationships between arbitrary functions in the general sense it is necessary to build a large corpus of binaries with debugging information from different software packages. We construct this dataset from thousands of open-source software packages from the Debian repositories.

This produced 188,253 binaries with debugging information from 14,000 different software packages resulting in 82GB of executables; we make the tools used to build this comprehensive dataset available[11]. Of the 188,253 ELF binaries, 17,549 binaries were compiled from the C language. We limit ourselves to C binaries only as we expect different relationships between functions across different languages. We are able to infer the source code language of an executable by looking at the `.comment`, `.debug`, and `.gnu.version` ELF header sections. Using the 17,549 C binaries, we randomly split the list of executables into 10 equally sized groups and perform 10-fold cross validation to evaluate our approach.

*Explanation of Results.*

In evaluating the performance of our probabilistic graphical model, we used the same metrics as in (Section 3.5.1), however we use two different measures of correctness. The first being an exact match between the canonicalised ground truth and our inferred function name, the second

[11] `https://github.com/punstrip/debian-unstripped`

being our NLP matching scheme. The reason for doing so is that the sparsity of function names across our corpus gives many names that are used in similar ways whilst still not evaluating as similar in our NLP matching scheme.

Table 3.5 displays the results of our large-scale inference experiment using 10-fold cross validation[12] From a detailed analysis of the results our NLP matching schemes correctly pick up meaningful inferred function names where the exact correct name is not present in the training set. Both tools perform worst on small dynamically linked binaries with little recognisable relationships. Furthermore, it is evident that *Punstrip* may infer symbol names that are structurally close on a micro-level to the correct names, however, they lie in a different or rotated graphical orientation. Symbol names with strong relationships between each other are often predicted locally correct as a group but not necessarily in the correct structural order which reduces our accuracy. For example, the functions `vfprintf`, `__vfprintf_internal`, and `buffered_vfprintf` are found close together in the call graph and jointly implement the functionality of `vfprintf`.

We build pairwise relationships up to the third degree and store factors involving up to three functions. To improve our tool's accuracy, we could trivially increase the dimensions of relationships stored between. We choose to use pairwise relationships up to the third degree as we were limited in our computational and storage resources for our large-scale experiments.

[12]Our results achieve significantly lower $F_1$ scores than our prior experiment due to the lack of common source code between binaries in the dataset.

## 3.6 Limitations

Throughout our approach, we rely on previous work [11] on function boundary detection to justify the conditions for function boundaries to be known. In real world environments, further errors may be introduced in the function boundary extraction stage which could have undesired effects in probabilistic inference due to the incorrectness of the recovered graphical structure. In the event of an erroneous recovered structure, we believe that sufficient randomisation of belief updates and our greedy algorithm's objective of maximising the joint likelihood across all unknown nodes simultaneously would mitigate incorrect inference.

Our probabilistic fingerprint may succeed when faced with small changes to machine code; however, there are often large unknown functions in previously unseen or obfuscated binaries. It is highly likely that *Punstrip* would perform poorly on executables that are highly obfuscated or contain handwritten assembly code. *Punstrip* is limited by the correctness of binary analysis; we make use of program analysis to recover features and relationships between data and code. Techniques which aim to mislead or impede program analysis are out of scope, however trivial obfuscation techniques such as junk code insertion should be overcome by the VEX IR optimisation step. *Punstrip* may be combined with existing reverse engineering software suites or debuggers to analyse regions of memory containing unlabelled code; the prime example being recognising functions during software *unpacking* at runtime.

## 3.7 Related Work

We examine related work across function identification (Section 3.7.1) and function fingerprinting (Section 3.7.2). Finally, we look at probabilistic

models for computer programs (Section 3.7.3).

### 3.7.1 Function Identification

The problem of matching sequences of binary code while allowing for variation presents itself in several domains. *Code clone detection* [83, 34, 124, 74, 57], *vulnerable code identification* [40], *code searching* [53], and *software plagiarism detection* [95] address the problem of finding exact source code matches between software components. They focus on finding a fixed set of previously seen functions with the main contributions drawn from methods of identifying semantically equivalent code that have undergone various software transformations; these transformations are typical of source code compiled with different compilers or compilation optimisations. Techniques typically adopt static or dynamic approaches that build features of a functions interpreted execution or rely on fixed properties of compiler generated machine code. *Patch code analysis* [74, 162] borrows the same techniques from the problem domain for feature collection, however, it requires an existing executable with prior information to perform analysis on differential updates.

State of the art reverse engineering tools, such as the IDA Pro disassembler, use databases of function signatures to reliably identify standard functions such as those included by a statically linked C run-time. This works well for systems where libraries are standardised and rarely recompiled. IDA Pro for example, maintains a directory of FLIRT signature files for the most common Windows libraries replicated across the most common compilers and instruction set architectures (ISAs). Reverse engineers also manually create custom databases of such signatures because they can immediately identify many functions which otherwise would have to be rediscovered in new binaries through costly manual analysis. Such signature-based mechanisms can allow for some variation in the exact byte

sequence matched, but they do not go further than relatively simple wildcard mechanisms.

### 3.7.2 Function Fingerprinting

Work in binary function identification predominantly focuses on the problems of clone or exact function detection. Code clone detection focuses on the recognition of previously seen functions [83]. *Punstrip* infers semantically similar names for previously unseen functions based on modified known examples.

*Unstrip* [75] aims to identify functions in stripped binaries and focuses on labelling wrapper functions around dynamic imports. BinSlayer [24], BinGold [9] and BinShape [138] identify and label functions in stripped binaries. They collect large numbers of features such as system calls, control flow graphs and statistical properties to fingerprint functions. Static approaches such as Genius [58] and discovRE [55] extract features from a binary's Control Flow Graph (CFG) and rank the similarity of functions based on the graph isomorphism problem. In contrast, *Punstrip* utilises a probabilistic graphical model that uses higher-level features to infer structure in stripped binaries; combined with our NLP analysis we suggest semantically similar function names. Structural Comparison of Executable Objects [60] finds vulnerabilities through analysing security patches. Gemini [155] creates a feature embedding based on Structure2vec [43] for code clone detection.

Dynamic approaches such as BLEX [53] and Exposé [115] use symbolic execution and a theorem prover to rank the similarity between *pairs* of individual functions. Egele et al. [53] employ symbolic execution and compare dynamic traces from functions to detect similar components. Others such as BinGo [34] and Multi-MH [124, 74] try to describe a functions behaviour by sampling each function with random inputs to match

known vulnerabilities across architectures and operating systems. Gupta et al. [109] use a dynamic matching algorithm for comparing control flow and call graphs.

### 3.7.3 Probabilistic Models

The seminal work of Bichsel et al. [22] in building probabilistic models is closely related to this work. They describe the process of building linear chain condition random fields for sections of Java bytecode based on a program dependency graph and utilise high-level information such as types, method operations and class inheritance to build relationships for inference. When applying a similar technique to machine code, the problem is exacerbated by the lack of access to concrete information on which to build features or known relationships to describe the semantics of code. The work was built on the JavaScript deobfuscation framework, JSNice [129], which infers local variable names for JavaScript programs using CRFs. Other works utilise probabilistic graphical models to infer properties of programs, e.g., specification [19] [96], verification [67] and bug finding [67]. The closest work to ours that labels functions in stripped binaries is Debin [70] which infers names of DWARF debugging information and function names simultaneously.

Recent advances in function boundary detection in stripped binary executables form a foundation of this work. We utilise Nucleus [11] when inferring function names without known function boundaries; a tool which uses spectral clustering to group basic blocks into function boundaries and results are an improvement over work by Rosenblum et al. [130] and Shin et al. [137], the former uses a CRF and the later use neural networks to detect function boundaries in binaries; however, neither performs the task of function naming.

# Distributed Representations of Binary Functions

# 4

This chapter introduces distributed representations of binary functions for use in machine learning applications. We present two different distributed representations: the first, *Symbol2Vec*, is a condensed representation over the label space of function names found in our experiments from Chapter 3; the second, *DEXTER*, is an embedding over the feature space of binary functions that improves on the sparse nature of *Punstrip*'s fingerprint using a deep learning approach.

*Symbol2Vec* aims to produce vector representations of function names found in binaries such that similarly used names are grouped together. It is useful for calculating the numerical similarity of inferred function names in the *label space* for traditional multi-class classifiers where no lexical similarity between the inferred and correct names exists. This overcomes one of the major limitations to the correctness of the approach previously described in Chapter 3. *DEXTER*, on the other hand, provides a new deep learning approach to *Punstrip*'s fingerprint that produces a dense vector representation of functions in which functions that have similar features have similar real values in the resultant embedded vector space. This may be used for a variety of applications including building a semantic similarity search engine for binary functions.

This chapter is based on papers originally published at ACSAC 2020 [119] and work submitted to S&P 2022 [120], both of which, the

author of this thesis was first author. We provide extended descriptions of *Symbol2Vec* (Section 4.1), *DEXTER* (Section 4.2), and evaluate both approaches (Section 4.3) in their respective settings. Finally, we review the related work (Section 4.4).

## 4.1 Symbol2Vec

Choosing an appropriate name for a function is a subjective goal in which different entities may choose different names for the same function. The majority of the time we would hope that these names are similar for functionally equivalent code and exhibit a subset of natural language features so that they can be compared by *Punstrip's* NLP matching stage (Section 3.4). Unfortunately, one programmer may choose a different name to that of another that does not match in the NLP comparison whilst still being functionally relevant. For example, consider the two real-world functions that start a network connection to a remote server and return a file handler named `init_connection` and `get_resource_handler`. The two functions share no lexical similarities and would be matched as different function names under normal conditions.

In the design of *Symbol2Vec* we create a numerical method to serve as a metric for name similarity which can alleviate this problem. We do this by projecting symbol names into a high dimensional vector space such that functions which are semantically similar appear close in the vector space and functions that differ are far apart.

### 4.1.1 Approach

With inspiration from *Word2Vec* [102], we modify the Continuous Bag Of Words (CBOW) and Skip-Gram model in order to create *Symbol2Vec*; a vector representation of function names. We replace the CBOW model with

our own Continuous Bag Of Functions (CBOF) that uses the call graph of binaries to generate sequences of function names that are related to each other through the *caller-callee* relationship. These sequences are then used in order to predict a target function name given the surrounding context function names. By using the dataset from our *Punstrip* experiments (Chapter 3) we were able to utilise a large collection of analysed binaries with ground truth symbol information. Sampling valid paths through a binary's call graph allows us to generate sequential data needed by contemporary sequence models.

In our CBOF model, as depicted in Figure 4.1, we use a context window of 2 and randomly sample a target function name with the associated context function names that are either a callee or caller of the target function. Due to the categorical nature of function names, we represent unique names in the *input* and *output* layers of our CBOF as one-hot-encoded vectors.

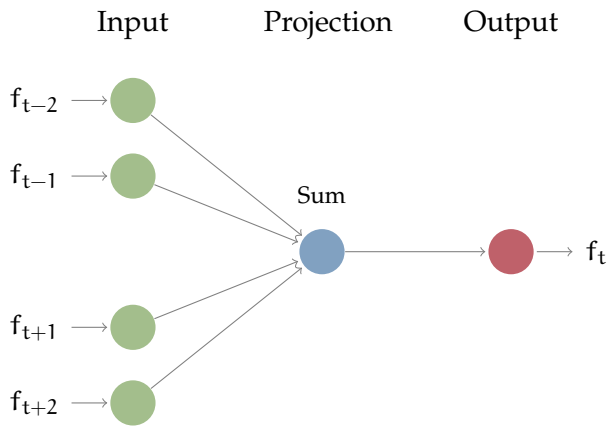



Figure 4.1: The CBOF model architecture predicts the current function name based on the context functions. The diagram shows how a sliding window of 2 is used to sum the context words in the projection layer.

Table 4.1: Examples of five target words and their closest vector representations in *Symbol2Vec* using the cosine distance.

| **Target Symbol** | **Closest Vectors in *Symbol2Vec*** |
|---|---|
| grub_error | grub_video_capture_set_active_render_target, grub_crypto_gcry_error, grub_font_draw_glyph, grub_disk_filter_write |
| opendir | readdir, closedir, dirfw, rewinddir, readdir_r, fdopendir |
| tls_init | tls_context_new, mutt_ssl_starttls, tls_deinit, eap_peer_sm_init, initialize_ctx |
| tor_x509_cert_get_cert_digests | tor_tls_get_my_certs, should_make_new_ed_keys, router_get_consensus_status, we_want_to_fetch_unknown_auth_certs |
| clock_start | clock_stop, lindex_update_first, lindex_update, index_fsub, index_denial |

After training the CBOF model, *Symbol2Vec* vectors may then be obtained by calculating the weights at the projection layer for individual one-hot-encoded vectors. In this way, we map unique function names in our corpus from one-hot-encoded vectors to condensed, distributed vectors.

### 4.1.2 Implementation

Our approach is implemented in Python 3 and TensorFlow [56] to create a autoencoder using 150 hidden nodes and 800,000 input and output nodes; one to represent each function name using one-hot encoding after a sub-sampling stage. We then train our neural network using Stochastic Gradient Descent (SGD) and Noise Contrastive Estimation (NCE) with negative sampling on a server with 256 GiB RAM and a Intel(R) Xeon(R) Gold 6142 CPU for three days to minimise the loss between predicting the pivot words from its context. The resulting weights of the hidden layer when activated by the input function name form the vector representation for each function name. We display function names and their nearest neighbours in our *Symbol2Vec* vector space for selected functions in Table 4.1, which demonstrates how semantically similar function names are grouped close together.

For a large corpus of function names, very common functions provide less information than rare function names. Therefore, we use a subsampling approach as per Mikolov et al. [104] to discard function names based on the probability defined by the following formula:

$$p(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{4.1}$$

where $f(w_i)$ is the frequency of the function name $w_i$ in our corpus and $t$ is an arbitrary threshold that we took to be $10^{-5}$. This step reduced our set of unique function names removing the likes of `malloc`, `free`, and `csu_init`.

## 4.2 DEXTER

Learning to classify abstract objects that are subject to many different features suffer from the curse of dimensionality. It is often easy to concretely describe features of objects but difficult to ascertain core features that strongly determine the class of an object. Traditional machine-learning techniques such as Principal Component Analysis (PCA) try to reduce the feature space by projecting along its eigenvectors in order to generate a condensed and informative feature vector that can be used in large-scale and real-world applications of machine learning. In this section, we use a deep learning approach that produces a condensed representation of binary functions, their calling context, and binaries themselves. We condense millions of sparsely populated features from our binary analysis into a single, 192-element, distributed vector representation for each function.

We start by giving an overview of our new DEXTER embedding (Section 4.2.1) before detailing its feature engineering (Section 4.2.2), the representation of context (Section 4.2.3), and the training process (Section 4.2.4).

### 4.2.1 DEXTER Overview

To create a distributed representation of functions we start by analysing binary code and creating a large collection of features for each function in a dataset. We separate the collected features into two sets; a dense set of numerical hand-crafted features such as the number of function arguments, and a large collection of sparse features representing categorical data automatically generated from our analysis (Table 4.2). We first design a feature set that captures as much information about a function and its context as possible. Our features are selected with the intuition that they are likely to remain similar through different compiler optimisations and compilation environments. We then build a deep autoencoder that embeds each function's features, the features of each function's context, and features from the binary into a compact distributed representation. The autoencoder learns the importance of individual features from our very large set and creates a distributed and condensed representation that can be used in machine-learning models and scales to perform on real-world problems[1].

### 4.2.2 Feature Engineering

To prepare the dataset for processing, we parse the ELF file headers of each binary and extract the boundaries of functions to obtain their binary code for training DEXTER embeddings. Where symbol information is available, function boundaries can be read directly from the binaries' program header. When extracting features from closed-source binaries, we are able to recover function boundaries and extend our analysis tool, *desyl*, to include stripped function extraction using Ghidra[2], IDA Pro, and Radare 2. However, to factor out the performance of function boundary prediction

[1] In particular we use the DEXTER embedding with a modified version of *Punstrip* to perform multi-label classification for function names in Chapter 5.

[2] `https://www.ghidra-sre.org`

from the overall task performance, we remove this step throughout our evaluation. For all tools, we take boundary data from the symbol table also for the binaries in the testing set. We limit the set of functions to *function symbols* that have a *GLOBAL* symbol binding in the ELF symbol table. This mitigates function definitions in the symbol table that are used as reference pointers to internal objects or have abnormal functional properties, e.g., a function with a size of 0 is impossible to detect and recover from a stripped binary.

We include in our feature set those derived by the analysis framework in Chapter 3, which lifts each function to an intermediate representation (VEX) and performs a symbolic analysis of each function in isolation. We analyse live memory addresses and register values used in each function to determine the number of input arguments. Finally, we extend our previous analysis to taint each input argument recovered and calculate flows to the individual arguments of callee functions. The full list of features we extract is shown in Table 4.2.

The feature vector for a function is built from concatenating two distinct vectors. The first, $\mathbf{f}_q$, represents dense, quantitative information that richly describe features of functions in executables. The second, $\mathbf{f}_c$, represents sparse, categorical data that is indicative of realised features that have no numerical basis, typically represented in a one-hot-encoded form. The two feature vectors are created by stacking the corresponding numerical and categorical features as depicted in Table 4.2. Each quantitative feature vector contains 512 elements such that $|\mathbf{f}_q| = 512$, and each categorical feature vector contains 3 million elements such that $|\mathbf{f}_c| = 3 \times 10^6$. After we have extracted $\mathbf{f}_q$ and $\mathbf{f}_c$ for each function in a binary, we use each function's feature vectors alongside the extracted binary call graph to build the DEXTER embedding.

Table 4.2: Features extracted from binary code that are used in the creation of DEXTER embeddings.

| Feature | Description |
|---|---|
| | *Numerical Features* |
| Size | Size of the symbol in bytes. |
| VEX instructions | Number of VEX IR instructions. |
| VEX jumpkinds | Sum of one-hot-encoded vector representing VEX IR jumps inside a function, e.g., *fall-through*, *call*, *ret* and *jump* |
| VEX temporary variables | Number of temporary variables used in the VEX IR. |
| VEX IR | Sum of one-hot-encoded vectors representing VEX IR Statements, Expressions and Operations. |
| Callers | The number of functions that call this function. |
| Callees | The number of functions this function calls. |
| Transitive Closure | The number of symbols reachable under this function. |
| Basic Block ICFG | Graph2Vec vector representation of the ICFG labelling with jumpkinds. |
| Stack bytes | Number of bytes referenced on the stack. |
| Heap bytes | Number of bytes referenced on the heap. |
| Arguments | Number of function arguments. |
| Stack locals | Number of bytes used for local variables on the stack. |
| Thread Local Storage (TLS) | Number of bytes referenced from TLS. |
| Tainted register classes | One-hot encoded vector of tainted register types, e.g., stack pointer, floating point. |
| Tainted heap | Number of tainted bytes of the heap. |

*Continued on the next page*

Table 4.2: Features extracted from binary code that are used in the creation of DEXTER embeddings (cont.).

| Feature | Description |
|---|---|
| Tainted stack | Number of tainted bytes of the stack. |
| Tainted stack arguments | Number of tainted bytes that are function arguments to other functions |
| Tainted jumps | Number of conditional jumps that depend on a tainted variable. |
| Tainted flows | Number of tainted flows to other functions. |
| | *Categorical Features* |
| Hash | Common SHA-256 hashes of binary code. |
| Opcode Hash | Common SHA-256 hashes of assembly opcodes. |
| Constants | Common constants referenced. |
| Dynamic Callees | The names of dynamically linked callee functions. |
| Transitive closure | Known function names reachable under this function in the binary call graph. |
| Data XREFS | References to known data objects in dynamically linked libraries. |
| Tainted flows | The names of dynamic functions and argument registers tainted when all input arguments are tainted. |

### 4.2.3 Function Context

DEXTER captures information of a function's local and global context; we pre-compute averaged vectors for a function's local context and binary context and concatenate them with our input feature vector.

First, we append the summation of feature vectors from each function's callers and callees in the binary's call graph. This allows DEXTER to cap-

ture complex information from surrounding functions to help identify it 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 that initialise parameters and then jump to the larger procedure implementing the actual functionality.

Second, we concatenate the summation of every function in the binary to attend to information recovered from our analysis of the whole binary. We build a modified feature vector, $\mathbf{f}_{mod}$, for each function in our dataset:

$$\mathbf{f}_{mod} = \mathbf{f}_{q} \oplus \mathbf{f}_{c} \oplus \frac{1}{|\mathcal{C}|} \sum_{c}^{\mathcal{C}} (\mathbf{c}_{q} \oplus \mathbf{c}_{c}) \oplus \frac{1}{|\mathcal{B}|} \sum_{b}^{\mathcal{B}} (\mathbf{b}_{q} \oplus \mathbf{b}_{c})$$

Here, $\mathcal{C}$ and $\mathcal{B}$ represent the set of functions in the target function's context (callees and callers) and all functions in the binary respectively. The vectors $\mathbf{c}_{q}, \mathbf{b}_{q}$ represent the quantitative function feature vectors of the corresponding function in each set, and $\mathbf{c}_{c}, \mathbf{b}_{c}$ represent the categorical function feature vectors.

### 4.2.4 Autoencoder Training

An unsupervised, deep autoencoder is then trained on our modified feature vectors for each function to create a dense, distributed embedding of functions. Our model's architecture is depicted in Figure 4.2. The model first creates dense representations for the function, the function's context, and the binary before combining them into a single embedding. Our methodology captures structural information not present when using features from each function in isolation. To enforce generalisation, we first connect the input layer into three dense sub-layers, each with 768 nodes, before performing batch normalisation and creating a dropout layer. These layers along with $L_1$ and $L_2$ regularisation on each dense sub-

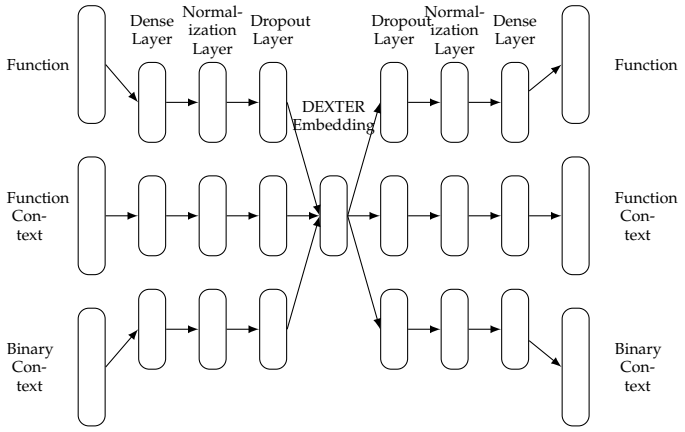



Figure 4.2: Overview of the autoencoder design used to generate DEXTER embeddings. The function context averages feature vectors from a target functions immediate callers and callees. The binary context averages feature vectors for every function from the target binary. The resultant middle layer forms our DEXTER embeddings once the network has been trained.

layer aim to prevent our model from over-fitting and force our model to learn an embedding that generalises well. Finally, we connect the output from all three dropout layers into a 192 node dense layer that we use as our embedded representation. The model architecture is reversed for predicting the output feature vector.

Training neural networks using standard loss metrics such as the Binary Cross Entropy or Hamming Loss is ineffective when there are an extremely large number of classes or we have sparse output data. To this end, we experimented optimising our model using the Adam optimisation algorithm with max-margin contrastive loss [68] and triplet loss [71]. We use these losses to produce an embedding in which functions with similar features appear close together in the embedded space.

Our model is created using Tensorflow [56] and aims to minimise the

contrastive loss between an input feature vector and a corresponding output feature vector. The contrastive loss is similar to the SoftMax function with the addition that the distance to positive samples is minimised and the distance to negative samples are maximised. When feeding our model pairs of input and output vectors, we randomly sample modified feature vectors from a dataset of analysed binaries. We experimented with subsampling feature vectors to achieve an equal number of vectors per function name and limited the number of data samples for the most common function names, however, no significant improvement in generalisation or performance was found.

## 4.3 Evaluation

In this section we evaluate our distributed representations for both the *names of functions* and *functions* themselves. We first present our evaluation approach for *Symbol2Vec* (Section 4.3.1) before evaluating DEXTER (Section 4.3.2).

### 4.3.1 Namespace Embedding

To evaluate *Symbol2Vec* we use a dataset that uses all function names found in or referenced by the code section of ELF binaries for C executables in Debian which resulted in 17,549 binaries, 5 million unique symbol names, and 1.1 million function names after our naming pre-processing step. Analogous to the classic *Word2Vec* analogy $King - Man + Woman = Queen$, we can reveal analogical relationships between function names. One such analogy is that between hash functions used in the *Nginx* software package, where the closest vector resulting from `ngx_md5_update` − `ngx_md5_init` + `ngx_sha1_init` is the function name `ngx_sha1_update` as shown in Figure 4.3.

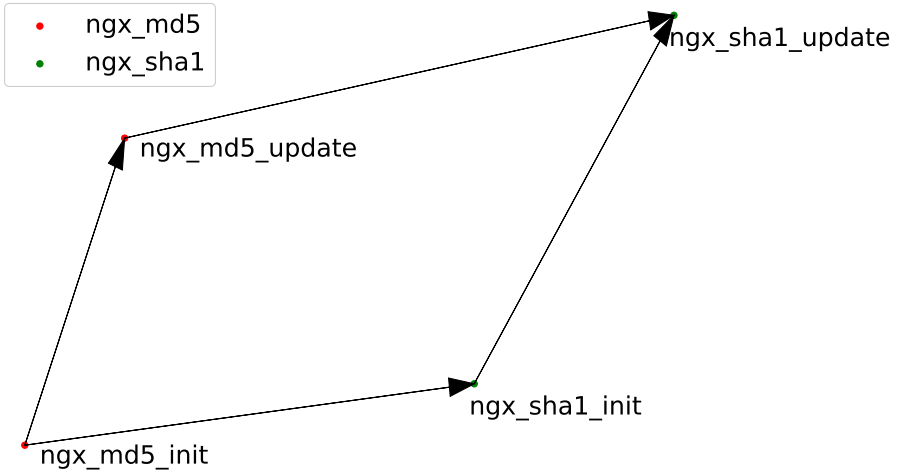



Figure 4.3: t-SNE plot of the closest vectors for the SHA-1 and MD5 hash algorithm relationship existing in *Symbol2Vec*.

The distributed representation of function names by *Symbol2Vec* allows us to numerically express the similarity of function names by calculating the distance between vectors in this vector space. We evaluate the effectiveness of our distributed representation by comparing the correlation between the generated representation and a manually created list of lexical relationships. Our analogies are in the form $\mathbf{a} - \mathbf{b} + \mathbf{c} \approx \mathbf{d}$. The full list of analogies can be seen in Table 4.4.

We measure the distance between function name vectors in the ordinal sense as we are unable to produce an exact continuous measure for our manual analogies. For each of our analogies we evaluate the preceding formula on the vector representations of each word and then rank each word vector based on its cosine distance to the new point with the closest vector having rank, and hence distance, 0.

$$r_s = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{4.2}$$

The Spearman's rank correlation is then calculated as per Equation 4.2 with $n$ being the number of observations and $d_i$ is the ordinal distance **d** is away from $\mathbf{a} - \mathbf{b} + \mathbf{c}$ when compared to all function name vectors for analogy $i$.

We show that *Symbol2Vec* is a meaningful representation and the distance between vectors in this space strongly correlates to their semantic similarity. We achieve a Spearman's rank correlation coefficient of 0.97 between *Symbol2Vec* and our manually crafted analogies. Thus, proving a strong correlation between our semantic analogies created by a human analyst and *Symbol2Vec*. In order for its use in the community and other similar applications, we release *Symbol2Vec* [78] open source.

*Punstrip Evaluation*. We use our *Symbol2Vec* function name representations to re-evaluate the results from Chapter 3. We use the cosine distance to calculate the five nearest neighbours from an inferred function name's *Symbol2Vec* representation. If the ground truth function name is contained within the closest five vectors, then we consider our inferred name correct. The re-evaluation of Punstrip with our *Symbol2Vec* metric can be seen in Table 4.3 and slightly improves our results. Metrics denoted with † are directly taken from our evaluation in Chapter 3.

When evaluating Debin and Punstrip using five nearest neighbours in *Symbol2Vec*, the resultant precision and F1 score for each tool increased. We hypothesise that function names matched in *Symbol2Vec* shared semantic meaning that could not be matched by the other metrics. We consider this metric useful to a reverse engineer who is presented with a list of 5 semantically similar function names whereby the correct function name

Table 4.3: 10-fold cross validation against Debin and *Punstrip*.

| Metric | Debin | | | *Punstrip* | | |
|---|---|---|---|---|---|---|
| | P | R | $F_1$ | P | R | $F_1$ |
| Exact† | 0.63 | 0.66 | 0.51 | 0.65 | 0.92 | 0.73 |
| NLP† | 0.66 | 0.67 | 0.55 | 0.68 | 0.92 | 0.75 |
| *Symbol2Vec* | 0.68 | 0.69 | 0.57 | 0.70 | 0.93 | 0.77 |

will be contained.

### 4.3.2 Featurespace Embedding

We evaluate DEXTER against the two state-of-the-art binary code embedding techniques Asm2Vec [49] and SAFE [100]. Both Asm2Vec and SAFE build binary code embeddings by modelling the language of assembly code and do not take into account complex features such as a function's ICFG. We show that our approach provides a distributed representation that captures more information and is more effective when used to predict labels of function names.

Using our dataset described in Section 3.5.1 we generate embeddings for all functions and randomly split them into a training, validation, and test set using a 90:5:5 ratio. Each function's embedded representation is then used to train a multi-label classifier. The multi-label classifier, PfastreXML, is used to predict string tokens that exist in function names rather than using Punstrip to predict whole function names[3]. Our evaluation compares DEXTER's impact when used to train a classifier in comparison to state-of-the-art function embeddings. We cannot prove that a particular embedding technique is inherently *better* for all tasks, however, we show

[3]Predicting labels such as *"str"*, *"network"*, or *"initialise"*, is explored in more detail in Chapter 5.

the relative performance difference for predicting textual name information when each embedding is used to train a classifier. The same training, validation, and test dataset are used for DEXTER, Asm2Vec, and SAFE. We run multiple experiments with varying sizes of label spaces to determine how increasing the size of the label space affects information gain[4].

Our results show the corresponding mean nDCG@5, DCG@5, CG@5 and Precision@5 for all embeddings as depicted in Figure 4.4. We can see that DEXTER outperforms both Asm2Vec and SAFE embeddings in a measure of rank scenario when inferring textual labels in the names of functions. We vary the size of the predicted label space between 512 and 4096 elements to understand how a larger label set affects performance. All embeddings exhibit a similar trend with a reduction in the nDCG as more information is needed to be learnt for the larger label spaces. With the increasing number of labels, the total amount of information needed to be learned and inferred increases, thus, information gain increases. At the same time, the proportionate amount of information gain recovered from the total information decreases due to the increased difficulty of the prediction task. We conclude that the semantic information gained from static analysis by DEXTER cannot be inferred from techniques such as Asm2Vec and SAFE. Thus, training a multi-label classifier to predict labels exhibited in function names performed better with DEXTER than the other methods.

## 4.4 Related Work

We structure this section to discuss the related work pertinent to both *Symbol2Vec* (Section 4.4.1) and DEXTER (Section 4.4.2).

[4]For details on how we generate label spaces and a detailed explaination of nDCG, DCG, and CG, see Chapter 5.

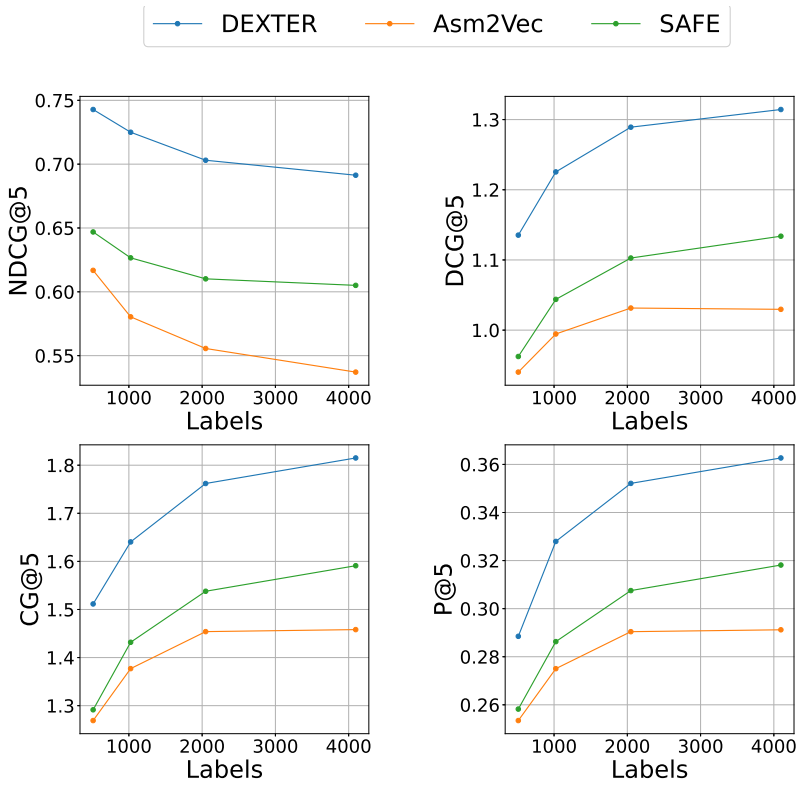



Figure 4.4: A comparison of information gain between different function embeddings across different sizes of label spaces. The Pfastre-XML algorithm was trained on DEXTER, Asm2Vec, and SAFE and used to rank every label for each data point. We record the Normalised Discounted Cumulative Gain, Discounted Cumulative Gain, Cumulative Gain and Precision of the ranked results.

### 4.4.1 Symbol2Vec

Independently of us, Daniel De Freez et. al. [47] implemented path-based function embeddings in a similar manner to *Symbol2Vec*. Prior distributed representations of binary functions exist, however, we believe we were the

first to build distributed representations for the *names* of binary functions.

### 4.4.2 DEXTER

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. The two major competing embeddings for binary code are SAFE [100] and Asm2Vec [49], which we use to compare against DEXTER.

SAFE uses a self-attentive neural network derived from recent developments in natural language processing. First, SAFE models assembly instructions using an adapted skip-gram method to retrieve vectors for each assembly instruction. It then models these sequences as a vector which is used to train a neural network and captures the inter-dependencies of specific instructions using an attention mechanism. This has the purpose of clustering similar functions in the resulting embedding space.

Asm2Vec follows a Paragraph2Vec-like approach [104], adapting it 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 [91]. Both Asm2Vec and SAFE are syntactic in nature, although they use static analysis for constructing the CFG and retrieving feasible traces of assembly instructions.

## 4.5 Summary

This chapter presented the design, implementation, and evaluation of two distributed representations: *Symbol2Vec* and DEXTER. Both embeddings are tangential to one another. We designed *Symbol2Vec* to overcome the limitation of matching the names of functions in binaries that shared no lexical properties. It uses the caller and callee relationship of a binary's call graph to match similarly named functions based on their contexts. Our results show that *Symbol2Vec* is useful for measuring the relative similarity between different *names* of functions and it provides us with an absolute numerical similarity metric to the subjective name similarity task by computing the distance between vectors.

DEXTER, on the other hand, was designed to provide a condensed representation of functions that captured information from a targets calling and binary contexts and is useful for identifying functions, and their names. It condenses sparsely populated categorical features, such as unique constants used in cryptographic functions, into a 192-element distributed vector representation. Our evaluation in this chapter compared DEXTER against two state-of-the-art binary function embedding techniques by using their associated vectors to train a multi-label classifier to predict tokens in the names of functions. The results show that the multi-label classifier was able to recover more tokens from function names using DEXTER than the other two approaches. We hypothesise that our embeddings are better suited for identifying and naming functions than the current state-of-the-art methods.

Table 4.4: Lexical analogies in *Symbol2Vec*, where $\mathbf{a} - \mathbf{b} + \mathbf{c} \approx \mathbf{d}$.

| a | b | c | d |
|---|---|---|---|
| sha1_init_ctx | md5_init_ctx | md5_update | sha1_update |
| realloc | malloc | xmalloc | xrealloc |
| fopen | open | close | fclose |
| icmp_open | open | close | icmp_close |
| hci_connect | connect | disconnect | hci_disconnect |
| close_log_file | fclose | fopen | open_log_file |
| sendmsg | write | recv | recvmsg |
| closepipe | close | open | openpipe |
| nxt_recv_file | recv | send | nxt_send_file |
| gethostent | get | set | sethostent |
| sha2_hmac_update | sha2_update | md5_update | md5_hmac_update |
| http_server_init | init | close | http_server_close |
| usb_bulk_write | write | read | usb_bulk_read |
| OPENSSL_CTX_init | init | free | OPENSSL_CTX_free |
| ssh_connect | connect | disconnect | ssh_disconnect |
| dhcpcd_config_get | dhcpcd_config_set | fopen | fclose |
| socket_accept | accept | close | socket_close |
| SQLConnect | connect | disconnect | SQLDisconnect |
| gethostname | get | set | sethostname |
| csu_fini | fini | csu_init | init |
| usb_bulk_send | send | recv | usb_bulk_recv |
| SHA384File | SH384_Init | SHA512_Init | SHA512File |
| Hread | read | write | Hwrite |
| SHA1File | SHA1_Init | MD5_Init | MD5File |
| btconnect | connect | disconnect | btdisconnected |
| EndDocFile | fclose | fopen | BeginDocFile |
| nxt_send_buf | send | recv | nxt_recv_buf |
| SHA256File | SH256_Init | SHA1_Init | SHA1File |
| cprng_deinit | free | malloc | cprng_init |
| unix_sock_open | open | close | unix_sock_close |

# Labelling Functions in Stripped Binaries

# 5

Chapter 3 addressed the problem of identifying function names in stripped binaries and Chapter 4 gave new foundations to matching similar names when performing multi-class classification of whole function names. In this chapter, we discuss the fundamental limitations to learning whole function names and introduce eXtreme Function Labelling (XFL), an eXtreme Multi-Label (XML) 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 demonstrate that XFL outperforms state-of-the-art approaches to function labelling on a dataset of over 10,000 binaries from the Debian project, achieving a precision of 82.5%. We also apply our labelling procedure to state-of-the-art function name prediction tools that perform multi-class classification to compare against relevant approaches in a multi-label setup. As a result, we are able to show that recovering textual information from functions names is best phrased in terms of multi-label learning, and that binary function embeddings benefit from moving beyond just learning from syntax.

This chapter comprises extended work of a paper written in collaboration with Moritz Dannehl and Johannes Kinder, for which the author of this thesis was the first author. My contributions to this work include the approach, implementation, and evaluation of XFL against other tools

and embeddings. We structure the following sections by first discussing the problem with multi-class classification for function name identification (Section 5.1) that is inherent with work presented in previous chapters. Then we describe our approach to multi-label learning (Section 5.2), evaluate our overall approach (Section 5.3) against the state of the art and discuss potential threats to validity (Section 5.4). Finally, we compare to related work (Section 5.5).

## 5.1 The Multiclass Classification Problem

Machine learning promises to enable 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 [70, 119] as done in previous chapters. The resulting systems learn models of the contents and structure of binary 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 unlimited. Even worse, the classes are heavily imbalanced, with most classes having a single sample and a minority of classes being over-represented (e.g., `main`). Normalising function names to remove differences in coding style such as `CamelCase` vs. `snake_case` can alleviate the problem to some extent [119], but still no such approaches can accurately predict function names that remain unseen after normalisation.

The drastic class imbalance is visible in Figure 5.1, which plots the fre-

quencies 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 (standardised names like `main`, `libc_csu_init`, etc.). Over 98% of function names occur fewer than 10 times, and 73% of function names occur only once. This long tail of single-sample classes demonstrates that any model for whole function names is doomed to mis-predict 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 5.1 shows that when this labelling approach is applied to function names in our dataset, 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.

An equivalent problem exists when trying to automatically tag text with the most relevant subset of labels from a large label set, which has motivated the research area of eXtreme Multi-label Learning (XML). Based on this insight, we show how to leverage state-of-the-art algorithms from XML for labelling functions in stripped binaries with *XFL* (eXtreme Function Labelling). *XFL* scales to millions of data points, features, and labels. It takes into account the power law distribution over labels and recognises infrequently occurring tail labels with little training data.

The architecture of *XFL* is shown in Figure 5.2. *XFL* is parameterised by a given *function embedding*, a distributed representation of each binary function. Intuitively, functions that a reverse engineer would consider similar should be mapped to vectors that lie closely together within the embedded space, and further away from vectors of dissimilar functions. *XFL* is compatible with state-of-the-art binary function embeddings such

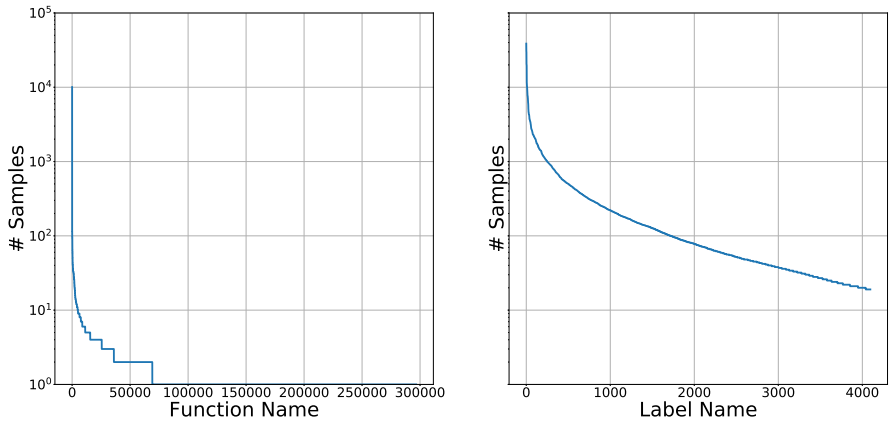



Figure 5.1: Semi-log plot showing the number of samples in each class when using function names and labels respectively (y-axis is shared). When predicting function names, the majority of function names have only a single sample.

as DEXTER, SAFE [100] and Asm2Vec [49].

With XFL, we introduce extreme multi-label learning as a solution to the problem of labelling binary functions (Section 5.2). XFL naturally solves the fundamental problems of sparsity and class imbalance in binary function labelling and provides information-theoretic metrics for a meaningful evaluation. In an extensive evaluation on a dataset with over 741,724 functions from 10,047 binaries, we demonstrate that our implementation significantly outperforms the state of the art in recovering textual information from the names of functions.

## 5.2 XML for Function Binaries

*XFL* uses PfastreXML and DEXTER (Section 4.2) to efficiently perform multi-label classification for tokenised labels of function names. We first

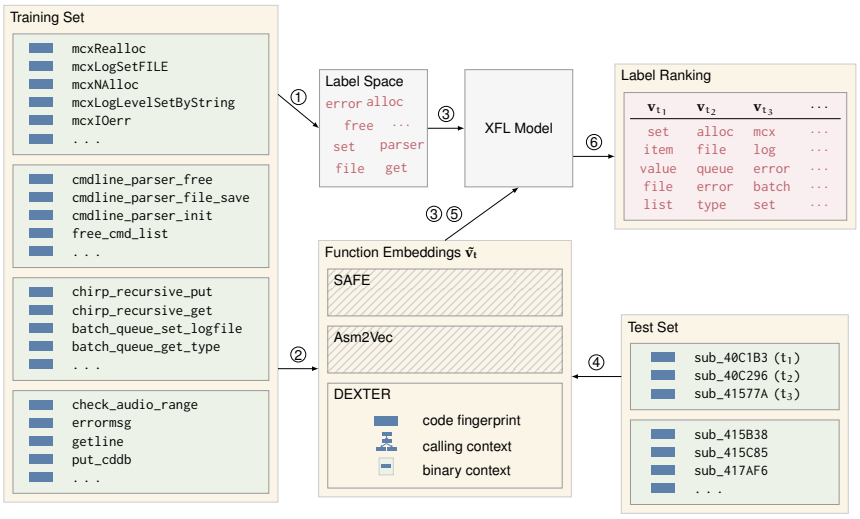



Figure 5.2: Overview of the XFL training and inference process. Function names in the training dataset are preprocessed to create the label space ①. Function binaries are used to train the embeddings ②. Function embeddings and the label space serve as input for training an XFL model ③. To infer labels for functions in a binary, an embedding $\mathbf{v}_{t_i}$ 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.

detail how *XFL* splits function names into tokens (Section 5.2.1) and generates a label space (Section 5.2.2), before describing how *XFL* uses Pfastre-XML to rank associated labels (Section 5.2.3) for functions.

### 5.2.1 Tokenising 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, consider 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 analysing the union of the resulting canonical token sets from the corpus of function names in a 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 recognise 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 *initialisation*. For example, `mkdirs` $\mapsto$ {`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 character

sub-sequences to find the largest collection of British and American English words. Our algorithm scores longer length English words higher over two or more same-length sub-sequences, e.g., $\{\texttt{background}\} > \{\texttt{back}, \texttt{ground}\}$. However, a longer length of total characters scores higher, e.g., $\texttt{foreach} \mapsto \{\texttt{for}, \texttt{each}\}$ and not $\{\texttt{reach}\}$.

The rod cutting algorithm is used to find the optimal method of cutting a single rod of length $n$ into multiple pieces where each length of rod has a price $p_i \in \mathbf{p}$. The goal is to maximise the revenue achieved by cutting the rod into smaller, integer length sections. To make longer scoring words score higher, our scoring metric, and hence rod price, is based on the square of the length of characters. The solution to this problem may be solved with a recursive pseudo code algorithm as detailed in Algorithm 1.

**Algorithm 1** Best Cut of The Rod

**Require:** Price Array $\mathbf{p}$
**Require:** Rod length $n$
  **function** CUT-ROD($\mathbf{p}, n$)
    **if** $n == 0$ **then**
      **return** $0$
    $q :=$ MinLen
    **for** $i = 1, \ldots, n$ **do**
      $q := max(q, \mathbf{p}(i) + \textit{CUT-ROD}(\mathbf{p}, n - i))$
    **return** $q$

We have confirmed that in practice this algorithm achieves splits of function names that correspond to common developer intuitions. However, we are unable to provide a quantitative measure for the accuracy of splitting textual strings into informative labels relevant for a reverse engineer. Providing this metric is an open question which one may wish to solve in future work.

### 5.2.2 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 explore the impact of an increasing number of labels or to restrict labels to a subset of the most informative and relevant string tokens. 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 our complete canonical token set such that $L_n \subseteq L$. While we cannot guarantee that the most informative labels are used most frequently and are thus retained in each label sub-space, we found that this method works well in practice.

### 5.2.3 Training an XML Model

After training all embeddings and generating a fixed size label space, we use the PfastreXML algorithm to train a model to learn labels from each function's canonicalised set. To obtain the ground truth of labels expressed for each function name, *XFL* first projects each name's canonical set $L_c$ onto those labels that exist in the generated label space $L_n$ as the intersection $L_c \bigcap L_n$.

To remove popularity bias (from our model) and recommend rare and novel labels we need to weight each label inversely to its popularity. Following the design and theory of PfastreXML, *XFL* optimises Equation 2.9 such that the log distribution of label occurrences in our training set matches the sigmoid function. Through the recommendation of the PfastreXML authors, we perform a grid search close to known good parameter values that optimises the nDCG calculated on our model's validation dataset when trained on a training dataset as discussed in Section 5.3.3.

The PfastreXML model predicts the probability distribution of labels in a label space being assigned to each function embedding. To produce a finite set of predicted labels for multi-label classification we optimise a threshold $p_t$, which selects a variably sized set of labels with a probability greater than $p_t$. The PfastreXML model acts independently and may be trained using arbitrary embeddings for binary code, such as DEXTER, Asm2Vec, or SAFE.

## 5.3 Evaluation

We now present our evaluation of *XFL*. We first explain the makeup of our dataset (Section 5.3.1) and discuss the metrics used in the evaluation (Section 5.3.2). Then, we evaluate *XFL* against comparable state-of-the-art approaches in function name prediction (Section 5.3.3).

*XFL* is implemented in Python and TensorFlow, and makes use of the PfastreXML classifier. We make our code available as open source to foster further research and enable like-for-like comparisons. All our experiments were carried out on a machine with dual AMD EPYC 2 64-Core CPUs and 1TB of RAM. The *XFL* project requires access to both Redis and PostgreSQL databases that store the results from our pre-processed binary analysis stage. Throughout our evaluation we used dedicated machines for each database server that contained at least 32 hyper-threaded cores and 256 GB of RAM.

### 5.3.1 Dataset

Training and testing require a set of ELF binaries with ground truth symbol information that is sufficiently large for generalising semantics of binary functions for each label. We use the Debian-based Punstrip dataset (Chapter 3), which contains over 1 million functions from 20,000

C binaries, compiled with a mixture of compilers and compiler versions from individual package maintainers. As we would expect a large amount of duplicated functions across statically linked executables, our dataset is formed from dynamically linked binaries only. This allows us to focus on learning a general model of functions from different source code rather than exact copies of replicated functions.

When using the ELF symbol table as the ground truth for obtaining function boundaries and their corresponding names, it became apparent that using all function definitions is unhelpful and misleading; with many entries corresponding to overlapping functions, zero-sized non-existent functions, or functions that are used as virtual address definitions with relative jump offsets. Therefore, we limit the set of function definitions used in our evaluation to those with a *global* symbol binding. Furthermore, *locally* bound function symbols allow compilers to modify procedures beyond standard definitions for optimisation, potentially violating the application binary interface. In practice, this limited our target set of functions to those that were well-defined and whose boundaries were much more likely to be correctly recovered by external function boundary prediction tools when applied to stripped binaries[1].

When training an XFL model, for each embedding we first trained a model on our training set. Next, we performed a grid search to fine-tune the models' hyper-parameters over our validation dataset. Using values close to known good parameters, we optimise the $\alpha$ and $\gamma$ hyper-parameters[2] of PfastreXML to minimise our nDCG loss as shown in Figure 5.3. The application-specific values that optimised our loss for the DEXTER were $\alpha = 1.025$ and $\gamma = 28.75$.

[1] We do not implement any mechanism to ensure that exact function duplicates are removed from our dataset.

[2] $\alpha$ and $\gamma$ are PfastreXML hyper-parameters that affect the final re-ranking of labels and the hyper-spherical decision boundary of the tail label classifiers. The full details of how these parameters effect the prediction results can be found in Jain et al.[77].

Table 5.1: Analysis of the embedding datasets. Average labels per point and points per label were calculated for a label space of size 4096.

| Property | DEXTER | Asm2Vec | SAFE |
|---|---|---|---|
| Train Samples | 400,357 | 396,796 | 342,610 |
| Embed. Dimensions | 512 | 50 | 100 |
| Test Samples | 22,422 | 22,224 | 19,035 |
| Average Labels per Point | 2.9271 | 2.9254 | 2.9295 |
| Average Points per Label | 320.45 | 317.43 | 255.24 |

### 5.3.2 Metrics

We evaluate *XFL* using metrics widely adopted for XML and ranking tasks [21, 3, 72, 127, 151, 152, 157]. One such metric is multi-label precision at k (P@k), which is used to count the fraction of correct predictions in the top k scoring labels. We include the precision@k for comparison against other tools and datasets; however, it has notable problems as an evaluation metric. The principal issue with precision@k is that it ignores the order of predicted labels for a given k. Therefore, we include the more intuitive metrics previously described in Section 2.4.2: cumulative gain at k, discounted cumulative gain at k, and normalised discounted cumulative gain at k. Based on our analysis of the dataset as shown in Table 5.1, we find each sample contains an average of 2.9 labels, and each label contains an average of 297 training samples. Few function names consist of more than five meaningful sub-tokens, and five labels with their associated probabilities would be easily processed by an analyst in real-world applications. Therefore, we set $k = 5$ for all of our evaluation metrics. We feed each tool the same dataset to learn and generate binary code embeddings from; the difference in training sample sizes were due to errors in processing the dataset and different approaches to handling or recovering

function boundaries[3].

In Section 5.3.3 we evaluate *XFL* in the traditional multi-label classification setup and use it to predict the relevant subset of correct labels assigned to each function name as opposed to ranking all labels. In this case we report the micro-average precision, recall, and $F_1$ for multi-label classification as defined below:

$$P_\mu = \frac{\sum_l^L TP_l}{\sum_l^L TP_l + \sum_l^L FP_l} \quad (5.1) \qquad R_\mu = \frac{\sum_l^L TP_l}{\sum_l^L TP_l + \sum_l^L FN_l} \quad (5.2)$$

$$F_{1\mu} = \frac{2 \cdot P_\mu \cdot R_\mu}{P_\mu + R_\mu} \quad (5.3)$$

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 (for our specific experiment definitions, see Section 5.3.3). In choosing micro-averaged over macro-averaged, we mitigate problems with label bias from the heavily imbalanced dataset[4]. The equations given sum over all labels for each metric rather than calculating metrics per label and then averaging. With heavily imbalanced datasets, division by zero errors may occur if there are no true positives and no false positives for a single label in the test set.

[3] A more precise comparison would entail training on the intersection of functions that embeddings were generated for across all tools.

[4] Note that *macro*-averaging is very sensitive to labels with few occurrences. Macro-averaged scores compute the metric independently for each label and then compute the average, which treats all labels equally. *Micro*-averaged scores aggregate the contributions of all labels to compute a single metric. In choosing micro-averaged over macro-averaged, we therefore mitigate problems with label bias from the heavily imbalanced dataset.

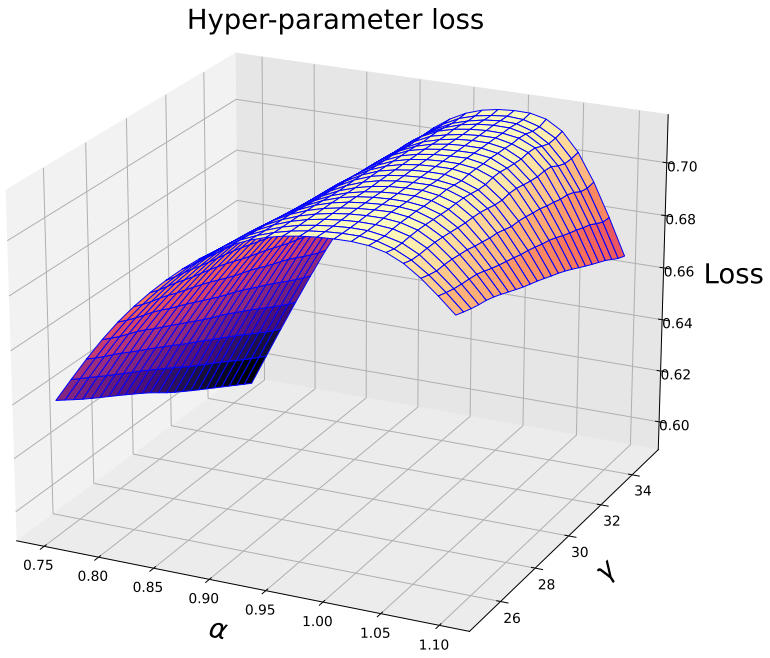



Figure 5.3: Grid search of hyper-parameters on the validation dataset for $\alpha$ and $\gamma$ when calculating the nDCG@5 loss.

### 5.3.3 Function Labelling

We evaluate *XFL* against four state-of-the-art function name prediction tools: Dire, Debin, Nero, and Punstrip. On the recommendation from the Dire authors, we did not use their tool for function name prediction; however, we include the Dire results reported in the Nero paper [45], which are based off a customised version. Nero [45] combines heavy static analysis to obtain an Augmented Control-Flow Graph (A-CFG) and feeds this representation into state-of-the-art neural networks. In their work, the variant based on a Graph Neural Network (GNN) performs best, so we use this variant to compare against. Debin [70] and Punstrip [119] both make predictions for function names based on Conditional Random Fields to

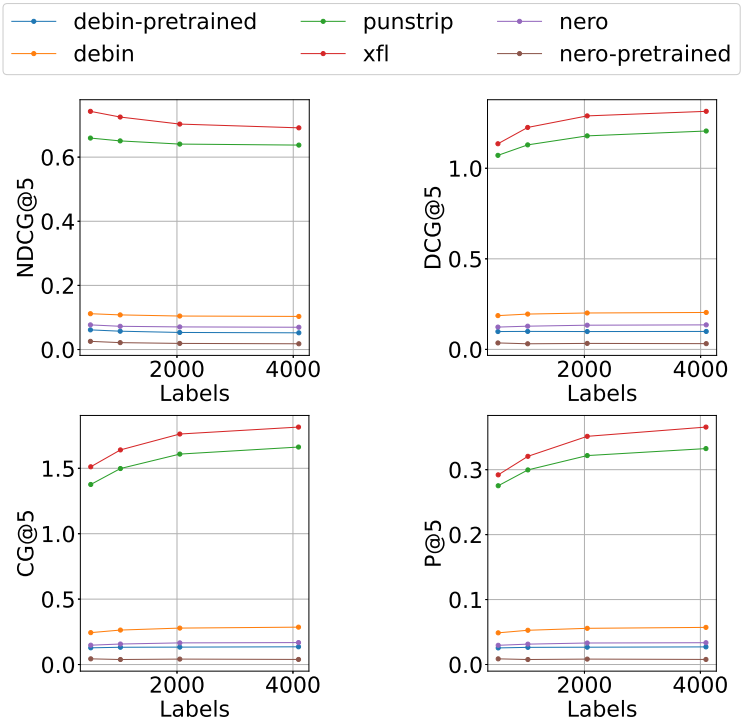



Figure 5.4: An information theoretic comparison between Debin, *XFL*, Nero, and Punstrip across multiple sized label spaces. All metrics were taken @5 and a default order was added where tools unable to rank all classes predicted less than five labels.

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.

We first randomly split our dataset obtained in Section 5.3.1 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 the tools. We generate label spaces for $L_n$ with $n \in \{512, 1024, 2048, 4096\}$.

We evaluate all tools in two parts, exploring *different* discrete problems. In both problems we use each tool to predict relevant labels contained in function names. The first problem 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 evaluates each tool in a multi-label classification setting predicting a finite set of relevant labels.

#### Measure of Rank

In evaluating against the measure of rank, we require each tool to rank each label in the correct order for all data points. We evaluate this experiment using the information gain metrics nDCG@5, DCG@5, CG@5, and P@5 as depicted in Figure 5.4. Intuitively, an nDCG@5 of 1.0 is achieved by perfectly ranking the top five labels for each sample. An nDCG@5 of 0.0 is achieved by ranking labels such that no relevant labels are in the top five for each data point. For tools that do not predict labels, we apply the inferred function name to our canonical set generation and project it over the given label space to carry out a like-for-like comparison between all tools. If there are no valid labels for a sample in the label space, as may happen in smaller label space sizes, we skip the calculation.

Every tool evaluated performed weaker in terms of nDCG as the size of the label space increased. This is explained by the fact that by increasing the size of the label space, we increase the total information that each model has to learn. Precision and other cumulative gain metrics increased with an increase in the size of the label space as more tokens were available to be predicted and thus increased the overall information.

Where tools that predicted whole function names inferred the correct

name, they also correctly predicted the entire set of labels for the data point. We suspect this influences Punstrip's high performance in multi-label classification and that its label predictions were more skewed to polar predictions in having all labels correct or incorrect. The other tools results were much less binary showing the ability to pick out individual relevant labels on more test samples.

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 (Section 5.3.3), 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 publications to machine learning on binary code, which we attempt to counter with an increased dataset size and using an embedding-based approach that learns relative distances between functions.

#### 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 val-

Table 5.2: Multi-label classification evaluation against state-of-the-art tools for the Debian and Nero datasets. Results marked with † are taken directly from David et al.[45] without re-running the experiment.

| | **Debian Dataset** | | | **Nero Dataset** | | |
|---|---|---|---|---|---|---|
| **Tool** | **Prec.** | **Recall** | **$F_1$** | **Prec.** | **Recall** | **$F_1$** |
| *XFL* | 0.8248 | 0.5650 | 0.6706 | 0.8561 | 0.4293 | 0.5718 |
| 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† |

idation, we set $p_t$ to a value that maximises the $F_1$ score on the validation dataset.

Our results as shown in Table 5.2 display the micro-averaged multi-label precision, recall, and $F_1$ score on both our Debian dataset and the Nero dataset using a label space with 1024 elements. *XFL* outperforms other state-of-the-art tools in precision and $F_1$, 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 on 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 generalised results, 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 optimisation levels are not fixed.

By ranking relevant labels by their confidence and joining them by an underscore or in camel case, we can reconstruct and predict whole function names. Table 5.3 shows complete examples of such predictions, including typical errors that can be introduced through the function name generation process. Some strings may be split along boundaries that a reverse engineer would not, e.g., due to spelling mistakes (see the `GAME_IsMissionAccomplied` example). Finally, labels in the generated name are ordered by their confidence, which may seem unnatural or even change the meaning of the associated function name. We leave the clean-up of ground truth data and the generation of more palatable whole function names to future work.

## 5.4 Threats to Validity

We now discuss potential threats to validity of our findings, based on limitations in tooling and dataset composition.

### 5.4.1 Differences in Ground Truth

*XFL*'s symbolic analysis is based on Punstrip and angr's Claripy [140]. It depends on the Radare2 API for reading symbol information and extracting raw data from binaries. Our choice to use the ELF symbol table as our ground truth is a decision that may differ between other tool implementations. Nero, for example, extracts ground truth information from IDA Pro's own analysis which renames common routines; e.g., the entry function is renamed to `.init_proc`. In evaluating against Nero, we took care of mapping function's original names as defined in the ELF symbol table to

Table 5.3: 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. Labels from the canonicalisation process that did not make it into the label space are crossed out.

| **Original Name** | **Ground Truth Tokens** | **Predicted Labels** | **Output Name** |
|---|---|---|---|
| `__libc_csu_init` | {libc, csu, init} | {libc, csu, init} | `libc_csu_init` |
| `grub_crypto_cbc_decrypt` | {decrypt, grub, crypt, cb, ~~e~~} | {decrypt, grub, crypt, cb, odisk} | `grub_decrypt_crypt_cb_odisk` |
| `make_smooth_colormap` | {make, color, map, smooth} | {make, color, map, random} | `color_make_map_parse_random` |
| `mcxRealloc` | {realloc, mcx} | {realloc, mcx} | `mcx_realloc` |
| `check_audio_range` | {range, audio, check} | {audio, range, check} | `check_range_audio` |
| `HIDSetItemValue` | {set, item, hid, value} | {set, item, hid, value, is} | `set_item_value_hid_is` |
| `mcxTingRelease` | {release, mcx, ~~ting~~} | {release, mcx} | `mcx_release_free` |
| `chirp_recursive_put` | {put, recursive, chirp} | {put, recursive, chirp, ticket} | `recursive_chirp_ticket_put` |
| `lkfopendata` | {lkf, data, open} | {lkf, open, switches, data} | `lkf_open_switches_data` |
| `SECU_PrintCertNickname` | {nickname, print, cert, sec, ~~u~~} | {nickname, cert, sec, print, collision} | `nickname_cert_sec_print_collision` |
| `cmdline_parser_init` | {line, init, cmd, parser} | {parser, init, cmd, line, csu} | `parser_init_cmd_line_csu` |
| `GAME_IsMissionAccomplied` | {complied, ac, is, mission, game} | {complied, game, mission, msrs, is} | `complied_game_mission_msrs_is` |

those retrieved by different analysis platforms, such as IDA Pro, and used the tool's ground truth data in order to give a fair comparison.

### 5.4.2 Quality of Training Data

This work is based upon the hypothesis that programmers usually provide useful function names that contain reasonable information. Even though this approach is not fault-proof because the splitting of names into labels when generating the ground truth can go wrong, we are trusting the programmer's intuition and assume this to be a robust method compared to, say, using a doc2vec summation of man page descriptions of the functions or comments in the source code. Table 5.3 specifically shows an error in our unsupervised label space generation algorithm with `grub_crypto_cbc_decrypt` function. The acronym for *Cipher Block Chaining* (CBC) is unknown to our tokenisation process and thus is split into `cb` which is commonly used for *callback* routines.

### 5.4.3 Label Spaces

As Debin and Punstrip only predict whole function names, we can impose predicted labels in our generated label space by projecting string tokens in our canonicalisation step. Nero, however, is capable of predicting labels from the names of functions using their own method. While our results impose our own label space from their predictions, the values of $F_1$, precision, and recall differ only slightly from those reported by their tool on their own label space.

### 5.4.4 Dataset Biases

The results of an evaluation may be biased towards the dataset used and models may over-fit learning parameters. The problem of learning whole

function names is that any predicted name must be seen in the training set. Likewise, we must be careful when separating training, validation, and testing datasets to ensure a sufficient separation of common code from the same software packages and statically linked libraries. Our results for multi-label classification show that Nero is outperformed by Debin; a surprising outcome given Nero's own evaluation against Debin purported much better results. After contacting the Nero authors about this discrepancy, they suggested that it may occur for the following reasons:

1. 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.

2. The vocabulary size in our dataset is significantly bigger and as a result Nero predicts empty labels 43% of the time.

3. Nero's configuration and implementation had not been fine-tuned to our dataset.

To give a similar comparison we evaluated *XFL* against Nero's dataset as reported in Section 5.3.3. Furthermore, for fair comparison to our own work we make our own dataset and tool publicly available.

### 5.4.5 Obfuscation and Multiple ISAs

We suspect *XFL* would not perform well on labelling functions in obfuscated malware. However, it may provide a base for identifying common unpacking routines such as those provided in a UPX packed binary and could be used to analyse memory dumps once malware has been unpacked. While we only evaluated *XFL* applied to non-obfuscated `x86_64` Linux ELF files, the same techniques also apply to Windows PE and Mach-O files with basic support for PE files already included. Our evaluation

did not cover these executable formats due to a variety of reasons: firstly, we were unable to obtain a large collection of prebuilt executables with symbol information that covered various compilers and optimisation levels sufficient to learn a generalised model of binary code. Secondly, APIs and programming styles change between Operating Systems and ISAs; we suspect an *XFL* model trained per executable format and ISA to give best results.

### 5.4.6 Pretraining

Presently, DEXTER utilises high level features extracted from binary functions. During the feature extraction and vectorisation process, Graph2Vec is used to generate a vector representation of each function's intra-procedural control flow graph (ICFG). This unsupervised pre-processing step requires all possible ICFGs to be known before a model can be trained, such that the stripped target binaries have to be available during training. Currently, this prevents full pretraining of the DEXTER embeddings, but could be mitigated by choosing to use a generative graph embedding in place of Graph2Vec.

## 5.5 Related Work

We now review related work on the problem of binary function labelling (Section 5.5.1), followed by the related research areas of binary code similarity (Section 5.5.2) and of learning source-level code representations (Section 5.5.3). Finally, we discuss related work in the area of extreme multi-label learning (Section 5.5.4).

### 5.5.1 Binary Function Labelling

The only work to predict textual tokens of function names as labels is Nero [45]. In their work, they build an *augmented control flow graph* (A-CFG) which extends a CFG with call sites, specifically engineered for procedure name prediction. The representation is fed into GNNs, LSTMs and Transformer architectures. Dire [88] 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 tokenised code and the AST from a decompiler and generates embeddings for each identifier which are used by the decoder to predict names.

### 5.5.2 Binary Code Similarity

Much recent work focuses on producing vector embeddings for basic blocks, functions, or binaries that aim to serve the task of *binary code similarity*. DeepBinDiff [50] 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 [155], different source code versions [94] or different compilers and compiler settings [49].

*Natural Language Processing* is often used as an analogy to compare assembly code to natural language. One example is SAFE [100], which utilises a recurrent neural network with Gated Recurrent Units (GRU) to model assembly instructions as sequences. An attention mechanism is added to put emphasis on the most relevant parts of the code. Asm2Vec [49] adapts the Word2Vec [104] model to assembly syntax to generate embeddings for tokens such as assembly opcodes, operands and functions. It does so using the distributed memory model similar

to the Paragraph Vector Distributed Memory (PV-DM) model of Paragraph2Vec [91].

Zuo et. al [161] rely on Word2Vec, but adapt *Neural Machine Translation* to handle single instructions as words and basic blocks as sentences. Sun et al. [142] use approaches found in bioinformatics such as longest common sub-sequence algorithms in conjunction with Word2Vec embeddings to measure binary semantic similarity.

### 5.5.3 Source Code-based Approaches

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 utilise syntax information from Abstract Syntax Trees [23] [8]. Program representations may be constructed using generative models [25], graph neural networks [5], graph models enriched with sequence encoders [59], or attention-based models [7]. A convolutional network is used to summarise source code to tokens [6].

A different family of problems are those dealing with the authorship of code. Those approaches focus on identifying, verifying, or clustering authors as well as analysing the evolution of programmer skill [82].

### 5.5.4 eXtreme Multi-label Learning (XML)

Two main approaches exist to solve the XML problem: embedding-based and tree-based methods.

#### Embedding-based methods

Bhatia et. al. [21] present a state-of-the-art embedding method that aims to address the challenges of XML. Traditional embedding based approaches project a label space L into a linear lower-dimensional subspace $\hat{L}$. The linear constraint is required for post-processing in the prediction stage in which a decompression matrix is used to lift embedded label vectors back to the original label space. SLEEC [21] disregards the global linear low-rank subspace used by embedding methods and learns an ensemble of non-linear embeddings which preserve pairwise distances between label and embedding spaces for only the nearest neighbours of individual data points. During prediction, rather than using a decompression matrix, SLEEC uses a k-nearest neighbour classifier in the embedding space. Recent advancements by DeepXML [42], Slice [76], and DECAF [107] bring deep learning approaches to the XML field and report the highest embedding based prediction accuracies.

#### Tree-based methods

Tree-based XML methods such as PfastreXML and Parabel [126] aim to learn a hierarchy of labels using a node partition function. The tree's root node is initialised to contain the entire label set and child nodes partition the parent's labels until leaf nodes contain only a few labels. A multi-label classifier of choice is then trained with the subset of labels present in each leaf node. Traditional tree-based XML classifiers use task independent measures to learn this hierarchy such as Multi-Label Random Forest's (MLRF's) [3] Gini index and Label Partitioning by Sublinear Ranking's (LPSR's) [152] clustering error. However, FastXML [127], a direct precursor of PfastreXML, optimises a nDCG based ranking loss function to partition the feature space which leads to more accurate predictions. The intuition behind forming a hierarchy over the feature space rather

than the label space is from the observation that only a small number of labels are present in each region of feature space. Label prediction can then be efficiently computed by carrying out multi-label classification on a small subset of labels active in the region of feature space learnt in the hierarchy. PfastreXML, along with SwiftXML [125], provide significant improvements in prediction accuracies over embedding based methods and are state-of-the-art approaches to solving the XML ranking problem.

## 5.6 Summary

This chapter explored the problem of recovering tokens of function names in stripped binaries. It built on our binary analysis framework presented in Chapter 3 and binary code embeddings presented in Chapter 4. We have shown that XFL can be used to learn a general model of informative labels contained in function names and are able to infer tokens in previously unseen stripped software. Whilst we show that learning an XML model to infer naming components out-performs the state-of-the art in terms of information-theoretic metrics, we believe the real benefit of XFL over Punstrip is in its generalisation of inferred tokens. In our evaluation, tools that did not perform multi-label classification were modified to do so by applying our tokenisation procedure on the inferred symbol name. This allowed those tools to correctly predict every label if they matched the inferred function name correctly. We believe that this made Punstrip very precise but at the lack of generality. Therefore, we suggest that Punstrip would be better suited to *code clone search* as it is able to uniquely identify whole function names. On the other hand, XFL provides a more general approach to labelling functions that is useful for reverse engineers to quickly understand the semantics of binary code.

# Probing USB Drivers Through Symbolic Fault Injection of Known Functions

# 6

In this chapter we explore a theoretical use case of *symbol recovery* presented in previous chapters, and apply it to software fuzz testing to develop a new approach to finding software bugs in kernel drivers. We present the design and implementation of *POTUS* (**p**robing **o**ff-**t**he-shelf **U**SB drivers with **s**ymbolic fault injection), a system for automatically finding vulnerabilities in USB device drivers for Linux, which is based on fault injection, concurrency fuzzing, and symbolic execution. We target Linux USB drivers due to the popularity of USB hardware and the ease of development for our testing framework in the GNU/Linux environment; however, with techniques described in previous chapters, one could apply the approaches presented in this chapter to closed source drivers from other operating systems, simply by adding *recovered symbol information*.

## 6.1 Introduction

Device drivers are a critical part of any modern operating system (OS) due to their privileged access to hardware and the OS kernel. At the same time, drivers are challenging to maintain and keep bug free due to the number of devices requiring support. As a result, device drivers commonly con-

tain many more bugs than other parts of the kernel: in a classic study, 70% of Linux device drivers were reported to have an error rate three to seven times higher than that of the core kernel [36].

Several mechanisms have been introduced to protect against kernel-level exploits, but as long as bugs persist in drivers, they pose a Denial of Service (DoS) vulnerability at best and a stepping stone for a multi-step exploit at worst. A wide range of defence techniques are available today, including Kernel Address Space Layout Randomization (KASLR), Data Execution Prevention (DEP), Return Address Protection (RAP) [121], System Mode Execution Prevention (SMEP), System Mode Access Prevention (SMAP), and Control Flow Integrity [41]. All of these helps raise the bar for developing a working exploit but do not prevent attacks entirely.

Drivers for devices on the Universal Serial Bus (USB) have recently received particular attention in the vulnerability research community [148, 64, 132, 99, 46]. The plethora of USB devices and the widespread adoption of the USB standard makes them a high value target; in particular, a working exploit against a USB device driver can permit malware to spread to air-gapped devices.

The main avenues for eliminating bugs from device drivers are device testing and static analysis. This is due in part to the difficulty in modelling the operating system and the hardware it interacts with. Dynamic approaches for testing kernel drivers mainly focus on injecting errors via bit-flips in the communications channel with the hardware. In the case of Linux's USB stack this typical results in code coverage of the core USB module (*usbcore*) instead of exercising paths corresponding to high-level logic of individual drivers. Many testing approaches for the Linux USB stack exist including the USB test suite of the Linux Test Project [97] and hardware-based approaches such as FaceDancer [64], Teensy [28], and FrisbeeLite [12]. However, all these approaches struggle to model the large number of possible interactions between device and driver. Static

approaches such as Coverity [20] regularly scan the Linux kernel for software faults. However, like other generic static analysis tools, it will miss bugs that require knowledge about the runtime environment and hardware, or that are based on complex and concurrent interactions between different components.

We describe the concepts behind our tool *POTUS* that focuses on finding deep bugs requiring complex interactions with a matching hardware device. We base the design of our tool on an attacker model of a user with physical access to a USB port and guest user privileges; a typical real-world scenario would be an attacker plugging in a custom USB device while a user is logged in (necessary commands can be typed by sending keystrokes from the device masquerading as a keyboard). For the purposes of our analysis we assume that *symbol information* is present in the target client drivers. When analysing proprietary drivers that are *stripped*, this information could be recovered using work presented in previous chapters.

Given knowledge of the Linux USB stack (Section 2.6.1) and Selective Symbolic Execution (Section 2.1), we focus our analysis to the myriad of USB client device drivers by creating a tool capable of generalising USB interactions. Many of the approaches before us focus on mutation-based fuzzing at this stage; something which is common to all USB device drivers and is well tested, and cannot expose bugs which require contextual interactions between userspace and hardware. In the remainder of the chapter, we introduce the design of *POTUS* (Section 6.2), its implementation (Section 6.3), and present our findings about *zero-day* vulnerabilities we discovered in Linux USB drivers (Section 6.4). Finally, we discuss limitations (Section 6.5), and contrast with related work (Section 6.6).

## 6.2 Design

We now introduce the design of *POTUS*. We start by explaining the underlying attacker model (Section 6.2.1) and then give a high-level overview of its components (Section 6.2.2).

### 6.2.1 Attacker Model

Our approach implements an attacker model where the adversary has a user account and physical access to USB ports. A typical scenario would be a workplace environment in which an employee has access to a machine and wants to achieve privilege escalation; or a visitor using a brief moment to plug in a USB device that implements both a human interface device to send keystrokes and a device targeting an exploitable driver.

Following this intuition, our attacker model has several degrees of freedom: first, it incorporates any possible interaction between devices and the operating system, which allows to trigger the driver loading, device enumeration, and driver-device setup stages. This dimension has been the focus of related work in the area [87, 35, 132]. Second, our attacker model includes system calls from user mode, i.e., the capability to interact with files created by device drivers. In particular, our attacker, as implemented in the driver exerciser, has the capability of (concurrently) performing system calls on file descriptors that trigger execution paths within the device driver. Third, our attacker model includes the ability to influence scheduling and cause memory allocation failures, which an attacker can achieve in practice by placing the CPU under load or exhausting memory.

### 6.2.2 System Overview

An overview of *POTUS*'s architecture is shown in Figure 6.1. *POTUS* builds on $S^2E$ (which in turn builds on QEMU) to run a full guest OS in

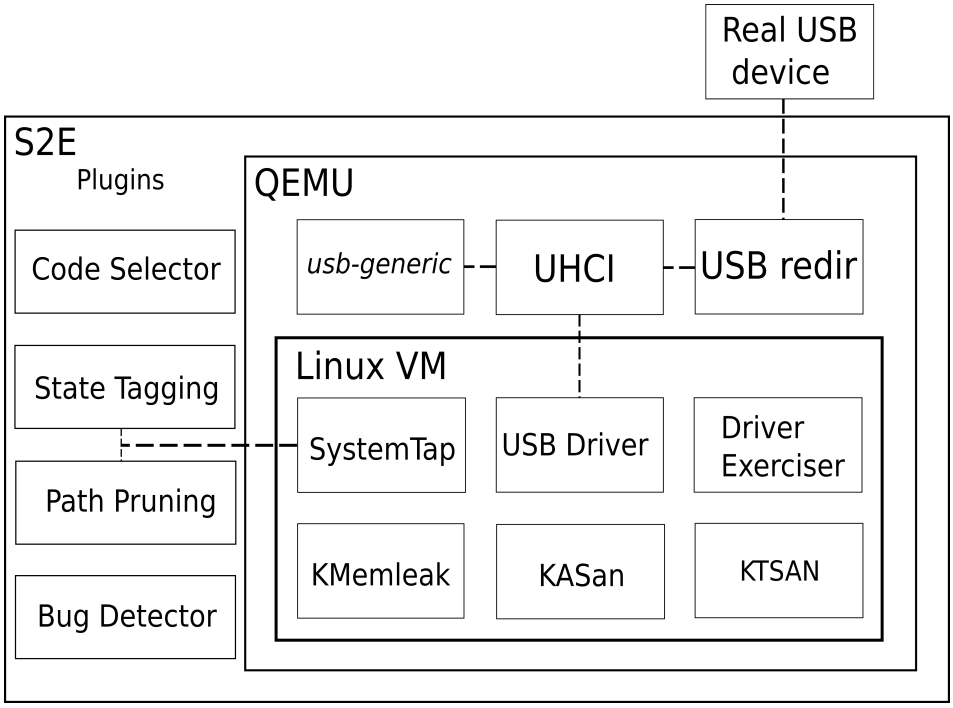



Figure 6.1: System overview of *POTUS*.

a virtual machine. It allows us to symbolically execute the specific client USB device driver module under analysis. This Linux USB software stack is shown in Figure 2.8, which explains the boundaries of interactions between hardware, kernel space and user space for the USB stack under Linux. As we can analyse Linux client USB device drivers dynamically and inside the environment of execution, it allows for the detection of software bugs arising from the complexities of interactions embodied in the entire kernel and hardware configurations.

The *USB driver* under test resides in the VM and receives inputs from the *driver exerciser* and one or several USB devices. The USB devices connect through QEMU's Universal Host Controller Interface (UHCI) and may be

either virtual usb-generic devices or real devices passed through QEMU's USB redirector (USB redir). For our case studies we relied on virtual devices only, but real devices may help to explore deep paths in involved protocols without modelling overhead. The driver exerciser runs inside the guest and enumerates—in a form of fuzz testing—possible interactions of a user process with the USB device.

We mark data sent from the device as symbolic to allow exploring all possible responses in $S^2E$, and we inject faults in the form of symbolic error codes. We use SystemTap scripts to interface between guest code and $S^2E$ and inject data at function call or return sites. We implemented $S^2E$ plugins for state tagging and path pruning to limit path explosion and prioritise states exploring different types of and locations of faults.

We rely on existing instrumentation tools to detect kernel-level bugs, namely Kernel Address Sanitizer (KASAN) for memory errors, Kmemleak for memory leaks, and Kernel Thread Sanitizer for data races. While the combination of these tools adds significant performance overhead, it allows us to have high confidence that we can detect any security-relevant errors as they occur. We also built a bug detector plugin for $S^2E$ that intercepts kernel bug functions such as `BUG_ON()` and `panic()` calls and retrieves the kernel log file, the driver exerciser log, and various other system log files to help debug and interpret the results.

## 6.3 Implementation

We now describe the implementation of *POTUS*, the generic virtual USB device (Section 6.3.1), the mechanism for fault injection (Section 6.3.2), the driver exerciser (Section 6.3.3), and its search strategy (Section 6.3.4).

### 6.3.1 The usb-generic Device

To exercise a USB device driver under all possible configurations, we created the virtual usb-generic device. It implements a generic USB device with configurable device descriptors and a symbolic model for data transfer.

The USB specification contains five descriptors: string descriptors, device descriptors, interface descriptors, configuration descriptors, and endpoint descriptors. usb-generic can be configured to impersonate any possible USB device by configuring its descriptors in a set of configuration files. In principle, this data could also be extracted from the target driver automatically. Once the descriptors are configured, the usb-generic device is ready to be used for testing any driver; further customisation is not required, but possible.

There are four URB types used for communication between a USB gadget and the host controller interface (see Section 2.6.1). By default, usb-generic ignores the content of OUT URBs (packets sent to the virtual device) and responds to IN URBs by writing the requested amount of data and marking it as symbolic for $S^2E$. We found that in practice the combination of symbolic data and a host-specified content length is sufficient to thoroughly exercise our target drivers, given enough time. However, usb-generic also provides the option for further (compile-time) customisation by installing driver-specific callbacks to respond differently to each URB type. We can further separate functionality by device, interface, class, and endpoint request codes within per Device ID and Vendor ID segments. Finally, the usb-generic device can also be instructed to respond negatively to individual URBs to influence packet scheduling, inject delays, or intentionally violate the USB specification.

Our usb-generic device is implemented as a QEMU hardware device that may be added to any virtual machine run by QEMU. Thus, it may be

```
{
        'idVendor' :            7504,
        'idProduct' :           24737,
        'bcdDevice' :           0,
        'iManufacturer':        1,
        'iProduct':             2,
        'iSerialNumber':        3
}
```

Listing 3: Content of `id.json` for an AirSpy USB device.

used across many different architectures and platforms for both the *host* and *guest* environments. During virtual device creation, four configuration files may be passed to specify descriptor values as JSON files. By default, `id.json`, `device.json`, `iface.json`, and `strings.json` are used to specify configuration optioned pertinent to ID descriptors, Device descriptors, Interface descriptors, and String descriptors. This runtime initialisation allows different USB configurations to be used without the recompilation of QEMU. An example of `id.json` for an AirSpy USB device can be seen in Listing 3.

Further control over the behaviour and data of virtual USB devices may be achieved by modifying the virtual device's C source code. Outside of vulnerability discovery, we believe that usb-generic can also be used by developers to aid writing device drivers in absence of a physical device, using fine-granularity callbacks.

### 6.3.2 Fault Injection

*POTUS* injects faults into the running kernel using SystemTap [128]. SystemTap allows to place *probes* at *known* or *recovered kernel symbol locations* to gather information or inject compiled C code to modify kernel data structures. *POTUS* contains SystemTap modules for automatically injecting

faults into core kernel submodules used by USB client device drivers such as *usbcore* and memory allocation mechanisms, an example of which can be seen in Listing 4. In particular, this example shows how fault injection interacts with symbolic execution: the function will fork the current state (as long as the per-path fault limit is not exceeded (see Section 6.3.4) and return success in one state and a symbolic fault code in the other. The symbolic fault code allows to explore all possible error codes at once, which would be impossible with purely concrete fault injection. The Linux kernel uses negative `errno` return codes to indicate errors; we therefore constrain the symbolic expressions for the return value on error paths to be negative. In addition to returning symbolic error codes, our fault model also injects the maximum admissible *delay*. While this is by no means a complete approach to verifying concurrent code, it is often just enough "fuzz" to expose concurrency bugs.

Some device drivers register asynchronous callbacks from the resulting URB inline; in that case we must manually write a short SystemTap probe for each. Our library of fault injectors works on a per module basis, i.e., it only injects faults into kernel threads that have a call stack associated with the target module under test. New SystemTap files can easily be created for testing new client drivers by including our library files and setting configuration values for fine-tuning fault injection, symbolic data injection and path pruning. Our full library for injecting faults can be found on Github[1].

### 6.3.3 Driver Exerciser

The driver exerciser in *POTUS* initiates operations on the device from user space. It randomly invokes file operations on the target device driver's file descriptors; to expose concurrency bugs, it can initiate multiple concurrent

[1] `https://github.com/p0tus/s2e`

Table 6.1: File descriptor operations implemented in the driver exerciser.

| Function | Description |
|---|---|
| `open()` | Open a new file descriptor corresponding to one of the exposed device files. Upon being called, the guest driver exerciser forks into two processes, with both accessing the device concurrently to try and trigger concurrency bugs. |
| `close()` | Close the active file descriptor. |
| `connect()` | Simulate a physical hardware connection. |
| `disconnect()` | Simulate a physical hardware disconnection. |
| `read()` | Perform a `sys_read` on the currently active file descriptor of random length. Discard the data read. |
| `write()` | Perform `sys_write` on the currently active file descriptor of random length. Data written is made symbolic. |
| `poll()` | Perform `sys_poll` on the currently active file descriptor. Request all events for a randomised timeout. |
| `lseek()` | Perform `sys_lseek` on the currently active file descriptor. Seek to a random offset for the current active file descriptor. |
| `ioctl()` | Perform `sys_ioctl` on the currently active file descriptor. Currently implemented for driver specific ioctl calls and arguments however may be made for generic by providing a symbolic call code and argument. |
| `send()` | Perform `sys_send` on the currently active socket. Send a buffer of symbolic data with a random length and symbolic or concrete flags. Allows for sending data on sockets of domain `AF_INET` and type `SOCK_STREAM` or `SOCK_DGRAM`. |
| `recv()` | Perform `sys_recv` on the currently active socket. Receive a random sized buffer with symbolic or concrete flags and discard any data read. Allows for receiving data on sockets of domain `AF_INET` and type `SOCK_STREAM` or `SOCK_DGRAM`. |

```
probe module("usbcore")
  .function("usb_bulk_msg").return {
  nfaults = potus_get_annotation(@FAULT_KEY)
  if (nfaults < @FAULT_LIMIT) {
    child = s2e_fork_state(__FUNC_NAME__ . " fork")
    if(child) {
      potus_annotate(@FAULT_KEY, nfaults+1)
      if (@SYMBOLIC_FAULTS)
        $return = potus_get_symb_fault(32)
      else
        $return = -1
      next
    }
  }
  ...
}
```

Listing 4: SystemTap probe for injecting faults into all `usb_bulk_msg` functions.

operations on the same or different file descriptors. We implemented this by using symbolic execution to search depth first through a weighted tree of operations (see Table 6.1), initially instantiating an active file descriptor with `open()`. Currently we support device drivers exposing a socket, character device or block device file, which covers the vast majority of USB device drivers.

Figure 6.2 shows an example execution path for a guest user opening a character device file owned by the *legousbtower* driver. The user initiates the interaction by calling `sys_open` on a file whose file operations for `open()` map to a callback within the *legousbtower* driver. Should the driver attempt to allocate memory or call other kernel subsystems for which fault injection is currently active, *POTUS* forks the system state and returns a symbolic fault in one state, indicated by the red path. In the fault-free path, the driver callback calls `usb_submit_urb` with a device request of

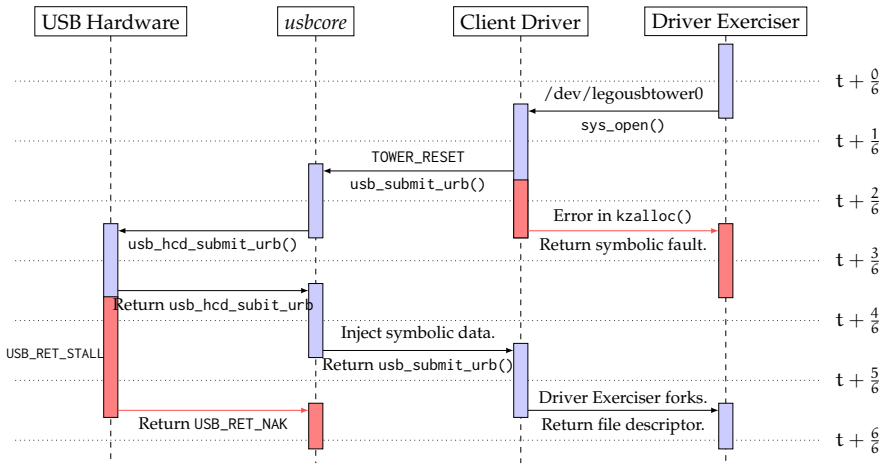



Figure 6.2: Example of flows for viable code paths from the driver exerciser through the Linux USB stack. The diagram shows fault injection and fork points for the *legousbtower* client driver.

a custom `TOWER_RESET` device reset code, which is passed to the HCI via `usb_hcd_submit_urb`. At this point, the system state is forked again, with the fault-free path returning successfully, and the other path failing the URB transfer after the maximum delay.

### 6.3.4 Path Prioritisation

Since we see *POTUS* as complementary to existing testing and bug finding approaches, we particularly focus our vulnerability search on deep bugs arising from concurrency errors, faults, and their interplay. Our intuition is that those bugs that are detectable by light-weight static analysis will have already been found in the Linux kernel. Still, the multitude of low-level concurrency primitives in a Linux kernel running on modern hardware harbour great potential for lingering concurrency bugs.

The combination of symbolic faults, concurrency fuzzing with delays,

Table 6.2: List of device drivers tested.

| Driver Tested | LOC | Driver-specific Probes | Functions in Driver | Exposed Interface |
|---|---|---|---|---|
| AirSpy | 1,108 | 1 | 32 | *v4l2* device |
| Apple USB Display | 369 | 1 | 8 | Backlight device |
| Chaos Key | 579 | 0 | 11 | Character file |
| Cytherm | 407 | 0 | 13 | *Sysfs* files |
| IO Warrior | 919 | 2 | 15 | Character file |
| Lego USB Tower | 982 | 2 | 15 | Character file |

and symbolic data aggravates the state explosion problem in symbolic execution. Since we are interested in exploring deep paths, we prioritise a primary path without faults or delays, and use it to spawn new states at potential fault and delay injection points. *POTUS* attaches a map of annotations to each state, which is cloned upon forking; we use this map to track the number of faults already injected into the state (see Listing 4, line 3) and the number of children created. We use this to limit the number of fork points by preventing further fault injection when the limit of faults is reached. This balances general code coverage with the exploration of fault routines. The intuition behind this optimisation is that paths with a high number of faults will likely lead to exploring the same code numerous times without exploring new error handlers. We thus prioritise complex driver-device exchanges over exploring the entire, potentially infinite, set of possible interactions.

## 6.4 Evaluation

We now present a preliminary evaluation of our approach; we were interested in evaluating *POTUS*'s ability to effectively find bugs and its potential for generalising to different types of USB drivers. We first discuss our setup and methodology (Section 6.4.1), explain how *POTUS* applies to six target USB device drivers (Section 6.4.2), and then provide details and an assessment of exploitability for the two previously unknown vulnerabilities we discovered: CVE-2016-5400[2] resulted in a denial-of-service attack on Linux kernels 3.17–4.6 (Section 6.4.3); CVE-2017-15102 allows an arbitrary write/read primitive affecting Linux kernels pre-2.6–4.6 since 2003 (Section 6.4.4).

### 6.4.1 Experimental Setup

All of our experiments were run on a Ubuntu 12.04 LTS with dual Intel(R) Xeon(R) CPU E5-2640. We used our own fork of S$^2$E based on the latest version as of 2016-11-01 and Debian Sid as the guest OS running a custom vanilla Linux 4.6 kernel, which only enables the required modules and keeps a minimal guest OS. We dynamically loaded `usb-generic` devices through QEMU's monitor interface and executed S$^2$E on all the available CPU threads of the host platform.

We developed an automation framework to control experiments, including booting the OS, inserting SystemTap probes, and loading client device drivers. The guest OS was executed with a QEMU emulated Core 2 Duo CPU and 1 GB of RAM. Overall, we ran each driver for up to one hour, exploring in the order of hundreds of states per experiment. We used S$^2$E in concolic mode, exploring a state until termination before switching to a new state, to explore deeper code paths.

[2] `https://nvd.nist.gov/vuln/detail/CVE-2016-5400`

*POTUS*'s memory requirements are kept manageable (considering it is based on full-VM symbolic execution) by our path pruning strategy. For instance, in testing the Lego USB Tower driver for one hour with 32 $S^2E$ processes, *POTUS* forked 488 different VM states and used 6.6 GB of RAM. As *POTUS* runs $S^2E$ in concolic mode, it executes the driver exerciser to a fixed number of operations before terminating the state, which improves *POTUS* overall memory footprint as all the states resources do not have to be saved simultaneously.

### 6.4.2 Adapting to Target Device Drivers

To evaluate our claim of a generic testing tool suite for USB device drivers, we consider the effort it takes to test new drivers. We selected six open-source USB device drivers that are included in the mainline Linux kernel, touch on several of its different subsystems, and have a significant number of lines of code. Table 6.2 lists the device drivers that we tested along with their lines of code, number of functions, the exposed interface, and the number of driver-specific probes we had to write.

We could test Chaos key and Cytherm entirely with *POTUS*'s default libraries for injecting faults, because these drivers rely only on synchronous `usbcore` library functions such as `usb_bulk_msg` to transfer data. Although the driver exerciser must be instructed to point to the corresponding device files, there was no need for additional SystemTap probes. Conversely, drivers that also rely on asynchronous `usbcore` library functions expose specific callbacks that have to be instrumented and required one to two driver-specific probes (see Table 6.2). It is feasible to automatically address such contexts (e.g., we can modify our SystemTap libraries with inline C code to dereference forward referencing functions), but the currently necessary manual effort is minimal.

### 6.4.3 Airspy (CVE-2016-5400)

CVE-2016-5400 represents a memory leak vulnerability in a USB device driver for communicating with an Airspy Software Defined Radio (SDR), located under `drivers/media/usb/airspy/airspy.c` in the Linux kernel source tree. The memory leak can be triggered purely from hardware to perform a Denial of Service (DoS) attack, crashing the host by plugging in a specially crafted USB device. The USB device driver interacts with the Video For Linux 2 (V4L2) subsystem and, as a result, requires the allocation of `v4l2` device structures and registration with the subsystem. The programming error that led to the memory leak was situated in the drivers `probe` function; a function that is called when a new device associated with the driver is plugged into the host. The relevant code snippet can be seen in Listing 5 and shows that if the `video_register_device` function fails, the driver fails to free any of the control variables registered with the `v4l2` subsystem. *POTUS*'s automatic fault injection identified this memory leak.

*Exploitability*.

The Airspy kernel module was installed by default in most Linux distributions, including, but not limited to Ubuntu, Debian, Arch Linux, and Trisquel; it loads whenever a USB device with the Airspy device descriptor is plugged in.

An attacker can make `video_register_device` fail with a specially crafted hardware as the Linux kernel only supports a maximum of 64 minor numbers for `VFL_TYPE_SDR` type devices attached to a host at any given time. By creating a USB device that acts as a hub and attaches 65 of the same devices, we can trigger the memory leak vulnerability. The sequence of connection and disconnection operations on the 65th device consumes all the available RAM and effectively triggers a DoS attack. We successfully verified the feasibility of this attack under the *POTUS* testing

```
1  static int airspy_probe(struct usb_interface *intf,
2          const struct usb_device_id *id)
3  {
4      ...
5      v4l2_ctrl_handler_init(&s->hdl, 5);
6      ...
7      ret = video_register_device(&s->vdev,
8              VFL_TYPE_SDR, -1);
9      if (ret) {
10         dev_err(s->dev, "Failed to...");
11         goto err_unregister_v4l2_dev;
12     }
13     dev_info(s->dev, "Registered as ...");
14     return 0;
15 err_free_controls:
16     v4l2_ctrl_handler_free(&s->hdl);
17 err_unregister_v4l2_dev:
18     v4l2_device_unregister(&s->v4l2_dev);
19 err_free_mem:
20     kfree(s); return ret;
21 }
```

Listing 5: Airspy probe function.

framework.

### 6.4.4 Lego USB Tower (CVE-2017-15102)

CVE-2017-15102 is a Use-After-Free vulnerability that has existed in the Linux kernel's Lego USB Tower driver since 2003 (`drivers/usb/misc/legousbtower.c`). The driver is quite pervasive: it is compiled and available with the majority of Linux distributions, including the latest server editions of Ubuntu 16.04 LTS and Fedora 25. The vulnerability is a race condition that leads to a NULL pointer dereference; if remapped to a user-controlled memory location, it can be abused to escalate privileges or execute arbitrary code.

```
1  static int
2  tower_probe(struct usb_interface ...)
3  {
4  ...
5    /* register the device now, as it is ready */
6    usb_set_intfdata (interface, dev);
7    retval = usb_register_dev (interface, ...);
8  ...
9    /* get the firmware version and log it */
10   result = usb_control_msg (udev,
11       usb_rcvctrlpipe(udev, 0),
12       LEGO_USB_TOWER_REQUEST_GET_VERSION,
13       USB_TYPE_VENDOR | USB_DIR_IN | USB_RECIP_DEVICE,
14       0, 0, &get_version_reply,
15       sizeof(get_version_reply), 1000);
16   if (result < 0) {
17     dev_err(idev, "LEGO USB Tower get\
18             version control\
19             request failed\n");
20     retval = result;
21     goto error;
22   }
23 ...
24 error:
25   tower_delete(dev); return retval;
26 }
```

Listing 6: Lego USB Tower probe function.

Listing 6 shows the driver probe function and entry point into the program. The function registers a character device file `/dev/usb/legousbtower[0-9]+` and proceeds to submit a request for the device firmware version. If this URB request fails, the driver then calls `tower_delete`, which deletes the device structures associated with the driver without checking for any active connection. Registering the device file grants file operations from user space, an action which could happen before the probe function terminates. Listing 7 details the `tower_write`

```
1  static int write_buffer_size = 480;
2  ...
3  static ssize_t
4  tower_write (struct file *file,
5      const char __user *buffer, size_t count, ...)
6  {
7      struct lego_usb_tower *dev;
8      size_t bytes_to_write;
9      ...
10     /* verify that the device wasn't unplugged */
11     if (dev->udev == NULL) {
12         retval = -ENODEV;
13         pr_err("No device or device\
14                 unplugged %d\n", retval);
15         goto unlock_exit;
16     }
17     /* wait until previous transfer is finished */
18     while (dev->interrupt_out_busy) {
19         if (file->f_flags & O_NONBLOCK) {
20             retval = -EAGAIN;
21             goto unlock_exit;
22         }
23     }
24     /* write the data into interrupt_out_buffer
25        from userspace */
26     bytes_to_write = min_t(int, count,
27             write_buffer_size);
28     if (copy_from_user (dev->interrupt_out_buffer,
29                 buffer, bytes_to_write))
30     ...
31 }
```

Listing 7: Lego USB Tower `tower_write` function.

function, which maps to the `sys_write` system call and checks that the device is still connected before copying data from user space into a local kernel buffer pointed to by `dev->interrupt_out_buffer`. If `tower_delete` is called after the write function checks that the device is connected, it will delete the `dev` structure, setting its value to NULL and causing a NULL pointer dereference in `tower_write`.

*Exploitability*.

An attacker can create a USB device that will hold open or drop the control message for the board's firmware version, providing with the time necessary to exploit the race condition. As the kernel executes in the same address space as user space, an unprivileged user may map the NULL page (or use alternative techniques to work around limitations) to control the location of the data being written to. As an attacker also controls the data, it is possible to write an arbitrary payload to arbitrary memory locations, thus overwriting the local user id for the process to gain root privileges.

If the NULL page is mappable through the `sysctl` setting `mmap_min_addr` or by using a user account with the Linux personality of `MAP_PAGE_ZERO`, an adversary can easily force the location and data on a buffer written inside the kernel. Other methods, such as those that execute a *setuid* binary to remap existing memory have previously been shown to circumvent this protection[3]. Linux kernels before 2009 have no protection against mapping the NULL page and are thus easily exploitable using a specially crafted USB device and a low-privileged guest user account to trigger the race condition. Upon further inspection, the device file exposed by this driver is made globally readable and writable on most systems by *udev*; something which happens after the probe function finishes and closes the race condition. This significantly

[3] http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html

lowers the impact of the vulnerability, but it may be used in a multi-stage exploit or to escape containers where the user already has an *fsuid* of 0.

*Bypassing SMEP, SMAP and RAP*.

To assess the exploitability of CVE-2017-15102 on a modern, security-hardened kernel, we decided to build a proof of concept of a local privilege escalation exploit that would work on the latest kernel at the time of development: Linux 4.6 with PaX's GrSecurity patches applied. The underlying idea is to reallocate the same memory used by the struct deletion described above to control the location of the output buffer.

To be able to remap the same memory location as the kernel space `dev struct`, we abused the Linux kernel's SLUB memory allocator that reprovisions previous allocations of the same size. Invoking `sys_sendmsg` allowed us to force the kernel to allocate arbitrarily sized memory in the general kernel cache. Once we identified the size of the message to send, we created a USB device to insert a one second delay for the `tower_probe`'s device firmware URB request, which enabled us to consistently remap the same memory freed from `tower_delete`. Unfortunately, `sys_sendmsg` does not allow us to control the first 40 bytes of memory allocated, due to it being reserved for the messages metadata. In our case, while the main data structures were outside this region, it overlapped with a device mutex, which would block execution of the exploit indefinitely.

We relaxed the condition for exploitation and created a kernel module that would first leak memory addresses to bypass KASLR and discover the running process' `task struct` to overwrite its credentials and increase privileges and, secondly, allow us to allocate memory of arbitrary size in the kernel. Under such assumptions, we were able to dereference a pointer to the current processes `credentials struct` and overwrite `{u,g}id`, `e{u,g}id`, `s{u,g}id`, `fs{u,g}id` and `capabilities`. Our final exploit bypasses SMAP through the kernels own use of `copy_from_user`, temporarily disabling it without performing any buffer overflow or con-

trol flow hijacking, thus remaining unaffected by both SMEP and RAP. We believe that our relaxed exploitation conditions are realistic [79] and do not affect the feasibility of a successful attack. For instance, the second condition can be addressed by adjusting the `bMaxPacketSize` of the device descriptor to load data into that memory location to read and write data to the device.

## 6.5 Limitations

QEMU's current UHCI implementation supports USB devices up to USB 1.1. Although many devices are backwards compatible and simply transfer data at lower speeds, they may use some features of newer USB specifications that we therefore cannot test. For instance, usb-generic currently does not support a multi-master device setup and hence does not support the USB On-The-Go extension.

The implementation of `usb_submit_urb` in *usbcore* contains an `interval` parameter which specifies a time period for periodically polling the device, indefinitely. If we are masking data input from the device as completely symbolic data, and if each URB will result in at least one state being created from injected faults, then this presents an infinite set of possible paths to explore and further increases path explosion.

Furthermore, the high runtime overhead of symbolic execution in $S^2E$ increases the frequency of timer interrupts relative to the execution of other instructions. As a result, we had to slow down QEMU's internal clock by a factor of five to avoid exploring only device polling in drivers using short intervals.

## 6.6 Related Work

Our approach draws on a range of previous work, which we compare and contrast to in this section.

Symbolic execution has been widely used for vulnerability discovery (e.g., [32, 61, 33]). Most closely related are $S^2E$ itself [35] and in particular its predecessor project DDT [87]. Both have been used to find bugs in user space applications and device drivers. *POTUS* builds on $S^2E$ and expands it with features specific to the problem domain of Linux USB drivers. In a way, *POTUS* is a sister project to DDT in that it allows testing USB drivers on the latest Linux versions similarly to how DDT tested PCI drivers for Windows XP. However, DDT used fully symbolic PCI devices that would be too generic to allow meaningful exploration of devices communicating via a USB host controller.

Tonder and Engelbrecht [148] describe a hardware-based mutation fuzzing scheme for USB. Their approach builds on the Facedancer project [64] to mutate the interactions of existing USB devices with the host. A pure software approach is inherently easier to deploy (e.g., where no related device is available) and more flexible, e.g., the reported 300ms delay in control transfers would make it difficult to discover timing-sensitive race conditions such as CVE-2017-15102. Furthermore, a hardware-only approach that uses random bit flips ignores driver logic and is likely to only exercise a very limited portion of the USB subsystem, in particular the USB device enumeration in *usbcore*.

Jodeit et al.'s [81] combined a physical USB device with a mutational based fuzzer to test an Apple iPod on Windows and found multiple software bugs in Windows XP drivers. Schumilo et al. [132] presented the software USB fuzzer vUSBf, which relies on QEMU's usbredir server to redirect URB packets from host emulated devices into the guest operating system. vUSBf mainly focuses on fuzzing values in USB descriptors

and USB HID drivers and provides no systematic way of exercising device drivers.

NCC's *umap2*[4] allows to fuzz USB device drivers by recording traces from emulated devices and then fuzzing replays of the traces. The project relies on gadgetfs or Facedancer and a Python program to describe each device. The project is currently able to emulate 13 device classes, each specified in hundreds of lines of Python code. In contrast, *POTUS* provides significantly more automation, requiring typically to only adapt a few SystemTap scripts and configure the virtual device.

Software-implemented fault injection (SWIFI) is a widely used technique for testing the robustness of software. Natella et al. [113] provide a recent survey of the area. A flexible framework for fault injection at the level of libraries was presented in LFI [98]. LFI automatically generates error models for libraries, which we aim to also achieve at kernel level for *POTUS* in future work. A comparative study of fault injection techniques by Jarboui et al. [80] showed that internal software faults or faults caused by device drivers could not be easily emulated by injecting faults at the system call level only. This validates our design choice in *POTUS* to allow fault injection at any point in the kernel, and most importantly at the level of the kernel subsystem APIs. The impact of device drivers on the Linux kernel is a known cause for concern. For example, Albinet et al. [4] characterised the kernel's robustness based on the impact of faulty device drivers.

[4] `https://github.com/nccgroup/umap2`

# 7 Conclusions

Our goal in this thesis has been to explore information recovery in compiled computer programs and use it to aid software analysis. Throughout our work, we built on top of previous state-of-the-art techniques and tools in program analysis and machine learning. Where possible we have contributed back to open-source projects and to the research community. For the most part, software created to form this research was added to the binary analysis toolkit, *desyl*, which provides the implementation of techniques covered in this thesis and is released open-source. A divergent software project forked from $S^2E$ and QEMU is also released open-source under *usbdt*.

We started in Chapter 3 which presented *Punstrip*, a novel approach for naming functions in stripped executables that combines program analysis and machine learning to infer symbol information. We demonstrated that *Punstrip* is a viable approach to learn a function fingerprint that is capable of inferring symbols between multiple compilers and optimisation levels. Secondly, we combined our fingerprint with structured prediction using $3^{rd}$ order general graph-based conditional random fields to predict symbol information in binaries using all known relationships simultaneously rather than considering each function in isolation. We carried out an extensive 10–fold cross-validated evaluation against C ELF binaries built from different environments and compilers in the Debian Sid repositories

and released the dataset to the research community. We have shown that it is possible to learn intrinsic relationships between functions and transpose that information to other *previously unseen* stripped binaries. Where the original function name has never been seen before, we suggest meaningful names in order to aid the reverse engineering process.

Our work on *Punstrip* highlighted fundamental problems with building machine learning models to name components of binary code. In light of this, we built deep learning models that captured the semantics of binary functions and their names. Our label space function name model, *Symbol2Vec*, mitigates the subjective nature of naming functional components by human beings and expresses abstract relationships through its vector space. We were able to prove that our *Symbol2Vec* vector space is meaningful by computing the nearest neighbours in this space. This enabled us to have an absolute numerical metric to compare the similarity of function names by computing the distance between vectors.

The development of our DEXTER embeddings proves as a useful vector representation of binary functions that can be used in machine learning models or binary code search engines. Our technique creates a distributed representation of binary code that concisely captures the semantics of functions. In doing so, DEXTER condenses 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 many features from static analysis for learning embeddings improves performance over relying on more syntactic features, as done in previous approaches.

Chapter 5 presented *XFL* to solve the function naming problem in a way that addresses limitations inherent in previous methods. *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. Our model can predict infrequently occurring "tail labels" and we showed that our approach outperforms existing approaches to function name prediction by evaluating our embeddings using information-theoretic metrics from the XML field and traditional metrics for multi-label classification. As *XFL* predicts labels instead of whole function names, it is able to predict names for functions even when no function of that name is contained in the training set; an intrinsic weakness of all other approaches from their lack of generalisation.

Finally, we investigated how information recovery could be useful when analysing kernel drivers. Chapter 6 presented *POTUS*, a new approach for testing USB device drivers capable of finding long-existing bugs that previous state-of-the-art tools failed to find. Our approach is built on top of free and libre open-source software, is easily extendable, and can work together with existing open-source projects to provide further functionality. As a result, we found two critical vulnerabilities in the latest version of the Linux kernel at the time of completion and built proof of concept exploits to explore their severity.

We release all our tools to the community to improve the area of software analysis, reverse engineering, bug finding, and USB driver development.

*Future Work*. Techniques developed in this thesis aid the reverse engineering process and develop new tools to analyse kernel USB drivers. The drivers targeted in Chapter 6 were open-source drivers from the Linux kernel which enabled us to focus on our approach to finding deep bugs in the logic of each driver. Naturally, one would hope to extend this work by recovering symbol information in proprietary Linux, Windows or Mac OS kernel drivers, before analysing them using *POTUS*. This would enable a new insight into targeted software analysis of closed source software that has not been seen before. Another extension to the work presented in this thesis could lie in the use of recent natural language text embeddings (e.g,

GPT-3). One could try to use semantic knowledge inherent in large pre-trained language models to generate higher quality function names from a finite set of string tokens predicted by *XFL*.

# Bibliography


[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In V. Atluri, C. A. Meadows, and A. Juels, editors, Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS, pages 340–353. ACM, 2005.

[2] B. Adhikari, Y. Zhang, N. Ramakrishnan, and B. A. Prakash. Sub2vec: Feature learning for subgraphs. In Advances in Knowledge Discovery and Data Mining, volume 10938 of Lecture Notes in Computer Science, pages 170–182. Springer, 2018.

[3] R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma. Multi-label learning with millions of labels: recommending advertiser bid phrases for web pages. In 22nd International World Wide Web Conference, WWW '13, pages 13–24. WWW Steering Committee / ACM, 2013.

[4] A. Albinet, J. Arlat, and J. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), pages 867–876, 2004.

[5] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In International Conference on Learning Representations, 2018.

[6] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In International conference on machine learning (ICML), pages 2091–2100. PMLR, 2016.

[7] U. Alon, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In International Conference on Learning Representations (ICLR), 2019.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. A general path-based representation for predicting program properties. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, page 404–419. Association for Computing Machinery, 2018.

[9] S. Alrabaee, L. Wang, and M. Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). Digital Investigation, 18:S11–S22, 2016.

[10] J. Alves-Foss and J. Song. Function boundary detection in stripped binaries. In D. Balenson, editor, Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, pages 84–96. ACM, 2019.

[11] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In 2017 IEEE European Symposium on Security and Privacy, EuroS&P, pages 177–189, 2017.

[12] N. Andy Davis. Fuzzing USB Devices using FrisbeeLite. Available at `https://www.nccgroup.com/globalassets/our-research/uk/whitepapers/fuzzing_usb_devices_using_frisbee_lite.pdf`, 2012.

[13] K. Balasubramanian and G. Lebanon. The landmark selection method for multiple output prediction. In Proc. 29th International

Conference on Machine Learning (ICML). icml.cc / Omnipress, 2012.

[14] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: learning to recognize functions in binary code. In Proceedings of the 23rd USENIX Security Symposium, pages 845–860, 2014.

[15] I. Batatia. A deep learning method with CRF for instance segmentation of metal-organic frameworks in scanning electron microscopy images. In 28th European Signal Processing Conference, EUSIPCO, pages 625–629. IEEE, 2020.

[16] F. Bellard. QEMU, a fast and portable dynamic translator. In Proc. USENIX Annual Technical Conference, FREENIX Track, pages 41–46, 2005.

[17] E. Bendersky. PyELFTools - A Python Library for parsing ELF files. `https://github.com/eliben/pyelftools`, 2018.

[18] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. Journal of Maching Learning Research, 3:1137–1155, 2003.

[19] B. N. Bershad and J. C. Mogul, editors. Operating Systems Design and Implementation (OSDI). USENIX Association, 2006.

[20] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM, 53(2):66–75, 2010.

[21] 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), pages 730–738, 2015.

[22] B. Bichsel, V. Raychev, P. Tsankov, and M. T. Vechev. Statistical deobfuscation of android applications. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS, pages 343–355, 2016.

[23] P. Bielik, V. Raychev, and M. Vechev. Phog: Probabilistic model for code. In Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML) - Volume 48, page 2933–2942. JMLR.org, 2016.

[24] M. Bourquin, A. King, and E. Robbins. BinSlayer: accurate comparison of binary executables. In Program Protection and Reverse Engineering Workshop, page 4. ACM, 2013.

[25] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov. Generative code modeling with graphs. In International Conference on Learning Representations, 2019.

[26] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In Proceedings of Computer Aided Verification - 23rd International Conference, CAV, pages 463–469, 2011.

[27] B. R. Buck and J. K. Hollingsworth. An API for runtime code patching. Int. J. High Perform. Comput. Appl., 14(4):317–329, 2000.

[28] E. Bursztein. Malicious USB, HID spoofing multi-OS payload for Teensy. Available at `https://github.com/ebursztein/malusb`, 2016.

[29] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. SIAM Journal of Scientific Computing, 16(5):1190–1208, 1995.

[30] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications.

In Proceedings of the Network and Distributed System Security Symposium, NDSS. The Internet Society, 2010.

[31] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the Operating System Design and Implementation (OSDI), pages 209–224, 2008.

[32] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In ACM Conf. Computer and Communications Security (CCS), pages 322–335. ACM, 2006.

[33] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In IEEE Symp. Security and Privacy (S&P), pages 380–394. IEEE Computer Society, 2012.

[34] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. BinGo: cross-architecture cross-OS binary search. In Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, pages 678–689, 2016.

[35] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In Proceedings of the International Conference of Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 265–278, 2011.

[36] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In Proceedings of ACM Symposium on Operating System Principles (SOSP), pages 73–88, 2001.

[37] M. Cissé, N. Usunier, T. Artières, and P. Gallinari. Robust bloom filters for large multilabel classification tasks. In C. J. C. Burges, L. Bottou, Z. Ghahramani, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 26, pages 1851–1859, 2013.

[38] U. S. B. Computer Security Lab and A. S. U. SEFCOM. Claripy: an abstraction layer for constraint solvers. Available at `https://github.com/angr/claripy`, 2020.

[39] J. Corbet, A. Rubini, and G. Kroah-Hartman. Linux Device Drivers. O'Reilly, 3rd edition, 2005. Available at `https://lwn.net/Kernel/LDD3/`.

[40] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In 22nd Annual Computer Security Applications Conference (ACSAC), pages 269–278, 2006.

[41] J. Criswell, N. Dautenhahn, and V. S. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In IEEE Symp. Security and Privacy (S&P), pages 292–307, 2014.

[42] 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.

[43] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In Proceedings of the 33nd International Conference on Machine Learning, ICML, pages 2702–2711, 2016.

[44] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity

protection. In K. Fu and J. Jung, editors, Proceedings of the 23rd USENIX Security Symposium, pages 401–416. USENIX Association, 2014.

[45] Y. David, U. Alon, and E. Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. Proceedings of the 2020 ACM Object-oriented Programming, Systems, Languages, and Applications (OOPLSA), 4:225:1–225:28, 2020.

[46] A. Davis. Lessons learned from 50 bugs: Common USB driver vulnerabilities. `https://www.nccgroup.trust/globalassets/our-research/uk/whitepapers/usb_driver_vulnerabilities_whitepaper_v2.pdf`, 2013.

[47] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-based function embedding and its application to error-handling specification mining. In Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 423–433, 2018.

[48] 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 2019 IEEE Symposium on Security and Privacy, SP, pages 472–489. IEEE, 2019.

[49] 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, pages 472–489. IEEE, 2019.

[50] Y. Duan, X. Li, J. Wang, and H. Yin. Deepbindiff: Learning program-

wide code representations for binary diffing. In Network and Distributed System Security Symposium, 2020.

[51] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis transformation. In Proceedings of CanSecWest, 2009.

[52] P. Efstathopoulos, M. N. Krohn, S. Vandebogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, M. F. Kaashoek, and R. T. Morris. Labels and event processes in the asbestos operating system. In A. Herbert and K. P. Birman, editors, Proceedings of the 20th ACM Symposium on Operating Systems Principles SOSP, pages 17–30. ACM, 2005.

[53] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In Proceedings of the 23rd USENIX Security Symposium, pages 303–317, 2014.

[54] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In B. N. Bershad and J. C. Mogul, editors, 7th Symposium on Operating Systems Design and Implementation (OSDI, pages 75–88. USENIX Association, 2006.

[55] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In 23rd Annual Network and Distributed System Security Symposium, NDSS, 2016.

[56] M. A. et al. TensorFlow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI, pages 265–283, 2016.

[57] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi. Binclone: Detecting code clones in malware. In Eighth International Conference on Software Security and Reliability, SERE, pages 78–87, 2014.

[58] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 480–491, 2016.

[59] P. Fernandes, M. Allamanis, and M. Brockschmidt. Structured neural summarization. In International Conference on Learning Representations (ICLR), 2019.

[60] H. Flake. Structural comparison of executable objects. In U. Flegel and M. Meier, editors, Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, volume P-46, pages 161–173, 2004.

[61] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In Proceedings of Network and Distributed System Security Symposium (NDSS). The Internet Society, 2008.

[62] T. Gonçalves and P. Quaresma. The impact of nlp techniques in the multilabel text classification problem. In Intelligent Information Processing and Web Mining, pages 424–428. Springer, 2004.

[63] T. Goodspeed. Facedancer: emulating USB devices with python. Aailable at `https://travisgoodspeed.blogspot.com/2012/07/emulating-usb-devices-with-python.html`, 2012.

[64] T. Goodspeed and S. Bratus. Emulating USB devices with python. `https://travisgoodspeed.blogspot.de/2012/07/emulating-usb-devices-with-python.html`, September 2012.

[65] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 855–864. ACM, 2016.

[66] GrSecurity and the PaX team. Academic research in software defenses. Available at `https://www.grsecurity.net/research`, 2021.

[67] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pages 277–289, 2007.

[68] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In Computer Society Conference on Computer Vision and Pattern Recognition CVPR 2006), pages 1735–1742. IEEE Computer Society, 2006.

[69] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. SIGARCH Comput. Archit. News, 33(5):63–68, 2005.

[70] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. T. Vechev. Debin: Predicting debug information in stripped binaries. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS, 2018.

[71] E. Hoffer and N. Ailon. Deep metric learning using triplet network. In Similarity-Based Pattern Recognition - Third International Workshop, SIMBAD, volume 9370 of Lecture Notes in Computer Science, pages 84–92. Springer, 2015.

[72] D. J. Hsu, S. M. Kakade, J. Langford, and T. Zhang. Multi-label prediction via compressed sensing. In Y. Bengio, D. Schuurmans,

J. D. Lafferty, C. K. I. Williams, and A. Culotta, editors, Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems, pages 772–780. Curran Associates, Inc., 2009.

[73] X. Hu, T. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS, pages 611–620. ACM, 2009.

[74] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME, pages 104–114, 2018.

[75] E. R. Jacobson, N. E. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools, PASTE'11, pages 1–8, 2011.

[76] H. Jain, V. Balasubramanian, B. Chunduri, and M. Varma. Slice: Scalable linear extreme classifiers trained on 100 million labels for related searches. In J. S. Culpepper, A. Moffat, P. N. Bennett, and K. Lerman, editors, Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining, WSDM, pages 528–536. ACM, 2019.

[77] 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, pages 935–944. ACM, 2016.

[78] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Feature Embeddings for Binary Symbols with Symbol2Vec. `https://www.github.com/punstrip/symbol2vec`, 2020.

[79] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with Intel TSX. In ACM Conf. Computer and Communications Security (CCS), pages 380–392. ACM, 2016.

[80] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental analysis of the errors induced into Linux by three fault injection techniques. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), pages 331–336, 2002.

[81] M. Jodeit and M. Johns. USB device drivers: A stepping stone into your kernel. In Proc. European Conf. Computer Network Defense (EC2ND), pages 46–52. IEEE, 2010.

[82] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina. Code authorship attribution: Methods and challenges. ACM Computing Surveys (CSUR), 52:1–36, 2019.

[83] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7):654–670, 2002.

[84] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE, pages 353–364, 2017.

[85] D. Koller and N. Friedman. Probabilistic Graphical Models: Principles and Techniques. MIT Press, 2010.

[86] S. Kumar and M. Hebert. Discriminative fields for modeling spatial dependencies in natural images. In S. Thrun, L. K. Saul, and B. Schölkopf, editors, Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS, pages 1531–1538. MIT Press, 2003.

[87] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In USENIX Annual Technical Conference. USENIX Association, 2010.

[88] 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, pages 628–639. IEEE, 2019.

[89] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Proceedings of the Eighteenth International Conference on Machine Learning (ICML, pages 282–289. Morgan Kaufmann, 2001.

[90] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO, pages 75–88, 2004.

[91] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. In Proceedings of the 31th International Conference on Machine Learning, ICML, volume 32 of JMLR Workshop and Conference Proceedings, pages 1188–1196, 2014.

[92] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of webassembly. In S. Capkun and F. Roesner, ed-

itors, 29th USENIX Security Symposium, USENIX Security 2020, pages 217–234. USENIX Association, 2020.

[93] Y. Li and B. Liu. A normalized levenshtein distance metric. IEEE Transactions on Pattern Analysis and Machine Intelligence, 29(6):1091–1095, 2007.

[94] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou. $\alpha$diff: Cross-version binary code similarity detection with dnn. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, page 667–678. Association for Computing Machinery, 2018.

[95] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 872–881, 2006.

[96] V. B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. In Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pages 75–86, 2009.

[97] LTP contributors. Linux Test Project. Available at `https://linux-test-project.github.io/`, 2017.

[98] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In Proceedings of the International Conference on Dependable Systems and Networks (DSN), pages 379–388, 2009.

[99] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham. Mouse trap: Exploiting firmware updates in USB peripherals. In USENIX Workshop on Offensive Technologies (WOOT), 2014.

[100] L. Massarelli, G. A. D. Luna, F. Petroni, R. Baldoni, and L. Querzoni. SAFE: self-attentive function embeddings for binary similarity. In R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, editors, Detection of Intrusions and Malware, and Vulnerability Assessment, volume 11543, pages 309–329. Springer, 2019.

[101] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V application binary interface - amd64 architecture processor supplement. `https://www.uclibc.org/docs/psABI-x86_64.pdf`, 2014.

[102] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In 1st International Conference on Learning Representations, ICLR 2013, Workshop Track Proceedings, 2013.

[103] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur. Recurrent neural network based language model. In T. Kobayashi, K. Hirose, and S. Nakamura, editors, INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, pages 1045–1048. ISCA, 2010.

[104] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In Advances in Neural Information Processing Systems, pages 3111–3119, 2013.

[105] T. Mikolov, W. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In L. Vanderwende, H. D. III, and K. Kirchhoff, editors, Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, pages 746–751. The Association for Computational Linguistics, 2013.

[106] G. A. Miller. WordNet: A Lexical Database for English. In Communications of the ACM, volume Vol. 38, No. 11:, pages 39–41, 1995.

[107] 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.

[108] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI), 1999.

[109] V. Nagarajan, R. Gupta, M. Madou, X. Zhang, and B. D. Sutter. Matching control flow of program versions. In 23rd IEEE International Conference on Software Maintenance (ICSM), pages 84–93, 2007.

[110] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In M. Hind and A. Diwan, editors, Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, pages 245–258. ACM, 2009.

[111] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. CETS: compiler enforced temporal safety for C. In J. Vitek and D. Lea, editors, Proceedings of the 9th International Symposium on Memory Management, ISMM, pages 31–40. ACM, 2010.

[112] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal. graph2vec: Learning distributed representations

of graphs. In Proceedings of the 16th International Workshop on Mining and Learning with Graphs (MLG), 2017.

[113] R. Natella, D. Cotroneo, and H. Madeira. Assessing dependability with software fault injection: A survey. ACM Computing Surveys, 48(3):44:1–44:55, 2016.

[114] N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI, pages 89–100, 2007.

[115] B. H. Ng and A. Prakash. Expose: Discovering potential binary code re-use. In 37th Annual IEEE Computer Software and Applications Conference, COMPSAC, pages 492–501, 2013.

[116] C. Nguyen Anh Quynh. Capstone: Next-Gen Disassembly Framework. In Proceedings of the 17th BlackHat USA, 2014.

[117] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In IEEE Symposium on Security and Privacy, SP, pages 601–615. IEEE Computer Society, 2012.

[118] J. Patrick-Evans, L. Cavallaro, and J. Kinder. POTUS: probing off-the-shelf USB drivers with symbolic fault injection. In 11th USENIX Workshop on Offensive Technologies, WOOT. USENIX Association, 2017.

[119] J. Patrick-Evans, L. Cavallaro, and J. Kinder. Probabilistic naming of functions in stripped binaries. In ACSAC '20: Annual Computer Security Applications Conference, pages 373–385. ACM, 2020.

[120] J. Patrick-Evans, M. Dannehl, and J. Kinder. XFL: extreme function labeling. CoRR, abs/2107.13404, 2021.

[121] PaX GrSecurity. RAP: RIP ROP, 15 years of intrusion prevention. `http://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf`, 2017.

[122] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830, 2011.

[123] F. Peng and A. McCallum. Accurate information extraction from research papers using conditional random fields. In J. Hirschberg, S. T. Dumais, D. Marcu, and S. Roukos, editors, Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL, pages 329–336. The Association for Computational Linguistics, 2004.

[124] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In 2015 IEEE Symposium on Security and Privacy, SP, pages 709–724, 2015.

[125] Y. Prabhu, A. Kag, S. Gopinath, K. Dahiya, S. Harsola, R. Agrawal, and M. Varma. Extreme multi-label learning with label features for warm-start tagging, ranking and recommendation. In Proceedings of the ACM International Conference on Web Search and Data Mining, February 2018.

[126] Y. Prabhu, A. Kag, S. Harsola, R. Agrawal, and M. Varma. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. In P. Champin, F. Gandon, M. Lalmas,

and P. G. Ipeirotis, editors, Proceedings of the 2018 World Wide Web Conference on World Wide Web, pages 993–1002. ACM, 2018.

[127] 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, pages 263–272. ACM, 2014.

[128] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In Proc. Ottawa Linux Symposium (OLS), pages 49–64, 2005.

[129] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, pages 111–124. ACM, 2015.

[130] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI, pages 798–804, 2008.

[131] K. Sato and Y. Sakakibara. RNA secondary structural alignment with conditional random fields. In ECCB/JBI'05 Proceedings, Fourth European Conference on Computational Biology/Sixth Meeting of the Spanish Bioinformatics Network (Jornadas de BioInformática), page 242, 2005.

[132] S. Schumilo, R. Spenneberg, and H. Schewartke. Don't trust your USB! How to find bugs in USB device drivers. In Blackhat Europe, 2014.

[133] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In USENIX Annual Technical Conference, pages 309–318, 2012.

[134] B. Settles. ABNER: an open source tool for automatically tagging genes, proteins and other entity names in text. Bioinformatics, 21(14):3191–3192, 2005.

[135] F. Sha and F. C. N. Pereira. Shallow parsing with conditional random fields. In M. A. Hearst and M. Ostendorf, editors, Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, HLT-NAACL. The Association for Computational Linguistics, 2003.

[136] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. Journal of Machine Learning Research, 12:2539–2561, 2011.

[137] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In 24th USENIX Security Symposium (USENIX Security 15), pages 611–626, Washington, D.C., 2015. USENIX Association.

[138] P. Shirani, L. Wang, and M. Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 301–324. Springer, 2017.

[139] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In 22nd Annual Network and Distributed System Security Symposium, NDSS, 2015.

[140] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In IEEE Symp. Security and Privacy (S&P), 2015.

[141] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of Network and Distributed System Security Symposium (NDSS). The Internet Society, 2016.

[142] W. Sun, Y. Chen, Z. Shan, Q. Wang, et al. Binary semantic similarity comparison based on software gene. Journal of Physics: Conference Series, 1325:012109, 2019.

[143] C. A. Sutton and A. McCallum. An introduction to conditional random fields. Foundations and Trends in Machine Learning, 4(4):267–373, 2012.

[144] F. Tai and H. Lin. Multilabel classification with principal label space transformation. Neural Computation, 24(9):2508–2542, 2012.

[145] The GCC team. GENERIC Intermediate Representation, GCC internals reference mannual. Available at `https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html#GENERIC`.

[146] The GCC team. GIMPLE Intermediate Representation, GCC interals reference mannual. Available at `https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html`.

[147] A. N. Tikhonov, A. Goncharsky, V. Stepanov, and A. G. Yagola. Numerical methods for the solution of ill-posed problems., volume 328. Springer Science & Business Media, 2013.

[148] R. van Tonder and H. A. Engelbrecht. Lowering the USB fuzzing barrier by transparent two-way emulation. In USENIX Workshop on Offensive Technologies (WOOT), 2014.

[149] F. Wang and Y. Shoshitaishvili. Angr - The Next Generation of Binary Analysis. In IEEE Cybersecurity Development, SecDev, pages 8–9, 2017.

[150] Z. Wang and J. Xu. A conditional random fields method for RNA sequence-structure relationship modeling and conformation sampling. Bioinformatics, 27(13):102–110, 2011.

[151] J. Weston, S. Bengio, and N. Usunier. WSABIE: scaling up to large vocabulary image annotation. In T. Walsh, editor, IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, pages 2764–2770. IJCAI/AAAI, 2011.

[152] J. Weston, A. Makadia, and H. Yee. Label partitioning for sublinear ranking. In Proc. 30th International Conference on Machine Learning (ICML), volume 28 of JMLR Workshop and Conference Proceedings, pages 181–189. JMLR.org, 2013.

[153] P. Willett. The porter stemming algorithm: then and now. Program, 40(3):219–223, 2006.

[154] W. Xu and A. Rudnicky. Can artificial neural networks learn language models? In Sixth International Conference on Spoken Language Processing, ICSLP/ INTERSPEECH, pages 202–205. ISCA, 2000.

[155] 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 Proceedings of the 2017 ACM SIGSAC

Conference on Computer and Communications Security, CCS, page 363–376. Association for Computing Machinery, 2017.

[156] P. Yanardag and S. V. N. Vishwanathan. Deep graph kernels. In L. Cao, C. Zhang, T. Joachims, G. I. Webb, D. D. Margineantu, and G. Williams, editors, Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1365–1374. ACM, 2015.

[157] H. Yu, P. Jain, P. Kar, and I. S. Dhillon. Large-scale multi-label learning with missing labels. In Proc. 31st International Conference on Machine Learning (ICML), volume 32 of JMLR Workshop and Conference Proceedings, pages 593–601, 2014.

[158] K. Yu, S. Yu, and V. Tresp. Dirichlet enhanced latent semantic analysis. In R. G. Cowell and Z. Ghahramani, editors, Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics, AISTATS. Society for Artificial Intelligence and Statistics, 2005.

[159] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In B. N. Bershad and J. C. Mogul, editors, 7th Symposium on Operating Systems Design and Implementation (OSDI, pages 263–278. USENIX Association, 2006.

[160] A. Zhila, W. Yih, C. Meek, G. Zweig, and T. Mikolov. Combining heterogeneous models for measuring relational similarity. In L. Vanderwende, H. D. III, and K. Kirchhoff, editors, Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, pages 1000–1009. The Association for Computational Linguistics, 2013.

[161] 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 Network and Distributed Systems Security (NDSS) Symposium 2019, 2019.

[162] Zynamics. Using BinDiff v1.6 for Malware analysis. Available at `https://www.zynamics.com/downloads/bindiff_malware-1.pdf`, 2019.