Statistical Approaches to Reverse Engineering

Omer Katz
Statistical Approaches to Reverse Engineering

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Omer Katz

Submitted to the Senate of the Technion — Israel Institute of Technology
Nisan 5779 Haifa April 2019
This research was carried out under the supervision of Prof. Eran Yahav, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:

<table>
<thead>
<tr>
<th>Title</th>
<th>Conference/Journal</th>
</tr>
</thead>
</table>

Acknowledgements

The research leading to the results presented in this thesis is partially supported by the European Union’s Seventh Framework Programme (FP7) under grant agreement no. 615688 (PRIME) and the Israel Science Foundation grant no.1319/16.

The Technion’s funding of this research is hereby acknowledged.
Contents

List of Figures

Abstract 1

1 Introduction 3
  1.1 Tools Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
  1.2 Architecture of Statistical Solutions . . . . . . . . . . . . . . . . . . . 8
  1.3 Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  1.4 Thesis Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2 Estimating Types and Virtual Call Targets 11
  2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
  2.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.3 Object Tracelets . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
    2.3.1 Virtual Functions and Dynamic Dispatch . . . . . . . . . . . . . . 20
    2.3.2 Tracelets and Events . . . . . . . . . . . . . . . . . . . . . . . . . 21
    2.3.3 Extracting Tracelets . . . . . . . . . . . . . . . . . . . . . . . . . 21
  2.4 Classification by SLMs . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
    2.4.1 Statistical Language Models . . . . . . . . . . . . . . . . . . . . . 22
    2.4.2 Correlating Implicit and Explicit Types . . . . . . . . . . . . . . 24
    2.4.3 Ranking Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    2.4.4 Other Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
  2.5 Implementation Details . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
    2.5.1 Base Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
    2.5.2 Finding Objects in the Code . . . . . . . . . . . . . . . . . . . . 30
    2.5.3 Identifying Events . . . . . . . . . . . . . . . . . . . . . . . . . . 30
    2.5.4 Applying Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    2.5.5 Finding Virtual Tables in the Binary . . . . . . . . . . . . . . . 31
    2.5.6 Determining Allocated Size of Types . . . . . . . . . . . . . . . 31
  2.6 Experimental Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 32
    2.6.1 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
    2.6.2 Experimental Design . . . . . . . . . . . . . . . . . . . . . . . . . 32

Technion - Computer Science Department - Ph.D. Thesis PHD-2019-03 - 2019
### 3 Reconstructing Class Hierarchies

3.1 Introduction .......................................................... 39
3.2 Overview .............................................................. 42
   3.2.1 Step I: Structural Analysis .................................. 44
   3.2.2 Step II: Behavioral Analysis ................................. 45
3.3 Behavioral Hierarchy Reconstruction ................................. 47
   3.3.1 Problem Definition ............................................. 47
   3.3.2 Finding the Most Likely Class Hierarchy ................. 48
3.4 Pruning Infeasible Class Hierarchies via Structural Analysis .......... 51
   3.4.1 Clustering Binary Types into Families .................... 52
   3.4.2 Eliminating Impossible Parents ............................ 53
   3.4.3 Handling Multiple Inheritance ............................. 53
3.5 Experimental Evaluation ............................................... 53
   3.5.1 Benchmarks .................................................... 54
   3.5.2 Ground Truth .................................................. 54
   3.5.3 Application Distance ........................................ 54
   3.5.4 Experimental Results ....................................... 55

### 4 Neural Decompilation

4.1 Introduction .......................................................... 60
4.2 Overview .............................................................. 63
   4.2.1 Motivating Example .......................................... 63
   4.2.2 Decompilation Approach .................................... 64
4.3 Neural Machine Translation ........................................... 66
4.4 Decompilation with Neural Machine Translation ...................... 67
   4.4.1 Decompiler Algorithm ....................................... 67
   4.4.2 Generating Samples ......................................... 67
   4.4.3 Improving Translation Performance with Canonicalization .. 69
   4.4.4 Evaluating Translations .................................... 71
   4.4.5 Stopping Decompilation ................................... 72
   4.4.6 Extending the Language .................................... 73
4.5 Filling the Template .................................................. 73
   4.5.1 Motivation .................................................... 74
   4.5.2 Finding assignments for constants ......................... 75
4.6 Experimental Evaluation ............................................... 76
   4.6.1 Implementation ................................................ 76
   4.6.2 Benchmarks .................................................... 77
   4.6.3 Experimental Design and Setup ............................ 78
   4.6.4 Results ....................................................... 79
List of Figures

1.1 General high-level architecture of our statistical solutions, consisting of two phases and relying on a combination of PL and ML techniques. .................................................. 8

2.1 Class definitions and example codes using MySocket and MyFile. Function sendInt send a value determined by input through a socket and function readLast returns last character of a file. .................................................. 16

2.2 Assembly code generated for function sendInt. Lines 7,16,23 match the function invocations at lines 33,35,37 of Fig. 2.1. .................................................. 17

2.3 Example tracelets for object o1 extracted from function sendInt, type MyFile from function readLast, and type MySocket. .................................................. 18

2.4 Assembly code generated for function readLast. Lines 12,20 match function invocations at lines 44,45 of Fig. 2.1. .................................................. 19

2.5 Coverage curves for 3 sample benchmarks. The curves show the percentage of evaluated virtual call sites for which the expected target was ranked at most x. To guarantee that for y% of call sites in a benchmark the expected target is examined, the top x ranked targets per call site should be examined. .................................................. 36

2.6 Histogram of types according to training set size for 2 sample benchmarks 37

2.7 Comparison of coverage curves for benchmark Smoothing.exe using different ranking methods, zoomed in on area of difference; the VMM curve covers 95.5% of the figure, while the fixed-order model covers 93.1% and the edit-distance covers 87.2% .................................................. 37

3.1 Methods used to read data from data sources. .................................................. 41

3.2 Class hierarchy of data sources. .................................................. 42

3.3 Class hierarchy as seen in source code. .................................................. 42

3.4 Class definitions and example code. .................................................. 43

3.5 Class definitions as observed in a stripped binary. .................................................. 44

3.6 Possible class hierarchy structures. .................................................. 45

3.7 Usage sequences for instances of the different classes (extracted from the stripped binary). .................................................. 46

3.8 Trained statistical language model of Class3. .................................................. 46
3.9 Class hierarchies for CGridListCtrlEx.exe: (a) ground truth, and (b) reconstructed hierarchy. ......................................................... 57
4.1 Example input (a) and output (b) of decompilation. ....................... 59
4.2 Steps for decompiling x86 assembly to C: (1) canonicalized x86 input, (2) NMT output, (3) templated output, (4) final fixed output. ............... 63
4.3 Schematic overview of the first phase of our decompiler .................. 64
4.4 Reducing vocabulary by splitting numbers to digits ....................... 70
4.5 Example of code structure alignment ....................................... 70
4.6 Example of Program Dependence Graph. Solid arrows for data depend-encies, dashed arrows for control dependencies. ........................... 72
4.7 Example of code structure alignment ....................................... 77
4.8 Success rate of x86 decompiler after a single iteration on various grammars, with compiler optimization enabled and disabled ................. 79
4.9 Cumulative success rate of x86 decompiler as a function of how many iteration the decompiler performed ................................. 82
4.10 Example of decompilation failure ............................................ 83
4.11 Failure due to redundant number ............................................ 84
4.12 Failure due to incorrect operator ............................................ 84
Abstract

Software affects all aspects of our lives. More and more industries and products incorporate software as a core component, and we routinely place our trust in software. If that software fails, either due to an attack or an error, it can have major implications for users, industries, and society. As the impact of software grows rapidly, it is imperative that we raise the level of confidence in its correctness.

One way to make sure that software is free from vulnerabilities and back-doors is to understand how it works, for example by inspecting its source code. However, the vast majority of software is delivered in binary form, without source code. When source code is not available, reverse engineering is required to lift software in low-level representation (typically machine code) to higher-level presentation that can be understood by humans. Reverse engineering is notoriously hard and requires extensive manual effort by experts.

This thesis focuses on techniques that assist reverse engineering. We tackle several reverse engineering challenges and provide statistical solutions. The problems we address in this thesis were left unsolved for many years. Some advances have been made but a solution that can be applied to realistic software remained unachievable, often due to computational complexity. The main idea behind our statistical solutions is to trade off precision for computational feasibility. These solutions are effective in focusing and directing reverse engineering efforts, and can be directly used for improving the level of software security.

We start by discussing the problem of identifying targets of virtual function calls in optimized stripped binaries [KEY16]. This is a crucial step towards understanding the control flow of the binary. It is crucial both for manual reverse engineering and for automatic analysis. Our solution relies on object tracelets — sequences of operations applied to an object — which represent how an instance of a type is used in practice, and the subtle differences between usages of different types.

We then tackle the problem of reconstructing type hierarchies in optimized stripped binaries [KRY18]. We measure a similarity between types based on similarity of usage patterns. We then elevate that similarity to a full hierarchy. Such a hierarchy is needed for understanding a given binary. Furthermore, it is essential for securing existing binaries with mechanisms such as control flow integrity.

Finally, we examine the challenge of automatic decompilation [KOGY19]. This challenge is the pinnacle of reverse engineering. Being able to automatically decompile
low-level code back to human-readable equivalent code makes it accessible to a wider range of programmers. It also allows modification and recompilation of the code as needed. We present a new approach and a new point-of-view for this problem by framing it as a language translation problem.

Throughout this thesis, we borrow insights and methods from machine learning and natural language processing. We combine those with techniques from the field of programming languages, resulting in solutions much more powerful than achievable by each field separately.

All of the approaches specified in this thesis were implemented as prototypes and evaluated using real-world binaries and code. Our evaluation shows the benefits of each approach compared to traditional solutions, and its effectiveness in understanding and securing software.
Chapter 1

Introduction

Software is all around us. It runs a wide range of devices, from watches and thermostats up to cars, medical devices, and nuclear reactors. More software is developed today than in previous years [Git18]. Today, almost anyone can write and publish software that has the potential to reach millions of users. This proliferation of software made our lives simpler and more convenient. At the same time, it increased our dependency on software and made us more vulnerable and susceptible to software bugs and exploitation.

Most software today reaches end-users as stripped binaries – i.e. without any source code, debug symbols, or any other form of documentation as to how the software operates. This unfortunately hinders attempts to properly and adequately secure software systems, thus leaving us vulnerable from a software security point of view. Software security reports indicate a constant increase in the number of reported vulnerabilities and CVEs over the years [Sym18, cve19, Sec19, WMR15].

A big portion of the effort to secure software systems is carried out by external specialists. White-hat hackers analyze and examine public software in search for potential bugs and vulnerabilities, a process known as vulnerability finding. Rather than exploiting them, they report vulnerabilities to developers. They do so in the hopes that reported issues will be resolved and fixed before malicious agents can exploit them. Even large software vendors, who often use dedicated in-house teams for exactly those purposes, rely on such outside specialists. They do so via bug bounty programs [goo10, mic13, moz04], in which people can report bugs and vulnerabilities for a cash reward.

While in some cases the external security researchers can obtain the source code of the software they are testing, that is not the general case. In many situations external researchers are forced to base their work solely on the publicly published binary executable for that software (in some settings it is even preferable to work on the binary as it allows for the discovery of bugs and vulnerabilities introduced by the compiler [BR10]).

When performing vulnerability finding on a stripped binary, the first and foremost tool available to researchers is reverse engineering. Reverse engineering (RE) is the process of examining the binary and understanding what it does and how it does it. RE
is not a single technique or method. It is a collection of questions, problems and puzzles that need to be solved, each one providing the researcher with a glimpse into the inner workings of the software. The combination of those questions ultimately answers the question “what does this program do?”.

Currently, very little of the RE process can be efficiently automated. Security firms employ specialists whose sole task is to reverse engineer binaries, be it benign binaries for vulnerability finding or malicious malware for effective mitigation.

Working as a reverse engineer is considered very difficult and tedious, thus placing reverse engineers, a crucial part of the software security effort, in constant risk of “burning out”. This results in fewer analyzed/examined binaries, and thus more vulnerabilities remaining undetected and unmitigated.

**Proposing a New Direction** Many reverse engineering problems remain unsolved due to computational infeasibility or cost.

For example, determining the exact set of types an object in some method can belong to would potentially require tracking all objects in the program and all the possible ways an object can reach that method. When dealing with big and complex binaries, Exhausting all paths through the program in order to guarantee perfect coverage is too expensive to be carried out in a reasonable time. Generally speaking, using existing techniques, one needs to choose between precision and cost (e.g. [Lan92]). A precise solution can be achieved, but would be computationally infeasible. Alternatively, reasonable-cost solutions are also possible, but their precision would hinder the usability of their results. Traditional techniques, such as those mentioned later in this thesis, cannot produce a solution that is both precise enough and incurs a feasible cost.

Our goal is to assist the reverse engineer by reducing the amount of work required to analyze a binary. Doing so would increase our overall security.

We recognize that, in some settings, having an approximate answer can be very beneficial to reverse engineers. Approximate statistical solutions can be used to direct and focus the reverse engineering efforts to improve effectiveness and productivity.

Our proposed solutions employ a combination of program analysis techniques (mostly static) with statistical machine learning tools. This combination allows us to improve on pure PL techniques (i.e., analyzing the binary) or pure ML techniques (i.e. using the binary itself as input to some predictive/generative model).

The premise of our approach lies in the reuse of software. Code is inherently repetitive. Many parts of programming can be considered as some sort of reuse. Be it the way we use type instances, how we use libraries, the way we tend to implement certain concepts, or even the way we compile code. This reuse results in patterns that repeat throughout our code/binaries. We rely on these patterns as a means of identifying various key factors in the program, and as a basis for our approximate answers.

We harness the above insights to provide statistical solutions to the following problems: (1) Identifying types of objects in a binary and Detecting targets of indi-
rect calls (Chapter 2), (2) Reconstructing class hierarchies in binaries (Chapter 3),
and (3) Decompiling low-level code to a high-level human-readable programming lan-
guage (Chapter 4).

1.1 Tools Overview

Following is a brief informal overview of the tools presented in this thesis. For each
tool we specify the problem it aims to solve, and intuitively describe its capabilities and
the output it provides the user. More formal and elaborate descriptions of the tools’
underlying techniques and mechanisms are provided in following chapters.

Note that the solutions we propose and corresponding tools are aimed mostly for
vulnerability finding. Thus we target standard benign binaries, built using off-the-shelf
compilers and with standard optimizations. We do not attempt to deal with hand-crafted,
obfuscated or packaged binaries, as is often the case with malicious binaries.

Esti  Described in detail in Chapter 2, Esti tackles the problem of identifying the
targets of virtual function calls. Esti takes as input a single stripped binary and deduces
from it the most likely targets for each virtual function call it finds in the binary. The
output of Esti is a ranked list of targets per call. Reverse engineers trying to follow
the flow of a program can use that output to focus their efforts by prioritizing which
possible targets they choose to explore. The prototype implementation of Esti is aimed
at 32-bit Windows binaries compiled from C++ source code.

The hypothesis behind Esti is that instances of different types are used in different
ways. Due to the nature of virtual function calls, the target of a call is directly determined
by the dynamic type of the object used in that call. Therefore, instead of directly
computing the most likely targets of a call, we can first compute the most likely type for
the relevant object, and deduce the target from it. Intuitively, when two types are used
in exactly the same way, it is usually beneficial to merge these types under a shared
interface. Thus, we can expect the usages of different types to differ in some way. We
leverage this differences in order to differentiate types and match objects to types.

Given an input binary, Esti starts by analyzing that binary and identifying all
virtual function calls and relevant objects. We then identify all usages of said objects to
extract usage patterns, referred to as object tracelets, for each object in the binary. Then,
by clustering the patterns for objects for which a type can be statically determined, we
obtain usage patterns for types. These usage patterns serve as our representation of
object and types. Essentially, the patterns describe an object/type not by what it can
do (i.e. what fields/methods it has), but by what it actually does (i.e. the sequence and
order of fields/methods as they are used in practice).

The usage patterns for types are further summarized by using them as training data
for a statistical language model. Given a query object o, the trained models are then
finally queried using the usage patterns of o to determine the probability of o belonging
to each type. These probabilities are used as the score by which the types are ranked for object $o$.

Evaluation of ESTI on a set of real world binaries demonstrated the tools usefulness. ESTI successfully ranked the correct most likely target for over 80% of calls in the top 3 targets. This reduces the burden of reverse engineers from dozens of possible target to only a few per call.

**Rock** The second problem discussed in this thesis is the reconstruction of class hierarchies in striped binaries, as described in Chapter 3. Similarly to ESTI, ROCK takes as input a single striped binary, from which it deduces the most likely hierarchy for that binary. Similarly to ESTI, ROCK is also aimed at 32-bit Windows binaries compiled from C++ source code.

ROCK shares some core components of ESTI, namely the analysis for extracting usage patterns and the statistical language models trained on these patterns. ROCK then extends on the foundations laid by ESTI by measuring similarity between the trained models. We use Kullback-Leibler divergence $[KL51]$ to measure the similarity.

Using the measure of model similarity, we can describe the class hierarchy reconstruction as a graph problem. The different types of the binary serve as the graph’s nodes, and a directed edge between two types, e.g. from $t_1$ to $t_2$, represents the possibility that $t_1$ inherits from $t_2$. The weight of an edge from $t_1$ to $t_2$ is set to the similarity measure between the model of $t_1$ and the model of $t_2$. Note that Kullback-Leibler divergence is asymmetric, allowing us to distinguish who inherits from who, not just that an inheritance relation exists. Finding the most likely class hierarchy for the binary is thus translated to finding a minimum-weight directed spanning-tree (arborescence) in the graph.

ROCK utilizes an additional optional optimization step designed to reduce the size of the graph and eliminate impossible hierarchies. ROCK uses existing structural cues found in the binary for that purpose. This same cues are also used by existing tools, but are often insufficient on their own and do not result in a single possible hierarchy.

We evaluated ROCK in the context of retro-fitting binaries with a Control-Flow Integrity (CFI) mechanism. CFI promises that only valid targets can be reached from indirect jumps (i.e. jumps for which the target is dynamically computed). For the case of virtual function calls, the set of valid targets directly deduced from a given class hierarchy. Our evaluation demonstrates that, compared to existing solutions, using ROCK can drastically reduce the amount of CFI false-positives, sometimes at the expense of very few additional false-negatives.

**TraFix** In Chapter 4 we provide a new point-of-view to the problem of decompilation, lifting low-level code back to the high-level source code from which it was compiled, as a language translation problem. To this end, we present the TraFix framework. TraFix takes as input a compiler $c$ and creates a decompiler that handles low-level
code compiled with \(c\). The TRAFIX framework is language agnostic and works over multiple languages and compilers.

At the core of the TRAFIX framework is a Neural Machine Translation (NMT) model. The framework surrounds the NMT model with various PL techniques. The PL techniques essentially rewrite high-level and low-level code (used either as training data or query input) in a reversible way that results in a simpler translation task. These include but are not limited to: (1) eliminating redundant information, (2) reducing the overall vocabulary, and (3) aligning high-level and low-level code so that they use a more similar sentence structure.

Given a compiler \(c\), TRAFIX automatically generates training examples by creating random high-level code and compiling it with \(c\). It then uses the random training data to train an NMT model. This creates a model trained for reversing the effects of the compiler \(c\). This results in a trained decompiler.

The trained decompiler \(d\) takes low-level statements as input \(I\) and output equivalent high-level code that can be recompiled to \(I\). For each \(i \in I\), \(d\) translates \(i\) from low-level to high-level using the trained NMT model. It then evaluates the resulting high-level code \(o\) to determine whether or not \(o\) is equivalent to \(i\). In case it is not, TRAFIX is capable of automatically advancing \(d\) by obtaining additional training data (both random and specialized to the relevant input) and retraining the NMT model. TRAFIX continues to iteratively improve \(d\) until it succeeds at decompiling the input.

Our implementation of the framework currently has two instantiations: (1) A decompiler from LLVM-IR to C based on the Clang compiler, and (2) A decompiler from x86 assembly to C based on GCC. Both instances use the same PL techniques to “wrap” the NMT model.

While this framework is not yet capable of outperforming existing decompilers, it has already proved a powerful and useful direction for solving the decompilation problem. Our evaluation shows that decompiling LLVM-IR reaches over 96% success rate. Decompiling x86 assembly, a considerably more difficult task compared to LLVM-IR, reaches over 80% using the initial training data, and exceeds 90% if allowed additional training.

Furthermore using the TRAFIX framework we can overcome significant pitfalls of traditional decompiler development – reliance on manually hand-crafted rules and patterns and limited ability to reuse such rules. Creating a decompiler from a given compiler using TRAFIX is entirely automatic, without human intervention and without modification to the framework. In the future this will help us drastically reduce the amount of time it takes to develop a decompiler (which currently takes years according to [ret17]).
1.2 Architecture of Statistical Solutions

The solutions we propose in this thesis all follow the same general architecture, as described in Fig. 1.1. All solutions follow a two-staged approach and utilize techniques from both the field of programming languages and the field of machine learning.

In the first stage, given some input, we use PL methods to generate/extract training data which we use to train a ML statistical model. The training data consists of information and examples required for solving each tackled problem. The model then “learns” from this data any relevant and applicable patterns and rules.

In the second stage, given some query input and relying on the model trained during the first stage, we query the model to obtain predictions relevant to the question we are trying to solve. We then analyze this predictions to obtain our final approximate answer. Note that the approximate part of the answer stems from the statistical nature of the predictions made by the model.

While all our solutions follow this general structure, specific parameters differ between solutions depending on the question we are trying to solve. Each of the parameters have to be specialized, suited and optimized for solving the specific problem each tool is tackling. In the first stage, these parameters include the kind of input we require, type of training data we use, how we obtain it and kind of model we train. For example, in Chapter 2 and Chapter 3, the input is a binary and training data is obtained by analyzing that binary, while in Chapter 4 the input is a pair of a grammar and a compiler and training data is obtained by sampling the grammar. In the second stage, the type of query input, type of predictions and the form of the final answer also differ. For example, in Chapter 2 the predictions will describe likelihood of an object belonging to a type and the final answer will deduce the most likely targets for a call based on said predictions. Similarly, in Chapter 3 the predictions are a measure of similarity between types, while the final answer is the most-likely hierarchy of a binary.
All of these solutions benefit greatly from the combination of PL and ML techniques. As further explained in the following chapters, using only PL or only ML to solve any of the problems we deal with, would result in inferior results, both in accuracy and in usability.

1.3 Challenges

In designing the solutions presented in this thesis, we had to address a number of challenges.

**Identifying Key Information** The first challenge in each of the problems we tackled was figuring out what information is actually needed for solving the problem. Since reverse engineering is traditionally a manual effort, we had to understand the thought process of a human reverse engineer. We focused on which aspects of the program are of interest, how they are utilized, and do they effect the final conclusions. This exploration lead us to the realization that the most important information for solving the problems of Chapter 2 and Chapter 3 is the manner in which different types are used, resulting in the introduction of “object tracelets”. Similarly, in Chapter 4 that same thought process helped us identify parts of the input that do not contribute to the decompilation and can ignored. Keeping these redundant parts of the input only hardens the translation problem.

**Learning from Small Code** Many works deal with “Big Code”. The premise of such works is that lessons learned on one program can be directly applicable to other unrelated programs. Unfortunately that is not the case in our work. In Chapter 2 and Chapter 3, anything we learn by analyzing a binary is applicable to only that binary. For example, object tracelets for instances of some type that is unique to that binary cannot be reused for other programs. While all the relevant information for solving these problems is found in the binary, it was not clear whether or not enough information could be extracted for a viable statistical solution. Feasibility of learning from a single binary, with the statistical tools available to us, was an open question. This lead to the challenge of exhausting all possible information from the binary. A lot of effort was put into maximizing the data that can be obtained and optimizing it for the problem at hand. We were able to demonstrate that this kind of learning is feasible and useful, even on small binaries that contain very little usable data.

**Remaining Useful** Given infinite time and resources, all the problems we tackled can be accurately solved. The biggest question we faced wasn’t whether or not a solution can be found. For example, exhaustive dynamic analysis could eventually identify all possible targets of a virtual function call. Similarly, exhaustively enumerating all possible high-level code snippets would eventually find one that matches some low-level
input for decompilation. None of these alternatives will be considered useful in any way since they are not guaranteed to terminate within a reasonable time. For a solution to be practically useful, it needs to be cost-effective in terms of how long it takes and how many resources it uses. The biggest question is thus whether or not a solution can be found effectively and in a timely and computationally feasible manner. The tradeoff between the cost of additional analysis/training and the risk of not having enough data was always on our minds. In Chapter 2 and Chapter 3 we managed to find a “sweet-spot” that reaches good results within a reasonable time. In Chapter 4 we took a different approach to this challenge and chose an iterative process that invests in additional training only when the decompiler determines it is necessary.

1.4 Thesis Contributions

The contributions of this thesis are as follows:

- In Chapter 2 we describe a statistical approach for identifying target of virtual function calls. This chapter introduces the concept of *object tracelets* – sequences of action applied to an object – as an effective means of representing objects, and statistical language models as a representation of types.

- In Chapter 3 we build on the representation of types as statistical language models by introducing a measure of type similarity (based on Kullback-Leibler divergence). This chapter describes a statistical solution for class hierarchy reconstruction by framing it as a graph problem, using the type similarity measure as edge weights.

- In Chapter 4 we tackle the problem of decompilation. This chapter provides a new perspective on the problem as a language translation problem. We leverage this insight and utilize neural machine translation to drive the decompilation. This chapter introduces a new decompilation framework that avoids drawbacks of traditional decompilers.

We believe this thesis provides a promising new approach that opens the door for practically useful statistical solutions to otherwise unsolved problems. The directions proposed in this thesis, relying on a combination of PL and ML techniques, hold much potential for future research.

The work described in this thesis was published in [KEY16, KRY18, KOGY19].
Chapter 2

Estimating Types and Virtual Call Targets

One of the first problems usually tackled by a reverse engineer is understanding the control and data flow of the given program. This task is made more difficult when dealing with binaries originating from object-oriented code [SY07]. A significant challenge is indirect calls to dynamically computed targets, the target of which will be reached at runtime. Finding the right target typically requires the reverse engineer to manually examine each one of the possible targets. The goal of this chapter is to assist the reverse engineer by automatically identifying the real targets of those indirect calls. Specifically, given a standard stripped (no debug information), possibly optimized, binary which uses dynamic dispatch, our goal is to \textit{statically} infer the likely targets for each indirect call site in the binary. We focus on indirect calls originating from virtual function calls. Since the target of a virtual call is directly determined by the type of the object used in the indirect call, we determine the targets by identifying the likely types of the objects.

2.1 Introduction

The Problem \hspace{1em} Given a stripped binary $bin(P)$ produced by a known compiler $CC$ from an unknown object-oriented source program, $src(P)$

1. Identify the objects in $bin(P)$, where objects are represented as location containing relevant pointers.

2. Identify the binary types in $bin(P)$, where binary types are represented as virtual function tables.

3. Construct a mapping from objects to rankings of binary types of $bin(P)$, such that for an object $o$ of type $vt_t$, the binary type corresponding to $vt_t$ is ranked highest in the ranking for $o$. 

**Our Approach**  
We present a framework for statically predicting likely types of objects in stripped binaries, and use it to determine likely targets of indirect calls.

We use **object tracelets**—statically constructed sequences of operations performed on an object—to approximately capture its potential runtime behaviors. Using statistical language models (SLMs) [Ros00], we measure the likelihood that sets of tracelets share the same source. We then rank the possible types for an object according to maximum likelihood. The underlying assumption is that objects with a similar set of object tracelets should be considered as being of a similar type.

This idea is similar in spirit to “duck typing,” used in dynamic languages where the type of an object is determined by its methods and properties (fields in C++) instead of being explicitly defined. However, rather than looking only at the presence of methods and fields, we consider their actual *usage sequences* as reflected in object tracelets. In fact, because there is no debug information, we cannot rely on correspondence between names of methods and fields, and can only rely on how they are being *invoked and accessed* as reflected in tracelets.

A distinguishing feature of our approach is that we do not require an exact match of sets of tracelets to consider two objects as being of the same type. Rather than relying on qualitative similarity metrics (e.g., identity or containment), we define a quantitative probability metric between sets of object tracelets. There is no guarantee developers will always use objects of a certain type in the same way. The quantitative probability metric allows us to overcome slight but legitimate changes in the way a objects of a certain type are used across the binary and slight changes and optimizations made by the compiler, by approximating the usage model from which the tracelets originated.

To the best of our knowledge, ours is the first work to use predictive modeling to capture and represent the behaviors of different types and objects in a binary and match objects to types based on these models.

The combination of object tracelets and predictive modeling provides a powerful tool to capture usage, behaviors and characteristics of objects and types in a binary. This combination can also be extended to measure similarity between whole programs, rather than between objects. The SLMs can also be used to compute likely shared models for type behaviors, from which a behavior-based type hierarchy can be deduced.

**Existing Techniques.**  
Static analysis of binaries is known to be a hard problem (e.g., [RLT+10, BR10, RBL06]). A lot of past work has focused on identifying variables and aggregate type information [BR07] in binaries with no debug information. Very little work has attempted to match targets to call sites or types to objects. We address this challenge, which we believe to be very valuable in practice. We do not attempt to recover all aggregate structures in the program, or to identify all variables. Nor do we try to force a single target for each call site or a single type for each object. Rather, we produce ranked lists of likely types per object, based on statistical language models (SLMs), and from these deduce ranked lists of likely targets per call site (see Section 2.4
for details). A similar problem was previously discussed in [BS96] but with poor results.

We draw some inspiration from classic work on reconstruction of aggregate types in COBOL [RFT99]. This work showed that unique usage and access patterns can be used to accurately split aggregate types into smaller atomic types. We rely on unique probabilities of sets of patterns and match them against types found in the binary.

We produce our rankings statically without executing the binary. Dynamic approaches cannot reach the same level of coverage of the binary as static approaches. Additionally, dynamic approaches cannot track all the events we wish to track and at the same granularity without prohibitively slowing down the program.

**Main Contributions** This chapter makes the following contributions:

- We introduce a new approach to dealing with the difficult task of reverse engineering binaries, using predictive modeling and statistical approaches. This approach can be also applied to other difficult reverse engineering tasks.

- We show that object tracelets can accurately determine the type of an object, and thus the targets of indirect calls, even in real-world binaries where extensive optimization and stripping have been applied and no debug information is present.

- We show that SLMs are a viable model for representing object behavior. We use them as a tool to measure correlation between objects and types and estimating types of objects.

- We implement our approach and evaluate it over real-world stripped binaries. We were able to reduce the number of likely targets to fewer than 3 for over 80% of virtual call sites over all benchmarks.

- We provide a simple static solution, with high success rates, diminishing the need to manually reverse engineer binaries.

**2.2 Overview**

We provide an informal overview of our approach. We use a simple example for illustrative purposes. More realistic examples can be found in Section 2.6.

**Type as a statistical model of behaviors.** We describe a type/object not by its name or its fields and functions but by the way it is used. The description of an object is the behavior it exhibits, as represented by sequences of actions applied to the object, and similarly for types. We refer to these descriptions as implicit types, and use them to train models that represent the types. In stripped binaries, there are no variable names, and no source language structure. As a result, there are fewer structural hints to rely on, and the behavior takes a more prominent role.
Using classification based on trained models, such object and type descriptions can be used to answer difficult questions in a statistical manner. We use it to determine probable targets for virtual calls, and probable types for objects, in stripped binaries.

We note that the choice of which actions are recorded as part of a behavior, which kind of statistical model to use, and how to solve the classification problem are all design choices. For example, the statistical model could be a fixed-rank n-gram model, some sort of a blended model employing Katz’s backoff [Kat87, CG96], or other variable-order Markov model (VMM) implementations. In our evaluation, we show that picking a variable-order model is superior to using a fixed-order (as expected), but even within the different choices of variable-order model algorithms, our experience indicates that certain implementations perform better than others.

**Virtual Function Tables (vtables)**. The virtual table of a C++ class contains function pointers to all of the class’s virtual method implementations. Virtual tables support dynamic dispatch by allowing a runtime choice of which function implementation to invoke. Whenever an object invokes a call to a virtual method, the actual call target is selected by referring to the virtual table of the appropriate class. The notion of an *explicit type* in this context refers to the low-level concept of a vtable created for each class by the compiler, and the size of memory allocated for the class’ instances.

**Problem Definition.** Given a program in the form of a standard stripped binary $b$, we denote by $VC(b)$ the set of all virtual calls in $b$, and by $VF(b)$ all the possible targets of virtual calls. Our goal is to estimate for each virtual call $vc \in VC(b)$ the likelihood that its target is some function $vf \in VF(b)$.

Towards that end, we have to solve the intermediate problem of estimating the likely types of objects. We use points-to analysis to obtain a set of abstract objects denoted by $Objects(b)$ (specifically pointers to objects) in the binary $b$, and extract the set $Types(b)$ of explicit types (virtual function tables) in the binary. We would like to estimate, for each $o \in Objects(b)$ and $t \in Types(b)$, the likelihood of $o$ being of type $t$, that is, $P_t(o)$ where $P_t(o)$ is the probability that $o$ matches the model $M(t)$ which represents type $t$.

**Motivating Example.** Consider the simple example `sendInt` shown in Fig. 2.1. This C++ function takes two parameters: a socket and an integer. If the int parameter is non-zero, the function sends its value using the socket. Otherwise, it sends a default value. The function returns an error code produced by the socket operations. This function yields the unoptimized x86 assembly code of Fig. 2.2. We assume the function was compiled using Microsoft’s Visual Studio compiler [Vis97] and uses the application binary interface (ABI) set by the compiler for 32-bit x86 binaries (see Section 2.5 for details). For this example, we use unoptimized assembly code as it is easier to understand. However, we note that our technique also works on optimized code and was evaluated on optimized binaries.

The code receives two arguments, $[\text{esp+4}]$ (corresponds to the Socket argument) and $[\text{esp+8}]$ (corresponds to the int argument). For simplicity, we omitted from
the assembly code some instructions, such as prolog, epilog, and other compiler added instructions that do not deal with the objects of the function.

The virtual calls to connect, send, and sendDefault result in the calls in lines 7, 16 and 23 respectively. These calls implement the C++ dynamic dispatch of virtual functions, where the target of the call is only determined at runtime. Our goal is to statically determine the possible targets for these calls by inferring the likely explicit types for the objects (corresponding to objects from the source code) on which they are invoked, \([\text{esp}+4]\) in this case.

**Illustrating Our Technique.** The main idea of our technique is to statically compute a set of object tracelets for each object in the program, and use the computed set of tracelets as an implicit object type. Note that in stripped binaries all types share a common namespace for fields and virtual functions – the offsets. Under this namespace all types structurally resemble each other and the only difference between the types is the order and sequences of actions (rather than the kinds of actions), as embodied in our tracelets.

Our technique consists of four steps:

1. Given a program in the form of a stripped binary \(b\), we identify explicit types \(\text{Types}(b)\) in \(b\). For each \(t \in \text{Types}(b)\), \(\text{instances}(t)\) is a set of objects of type \(t\) (see Section 2.4.2 for details). From \(\text{instances}(t)\) we extract \(\text{TT}(t) = tr_1, tr_2, ...,\) a set of tracelets for \(t\), known as “type tracelets,” such that \(tr_i = \sigma_{i,1}, \sigma_{i,2}, ...\) and \(\sigma_{i,j} \in \Sigma\), as defined in Section 2.3.2.

2. We build a set of statistical language models \(\{M(t) | t \in \text{Types}(b)\}\) by training \(M(t)\) on the corresponding \(\text{TT}(t)\).

3. For each \(o \in \text{Objects}(b)\) we extract \(\text{OT}(o)\), a set of tracelets for object \(o\), known as “object tracelets.”

4. For each \(o \in \text{Objects}(b)\) we rank all \(t \in \text{Types}(b)\) according to \(P_t(o) = \prod_{s \in \text{OT}(o)} P_t(s)\) such that \(P_t(s)\) is the probability of \(s\) originating from the model \(M(t)\). The highest ranked type \(t\) is pronounced the most likely type for \(o\).

**Extract Object Tracelets.** We begin by identifying the abstract objects in the function. We rely on a simple points-to analysis to determine aliasing between registers and memory locations that (may) correspond to object references. For the code of Fig. 2.2 there is a single object of interest, \([\text{esp}+4]\), referred to henceforth as \(o_1\) (identification of objects is described in Section 2.5.2):

- The register \(\text{eax}\) in lines 3, 12 and 19 points to \(o_1\).
- The register \(\text{edx}\) in lines 4, 13 and 20 points to \([o_1]\), which is the virtual table of \(o_1\), located at offset 0 in the object’s memory.
class BasicSocket { // socket interface
  virtual void connect() = 0;
  virtual void receive() = 0;
  virtual void close() = 0;
  int errorCode, address;
};
class MySocket : public BasicSocket{
  virtual void send(int n);
  virtual void sendDefault();
  int msgCounter;
};
class MinimalSocket : public BasicSocket {
  virtual void send();
};
class OneWaySocket : public BasicSocket {
  int msgCounter;
};
class MyFile {
  virtual void close();
  virtual void open();
  virtual void seek(int n);
  virtual void remove();
  virtual char* read();
  virtual void write(char* str);
  char *name, current, *path;
  int size;
  MyFile* parent;
};
int sendInt(MySocket* s, int x) {
  s->msgCounter = 0;
  s->connect();
  if (x) { s->send(x); } else { s->sendDefault(); }
  return s->errorCode;
}
int readLast(char* data) {
  MyFile* f = new MyFile();
  f->open();
  f->seek(f->size);
  return f->current;
}

Figure 2.1: Class definitions and example codes using MySocket and MyFile. Function sendInt send a value determined by input through a socket and function readLast returns last character of a file.
Figure 2.2: Assembly code generated for function sendInt. Lines 7, 16, 23 match the function invocations at lines 33, 35, 37 of Fig. 2.1.

- The register eax in lines 6, 15, and 22 eventually points to an entry in the virtual table of o1, and that entry is the call target.

**Statically Tracking Events for Objects of Interest.** We focus on object o1. Because eax in line 1 points to o1, we can determine that the “mov [eax+16], 0” instruction in line 2 assigns the value 0 to a field of o1. Similarly, we statically determine that, since eax in line 25 also points to o1, the “mov eax, [eax]” instruction in line 26 reads a value from a field of o1. Overall, we statically observe the following explicit events: (i) a write to field at line 2, (ii) a read from field in line 26, (iii) virtual calls at lines 7, 16, and 23.

We also observe 6 implicit events in this example: (i) an access to the object’s virtual table (read of the field in offset 0) in lines 4, 13, and 20, and (ii) the object is used as the this pointer (pointed to by ecx) in lines 5, 14, and 21. Overall, 11 events are performed on the tracked object.

We employ a points-to analysis that allows us to determine the accesses to objects in the binary (we discuss the analysis further in Section 2.5). Our analysis statically tracks the events as they appear in the function and extracts sequences of events (the object tracelets) as a representation of the object’s behavior. The tracked event sequences are illustrated in Fig. 2.3a. The nodes in the figure represent the events and the superscript numbers correlate to the relevant line of assembly code. The events are marked as follows:

1. W(x) – write to field of object at offset x
2. \( R(x) \) – read from field of object at offset \( x \)
3. \( C(x) \) – call to virtual function of object at offset \( x \)
4. \textit{this} – object was used as \textit{this} pointer

Additional kinds of events are explained in Section 2.3.

The two extracted sequences will be used as our object tracelets.

**Computing Reference Types.** To match the object tracelets we collected with an explicit type, we need reference data on which to train our SLMs. We build the reference data by collecting object tracelets correlating to objects for which the explicit type can be determined. We call these tracelets “type tracelets.” An explicit type can be determined when, for example, we observe the allocation or initialization of the object.

The function \texttt{readLast} in Fig. 2.1 is part of the same binary as \texttt{sendInt}. This function yields the x86 assembly code of Fig. 2.4. For simplicity, we omitted some instructions (prolog, epilog, etc.) from the assembly code. Similarly to the extraction of object tracelets for the object \( o_1 \) of \texttt{sendInt}, from this function we can extract the tracelet in Fig. 2.3b for the object represented by the value of \( [f] \). Because \( [f] \) holds the return value of the constructor of type \texttt{MyFile}, we know that \( [f] \) is of type \texttt{MyFile} and we associate the tracelet in Fig. 2.3b with type \texttt{MyFile} (See Section 2.4.2 for details).

In a similar manner we found additional locations where we could determine the types of objects and associate their corresponding tracelets with the relevant types. Fig. 2.3c shows an example tracelet collected for the type \texttt{MySocket}.

**Correlating Implicit and Explicit Types.** We match different objects (implicit types) and (explicit) types based on the probability that their sets of tracelets originated from the same model. We create a model based on the set of tracelets corresponding to the explicit type and match the tracelets of the object to that model, as described in Section 2.4.3.

Since we are looking to find \( o_1 \)’s actual allocated type, we can immediately eliminate
Figure 2.4: Assembly code generated for function readLast. Lines 12,20 match function invocations at lines 44,45 of Fig. 2.1.

BasicSocket as a candidate because it is an interface. Interfaces have purely virtual functions (without any concrete implementation), which have a unique representation in the binary and cannot be allocated.

Because some actions are not feasible for certain types, such as reading a field at an offset that doesn’t exist, some types are unlikely to be the type of an object. Such types cannot be candidates for o1’s type. Consider the set of tracelets we extracted for the object o1. These tracelets access a field at offset 12, the third field of the object, and call a function at offset 16, the fifth virtual function of the object. The type MinimalSocket does not have a third field and the type OneWaySocket does not have a fifth virtual function. We use this knowledge to determine that the types MinimalSocket and OneWaySocket cannot be candidates for o1’s type.

For the remainder of this example we focus only on the likely candidates of o1’s type, MySocket and MyFile.

The SLM models we use to match objects and types are VMMs. When sequences and dependencies in the data are not known to have a fixed length, as is the case in our scenario, VMMs are a natural model to use. Our VMMs are based on an n-gram model with smoothing and backoff mechanisms. We note that n-gram models are, in essence, Markov models of fixed-order n − 1, where the probability of an event is determined based solely on the n − 1 events that preceded it. The backoff mechanism transforms the fixed-order n-gram model to a variable-order Markov model by allowing to revert to a lower-order model when the current model doesn’t hold enough data. Specifically, we use the prediction by partial match (PPM) algorithm [CW84].

We compute the probability that each of o1’s tracelets from Fig. 2.3a originated from the resulting models (according to the formulas in Section 2.4.1) and multiply the
results to get a score for the entire set (as explained in Section 2.4.3). Using the trained models, the probability that $o_1$’s tracelets originated from MySocket’s model is found to be drastically higher than the probability that they originated from MyFile’s model. We thus see that MySocket’s model is more likely to be the origin model, meaning that MySocket is more likely to be $o_1$’s type. This result matches the actual type declared in the code in Fig. 2.1.

Knowing the likely type of $o_1$ is MySocket, we can now deduce that the likely targets of the virtual calls in lines 7, 16 and 23 of Fig. 2.2 are the relevant implementations of MySocket’s functions.

### 2.3 Object Tracelets

Our technique relies on the notion of object tracelets, as discussed here. First we briefly describe the dynamic dispatch mechanism used for virtual function calls. We then define the set of actions we track and describe the method used to extract object tracelets.

We assume the technical aspects of analyzing the binary, locating objects and types, and determining events are known. We only discuss the general notion of object tracelets as used in the context of our technique. The technical details can be found in Section 2.5.

#### 2.3.1 Virtual Functions and Dynamic Dispatch

Virtual functions are a mechanism of object-oriented code that allows the program to choose the desired implementation at runtime. When calling a virtual function, the implementation that will be executed is determined according to the actual runtime type of the object used and not necessarily according to its declared type. This mechanism is implemented using dynamic dispatch.

A type’s virtual functions are translated to a virtual function table, which contains pointers to the code of all the virtual functions of the type. Instances of that type hold a pointer to the virtual functions table, which is assigned by the type’s constructor at instance initialization. For example, in our setting, the constructor will assign the address of the virtual table to the object’s first field.

When calling a virtual function, the program first accesses the object’s stored pointer to get the relevant virtual function table. An offset into the table is selected, according to the required virtual function, from which the virtual function address is retrieved, to which the program then jumps. When examining a virtual function call in a binary, what can be observed is a pair of reads from memory and a jump to an address stored in a register. There are no indications of the actual virtual table and/or virtual function used.
### 2.3.2 Tracelets and Events

The notion of tracelets has been previously discussed. Nonetheless, the tracelets we use are unique and have never been used before. Previous research (such as [DY14, RVY14]) used tracelets as sequences of commands taken directly from the binary. Our notion of tracelets employs higher-level events that are tracked on objects, referred to as object tracelets.

Given a binary, we analyze all its functions and maintain a set of explicit object tracelets for each object found by our analysis. The events tracked are described in Table 2.1. These events are used as our alphabet $\Sigma$. An object tracelet is a sequence of events. We note that because we model assembly operations, an event may be the result of multiple assembly instructions.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C(i)$</td>
<td>Call a virtual function at offset $i$ in the object’s virtual table</td>
</tr>
<tr>
<td>$R(i)$</td>
<td>Read from a field at offset $i$ in the object</td>
</tr>
<tr>
<td>$W(i)$</td>
<td>Write to a field at offset $i$ in the object</td>
</tr>
<tr>
<td>$\text{this}$</td>
<td>Object passed as this pointer to a function. Object-oriented code uses a special pointer known as the this pointer for functions of class instances. This pointer is used to store the address of the class instance currently in use.</td>
</tr>
<tr>
<td>$\text{Arg}(i)$</td>
<td>Object passed as $i$-th argument to a function</td>
</tr>
<tr>
<td>$\text{ret}$</td>
<td>Object returned from called function</td>
</tr>
<tr>
<td>call($f$)</td>
<td>A call to a concrete function $f$. Marks the possible existence of actions on the object outside the scope of the current function. Relevant mostly when the object is used as an argument or as a this pointer.</td>
</tr>
</tbody>
</table>

Table 2.1: Descriptions of the events tracked by our analysis

### 2.3.3 Extracting Tracelets

By analyzing the binary, we identify and mark possible objects in the code. For each marked object $o$, we symbolically execute the binary using the object as a symbolic value $\hat{o}$. The symbolic execution tracks any usages, accesses and actions performed on the symbolic object. Each execution path taken by the symbolic execution leads to a sequence of events. These sequences, which were tracked on $\hat{o}$, are used as the object tracelets of the object $o$.

Our symbolic execution ignores branch conditions, meaning that even if it is impossible for a certain branch to be traversed in runtime, we will traverse this branch and extract sequences from it. In real (optimized) code such branches are highly unlikely as they would usually be optimized out of the binary.

Because our object tracelets are intra-procedural, when encountering calls and returns we do not follow them. Instead we record the relation between the call/return and the object as events (for example, virtual call on an object, object passed as argument, etc.) along with the call itself (for non-virtual calls, as a call($f$) event). The call($f$) event contains the address of the concrete function called. We evaluated whether
including the address of the concrete function has an effect on our results. Experiments
on a sample of our benchmarks showed the effects to be mostly marginal. Despite its
marginality, we decided to keep the concrete function address in the data in order to
leverage all the knowledge available to us from the binary, and reach optimal results.

The sets of type tracelets are constructed as the union of all tracelets of objects
predetermined to belong to the same type (as described in Section 2.4.2).

Object Tracelets of MySocket instance. We examine the function
sendInt from Fig. 2.1. This function deals with the object s of type
MySocket. From our symbolic execution for object s we get two ob-
ject tracelets: (i) W(msgCounter), C(connect), C(send), R(errorCode), and
(ii) W(msgCounter), C(connect), C(sendDefault), R(errorCode).

For simplicity, we left the events of the above example as high-level actions (with
field and function names). When translated to the appropriate stripped assembly level
object tracelets, we get the tracelets from Fig. 2.3a.

2.4 Classification by SLMs

In Section 2.3, we showed how to extract object tracelets for each object in the binary.
In this section, we show how to assign objects to their likely types. First, we give a brief
background of SLMs and VMMs in Section 2.4.1. In Section 2.4.2, we show how to map
a set of object tracelets to a specific type in cases where the type is explicitly used in
the program. Then, in Section 2.4.3, we use VMMs to build a ranking of possible types
for each object.

2.4.1 Statistical Language Models

In this work we use statistical language models to rank the possible types for an object.

Let \( \Sigma \) be a finite alphabet, and suppose we are given a training sequence \( q_1^N = q_1 q_2 \cdots q_N \), \( q_i \in \Sigma \), assumed to have emerged from some unknown stochastic source.
Our goal is to learn a probabilistic model \( P \) that will be able to assign probability
\( P(\sigma|s) \) to any future symbol \( \sigma \in \Sigma \) given any past \( s \in \Sigma^* \). The model \( P \) can then be
used to evaluate the likelihood of any test sequence \( x_1^T = x_1 \cdots x_T \) using the law of
total probability, \( P(x_1^T) = \prod_{i=1}^T P(x_i|x_1 \cdots x_{i-1}) \). We can thus utilize \( P \) for analysis,
prediction and inference. This ability is the main criterion a model has to meet to fit
our needs. Moreover, given a number \( M \) of different training sequences assumed to have
emerged from \( M \) different sources, we can build \( M \) separate models, one for each source,
and use them for classification and ranking of new test sequences.

In practice, a model considers sub-sequences of a certain length of \( x_1^T \). The length
of those sub-sequences is referred to as the order of the model. A bottleneck when
considering a fixed order model is that overestimating or underestimating the most

22
effective order $k$ can be harmful. On the one hand, the size of the training sample required to obtain an accurate model grows exponentially with the order $k$, so overestimation is problematic in the absence of a sufficiently large training sequence. On the other hand, small order modeling often fail to capture important behaviors that can only be characterized by sufficiently large contexts. In many real-world settings, any fixed-order model cannot faithfully approximate the sequences at hand. In such cases the (unknown) source can often be modeled more accurately by several sub-models having different orders.

Variable-order models alleviate the need to correctly guess a single fixed order and are able to handle variable length dependencies that simultaneously capture both small and large contexts. Generally, when variable order dependencies are present, simpler models, such as fixed-order models, are not sufficient and variable-order models are needed.

**Variable-order Markov Models** In this work we utilize Markov models, which are often the tool of choice for modeling complex sequences whose statistical characteristics are not well understood. The Markovian assumption, on which Markov models are based, states that the probability distribution of the next symbol in a sequence depends solely on the current state. This assumption is highly appropriate for describing real-world objects, as we explain in Section 2.4.3.

In a fixed-order Markov model over some finite alphabet $\Sigma$, the conditional probability $P(\sigma|s)$ is assumed to equal $P(\sigma|s')$, where $s'$ is a suffix of $s$ of some fixed length. Using a maximum likelihood approach, the training of order-$k$ Markov models involves estimating conditional distributions with respect to all possible suffixes of length $k$. Variable-order Markov model (VMM) algorithms can adaptively determine the effective dependency lengths based on the data itself, and therefore the set of fixed-order sub-models required to represent the data.

In this paper we rely on $n$-gram models with smoothing and backoff mechanisms (which we refer to as variable-order $n$-gram models), specifically the well-known prediction by partial matching (PPM) technique\textsuperscript{1} [CW84].

Typical variable-order $n$-gram models assume a known upper bound $D$ on the Markov order. The main idea is to blend together all Markov models of orders $0 \leq k \leq D$ using a backoff mechanism (which allows reverting to a lower-order model when the current model doesn’t hold enough information). When constructing $P_k$, the order-$k$ model, for each string $s$ of length $k$, we allocate a small probability mass $P_k(\text{backoff}|s)$ for all symbols that did not appear after context $s$ in the training sequence, and would require applying the backoff mechanism. This probability is used as the “cost” of using a lower-order model when needed. Thus, the backoff mechanism is constructed to satisfy

\textsuperscript{1}In the context of PPM-related literature, “backoff” is termed “escape”.

23
the following recursive relation,

\[
P_k(\sigma|s = x^k_1) = \begin{cases} 
    P_k(\sigma|s), & \text{if } s\sigma \text{ appeared in the training} \\
    P_k(\text{backoff}|s) \cdot P_{k-1}(\sigma|x^k_2), & \text{otherwise}. 
\end{cases}
\]

For the final model we define \( P(\sigma|s) = P_D(\sigma|s) \) and \( P(\sigma|\epsilon) = 1/|\Sigma| \), where \( \epsilon \) is the empty sequence.

Variable-order \( n \)-gram models differ in the way they assign probability mass to the smoothing and backoff mechanisms and to sub-model conditionals, \( P_k(\sigma|s) \). Our PPM implementation uses the well-known PPM-C method [Mof90], but quite a few other methods could be used instead, including Katz’s well-known backoff model [Kat87, CG96]. Since our approach does not depend on a specific VMM algorithm, it could benefit from integrating more advanced algorithms (e.g., PAQ8 [Mah05]).

VMMs (and some other Markov models) differ from most other machine learning tools (e.g., SVMs) in that VMMs do not rely on explicit feature generation. VMMs are supplied with sequences of letters from a pre-defined alphabet. The VMM then automatically generates its own feature set in the form of “contexts” – subsequences of alphabet letters. This means the choice of alphabet symbols can be viewed as kind of rudimentary feature generation. One of the beauties in VMMs is their ability to automatically aggregate symbols to contexts of variable length, upon which predictive conditional probabilities are extracted from the data.

### 2.4.2 Correlating Implicit and Explicit Types

We consider the set of tracelets associated with an object as the implicit type of the object. Explicit types are represented by the addresses of virtual function tables (vtables) in the program. We use \( VT \) to denote the set of starting addresses of vtables in the binary. For each address \( vt \in VT \), we define \( \text{functions}(vt) \) to be the number of functions in the vtable. For each address \( vt \in VT \), \( \text{size}(vt) \) captures the allocation size of objects that use this vtable (determining the size is explained in Section 2.5.6).

Our goal is to map the implicit type of an object into a ranking of likely explicit types. More formally, we use \( Objs \) to denote the set of objects. We then compute a mapping \( ltypes: Objs \to \mathcal{P}(VT \times \mathbb{R}) \), which maps each abstract object to pairs of explicit type and likelihood score.

Our technique is based on the observation that it is possible to automatically and statically determine the explicit types for some of the objects in the program, and thus automatically label some of the tracelets. That is, we can compute a function \( etypes: Objs \to VT \), mapping some objects to an explicit type, thus establishing a connection between a known explicit type (address of a vtable) and an implicit type.
(a set of tracelets). We use these labeled tracelets to train a statistical model used to predict types for other tracelets.

Identifying the explicit type of an object is possible in the following two cases:

- An address of a virtual table is assigned to a field of an object. Such assignments are typical in object allocation, as allocated objects contain a reference to their relevant virtual table.
- The object is represented by the initial value of the ecx register at the start of a virtual function vf. In such cases, we mark the object as belonging to the type associated with the virtual table containing vf since, as a convention of x86 binaries, ecx is used to pass the this pointer to member functions of classes.

We propagate the marked types to calling functions. Since propagating the types through the entire binary is infeasible, this is a limited propagation intended to return the types to functions that call constructors. The limited propagation propagates the marked types a few levels up the call hierarchy leading to the current function. Specifically, our implementation propagates to 2 levels.

We define a function $TT: VT \rightarrow P(\Sigma^*)$, which maps a type (vtable address) to a set of tracelets. The set of tracelets for $vt \in VT$ is defined as the union of tracelets for objects that use this vtable. That is, $TT(vt) = \bigcup_{\text{etype}(o) = vt} OT(o)$.

### 2.4.3 Ranking Types

We rank the possible types for an object according to the probability that the object’s tracelets are the output of the same model that created each type’s tracelets.

**Defining the SLMs.** We define the SLM’s alphabet symbols as the set of all unique actions we find in our tracelets. Each action corresponds to a different symbol. Thus the tracelets are sequences of letters that the VMM’s inner model can parse and analyze.

The contexts, and the states accompanying them, correspond to the sequences of past actions applied to an object. This notion perfectly describes the states of real-world objects. Consider, for example, an object instance of type Socket. The object can be in one of several states, either uninitialized, initialized, connected or closed. The state of the object is determined by the actions applied to it. If we apply a connect action, the object will be in a connected state. Each state of an object instance defines a set of legal and illegal actions. These actions can be directly translated to a probability distribution for that state. Therefore, the context (i.e. the “history” of an object) determines the probability distribution for future actions applied to an object. For example, the probability distribution of a Socket in the connected state will assign some probability to the actions send, receive and close, and 0% probability to actions such as init. Thus the Markovian assumption evidently holds for real-world objects.
Building the SLMs. We train a SLM instance for each type. Training is conducted by inputting the type’s tracelets to the SLM learning algorithm, which generates a probabilistic model for this type. We do that for each type separately, resulting in a trained SLM instance for each type. These SLMs are used as a reference point with which we compare the implicit types of the objects.

Calculating object-type score. The match score for each object-type pair is the probability that the tracelets corresponding to the object originated from the model corresponding to the type.

Given an object \( o \) and a type \( t \), we compute the probability

\[
P_t(o) = \prod_{s \in OT(o)} P_t(s)
\]

such that the function \( OT \) maps an object to the set of object tracelets for that object. Given a type \( t \), \( P_t(s) \) returns the probability that the sequence \( s \) originated from the statistical language model \( M(t) \), representing \( t \), as calculated by the SLM.

For each object \( o \) and each type \( t \) we compute the score \( P_t(o) \), which we use as a likelihood score and rank the possible types for each object according to that score.

Alternative match score. We considered an alternative means of calculating a match score of an object-type pair. In this alternative, in addition to training a model for each type, we would also train a model for each object. We would then calculate a distance between the models (for example, using the Jensen-Shannon divergence [Lin06] as our distance metric) and use that as the match score. We eventually decided against this approach as the objects have relatively fewer tracelets compared to most types, and this will result in degenerate models that will not properly describe the objects. We refer back to this idea in Chapter 3.

Compatibility of types. We split the set of possible types for each object into 2 subsets: (i) compatible types, and (ii) incompatible types. The incompatible types can be predetermined as not the correct type for an object; thus the final ranking will only consider compatible types. The ranking of the compatible types will be explained in Section 2.4.3.

The division into subsets is based on insights gained from our analysis as to the structure of each type as well as on additional methods described in Section 2.5.

Given an object tracelet \( t \), we define \( fields(t) \) to be the set of integer offsets of all \( W(i) \) and \( R(i) \) events in the tracelet. Given a set of object tracelets \( ot \), we define \( fields(ot) = \cup_{t \in ot} fields(t) \). Similarly, \( functions(t) \) is the set of integer offsets into the vtable used when calling a virtual function in \( C(i) \) events of the tracelet \( t \). We define \( functions(ot) \) as \( functions(ot) = \cup_{t \in ot} functions(t) \).
Given a set of object tracelets $ot$, corresponding to an object $o$, and given an explicit type $vt$, we say that the explicit type is an incompatible match to the implicit type when:

\[
\max(\text{fields}(ot)) \geq \text{size}(vt) \quad \text{or} \quad \max(\text{functions}(ot)) \geq \text{functions}(vt).
\]

That is, if the maximum offset used in field accesses in a tracelet is higher than the type’s allocated memory size, or if the maximum offset in a tracelet is higher than the size of the type’s virtual table, we know that this type cannot be a match. In such cases, the explicit type will be classified as incompatible.

In practice, only the remaining types, which were not marked as incompatible, need to be ranked.

### 2.4.4 Other Models

The choice of VMMs for classifying tracelets to a type is well motivated by the reported success of VMMs in modeling sequences in a variety of other application areas such as natural language processing [SS94], proteomics [BY01], web query recommendations [HJL+09], music analysis [PK09], and behaviometric identification [NYEYM03]. In such problems, the objects to be analyzed are naturally represented as sequences of variable length over some finite alphabet and can be effectively characterized via finite order Markov dependencies.

We now discuss a few other possible models we have tested/considered.

**Edit-Distance Metrics.** Edit-distance metrics are a well-known tool in sequence analysis and have been previously explored on related problems, such as the rewrite engine of [DY14].

The basis of an edit-distance metric is the assignment of a cost to each edit operation needed to align/match the two sequences. The costs of operations are typically preassigned using domain knowledge (i.e., they are not learned from the data). As such, deletion/replacement of an important letter usually costs the same as that of an unimportant letter. Assigning different costs to operations/letters according to the problem domain is possible. However, in our domain, the importance of letters differs between different parts of a problem and cannot be addressed in advance, as it might depend on both the context and the state of the problem.

Furthermore, when encountering an instance of a parent type, the metric can’t distinguish between the parent and types inheriting from it. This is because all valid uses of a parent type are also valid uses of any inheriting type, and thus the inheriting type’s set of tracelets will contain the set of the parent type. In such cases, the edit distance metric will assign the same rank to all related types.
Other Predictive Models. Machine learning offers a plethora of other approaches for learning variable length sequences beyond generative probability models such as VMMs and HMMs. Several families of techniques were proposed to allow the handling of sequences within discriminative models, such as support vector machines (SVMs) and other kernel methods, which are typically very effective in constructing sharp and accurate decision boundaries, but require fixed feature vector representation. The basic ingredient shared by all kernel methods, including SVMs, is the kernel function that, intuitively, quantifies the similarity between pairs of objects to be analyzed. Beyond the technical requirement (of positive semidefiniteness), such kernel functions should be efficiently computed without losing useful information. Notable examples of such kernel methods are string kernels [LSST+02, EWNL02, SVUA04, CV05] and Fisher kernels [JH+99].

Other viable probabilistic methods for structured prediction include graphical models such as Bayesian networks, factor graphs, and conditional random fields (CRFs); these methods have been effectively applied in various domains. While many graphical models are generative and attempt to explicitly model a joint probability distribution $P(y, x)$ over a vector $x$ of input variables and a vector $y$ of outputs, CRFs, much like logistic regression, are discriminative, and attempt to model only the conditional distribution $P(y|x)$, which is sufficient for classification. In particular, CRFs [SM11] have been recently used successfully in a related problem of predicting program properties [RVK15].

Comparing models. Most of the mentioned models are relatively complex and require either a relatively large training set and/or tuning of hyper-parameters (e.g., SVMs). We opted to use VMMs after concluding that other models lack the necessary training data or are too complex for our needs. VMMs provide us with predictive modeling that is simple enough to use and understand while still allowing the flexibility to capture dependencies and probabilities that would otherwise go unnoticed if a simpler model was used.

Similarly, edit-distance metrics are simply too crude to reflect the slight differences between different contexts which use the same letters. VMMs can learn weights and probabilities from the actual data and are thus able to adapt to changes mandated by different contexts. For example, when dealing with type hierarchies, sets of inheriting types can contain tracelets not present in the parent type’s set. The VMMs of the parent type and the inheriting types will, therefore, differ from one another. In such cases, VMMs will be able to reduce the number of similarly ranked types.

2.5 Implementation Details

We implemented this technique in a tool called Esti\textsuperscript{2}. The following describes the implementation and the different steps it consists of.

\textsuperscript{2}Source code available at https://bitbucket.org/omerktz/esti
2.5.1 Base Analysis

Our technique is based on a static intra-procedural Steensgaard-style points-to analysis [Ste96]. Given a binary, our analysis computes for each function the set of states possible in each location of the function. Each state holds the values for each register, memory address, and stack offset. Addresses, offsets, and values are represented by expressions uniquely determining their actual values in terms of the initial values given at function entry point. States corresponding to the same location may differ due to different execution paths. Our analysis is similar in spirit to symbolic execution. It can be viewed as a simple symbolic executor tailored specifically for the extraction of tracelets. It does not aim to mimic the abilities of other symbolic executors, such as floating-point arithmetic. Our analysis suffers from similar problems as symbolic execution, which it deals with using similar tricks. However, because our analysis is strictly intra-procedural, it remains practical.

Unlike other works that have used points-to analysis (such as [BR08, GBT+05] and others), we use a simpler analysis suited to our needs. Since we deal with binaries produced from a standard compilation model, we know that some operations will have no effect on our results and can be ignored. Such operations include, for example, binary-and and binary-or, which in our setting should not be part of an expression representing an actual object in the binary.

Our analysis was geared towards the ABI set for 32-bit x86 Windows binaries compiled using Microsoft's Visual Studio compiler [Vis97]. Most challenges our analysis faces are due to the C++ language and constructs. However, knowing the ABI does simplify certain aspects: the location of function arguments (last values pushed to the stack prior to the call) and the offset of the virtual table pointer in the object are determined by the ABI. The ABI is also reflected in the patterns we look for when searching for virtual function tables, objects, virtual function calls, field accesses, etc. The analysis is the only part of this work whose implementation is architecture specific: the remainder of our technique is oblivious to the ABI. The analysis was implemented to handle 32-bit x86 assembly. It can be modified to deal with other ABIs since both the patterns we look for and the locations we search can be easily switched. We note that, given a binary, there are known techniques for detecting most public compilers and ABIs, and therefore we do not consider our ABI assumption a drawback. Our analysis is based on the disassembly provided by IDA [IDA00] and can handle unoptimized and optimized builds. Compiler optimizations of binaries are, in fact, mostly beneficial to our technique, as the extensive inlining makes our tracelets longer and more descriptive.

We chose to focus on Windows binaries because we believe this is the common and harder case in which the source code is unavailable and our technique is needed.
<table>
<thead>
<tr>
<th>Event</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(i)</td>
<td>Matching the call target to the virtual function call pattern described in Section 2.5.2</td>
</tr>
<tr>
<td>R(i)</td>
<td>A field of the object is used as the right operand of an operation</td>
</tr>
<tr>
<td>W(i)</td>
<td>A field of the object is used as the left operand of an operation (such as mov)</td>
</tr>
<tr>
<td>this</td>
<td>ecx pointing to the object at the location of a call operation.</td>
</tr>
<tr>
<td>Arg(i)</td>
<td>Finding the object on the stack as maintained by our analysis at the location of a call operation</td>
</tr>
<tr>
<td>ret</td>
<td>Matching return values in registers (eax) to an object</td>
</tr>
<tr>
<td>call(f)</td>
<td>A call command to a known address</td>
</tr>
</tbody>
</table>

Table 2.2: Identification of tracked events by our analysis

### 2.5.2 Finding Objects in the Code

In order to find the objects used in the code we use the product of our base analysis. We go over the states collected during our analysis and search for call target expressions matching the pattern of a virtual call. The pattern we look for is of the form \([OBJ]+OFFSET\). Any expression used in a call instruction and matching this pattern is considered a call to a virtual function, and the part of the expression matching the OBJECT is considered a candidate for an object expression.

To eliminate as much noise as possible, we filter out some of the object expressions, keeping only those that match the following structure:

\[
Obj := \text{Regs} | Z | [Obj] | (Obj \ op \ Obj)
\]

where \(op \in \{+,-\}\). Other expressions (such as where \(op \in \{*,/\}\)) are extremely unlikely to result in a real object originating from the source code and are most likely noise resulting from compiler structures and the like.

The fact that our analysis is entirely static allows us to also easily and cheaply track accesses to fields of objects. After finding all objects in the code used for virtual function calls (and objects with a predetermined type, as explained in Section 2.4.2), we go over the states and search for field access patterns. The field access pattern is \([OBJ]+OFFSET\), that is, a dereference to an address positively offset to the beginning of the object. We find all expressions of the field access pattern whose OBJECT section matches an object expression and mark them as field accesses, either read or write.

### 2.5.3 Identifying Events

Table 2.2 explains for each of the events from Table 2.1 how we identify them and when we add them to our tracelets. The descriptions in the table assume the use of the ABI mentioned in Section 2.5.1.
2.5.4 Applying Analysis

Consider lines 19-23 from Fig. 2.2. By analyzing these lines, and assuming the initial value for esp is denoted by ESP, we will obtain the following mapping after line 23:

- edx→[[ESP+4]]
- ecx→[ESP+4]
- eax→[[ESP+4]]+16.

From the mapping above we can determine that the target of the call in line 23 is [[[ESP+4]]+16].

Using the pattern from Section 2.5.2, we determine that this call is to a virtual function of the object [ESP+4] at offset 16. Since [ESP+4] also matches our object filtering pattern we consider it as a valid object and will estimate a type for it. We mark this event as C(16).

Before type estimation, we need to collect all data for this object. By re-examining the mapping above, we identify several additional events: (i) the assignment to edx is a read of the object’s field at offset 0, marked as R(0), and (ii) the object is used as the this pointer as determined by the value of ecx, marked as this.

2.5.5 Finding Virtual Tables in the Binary

A virtual table is represented in the binary as a sequence of pointers to functions. The address of the first entry is used as the address of the virtual table. Those locations that use the virtual table address as immediate values are usually either constructors or destructors of the type. We analyze the binary and search for such sequences (of any length) which we mark as virtual tables.

2.5.6 Determining Allocated Size of Types

By iterating over the references to the virtual tables we found, we collect all the constructors and destructors of a class. The main difference between constructors and destructors is that the latter usually results in the object being deleted. To identify constructors and destructors, we search for locations that use the virtual table address as a constant (e.g. the constructor assigns the virtual table address to the constructed object). We then separate the destructors from the constructors by identifying the address of the delete operator (in stripped binaries as well), and backtracking up the call hierarchy leading to it.

Using our analysis we can determine the expression representing the object initialized by the constructor. We backtrack along the path leading to the constructor call until we find the point of allocation (static or dynamic) of the object.

In the case of a dynamic allocation, determined by a call to the new operator, we can determine the allocated size by checking the state of the stack as maintained by our analysis. The last value entered into the stack is the allocated size.
In the case of a static allocation we determine the allocated size as the distance between this object to the next statically allocated object. Distinguishing objects statically allocated on the stack can be done by observing the "base addresses" used to access fields (i.e. the address to which an offset is added whenever a field is accessed). This is a known technique utilized by exiting disassembler (e.g. [IDA00]). This process might result in the determined value being higher than the real value for the allocation size. When we use this value to classify incompatible type candidates (Section 2.4.3), the higher value might result in a false positive (meaning types are incorrectly considered as compatible candidates), but would never result in a false negative.

2.6 Experimental Evaluation

2.6.1 Benchmarks

We tested our implementation on 20 open source projects collected randomly from public internet source control repositories, such as sourceforge. The benchmarks included mostly executables (exe files) and a few dynamic libraries (dll files) written in C++. We compiled the benchmarks from source code as 32-bit binaries on a Windows machine using Microsoft’s Visual Studio compiler [Vis97] as release builds (which are optimized and stripped by default).

Other than the desired ABI, no aspect of the benchmarks was known in advance. In fact, most of our benchmarks were collected after our technique was implemented. We used a small fraction of our benchmarks for calibration and then evaluated it on all collected benchmarks.

2.6.2 Experimental Design

Our experiments were designed to emulate realistic use of our technique. Thus, we took into account the typical workflow of a reverse engineer, both in the design and evaluation of the experiments.

**Design.** After performing our analysis, as previously described, and extracting tracelets, we have a labeled sets of tracelets (for objects whose type can be predetermined by the analysis) and unlabeled sets of tracelets (for the rest of the objects). Because virtual calls sites whose target cannot be automatically determined will pose the greatest challenge to reverse engineers who use our technique, these calls are defined as our test set. We therefore obtain a natural division of tracelets into training sets and test sets.

We used all of our labeled sets of tracelets to train the SLMs. We then used the trained SLMs to classify the unlabeled sets of tracelets. To evaluate our results, we manually reverse engineered the benchmarks and manually determined the declared type of each object. The declared type was used as the expected top ranked type of the object. We then used this to determine the expected target of each virtual function call. The declared types in each benchmark were manually determined independently of
Table 2.3: Statistics of results from the benchmark evaluation.

The right-hand part contains statistics of the benchmark: binary size, number of types, objects and virtual call sites, total length of training data, portion of coverage figure covered by the curve; the left-hand part shows statistics of the rank of expected targets for virtual call sites.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>binary size (Kb)</th>
<th># types</th>
<th># objects</th>
<th># calls</th>
<th>total length of training data</th>
<th>area under curve</th>
<th>remaining targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>cppcheck.exe</td>
<td>97</td>
<td>12</td>
<td>1</td>
<td>1</td>
<td>2067</td>
<td>84.6</td>
<td>2 2 2 2 0</td>
</tr>
<tr>
<td>patl.exe</td>
<td>36.5</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>351</td>
<td>75.0</td>
<td>2 2 2 2 0</td>
</tr>
<tr>
<td>pop3.exe</td>
<td>24</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>12478</td>
<td>83.3</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>smtp.exe</td>
<td>26</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>12442</td>
<td>83.3</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>td_unittest.exe</td>
<td>101</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>21246</td>
<td>75.6</td>
<td>2 2 2 2 0</td>
</tr>
<tr>
<td>template_regtest.exe</td>
<td>69</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3719</td>
<td>80.0</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>AntispyComplete.exe</td>
<td>247</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>13745</td>
<td>85.7</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>tinyserver.exe</td>
<td>46</td>
<td>12</td>
<td>4</td>
<td>6</td>
<td>6280</td>
<td>89.2</td>
<td>1 2 1.4 1 0.24</td>
</tr>
<tr>
<td>echoparams.exe</td>
<td>58</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>15840</td>
<td>89.0</td>
<td>1 2 1.43 1 0.24</td>
</tr>
<tr>
<td>ShowTraf.exe</td>
<td>137</td>
<td>38</td>
<td>6</td>
<td>7</td>
<td>171967</td>
<td>76.6</td>
<td>1 19 9.00 10 29.14</td>
</tr>
<tr>
<td>yale.exe</td>
<td>68</td>
<td>26</td>
<td>5</td>
<td>7</td>
<td>33563</td>
<td>92.0</td>
<td>1 3 2.14 3 0.98</td>
</tr>
<tr>
<td>balprp.exe</td>
<td>529</td>
<td>32</td>
<td>7</td>
<td>8</td>
<td>2285</td>
<td>67.0</td>
<td>1 21 10.88 11 59.11</td>
</tr>
<tr>
<td>tinyxmlSTL.dll</td>
<td>88</td>
<td>75</td>
<td>14</td>
<td>17</td>
<td>4960</td>
<td>93.1</td>
<td>1 19 5.18 4 24.26</td>
</tr>
<tr>
<td>gper.exe</td>
<td>84</td>
<td>11</td>
<td>10</td>
<td>21</td>
<td>41085</td>
<td>90.8</td>
<td>1 3 1.10 1 0.18</td>
</tr>
<tr>
<td>MidiLib.dll</td>
<td>400</td>
<td>96</td>
<td>7</td>
<td>26</td>
<td>6301</td>
<td>98.0</td>
<td>1 9 1.88 1.5 2.56</td>
</tr>
<tr>
<td>tinyxml.dll</td>
<td>60</td>
<td>51</td>
<td>36</td>
<td>37</td>
<td>4449</td>
<td>82.7</td>
<td>1 42 8.89 4 128.91</td>
</tr>
<tr>
<td>libtemplate.dll</td>
<td>1233</td>
<td>229</td>
<td>39</td>
<td>60</td>
<td>17967</td>
<td>84.5</td>
<td>1 165 35.28 12 2247.22</td>
</tr>
<tr>
<td>Analyzer.exe</td>
<td>419</td>
<td>62</td>
<td>78</td>
<td>64</td>
<td>97275</td>
<td>86.8</td>
<td>1 34 8.13 9 34.48</td>
</tr>
<tr>
<td>Smoothing.exe</td>
<td>453</td>
<td>71</td>
<td>130</td>
<td>306</td>
<td>112027</td>
<td>97.5</td>
<td>1 25 1.71 1 5.76</td>
</tr>
<tr>
<td>CGridListCtrlEx.exe</td>
<td>151</td>
<td>35</td>
<td>220</td>
<td>406</td>
<td>163390</td>
<td>95.9</td>
<td>1 18 1.40 1 2.90</td>
</tr>
</tbody>
</table>

The right-hand part contains statistics of the benchmark: binary size, number of types, objects and virtual call sites, total length of training data, portion of coverage figure covered by the curve; the left-hand part shows statistics of the rank of expected targets for virtual call sites.
our automatic reverse engineering, and before we ran the tool on the benchmarks. This manual effort took around 2-3 days of expert work per benchmark.

If our manual reverse engineering procedure determined an object’s type to be that of a superclass, we defined this as the correct type even though, in practice, the object’s type could also be that of any of the children of that superclass. In practice, manual examination of a sample of results where the expected type was a superclass revealed that the children types were usually ranked similarly to the superclass.

We note that some of the virtual call sites found in the binary could not be used in our evaluation. These call sites relate to objects for which we were unable to manually set a type or whose type doesn’t exist in the binary (for example, dynamically linked types). We were unable to set a type for an object when the assembly operations on the object could not be associated with a command from the source code (most often these were compiler generated objects). In such cases, since the ranking for these objects could not be evaluated, we excluded them from our evaluation of the results.

We ran our experiments on a Linux machine with 64 AMD Opteron(TM) 6376 processors, each operating at 2.3GHz, and 128GB RAM, running Ubuntu 14.04. The symbolic analysis used to extract tracelets is expensive, and took a few of hours per benchmark. The training and prediction parts took less than 10 minutes.

Evaluation. We evaluated our results by the number of top ranked targets a reverse engineer would have to examine for each call site to ensure the correct target was examined from the ranking generated by our technique, as determined by the rank assigned to the correct target. This metric best captures what makes a solution to the problem we solve a useful one. As a numerical representation of our evaluation metric, we also use the area under our coverage curves (as described in Section 2.6.3).

A reduction in the number of possible targets for each call site directly translates to a reduction in the amount of work left to the reverse engineer. Since each examined target could result in an entire call hierarchy being examined in full, reducing the number of targets by even one could be a dramatic improvement.

2.6.3 Results

Table 2.3 presents the final results of the evaluation of our technique. The rows of the table correspond to the benchmarks we tested.

The left part of the table presents general benchmark statistics and consists of 6 values: (i) the size of the binary (in Kb); (ii) the number of types found in the binary; (iii) the number of objects used in the evaluation; (iv) the number of virtual call sites used in the evaluation; (v) the total sum length of all the labeled tracelets extracted from the binary (we later show that this is interesting as a criterion for the technique’s success); and (vi) the area covered by graphs such as in Fig. 2.5 (a numerical indication of the quality of our results, as will be explained shortly).

The right part presents statistics of our measure of success: the number of remaining
targets per call site that need to be examined, that is, the number of top ranked targets that need to be examined before the correct expected target is examined (assuming the user examines targets according to our ranking). This number is determined by the rank of the correct target and the number of targets that were ranked at least as high as the correct one. The statistics include the minimum and maximum number of targets, the mean, the median, and the variance.

The rows of the table are sorted by the number of call sites used in the evaluation. The benchmarks are split into two groups. Above the separating line are very small benchmarks, in which it is easier to track and verify the steps of our technique. Below the line are larger benchmarks, which we consider more interesting in the context of the evaluation, as they better illustrate the actual value of our technique.

The results in Table 2.3 show that in 14 of our benchmarks (70%) we are able to reduce the average number of targets that need to be examined to less than 3, even on large benchmarks where the original number of targets is very high. When measured on all call sites, across all benchmarks, we reduced the number of targets per call site to fewer than 3 in over 80% of the call sites.

Let’s examine the row of benchmark “libctemplate.dll.” For this benchmark the minimum number of remaining targets is 1 and the median is 12. That means that for most call sites there are no more than 11 targets that were ranked higher than the correct one, and there are call sites for which no other target was ranked higher than the correct one. The maximum number of remaining targets for this benchmark is the highest among all our benchmarks (a single call site had 165 remaining targets out of 172). Still, the benchmark exhibits a respectable reduction overall. This can be seen by comparing the number of remaining functions to the number of types in the benchmark (from the # types column), as (almost) every type results in a possible target (types which are part of the same hierarchy might share the same function implementations).

The graphs in Fig. 2.5 give a visual representation of the number of functions that need to be examined for each call site. Given a ranking \( t_1, t_2, \ldots, t_n \) of types generated by our technique, we select a value \( x \) for the x-axis, and examine the top \( x \) targets from the ranking, \( t_1, \ldots, t_x \). The graphs show the percentage out of all call sites in the benchmark for which \( t_1, \ldots, t_x \) contains the actual target. Alternatively, a reverse engineer could use the graphs to determine how many top ranked targets should be examined, in order to guarantee that the correct target is examined for \( y \% \) of the call sites. We aim to maximize \( y \) while keeping \( x \) reasonably small. The faster the curve rises (and thus covers a greater portion of the figure), the fewer the functions that need to be examined and, therefore, the better our results. We refer to these graphs as coverage curves. Table 2.3 we can see the percentage of the graph that is under the coverage curve; higher percentages mean better results.

The values of the x-axis vary between different graphs. The top value of the x-axis is set according to the maximum number of functions that needed to be examined before our technique was used, meaning that the top value of the x-axis is determined by the
Figure 2.5: Coverage curves for 3 sample benchmarks.
The curves show the percentage of evaluated virtual call sites for which the expected target was ranked at most $x$. To guarantee that for $y\%$ of call sites in a benchmark the expected target is examined, the top $x$ ranked targets per call site should be examined.

case in which all the functions need to be examined.

Smoothing.exe and CGridListCtrlEx.exe are two of our better benchmarks, and bafprp.exe is one of our worst. We can see from the graphs that to reach over 85% call site coverage, Smoothing.exe and CGridListCtrlEx.exe require using the top two ranked targets and the top ranked target for each call site, respectively, while bafprp.exe requires 21 targets.

We find a correlation between the sizes of the training sets used to train the SLMs and the quality of our results. As can be expected, our results improve as the size of our training set increases. In Fig. 2.6 we see the distribution of types by their training set size for the benchmarks Smoothing.exe and bafprp.exe. We can see that bafprp.exe has many types with very small training sets while for Smoothing.exe the portion of types with small training sets is relatively smaller. This can explain the results from Fig. 2.5. The small sizes of the training sets of bafprp.exe mean the resulting models are decayed and are too similar to other models to be effective.

Other ranking methods. As discussed in Section 2.4.4, we considered several other models for ranking the targets for each call site. Two such methods, which we implemented and tested, are edit-distance metrics and low-order fixed-length Markov models (smoothed with the KT estimator [BEY06, KT06]). Our edit-distance metric was based on the well-known Levenshtein-Damarau distance extended to sets of tracelets using min weight matching.
Figure 2.6: Histogram of types according to training set size for 2 sample benchmarks

Figure 2.7: Comparison of coverage curves for benchmark Smoothing.exe using different ranking methods, zoomed in on area of difference; the VMM curve covers 95.5% of the figure, while the fixed-order model covers 93.1% and the edit-distance covers 87.2%
Fig. 2.7 shows the result of these methods on the *Smoothing.exe* benchmark. The figure is zoomed in to focus on the area of difference between the coverage curves. The caption includes the percentage of the full graph that is under each coverage curve. In this figure, we see that a reverse engineer using our VMM-based approach will only have to consider the 2 top ranked targets to find the correct target of a virtual call in 92% of the cases. In contrast, using a fixed length model will require the reverse engineer to consider 9X more targets, that is, up to 18 targets to reach the same percentage of calls. Using the edit-distance would force the reverse engineer to consider up to 11X targets compared to our VMM-based approach. Working on the binary directly, without any tool assistance, would require the reverse engineer to consider all virtual functions (at the offset used by the call) as possible targets for this call, that is 71 targets in the worst case (roughly 30X more than when using VMMs).
Chapter 3

Reconstructing Class Hierarchies

Another fundamental challenge in reverse engineering of object-oriented code is reconstructing the class hierarchy of the original program. Obtaining the class hierarchy is a critical step in understanding the flow of control in a program, and a stepping stone for understanding the overall program architecture [SY07]. Additionally, knowing the class hierarchy helps to improve software security as it makes it possible to harden existing applications without breaking programs [PCvdV+17].

In this section present a technique for reconstructing the class hierarchy of an unknown object-oriented program from its stripped binary, i.e., from an executable code with no debug information. This task is challenging as it requires identifying types and their relationships based solely on assembly code with no explicit names. In particular, we have to determine the parent of each class and construct a consistent inheritance tree.\(^1\)

3.1 Introduction

**The Problem**  Given a stripped binary \(\text{bin}(P)\) produced by a known compiler \(\text{CC}\) from an unknown object-oriented source program, \(\text{src}(P)\)

1. Identify the binary types in \(\text{bin}(P)\), where binary types are represented as virtual function tables.
2. Construct a hierarchy over the binary types of \(\text{bin}(P)\), such that binary type \(\text{vt}_p\) is an ancestor of type \(\text{vt}_c\) if the source class compiled into the binary type \(\text{vt}_c\), denoted by \(\text{vtToSrc}(\text{vt}_c)\), is a subclass derived from \(\text{vtToSrc}(\text{vt}_p)\).

**Main Insight**  Given a non-optimized binary, determining the exact class hierarchy is fairly simple. In the case of stripped binaries, however, this problem becomes extremely challenging due to the lack of debug symbols or the structural information utilized by most existing techniques. Thus, we suggest using behavioral information to infer

\(^1\)To simplify presentation, we assume each class has at most one superclass. Our approach, however, does handle multiple inheritance (see Section 3.4.3).
subclass/superclass relations. The main idea is to capture the behaviors of each type using a statistical language model (SLM) [Ros00] and define pairwise similarity based on the Kullback-Leibler divergence (DKL) [KL51]. Since derived types inherit the behaviors of the parent type, the parent’s set of legitimate behaviors is contained in the set of each of its derived types. SLMs trained on similar languages are fairly similar; we therefore hypothesized and then experimentally validated that SLMs of related types are similar as well. This pairwise similarity is then lifted to infer the most likely class hierarchy.

The use of SLMs to capture pairwise similarity makes it possible to measure behavioral similarity between types. Behavioral similarity can be used to augment the structural similarity used in previous work.

**Our Approach** We present the first statistical approach for static reconstruction of the class hierarchy based on behavioral similarity between types. Our technique combines statistical modeling of the behaviors in each type with a preceding structural analysis, used to filter out cases where types cannot be in the same hierarchy, as described below.

- **Structural Analysis:** We partition the types in the program into different type families. Types in the same family might be related by inheritance. More importantly, types in different families are determined not to be related. The analysis in this stage is based on the structure of vtables (addresses of virtual functions) and observed instances of each type. We then eliminate infeasible class hierarchies within each family based on structural code features that can be extracted from the binary. Our structural analysis is similar to those utilized in previous work, e.g., [PCvdV+17].

- **Behavioral Analysis:** A static analysis that extracts sequences of operations performed on objects of each type. These sequences are used to train a statistical language model that captures the behaviors of each type. Once a model has been trained for each type, we use a distance metric based on DKL to define the likelihood of two types being related by inheritance. We use an arborescence algorithm to infer the most likely class hierarchy that can be constructed using the pairwise distance.

**Existing Techniques** Despite the importance of this problem to reverse engineering of modern software (as highlighted nicely in [PCvdV+17]), there has been relatively little work on the reconstruction of class hierarchies. Even though interest in the analysis of executables has increased in recent years [RLT+10, RBLT07, RBL06, RB08, BR07, BR10, GDN+15, DY14, SR14], most techniques have focused on the recovery of debug symbols and control-flow.

Existing techniques, e.g., [SY07, FDCT11, PCvdV+17], rely heavily on the existence of debug information or optional structural cues, e.g., calls to parent constructors or runtime type information (RTTI) records. This is unfortunate, as in many real-world
binaries this information is removed, either during stripping or as an optimization (e.g.,
inline calls to parent constructor).

Furthermore, existing techniques [SY07, FDCT11, PCvdV+17] do not reconstruct
an actual class hierarchy. Instead they compute a set of possible parents for each type,
effectively creating groups of related types. In contrast, we take an additional step by
lifting the local may-be-parent information to accurately reconstruct an actual, global,
class hierarchy.

For many applications, the precision of the extracted hierarchy is crucial. For example,
when the reconstructed hierarchy is used to harden binaries by applying control-flow
integrity (CFI) [ABEL05], as done in [PCvdV+17], imprecision in the hierarchy leads to
false positives, detrimental to security. For a simple example of a scenario in which type
grouping does not suffice, consider the C++ program shown in Fig. 3.1. The program
uses data from internal and external sources, each containing several types of data
(see Fig. 3.2 for a graphical depiction of the program’s class hierarchy). Internal sources
are trusted and secure, requiring no validation, while external sources are not. Therefore,
any data provided by an external source should be checked and validated before it
can be used. Clearly, an external data source should not be passed as a parameter to
readInternal(). Note that in this program, all the data sources reside in the same
hierarchy. Therefore, existing techniques would put them all in the same type group.
Applying CFI based on this result would not guarantee program security it as it would
allow reading unvalidated data from external sources. Indeed, an actual hierarchy of
the types is required to determine that external data sources cannot be a subtype of an
internal data source. As our results in Section 3.5 demonstrate, our technique generates
a more precise hierarchy than any existing technique.

Main Contributions This chapter makes the following contributions:

- A novel framework for statistical static reconstruction of a class hierarchy from
  stripped binaries. Our approach augments the technique of Chapter 2 with a
  notion of pairwise asymmetric distance between types based on the Kullback-Leibler
divergence.
A formalization of the problem of lifting the pairwise distances into the most likely class hierarchy as a natural problem in graph theory, finding a *minimum-weight maximal forest* in a directed weighted graph.

An implementation of our approach in a tool named ROCK, which we use to automatically reconstruct the class hierarchies of several real-world C++ binaries. We measure the quality of our algorithm using an applicative distance between the constructed hierarchy and a known ground-truth computed from source code. We usually manage to reconstruct the original class hierarchy with much higher accuracy than is possible using traditional non-statistical techniques.

### 3.2 Overview

We provide an informal overview of our approach using a simple illustrative example. A formal model and experimental results over realistic examples are provided later.

**Motivating Example** The code fragment shown in Fig. 3.4 defines the class hierarchy depicted in Fig. 3.3: (i) Stream, a base class containing a single method used to send values, (ii) ConfirmableStream, an extension of Stream that requires confirmation of each send, and (iii) FlushableStream, another extension of Stream that requires explicit flushing and closing of the stream. The useX methods demonstrate the proper way to use class X.

Compiling and stripping our motivating example produces a binary that does not contain any *meaningful names* and is missing the *inheritance relations* between the different classes. Fig. 3.5 depicts, in code, the generated stripped binary. Here, Class1, Class2, and Class3 correspond to Stream, ConfirmableStream, and FlushableStream, respectively. Note that the function names in Fig. 3.5 are generalized stripped names derived solely from the order of the functions in the code (such that f0 is the 1st function.
class Stream { // socket interface
    virtual void send(int n);
};

class ConfirmableStream : public Stream{
    virtual void confirm();
};

class FlushableStream : public Stream{
    virtual void flush();
    virtual void close();
};

int useStream(Stream* stream){
    stream->send(0);
    stream->send(1);
    stream->send(2);
}

int useConfirmableStream(ConfirmableStream* stream){
    stream->send(0);
    stream->confirm();
    stream->send(1);
    stream->confirm();
    stream->send(2);
    stream->confirm();
}

int useFlushableStream(FlushableStream* stream){
    stream->send(0);
    stream->send(1);
    stream->send(2);
    stream->flush();
    stream->close();
}

Figure 3.4: Class definitions and example code.


```cpp
class Class1{
    virtual void f0(int n);
};
class Class2{
    virtual void f0(int n);
    virtual void f1();
};
class Class3{
    virtual void f0(int n);
    virtual void f1();
    virtual void f2();
};
```

```cpp
int useClass1(Class1* s){
    s->f0(0);
    s->f0(1);
    s->f0(2);
}
int useClass2(Class2* s){
    s->f0(0);
    s->f1();
    s->f0(1);
    s->f1();
    s->f0(2);
    s->f1();
}
int useClass3(Class3* s){
    s->f0(0);
    s->f0(1);
    s->f0(2);
    s->f1();
    s->f2();
}
```

Figure 3.5: Class definitions as observed in a stripped binary.

of a class, f1 is the 2nd, etc.). There is no guarantee that f1 of classes Class2 and Class3 will point to the same implementation despite sharing the same name.

Our goal is to reconstruct the class hierarchy of the original source code (see Fig. 3.3), using only information from the stripped binary. We achieve our goal using a combination of structural analysis and behavioral analysis.

### 3.2.1 Step I: Structural Analysis

Structural analysis uses the content of virtual function tables and the relationships between tables to infer constraints on the class hierarchy. For example:

- **ConfirmableStream** and **FlushableStream** do not override **Stream**’s implementation of `send()`. Hence, in the stripped binary, all three classes use the same implementation for `f0()`. This information is encoded in the binary by having the entries corresponding to `f0` in all 3 virtual function tables (one for each class) point to the same implementation. The fact that entries of different virtual tables point to the same implementation is a structural hint that these classes are part of the same hierarchy.

- **Class1** contains a single virtual function, **Class2** contains two virtual functions, and **Class3** contains three virtual functions. In C++, a derived class cannot have
fewer virtual functions than its parent class. As a result, Class1 cannot be a derived class. Similarly, the only possible parent of Class2 is Class1. On the other hand, both Class1 and Class2 may be the parent of Class3.

In our motivating example, the use of structural information was sufficient to determine part of the hierarchy, but we are still left with the task of determining the correct parent of Class3. If we had any debug symbols containing the original function names, we could determine that f1 of Class2 and f1 of Class3 are unrelated (based on the functions’ names), making Class1 the only possible parent of Class3. However, that information is not available, and therefore both Class1 and Class2 are possible parents of Class3, and both of the hierarchies demonstrated in Fig. 3.6 are viable options.

Our structural analysis, described in Section 3.4, uses multiple structural hints in addition to the ones described above, e.g., calls to constructors or the destructor of the parent. We note that the first step of our approach can be thought of as the inter-procedural equivalent of the technique used in [PCvdV+17].

### 3.2.2 Step II: Behavioral Analysis

We reconstruct the class hierarchy accurately by static extraction of behavioral information that captures the way instances (i.e., objects) of classes are used: for each class, we capture behavioral information in the form of a statistical language model that assigns a probability to sequences of operations applied to instances of the class. We gather training data for the statistical language models using a static analysis that extracts object tracelets—an approximation of the sequences of operations applied to instances of each class.\(^2\)

For example, recall that the useX methods shown in Fig. 3.5 demonstrate the proper way to use each class X. Fig. 3.7 depicts the usage sequences extracted from the code of these methods.

A trained SLM (of depth 2) for Class3 is depicted in Fig. 3.8. The values along the edges show the probability assigned by the model to each of the possible outcomes of each state\(^3\). By multiplying the probabilities along a path, we can compute the

\(^2\)Object tracelets are formally defined in Section 2.3.

\(^3\)The escape outcome is used as a backoff mechanism. See Section 2.4.
Figure 3.7: Usage sequences for instances of the different classes (extracted from the stripped binary).

Figure 3.8: Trained statistical language model of Class3.

probability of the sequence correlating to that path. As can be seen from the figure, given a prefix of f0;f1, the most likely suffix is an invocation of f2.

From Fig. 3.7, we can observe that the tracelet matching Class3 seems more similar to the tracelet matching an instance of Class1 than to that of Class2. This is because the tracelet of Class1 is contained in the tracelet of Class3. To formalize this intuition and lift it from individual sequences to sets of behaviors, we measure a distance between the statistical language models trained for the different classes.

We measure the distance between statistical language models using the Kullback-Leibler divergence (DKL) [KL51]. DKL measures the similarity between two probability distributions and can be applied to measure distances between statistical language models. DKL, and other metrics based on it, have been successfully used to measure similarity of music [Sch14], evolution of languages [EYFT98], similarity of protein sequences [CS07], DNA sequences [YL02], and more. To the best of our knowledge, we are the first to use it to determine behavioral similarity in the context of programming languages.

DKL is computed by combining the measured probabilities for each of the sequences (see Section 3.3.2). For example, computing the DKL distance between different pairs of models of the classes in our motivating example, we get that the distance between Class3 and Class1 is 0.07, while the distance between Class3 and Class2 is 0.21, making Class1 the more likely parent. Therefore, we can determine that Fig. 3.6a, which corresponds to the original hierarchy tree in Fig. 3.3, is indeed the most likely class hierarchy structure.
3.3 Behavioral Hierarchy Reconstruction

Building on object tracelets (as established in Section 2.3) and SLMs (as described in Section 2.4), we next formalize the problem of reconstructing class hierarchies from stripped binary code and present an approach to solve it using a static probabilistic technique.

3.3.1 Problem Definition

Let $src(P)$ be the source code of a program $P$, assumed to be written in some object-oriented programming language, such as C++. The programming language is expected to include an inheritance mechanism for defining classes, allowing every class to inherit code and fields from other classes.

**Class Hierarchy as a Labeled Forest** A class $A$ can be defined either as the root of a class hierarchy or as a subclass of an already defined class. We refer to a class $B$ derived from class $A$ as a child of $A$, and refer to $A$ as the super-class or parent of $B$. The child relation between classes in $src(P)$ creates a node-labeled directed forest (NLD-forest), labeled using the names of the classes, which we refer to as the source class hierarchy of $P$.

During compilation, the compiler translates source classes to binary types, each represented by its virtual functions table. The compilation output is the binary code $bin(P)$. We assume that every binary type $vt$ has a source class counterpart, denoted by $vtToSrc(vt)$. In practice, this assumption does not directly hold, as the compiler may generate synthetic classes that do not have a source class counterpart. However, for the purpose of evaluation, we identify and remove synthetic classes to enable comparison of the hierarchies we compute with the source class hierarchy.

We refer to the hierarchy resulting from the compilation process as the induced binary type hierarchy, and denote it by $H_P$. We say that a binary type $vt_c$ is derived from a binary type $vt_p$ if $vtToSrc(vt_p)$ is an ancestor of $vtToSrc(vt_c)$ in the source class hierarchy.

Given a binary code $bin(P)$, the goal of this work is to find the binary type hierarchy of $P$ over the binary types in $bin(P)$.

**Optimized Class Hierarchies** We assume the compiler optimizes and strips the program, such that $bin(P)$ does not contain any source information or debug symbols.

One optimization applied by the compiler is inlining. Virtual classes, which cannot be instantiated, might be completely inlined in their child classes. Inlining might result in entire classes being absent from the binary. Furthermore, source inheritance trees might split into several binary inheritance trees if the root of the tree was inlined.

Interestingly, while we did not originally expect to overcome the latter challenge, our algorithm sometimes splices together such binary inheritance trees, based on their
behavioral and structural similarities. This allows us to infer relations between classes split into different hierarchies by the compiler, as demonstrated in Section 3.5.4.

We amend the parent-child relation such that \( vt_c \) is a child of \( vt_p \) if it is an immediate descendant of \( vt_p \) in the post-optimization induced binary type hierarchy, and we refer to \( vt_p \) as \( vt_c \)’s parent.

### 3.3.2 Finding the Most Likely Class Hierarchy

Without knowing the source class hierarchy of \( src(P) \), we can rarely find enough evidence in \( bin(P) \) to determine the true induced binary type hierarchy \( H_P \) of \( bin(P) \). Thus, a more reasonable approach is to treat the class hierarchy reconstruction problem as an optimization problem. Given some distance metric between NLD-forests \( forestDist(H_1, H_2) \), we say that a hierarchy \( H_1 \) is more precise than \( H_2 \) if, assuming that we do have \( H_P \), it turns out that \( forestDist(H_1, H_P) < forestDist(H_2, H_P) \). Obviously, we cannot look directly for a hierarchy \( H \) that minimizes the distance \( forestDist(H, H_P) \), as this would be \( H_P \). Instead, we rely on a measure \( w(\cdot) \) on NLD-forests, and we treat the NLD-forest that minimizes it as the most likely binary type hierarchy.

In this paper, we suggest using the distance between behaviors of types as the basis for a measure over NLD-forests. Because inheritance in object-oriented programs serves as a mechanism to implement subtyping, we expect that derived types will behave similarly to their parents.

*Hypothesis 3.3.1.* An NLD-forest that minimizes the distance between behaviors of a parent type and a child type is more likely than a NLD-forest with greater distances.

Expanding on the behavioral modeling described in Section 2.4, we now need a metric \( behaviorDist(vt_1, vt_2) \), which measures the distance between these behaviors, and a means of lifting this metric to the construction of a binary type hierarchy. We address these challenges next.

### Measuring Distance Between Types using Kullback-Leibler Divergence

Following Section 2.4, we model the behavior of binary types using statistical language models. A novel aspect of our work is that we lift the computed SLMs to distances between types. Specifically, we define \( behaviorDist(vt_1, vt_2) \) as the *Kullback-Leibler divergence* \( (D_{KL}) \) between the SLMs of \( vt_1 \) and \( vt_2 \).

The \( D_{KL} \) between two distributions \( A \) and \( B \), represented by SLMs defined over the same alphabet \( \Sigma \), denoted by \( D_{KL}(A \parallel B) \), is the relative entropy between the distributions. \( D_{KL}(A \parallel B) \) is measured over a set of words, \( W \), as:

\[
D_{KL}(A \parallel B) = \sum_{w \in W} \left( Pr(A_w) \ln \left( \frac{Pr(A_w)}{Pr(B_w)} \right) \right).
\]
\(D_{KL}\) is a standard measure used to evaluate model similarity. One can think of \(D_{KL}\) as the cost of encoding messages using an optimal code for a probability distribution \(P(B_w)\) of messages \(w \in W\), but the messages actually arrive with probabilities \(P(A_w)\). In this case, encoding each message requires, on average, an additional \(D_{KL}(A \parallel B)\) nats (where 1 nat, the natural measure for entropy, correlates to 1 bit) compared to the optimal encoding.

In our setting, the distance between behaviors of types is measured as a weighted sum of the differences between probabilities assigned to each behavior by the SLMs, such that popular behaviors weigh more than rare ones.

Due to the the nature of type inheritance, the set of tracelets for a parent type is usually contained in the set of a child type, resulting in high overlap. SLMs trained on similar sets of tracelets will output similar probabilities. Therefore, we expect the distance between related types to be smaller than between unrelated types.

**Remark.** Our approach is parametric in the criterion used to measure distances between the behaviors of types. Our algorithm only requires a ranking over the most probable child-parent relations. It is not even required that the criterion be a mathematical metric: the distances between types need not satisfy the triangle inequality, for example. Furthermore, given that criterion is used only as a relative metric for ranking parents, we also do not require a pre-set threshold to determine child-parent relations.

**From Pairwise Similarity to the Most Likely Hierarchy**

We translate our problem of finding the most likely hierarchy to the problem of finding a *minimum-weight spanning arborescence* in a directed graph.

**The Minimum-Weight Spanning Arborescence Problem** Given a directed graph \(G\), the spanning arborescence problem is the directed equivalent of the better-known spanning tree problem. A solution to the spanning arborescence problem is \(E'\), a subset of the edges set \(E\) of \(G\), such that:

1. the edges in \(E'\) form a tree;
2. there exists a root node \(r\) such that no edges in \(E'\) are directed towards it; and
3. all nodes in the graph are reachable from \(r\) via a directed path consisting only of edges in \(E'\).

Given a weight function \(w\) that assigns non-negative weights to edges in the graph, the minimum-weight spanning arborescence problem finds a spanning arborescence such that \(w(E') = \sum_{e \in E'} w(e)\) is minimal.

We note that the problem of finding a subset of edges with minimum weight can be trivially solved by returning the empty set. In our setting such a solution means that each type is a root residing in a separate hierarchy with no other parent or child types. This outcome is clearly undesired. This problem led us to the following design decision:
Heuristic 3.3.1. It is more plausible for a binary type to be a derived type than a root type.

Specifically, we require that any node for which there is a possible parent have a parent. This requirement resonates with the spanning arborescence problem, which requires all nodes in the graph to be part of the same arborescence, meaning that the arborescence should only have a single root.

Defining the Graph  We define a weighted directed graph \( G_P = (V, E, w) \) as follows:

- The nodes of \( G_P \) are the binary types of \( \text{bin}(P) \):
  \[ V = \text{VT}(P). \]

- An edge in the graph between \( vt_1 \) and \( vt_2 \) means that \( vt_2 \) is a possible parent of \( vt_1 \). We initialize the graph to contain an edge between all pairs of distinct types:
  \[ E = \{ vt_1 \rightarrow vt_2 \mid vt_1, vt_2 \in \text{VT}(P) \land vt_1 \neq vt_2 \}. \]

- The weight of an edge \( (vt_1 \rightarrow vt_2) \in E \) is defined as the distance between the behavioral models of \( vt_1 \) and \( vt_2 \):
  \[ w(vt_1 \rightarrow vt_2) = D_{KL}(\text{SLM}(vt_1) \parallel \text{SLM}(vt_2)). \]

A minimum-weight spanning arborescence for \( G_P \) constitutes a hierarchy tree that, according to Hypothesis 3.3.1, is the most likely binary type hierarchy.

Simplifying the Graph  For many type pairs we can determine a priori that a parent-child relation cannot exist. In this step we simplify the graph by eliminating impossible pairs, and therefore impossible edges.

We use a structural analysis to partition the types in the program into different type families. Types in the same family might be related by inheritance. More importantly, types in different families are determined to be not related. This allows us to split the graph \( G_P \) into several sub-graphs, one for each type family \( TF \). We then find a minimum weighted spanning arborescence for each sub-graph \( G_{TF} \) separately. The static analysis can also be used to eliminate specific edges from the graph, thus eliminating infeasible hierarchies from consideration. The specifics of this step are described in Section 3.4.

Finding a Minimum-Weight Spanning Arborescence  To find a minimum-weight spanning arborescence we use the well-known Edmonds’ algorithm [Edm67]. Given a graph \( G = (V, E) \) and a weight function \( w \), Edmonds’ algorithm starts by greedily choosing the a minimum-weight edge from \( E \) for each node in \( V \). This initial solution
has a single incoming edge for each node in $V$, but might contain cycles. The algorithm iteratively eliminates cycles by identifying a minimum-weight entry-point into the cycle from the rest of the graph, adding the corresponding edge $(u, v) \in E$ and removing any other edge incoming to $v$. Once all cycles are eliminated, what remains is a minimum-weight spanning arborescence.

Given a set of sub-graphs, one for each type family, we apply the algorithm to each $G_{TF}$. The result is a set of minimum-weight spanning arborescences, one for each type family, which constitutes a NLD-Forest that is used as the most likely binary type hierarchy.

Remark. The spanning arborescence algorithm may fail to find a single arborescence. While in the general setting this will be considered an error, we use this result as a hint that a type family contains more than a single hierarchy and needs to be further split to smaller families. We allow the algorithm to finish its execution and regard any types not part of the returned arborescence as part of a separate hierarchy.

For each of our benchmarks, it takes only a few minutes to construct the weighted graph and find an arborescence.

Handling Multiple Arborescences It is possible for the algorithm to find several arborescences with the same minimal weight. If that happens, we iteratively reduce the number of arborescences using a majority-vote heuristic. Assume the algorithm returns 3 arborescences such that 2 of them assign type $B$ to be the parent of type $A$ and the third assigns $C$ as the parent of $A$. Using our heuristic, we will prefer hierarchies in which $A$ is a child of $B$. Thus, we will eliminate the third arborescence. We note that the heuristic is not guaranteed to leave only a single arborescence. In such cases, several hierarchies will be returned to the user, and we will have to choose between them. In our evaluation, whenever we encountered this situation, we report the worst-case results: those obtained by choosing the least precise hierarchy.

3.4 Pruning Infeasible Class Hierarchies via Structural Analysis

In the following, we describe the preprocessing phase of our analysis, in which we construct the possible parent relation, possibleParent$_P \subseteq VT(P) \times VT(P)$, over the binary types of $bin(P)$. Recall that $(vt_1, vt_2) \notin$ possibleParent$_P$ indicates that $vt_2$ is deemed not to be the parent of $vt_1$ in the type hierarchy of $bin(P)$. This information focuses the behavioral analysis (see Section 3.3) towards computing the induced binary type hierarchy of $P$ by eliminating infeasible child-parent relations, and thus infeasible hierarchies.

The elimination process is based on information gained from a structural analysis of the binary code. As indicated by its name, the structural analysis looks for structural
artifacts that come from the compilation process. These features are mostly compiler independent. The structural analysis works in two phases:

**Phase I: Clustering of Types Families** In this phase the set of binary types $VT(P)$ is partitioned into different type families. Every family is composed of the types coming from one or more binary inheritance trees. Thus, they can be seen as a coarse clustering of related types. This ensures that the behavioral analysis need not consider child-parent relations between types from different families.

**Phase II: Elimination of Impossible Parents** In this more fine-grained phase, single child-parent relations are deemed impossible, placing further restrictions on the possible hierarchies that the behavioral analysis can produce and directing it further towards the induced type hierarchy.

Interestingly, in certain simple benchmarks, we found that the structural analysis is precise enough to determine the true parent of some derived classes, and sometimes was even sufficient for reconstructing the true induced hierarchy.

### 3.4.1 Clustering Binary Types into Families

When a class $A$ is derived from a superclass $B$, it inherits the vtable of $B$ in the following sense: unless $A$ redefines a function $f$ of $B$, the binary type $vt_a$ will contain a pointer to the same implementation of $f$ in the same position in the vtable as the binary type $vt_b$. Thus, pointers to shared virtual functions can be seen as a DNA fingerprint indicating that two classes come from the same inheritance tree. Therefore, we create an undirected graph $G_{DNA}^P$ over $VT(P)$ by placing an (undirected) edge between $vt_1$ and $vt_2$ if the intersection between their vtables is not empty.

We partition $VT(P)$ into families of binary types, by placing two types in the same family if and only if they belong to the same connected component in $G_{DNA}^P$.

We note that when a subclass redefines all the functions it inherits from its parent, the aforementioned heuristic will split the nodes coming from one binary inheritance tree into different families. However, in our benchmarks, originating from real-world C++ programs, we found no case where a child class redefined all functions inherited from a non-virtual superclass (recall that a virtual source class would not appear in $VT(P)$ since it would not be instantiated as a binary type). We believe that this is because inheritance from a non-virtual superclass is often for the purpose of reusing implementations it provided.

In some scenarios, compiler-generated functions and the compiler optimization passes might result in a pointer to the same function appearing in the vtables of two unrelated types. This unexpected behavior is an example of a problem faced by structural-only analysis, where noisy information due to arbitrary artifacts generated by the compiler must be dealt with. Our behavioral analysis partially overcomes this problem.
We initialize the possibleParent_P relation to hold all pairs of types that belong to the same family.

### 3.4.2 Eliminating Impossible Parents

The second phase of the structural analysis performs a fine-grained inspection of the code of every pair of binary types \(vt_1, vt_2 \in VT(P)\) that belong to the same family \(F\). The goal is to look for incriminating evidence that would allow us to remove \((vt_1, vt_2)\) or \((vt_2, vt_1)\) or both from the possibleParent_P relation. We remove \((vt_1, vt_2)\) from the relation possibleParent_P if at least one of the following structural features is identified:

1. \(vt_1\) contains more functions than \(vt_2\). In this case, \(vt_2\) cannot be a child of \(vt_1\) because a child class may only add functions to the vtable of its parent or replace existing ones.
2. \(vt_1\) contains a pointer to a virtual function (which does not have an implementation) at position \(i\) in its vtable, but the function pointed to by position \(i\) of \(vt_2\)'s vtable is not virtual. In this case, \(vt_1\) cannot be a child of \(vt_2\) because then it would have inherited the implementation of the \(i\)th function from \(vt_2\), or defined its own.
3. \(vt_1\)'s constructor calls the constructor of some other type \(vt_3\). This indicates a call to a superclass constructor and should be distinguished from an initialization of an object member. In this case it is determined that \(vt_3\) is the parent of \(vt_1\). If \(vt_1\) and \(vt_3\) are not part of the same type family, their respective families will be joined. The same logic applies to destructors as well.

### 3.4.3 Handling Multiple Inheritance

Under most ABIs, multiple inheritance is easy to detect from object layout. For example, in the MSVC [Vis97] ABI, a type that has multiple inheritance will be instantiated in memory as a concatenation of objects matching each of its parents. Therefore, if a type inherits from \(X\) different parents, we will observe assignments of \(X\) different vtable pointers in the initialization of its instances. These observations determine the number of parents a type should have. Given that we observe \(X\) assignments, we will choose the \(X\) most likely parents as the type’s parents. Similar approaches and other structural hints are also applicable to other ABIs we examined. Therefore, in this context, handling multiple inheritance is orthogonal to the more fundamental problem of ranking the most likely parents, and we thus focus our evaluation on the latter.

### 3.5 Experimental Evaluation

We implemented a prototype of our approach in a tool called Rock\(^4\). This section describes the evaluation of ROCK on a number of real-world stripped binaries.

\(^4\)Source code available at https://bitbucket.org/omerktz/rock
3.5.1 Benchmarks

We evaluated ROCK over 19 stripped binaries built from open-source projects. We used 19 out of the 20 benchmarks previously used by Section 2.6. The 20th benchmark contained only type families consisting of a single type (which either has no parent type or the parent type was optimized away); thus, there is no hierarchy to reconstruct and evaluate. While these benchmarks are relatively small, they allowed us to manually verify their results. We also successfully analyzed the binary of Skype [Cor03] (of size 21.6 Mb), but we do not report these results as we had no ground-truth to compare against. Furthermore, using these benchmarks also allowed us to reuse the framework and foundations already laid out by Chapter 2.

All benchmarks were built from source code as 32-bit binaries on a Windows machine using Microsoft’s Visual Studio compiler [Vis97], optimized and stripped, leaving no debug symbols in the binary.

We filtered out compiler generated classes and hierarchies containing a single type (which were not filtered in Section 2.6). We did so because the hierarchies for this classes can be trivially reconstructed. Additionally, including these cases would only improve our results, thus the results we present here serve as a lower limit.

3.5.2 Ground Truth

To obtain a ground truth for comparison, we relied on the source code, debug symbols and RTTI records generated during compilation of the benchmarks. We used RTTI records mainly to determine the sequence of ancestors for each type (as it exists in the binary), and we constructed the hierarchy from that sequence. When necessary, we correlated the binary and the source code using debug symbols to obtain missing information and complete the hierarchy. Alternatively, source types can replace RTTI records as the basis of the ground truth; doing so would not affect the numerical metrics reported in Section 3.5.4.

3.5.3 Application Distance

Consider the example from Section 3.1 and the application presented in Chapter 2: A reverse engineer trying to resolve the control flow graph of a program encounters a virtual call and needs to determine its possible targets. As the target of a virtual call is determined by the runtime type of the object used in the call, the reverse engineer can use the tool from Chapter 2 to deduce the most likely type of that object and infer from that the relevant target of the call. But this gives the reverse engineer only a partial answer. Given that an object \( o \) is an instance of type \( t \), it can also be an instance of any type \( t' \) derived from \( t \). Therefore, to identify all possible targets, the reverse engineer also requires some knowledge of the type hierarchy. In practice, the exact hierarchy is not important but only the set of types derived from each type.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>size (Kb)</th>
<th>num of types</th>
<th>Without SLMs</th>
<th>With SLMs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Missing</td>
<td>Added</td>
</tr>
<tr>
<td>AntispyComplete</td>
<td>247</td>
<td>3</td>
<td>0.0</td>
<td>0.33</td>
</tr>
<tr>
<td>bafrp</td>
<td>529</td>
<td>23</td>
<td>0.3</td>
<td>0.0</td>
</tr>
<tr>
<td>cpcheck</td>
<td>97</td>
<td>6</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>MidiLib</td>
<td>400</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>patl</td>
<td>36.5</td>
<td>4</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>pop3</td>
<td>24</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>smtp</td>
<td>26</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>tinyxml</td>
<td>60</td>
<td>9</td>
<td>0.89</td>
<td>0.0</td>
</tr>
<tr>
<td>tinyxmlSTL</td>
<td>88</td>
<td>15</td>
<td>0.6</td>
<td>0.27</td>
</tr>
<tr>
<td>yafe</td>
<td>68</td>
<td>15</td>
<td>0.0</td>
<td>0.2</td>
</tr>
<tr>
<td>Analyzer</td>
<td>419</td>
<td>24</td>
<td>0.21</td>
<td>6.79</td>
</tr>
<tr>
<td>CGridListCtrlEx</td>
<td>151</td>
<td>28</td>
<td>0.0</td>
<td>0.46</td>
</tr>
<tr>
<td>echoparams</td>
<td>58</td>
<td>4</td>
<td>0.0</td>
<td>2.25</td>
</tr>
<tr>
<td>gperf</td>
<td>84</td>
<td>10</td>
<td>0.0</td>
<td>3.8</td>
</tr>
<tr>
<td>libctemplate</td>
<td>1233</td>
<td>36</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>ShowTraf</td>
<td>137</td>
<td>25</td>
<td>0.04</td>
<td>0.4</td>
</tr>
<tr>
<td>Smoothing</td>
<td>453</td>
<td>31</td>
<td>0.19</td>
<td>7.9</td>
</tr>
<tr>
<td>td_unittest</td>
<td>101</td>
<td>2</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>tinyserver</td>
<td>46</td>
<td>4</td>
<td>0.0</td>
<td>2.25</td>
</tr>
</tbody>
</table>

Table 3.1: Application Distance from $H_P$. Structurally resolvable benchmarks are above the line and below it are unresolvable benchmarks.

We define our application distance to address this scenario so that it represents the usefulness of our technique in a real world setting. For each type $t$ we compute the set of types derived from it according to the ground-truth, denoted as $\text{successors}_{GT}(t)$. This is the optimal answer we could provide the reverse engineer given the question, Which types are derived from $t$? We compute a similar set of derived types using the type hierarchy $h$ constructed by our technique, $\text{successors}_h(t)$. To evaluate the constructed hierarchy, we measure for each type $t$ how many types from $\text{successors}_{GT}(t)$ are missing from $\text{successors}_h(t)$ and vice versa (denoted as added types). The measures of missing and added types are essentially the precision and recall, respectively, of our answer.

For every missing type, a possible target is “lost” to the reverse engineer, and for every added type a redundant target must be analyzed. Therefore, the missing types affect the validity of the results while the added types affect the payload (which is usually already very large).

### 3.5.4 Experimental Results

We ran ROCK on all 19 benchmarks.

We soon learned that our statistical approach is not always necessary. In some of our smaller benchmarks, with a relatively small number of types, the structural analysis is sufficient to eliminate all but 1 possible hierarchy. Note that these benchmarks are not necessarily free from error originating from the partition to type families. However,
since our current implementation does not attempt to repartition based on usage, our technique will not be beneficial in these cases.

Out of our 19 benchmarks, we found only 9 for which there was more than a single possible hierarchy after the structural analysis. We believe that this number will increase significantly as we scale up our technique to deal with increasingly larger benchmarks. To give a complete picture, the statistics of the 10 structurally solvable benchmarks are also included in Table 3.1. For the remainder of this section we will focus on the 9 benchmarks that were not structurally resolved.

Table 3.1 presents the results of our evaluation using the application distance from Section 3.5.3. We measure the application distance in two settings: (i) relying solely on the structural analysis, and (ii) using SLMs and DKL in addition to the structural analysis. For the case without SLMs, since we have no way to prioritize possible parents, we have to consider a type as a successor of each of its possible parents. A type might thus be counted as a derived type of several parent types.

For each benchmark we report (i) the number of types in the benchmark, (ii) the average number of missing types across all types, and (iii) the average number of added types across all types (for both settings).

The results show that most types in each benchmark do not suffer any missing or added types. Only a small number of types in each benchmark actually had an error in their results. This can be observed from the very small values for the average amounts of missing/added types.

To better understand the meaning of these results, consider the benchmark tinyxml, which had the highest average of missing types. This benchmark consists of 9 types. The ground-truth hierarchy contained a single root, such that all other 8 types are derived from it. The constructed hierarchy obtained from our technique had a single error originating from the structural analysis. The structural analysis found no evidence that the root was related to any of the other types and therefore placed it in a separate type family. As a result, the root type “lost” all of its children, 8 in total, while the other types had no missing types, resulting in an average of 0.89. We note, for 8 out of 9 types in the benchmark, there were no missing types, which we consider a good result in practice.

Similarly, consider the benchmark Analyzer. The constructed hierarchy contained a few deviations from the ground-truth, which manifested in a relatively high number of added types. However, for 19 out of 24 types in the benchmark there were no added types, and the additions occurred for only 5 types.

Additionally, a comparison of the results with and without using our technique shows a drastic decrease in the number of added types when using our model to build the most likely hierarchy. (This decrease comes at the cost of a slight increase in the number of missing types.) This means that a reverse engineer using our approach would have far fewer types to consider, thus drastically reducing time and cost.

We now examine some of the more interesting hierarchies in depth.
Consider the benchmarks CGridListCtrlEx and ShowTraf. In both benchmarks the structural analysis was sufficient to determine a parent for most types, leaving only 3 types with more than one possible parent. Additionally, in both benchmarks, our technique successfully ranked the correct parent, if one exists, as the most likely. However, our structural heuristics still resulted in some errors.

Fig. 3.9 shows partial type hierarchies, ground-truth and outputted by our technique, for CGridListCtrlEx. The errors were due to CAboutDlg and CGridListCtrlExDlg being incorrectly placed in the same type family, as were CGridEditorComboBoxEdit and CGridEditorText. The ground-truth tells us that each of these types should reside in a separate family, and therefore in its own hierarchy. From the source code we learn that these 2 pairs of types indeed each share a common parent. The types CGridEditorComboBoxEdit and CGridEditorText both inherit from a type called CEdit, and CAboutDlg and CGridListCtrlExDlg inherit from CDialog. Both these parent types are virtual types which cannot be instantiated and are therefore optimized out of the binary. As a result, they are not part of the ground-truth hierarchy as it exists in the binary. In essence, using our models and analysis, we have successfully deduced the relation between these pairs of types, even though no information about these relations exists in the binary. This result demonstrates the strength of our technique: the ability to learn relations between types even when those relations were eliminated during compilation. Similar results were obtained for ShowTraf.

The echoparams benchmark is another interesting case. This relatively small benchmark contains only 4 types, all part of the same type hierarchy. We expected this case to be easily solved using structural tools alone. However, we found that the structural analysis for this benchmark was incapable of eliminating any possible parents for any of the types since they are structurally equivalent. Thus, structural analysis alone resulted in 3 possible parents for each type, resulting in 64 equally likely possible hierarchies, whereas our technique correctly identified the parents, thus accurately reconstructing the correct type hierarchy. These results show that, in our context,
benchmark size is not a good indicator of the ease or difficulty of constructing the correct hierarchy. Relatively small benchmarks can still pose a significant challenge to purely structural techniques.

Another interesting case is Smoothing, where applying our technique resulted in a significant reduction in added types, from an average of 7.9 to 1.1, at the cost of only 0.04 missing types. This increase translates to only 1 additional missing type for only 1 of the 31 types in the benchmark.

Sources of Errors Our manual examination of the results identified 3 possible sources for the errors we encountered:

1. Due to optimizations, the compiler sometimes placed pointers to the same virtual function implementation in the virtual table of unrelated types, causing these types to be placed in the same family.

2. As another optimization, the compiler might omit entire classes and/or structural cues, causing related types to appear unrelated, thus causing the false splitting of a type family to subfamilies.

3. Our algorithm might choose a wrong parent for a type.

Generally, error 1 would result in added types and errors 2 and 3 would result in missing types. Note that purely structural techniques are just as susceptible to errors 1 and 2.

Other Metrics We experimented with other metrics for computing the pairwise similarity between types, such as JS-divergence (a symmetric extension of DKL) and JS-distance (a distance function based on JS-divergence). These other metrics performed poorly compared to the DKL metric we used. This is most likely because these are symmetric methods while our problem is inherently asymmetric.

Applying Control Flow Integrity Our results can be directly utilized for the strengthening of CFI policies by narrowing the set of legal targets based on the constructed hierarchy. We note that errors in the constructed hierarchy can lead to false negatives. However, we can trade off false negatives for false positives by assigning several parents to each type. Our algorithm supports this at the cost of increased computational complexity (while still polynomial).
Chapter 4

Neural Decompilation

Given the results presented in (Section 2.6.3 and Section 3.5.4), and the achievements of other works relating to statistical modeling of code (see Chapter 5), we conclude that statistical modeling, and especially when combined with analysis techniques, is effective at representing and summarizing code. We therefore decided to shift our focus from solving specific/localized RE problems and tackle a much harder challenge – Decompilation.

Given a low-level program in binary form (i.e. assembly) or in some intermediate representation, decompilation is the task of lifting that program to human-readable high-level source code.

Fig. 4.1 provides a high-level example of decompilation. The input to the decompilation task is a low-level code snippet, such as the one in Fig. 4.1(a). The goal of Decompilation is to generate a corresponding equivalent high-level code. The C code snippet of Fig. 4.1(b) is the desired output for Fig. 4.1(a).

Generating code that is either equivalent to the input or more readable is feasible. However, generating code that is both equivalent and more readable is challenging. As with the other problems we worked on, this task becomes even more challenging when dealing with stripped code. This is because the presence of debug information can
provide hints and clues to the nature of high-level code that should be generated.

There are many uses for decompilation. The most common is for security purposes. Searching for software vulnerabilities and analyzing malware both start with understanding the low-level code comprising the program. Currently this is done manually by reverse engineering the program. Reverse engineering is a slow and tedious process by which a specialist tries to understand what a program does and how it does it. Decompilation can greatly improve this process by translating the binary code to a more readable higher-level code.

Decompilation has many applications beyond security. For example, porting a program to a new hardware architecture or operating system is easier when source code is available and can be compiled to the new environment. Decompilation also opens the door to application of source-level analysis and optimization tools.

Decompilation is not usually considered a part of RE. However, we see it as a natural next step following RE. Decompilation provides a more readable and natural description of a given program. The output of decompilation can then be used by programmers, even those not experienced in RE, in order to understand a given piece of software, what it does and how. Thus, although decompilation has a scope much larger than RE, it still maintains the spirit and fulfills the goals of RE. Having a good decompiler can automatically resolve most (if not all) RE problems.

One can say that, when working on the previous problems we faced, we asked “is this summary (modeling) of the code sufficient to answer some question?”. We now wish to extend that question and ask “is this summary of the code sufficient to recreate the code?”. Note that in this context, decompilation is a very hard operating scenario as we try to recreate the code in an entirely dissimilar programming language (recreating the code in the same language, e.g. compressing the code, or in a similar programming language, e.g. [CLS18], is considered a simpler problem).

4.1 Introduction

The Problem Given a low-level program $P_l$ (in assembly or some intermediate representation) produced by a known compiler $CC$ from an unknown high-level source program, $P_h$, generate a high-level program $P'$ that is semantically equivalent to $P_h$ and compiles to $P_l$.

Neural Machine Translation Recent years have seen tremendous progress in Neural Machine Translation (NMT) [KB13, SVL14, CvMBB14]. NMT systems use neural networks to translate a text from one language to another, and are widely used on natural languages. Intuitively, one can think of NMT as encoding an input text on one side and decoding it to the output language on the other side (see Section 4.3 for more details). Recent work suggests that neural networks are also effective in summarizing source code [IKCZ16, APS16, MT14, ATGW15, HWLJ17, LMM17, ABBS15, ACR17].
Our Approach  We propose an automatic neural decompilation technique, using a two-phased approach. In the first phase, we generate a templated code snippet which is structurally equivalent to the input. The code template determines the computation structure without assignment of variables and numerical constants. Then, in the second phase, we fill the template with values to get the final decompiled program. The second phase is described in Section 4.5.

Our approach can facilitate the creation of a decompiler from $L_{low}$ to $L_{high}$ from every pair of languages for which a compiler from $L_{high}$ to $L_{low}$ exists.

Our technique is still modest in its abilities, but presents a significant step forward towards trainable decompilers and in the application of NMT to the problem of decompilation. The first phase of our approach borrows techniques from natural language processing (NLP) and applies them to programming languages. We use an existing NMT system to translate a program in a lower-level language to a templated program in a higher-level language.

Since we are working on programming languages rather than natural languages, we can overcome some major pitfalls for traditional NMT systems, such as training data generation (Section 4.4.2) and verification of translation correctness (Section 4.4.4). We incorporate these insights to create a decompilation technique capable of self-improvement by identifying decompilation failures as they occur, and triggering further training as needed to overcome such failures.

By using NMT techniques as the core of our decompiler’s first phase, we avoid the manual work required in traditional decompilers. The core of our technique is language-agnostic requiring only minimal manual intervention (i.e., implementing a compiler interface).

One of the reasons that NMT works well in our setting is the fact that, compared to natural language, code has a more repetitive structure and a significantly smaller vocabulary. This enables training with significantly fewer examples than what is typically required for NLP [KK17] (See Section 4.6).

Existing Decompilers  Existing decompilers, such as Hex-Rays [IDA00] and Phoenix [SLWB13b], rely on pattern matching to identify the high-level control-flow structure in a program. These decompilers try to match segments of a program’s control-flow graph (CFG) to some patterns known to originate from certain control-flow structures (e.g. if-then-else or loops). This approach often fails when faced with non-trivial code, and uses goto statements to emulate the control-flow of the binary code. The resulting code is often low-level, and is really assembly transliterated into C (e.g. assigning variables to temporary values/registers, using gotos, and using low-level operations rather than high-level constructs provided by the language). While it is usually semantically equivalent to the original binary code, it is hard to read, and in some cases less efficient, prohibiting recompilation of the decompiled code.

There are goto-free decompilers, such as DREAM++ [YEGS15, YDGPS16], that
can decompile code without resorting to using \texttt{gotos} in the generated code. However, all existing decompilers, even \texttt{goto}-free ones, are based on hand-crafted rules designed by experts, making decompiler development slow and costly.

Even if a decompiler from a low-level language $L_{\text{low}}$ to a high-level language $L_{\text{high}}$ exists, given a new language $L'_{\text{high}}$, it is nontrivial to create a decompiler from $L_{\text{low}}$ to $L'_{\text{high}}$ based on the existing decompiler. There is no guarantee that any of the existing rules can be reused for the new decompiler.

Recently, Katz et al. [KRS18] suggested using neural networks, specifically RNNs, for decompilation. Their approach trains a model for translating binary code directly to C source code. However, they did not compensate for the differences between natural languages and programming languages, thus leading to poor results. For example, the code they generate often cannot be compiled or is not equivalent to the original source code. Their work, however, did highlight the viability of using Neural Machine Translation for decompilation, thus supporting the direction we are pursuing. Chapter 5 provides additional discussion of [KRS18].

The technique suggested by [KRS18] attempted to apply NMT to binary code as-is, i.e. without any additional steps and techniques to support the translation. a trainable decompiler, and specifically an NMT-based decompiler, to be useful in practice, we need to augment it with programming-languages knowledge (i.e. domain-knowledge). Using domain-knowledge we can make translations simpler and overcome many shortcomings of the NMT model. This insight is implemented in our approach as our canonicalization step (Section 4.4.3, for simplifying translations) and template filling (Section 4.5, for overcoming NMT shortcomings).

\textbf{Mission Statement} \quad Our goal is to decompile short snippets of low-level code to equivalent high-level snippets. We aim to handle multiple languages (e.g. x86 assembly and LLVM IR). We focus on code compiled using existing off-the-shelf compilers (e.g. gcc [gcc87] and clang [cla07]), with compiler optimizations enabled, for the purpose of finding bugs and vulnerabilities in benign software. More specifically, we do not attempt to handle hand-crafted assembly as is often found in malware.

Many previous works aimed to use decompilation as a mean of understanding the low-level code, and thus focused mostly on code readability. In addition to readability, we place a great emphasis on generating code that is \textit{correct} (i.e., can be compiled without further modifications) and \textit{equivalent to the given input}.

We wish to further emphasize that the goal of our work is not to outperform existing decompilers (e.g., Hex-Rays [IDA00]). Many years of development have been invested in such decompilers, resulting in mature and well-tested (though not yet perfect) tools. Rather, we wish to shed light on trainable decompilation, and NMT-based decompilation in particular, as a promising alternative approach to traditional decompilation. This new approach holds the advantage over existing decompilers not in its current results, but in its potential to handle new languages, features, compilers, and architectures.
with minimal manual intervention. We believe this ability will play a vital role as decompilation will become more widely used for finding vulnerabilities.

**Main Contributions** This chapter makes the following contributions:

- A significant step towards neural decompilation by combining ideas from neural machine translation (NMT) and program analysis. Our work brings this promising approach to decompilation closer to being practically useful and viable.
- A decompilation framework that automatically generates training data and checks the correctness of translation using a verifier.
- A decompilation technique that is applicable to many pairs of source and target languages and is mostly independent of the actual low-level source and high-level target languages used.
- An implementation of our technique in a framework called TRA Fix (short for TRAnslate and FIX) that, given a compiler from $L_{high}$ to $L_{low}$ automatically learns a decompiler from $L_{low}$ to $L_{high}$.
- An instantiation of our framework for decompilation of C source code from LLVM intermediate representation (IR) [LLVM02] and x86 assembly. We used these instances to evaluate our technique on decompilation of small simple code snippets.
- An evaluation showing that our framework decompiles statements in both LLVM IR and x86 assembly back to C source code with high success rates. The evaluation demonstrates the framework’s ability to successfully self-advance as needed.

### 4.2 Overview

Using a motivating example, we provide an informal overview of our approach.

#### 4.2.1 Motivating Example

Consider the x86 assembly example of Fig. 4.1(a). Fig. 4.2 shows the major steps we take for decompiling that example.
The first step in decompiling a given input is applying *canonicalization*. In this example, for the sake of simplicity, we limited canonicalization to only splitting numbers to digits (Section 4.4.3), thus replacing 14 with 1 4, resulting in the code in block (1) of Fig. 4.2. This code is provided to the decompiler for translation.

The output of our decompiler’s NMT model is a canonicalized version of C, as seen in block (2). In this example, output canonicalization consists of splitting numbers to digits, same as was applied to the input, and printing the code in post-order (Section 4.4.3), i.e. each operator appears after its operands. We apply un-canonicalization to the output, which converts it from post-order to in-order, resulting in the code in block (3). The output of un-canonicalization might contain decompilation errors, thus we treat it as a code template. Finally, by comparing the code in block (3) with the original input in Fig. 4.1, we fill the template (i.e. by determining the correct numeric values that should appear in the code, see Section 4.5), resulting in the code in block (4). The code in block (4) is then returned to the user as the final output.

For further details on the canonicalizations applied by the decompiler, see Section 4.4.3.

### 4.2.2 Decompilation Approach

Our approach to decompilation consists of two complementary phases: (1) Generating a code template that, when compiled, matches the computation structure of the input, and (2) Filling the template with values and constants that result in code equivalent to the input.

**First Phase: Obtaining a Template**

Fig. 4.3 provides a schematic representation of this phase.

At the heart of our decompiler is the NMT model. We surround the NMT model with a feedback loop that allows the system to determine success/failure rates and improve itself as needed by further training.
Denote the input language of our decompiler as $L_{low}$ and the output language as $L_{high}$, such that the grammar of both languages is known. Given a dataset of input statements in $L_{low}$ to decompile, and a compiler from $L_{high}$ to $L_{low}$, the decompiler can either start from scratch, with an empty model, or from a previously trained model. The decompiler translates each of the input statements to $L_{high}$. For each statement, the NMT model generates a few translations that it deemed to be most likely. The decompiler then evaluates the generated translation. It compiles each suggested translation from $L_{high}$ to $L_{low}$ using existing of-the-shelf compilers. The compiled translations are compared against the original input statement in $L_{low}$ and classified as successful translations or failed translations. At this phase, the translations are code templates, not yet actual code, thus the comparison focuses on matching the computation structure. A failed translation therefore does not match the structure of the input, and cannot produce code equivalent to the input in phase 2. We denote input statements for which there was no successful translation as failed inputs. Successful translations are passed to the second phase and made available to the user.

The existence of failed inputs triggers a retraining session. The training dataset and validation dataset (used to evaluate progress during training) are updated with additional samples, and the model resumes training using the new datasets. This feedback loop, between the failed inputs and the model’s training session, drives the decompiler to improve itself and keep learning as long as it has not reached its goal. These iterations will continue until a predetermined stop condition as been met, e.g. a significant enough portion of the input statements were decompiled successfully. It also allows us to focus training on aspects where the model is weaker, as determined by the failed inputs.

The well-defined structure of programming languages allows us to make predictable and reversible modifications to both the input and output of the NMT model. These modifications are referred to as canonicalization and un-canonicalization, and are aimed at simplifying the translation problem. These steps rely on domain specific knowledge and do not exist in traditional NMT systems for natural languages. Section 4.4.3 motivates and describes our canonicalization methods.

**Updating the Datasets** After each iteration we update the dataset used for training. Retraining without doing so would lead to over-fitting the model to the existing dataset, and will be ineffective at teaching the model to handle new inputs.

We update the dataset by adding new samples obtained from two sources:

- Failed translations – We compile failed translations from $L_{high}$ to $L_{low}$ and use them as additional training samples. Training on these samples serves to teach the model the correct inputs for these translations, thus reducing the chances that the model will generate these translations again in future iterations.

- Random samples – we generate a predetermined number of random code samples in $L_{high}$ and compile these samples to $L_{low}$.
The validation dataset is updated using only random samples. It is also shuffled and truncated to a constant size. The validation dataset is translated and evaluated many times during training. Thus truncating it prevents the validation overhead from increasing.

**Second Phase: Filling the Template**

The first phase of our approach produces a code template that can lead to code equivalent to the input. The goal of the second phase is to find the right values for instantiating actual code from the template. Note that the NMT model provides initial values. We need to verify that these values are correct and replace them with appropriate values if they are wrong.

This step is inspired by the common NLP practice of *delexicalization* [HTY14]. In NLP, using delexicalization, some words in a sentence would be replaced with placeholders (e.g. NAME1 instead of an actual name). After translation these placeholders would be replaced with values taken directly from the input.

Similarly, we use the input statement as the source for the values needed for filling our template. Unlike delexicalization, it is not always the case that we can take a value directly from the input. In many cases, and usually due to optimizations, we must apply some transformation to the values in the input in order to find the correct value to use.

In the example of Fig. 4.2, the code contains two numeric values which we need to “fill” – 14 and 2. For each of this values we need to either verify or replace it. The case of 14 is relatively simple as the NMT provided a correct initial value. We can determine that by comparing 14 in the output to 14 in the original input. For 2, however, copying the value 2 from the input did not provide the correct output. Compiling the output with the value 2 would result in the instruction `sall 1, %eax` rather than the desired `sall 2, %eax`. We thus replace 2 with a variable `N` and try to find the right value for `N`. To get the correct value, we need to apply a transformation to the input. Specifically, if the input value is `x`, the relevant transformation for this example is `N = 2x`, resulting in `N = 4` that, when recompiled, yields the desired output. Therefore we replace 2 with 4, resulting in the code in Fig. 4.2(4).

Section 4.5 further elaborates on this phase and provides additional possible transformations.

### 4.3 Neural Machine Translation

Current Neural Machine Translation (NMT) models follow a sequence-to-sequence paradigm introduced in [BCB14]. Conceptually, they have two components, an encoder and a decoder. The encoder encodes an arbitrary length sequence of tokens $x_1, ..., x_n$ over alphabet $A$ into a sequence of vectors, where each vector represents a given input token $x_i$ in the context in which it appears. The decoder then produces an arbitrary length
sequence of tokens $y_1, \ldots, y_m$ from alphabet B, conditioned on the encoded vectors. The sequence $y_1, \ldots, y_m$ is generated a token at a time, until generating an end-of-sequence token. When generating the $i$th token, the model considers the previously generated tokens as well as the encoded input sequence. An attention mechanism is used to choose which subset of the encoded vectors to consider at each generation step. The generation procedure is either greedy, choosing the best continuation symbol at each step, or uses beam-search to develop several candidates in parallel. The NMT system (including the encoder, decoder and attention mechanism) is trained over many input-output sequence pairs, where the goal of the training is to produce correct output sequences for each input sequence. The encoder and the decoder are implemented as recurrent neural networks (RNNs), and in particular as specific flavors of RNNs called LSTM [HS97] and GRU [BCB14] (we use LSTMs in this work). Refer to [Neu17] for further details on NMT systems.

4.4 Decomposition with Neural Machine Translation

We describe the algorithm of our decompilation framework using neural machine translation. First, in Section 4.4.1, we describe the algorithm at a high level. We then describe the realization of operations used in the algorithm such as canonicalization (Section 4.4.3), the evaluation of the resulting translation (Section 4.4.4), and the stopping condition (Section 4.4.5).

4.4.1 Decompiler Algorithm

Our framework implements the process depicted by Fig. 4.3. This process is also formally described in Algorithm 4.1. The algorithm uses a Dataset data structure which holds pairs $(x, y)$ of statements such that $x \in L_{\text{high}}$, $y \in L_{\text{low}}$, and $y = \text{compile}(x)$.

The framework takes two inputs: (1) a set of statements for decompilation, and (2) a compiler interface. The output is a set of successfully decompiled statements.

Decomposition starts with empty sets for training and validation and canonicalizes (Section 4.4.3) the input set. It then iteratively extends the training and validation sets (Section 4.4.2), trains a model on the new sets and attempts to translate the input set. Each translation is then recompiled and evaluated against the original input (Section 4.4.4 and Section 4.5). Successful translations are then put in a Success set, that will eventually be returned to the user. Failed translations are put in a Failed set that will be used to further extend the training set. The framework repeats these steps as long as the stopping condition was not reached (Section 4.4.5).

4.4.2 Generating Samples

To generate samples for our decompiler to train on, we generate random code samples from a subset of the C programming language. This is done by sampling the grammar
Algorithm 4.1 Decompile

**Input**
- inputset, collection of statements in $L_{low}$
- compile, api to compile $L_{high}$ to $L_{low}$

**Output**
- Dataset of successfully decompiled statements in $L_{high}$

**Data Types**
- Dataset: collection of pairs $(x, y)$, such that $x = compile(y)$

```plaintext
1: procedure Decompile
2:     inputset ← canonicalize(inputset)
3:     Train ← newDataset
4:     Validate ← newDataset
5:     model ← newModel
6:     Success ← newDataset
7:     Failures ← newDataset
8:     while (?) do
9:         Train ← Train ∪ Failures ∪ random_samples()
10:        Validate = Validate ∪ gen_random_samples()
11:        model.retrain(Train, Validate)
12:        decompiled ← model.translate(inputset)
13:        recompiled ← compile(decompiled)
14:        for each i in 0...inputset.size do
15:            pair ← (inputset[i], decompiled[i])
16:            if equiv(inputset[i], recompiled[i]) then
17:                if fill(inputset[i], recompiled[i]) then
18:                    Success ← Success ∪ [pair]
19:                else
20:                    Failures ← Failures ∪ [pair]
21:            end if
22:        else
23:            Failures ← Failures ∪ [pair]
24:        end if
25:     end for
26:     end while
27:     return uncanonicalize(Success)
28: end procedure
```
of the language. The samples are guaranteed to be syntactically and grammatically
correct. We then compile our code samples using the provided compiler. Doing so
results in a dataset of matching pairs of statement, one in C and the other in $L_H$, that
can be used by the model for training and validation.

We note that, alternatively, we could use code snippets from publicly available code
repositories as training samples, but these are less likely to cover uncommon coding
patterns.

4.4.3 Improving Translation Performance with Canonicalization

It is possible to improve the performance of NMT models without intervening in the
actual model. This can be achieved by manipulating the inputs in ways that simplify
the translation problem. In the context of our work, we refer to these domain-specific
manipulations as canonicalization.

Following are two forms of canonicalization used by our implementation:

Reducing Vocabulary Size

The vocabulary size of the samples provided to the model, either for training or
translating, directly affects the performance and efficiency of the model. In the case of
code, a large portion of the vocabulary is devoted to numerical constants and names
(such as variable names, method names, etc.).

Names and numbers are usually considered “uncommon” words, i.e. words that do
not appear frequently. Descriptive variable names, for example, are often used within a
single method but are not often reused in other methods. This results in a distinctive
vocabulary, consisting largely of uncommon words, and leading to a large vocabulary.

We observe that the actual variable names do not matter for preserving the semantics
of the code. Furthermore, these names are actually removed as part of the stripping
process. Therefore, we replace all names in our samples with generic names (e.g. $x_1$
for a variable). This allows for more reuse of names in the code, and therefore more
examples from which the model can learn how to treat such names. Restoring informative
descriptive names in source code is a known and orthogonal research problem for which
several solutions exist (e.g. [RVK15, HIT+18, ABBS15]).

Numbers cannot be handled in a similar way. Their values cannot be replaced with
generic values, since that would alter the semantic meaning of the code. Furthermore,
essentially every number used in the samples becomes a word in the vocabulary. Even
limiting the values of numbers to some range $[1...K]$ would still result in $K$ different
words.

To deal with the abundance of numbers we take inspiration from NMT for natural
languages. Whenever an NMT model for NL encounters an uncommon word, instead of
trying to directly translate that word, it falls back to a sub-word representation (i.e.
process the word as several symbols). Similarly, we split all numbers in our samples to
digits. We train the model to handle single digits and then fuse the digits in the output into numbers. Fig. 4.4 provides an example of this process on a simple input. Using this process, we reduce the portion of the vocabulary dedicated to numbers to only 10 symbols, one per digit. Note that this reduction comes at the expense of prolonging our input sentences.

**Alternative Method for Reducing Vocabulary Size** We observe that, in terms of usage and semantic meaning, all numbers are equivalent (other than very few specific numbers that hold special meaning, e.g. 0 and 1). Thus, as an alternative to splitting numbers to digits, we tried replacing all numbers with constants (e.g. \texttt{N1, N2, ...}). Similarly to variable names, the purpose of this replacement was to increase reuse of the relevant words while reducing the vocabulary. When applying these replacements to our input statements, we maintained a record of all applied replacements. After translation, we used this record to restore the original values to the output.

This approach worked well for *unoptimized* code, but failed on *optimized* code. In unoptimized code there is a direct correlation between constants in high-level and low-level code. That correlation allowed us to restore the values in the output. In optimized code, compiler optimizations and transformations break that correlation, thus making it impossible for us to restore the output based on the kept record.

**Order Transformation**

Most high-level programming languages write code *in-order*, i.e. an operator appears between its 2 operands. On the other hand, low-level programming languages, which are "closer" to the hardware, often use *post-order*, i.e. both operands appear before the operator.

The code in Fig. 4.5 demonstrates this difference. Fig. 4.5a shows a simple statement in \texttt{C} and Fig. 4.5b the \texttt{x86} assembly obtained by compiling it. The different colors represent the correlation between the different parts of the computation.
Intuitively, if one was charged with the task of translating a statement, it would be helpful if both input and output shared the same order. Having a shared order simplifies "planning" the output by localizing the dependencies to some area of the input rather than spreading them across the entire input.

Similarly, NMT models often perform better when the source and target languages follow similar word orders, even though the model reads the entire input before generating any output. We therefore modify the structure of the C input statements to post-order to create a better correlation with the output. Fig. 4.5c shows the code obtained by canonicalizing the code in Fig. 4.5a.

After translation, we can easily parse the generated post-order code using a simple bottom-up parser to obtain the corresponding in-order code.

4.4.4 Evaluating Translations

We rely on the deterministic nature of compilation as the basis of this evaluation. After translating the inputs, for each pair of input $i$ and corresponding translation $t$ (i.e. the decompiled code), we recompile $t$ and compare the output to $i$. This allows us to keep track of progress and success rates, even when the correct translation is not known in advance.

Comparing computation structure   After the first step of our decompiler, the structure of computation in the decompiled program should match the one of the original program. We therefore compare the original program and the templated program from decompilation by comparing their program dependence graphs. We convert each code snippet to its corresponding Program Dependence Graph (PDG). The nodes of the graph are the different instructions in the snippet. The graph contains 2 types of edges: data dependency edges and control dependency edges. A data dependency edge from node $n_1$ to node $n_2$ means that $n_2$ uses a value set by $n_1$. A control dependency between $n_1$ and $n_2$ means that execution of $n_2$ depends on the outcome of $n_1$. Fig. 4.6b shows an example of a program dependence graph for the code in Fig. 4.6a. Solid arrows in the graph represent data dependencies between code lines and dashed arrows represent control dependencies. Since line 2 uses the variable $x$ which was defined in line 1, we have an arrow from 1 to 2. Similarly, line 8 uses the variable $z$ which can be defined in either line 4 or line 6. Therefore, line 8 has a data dependency on both line 4 and line 6. Furthermore, the execution of lines 4 and 6 is dependent on the outcome of line 3. This dependency is represented by the dashed arrows from 3 to 4 and 6.

We extend the PDG by adding a special node for each variable used in the code. Any instruction that reads from a variable $v$ before it is written, will have a data dependency on the special node corresponding to $v$. These nodes allow us to identify instructions that read from the same unwritten variable, while maintaining a separation between different variables.

We then search for an isomorphism between the 2 graphs, such that if nodes $n$
1: \( x = 3; \)
2: \( y = x \times x; \)
3: \textbf{if} \( y \% 2 == 0 \) \textbf{then}
4: \( z = x + 5; \)
5: \textbf{else}
6: \( z = x - 7; \)
7: \textbf{end if}\n8: \( w = z \times 2; \)

(a) Source code

Figure 4.6: Example of Program Dependence Graph. Solid arrows for data dependencies, dashed arrows for control dependencies.

and \( n' \) are matched by the isomorphism it is guaranteed that either (1) both \( n \) and \( n' \) are special nodes corresponding to variables, (2) both \( n \) and \( n' \) correspond to numeric constants, or (3) \( n \) and \( n' \) correspond to the same operator (e.g. addition, subtraction, branching, etc...).

If such an isomorphism exists, we know that both code snippets implement the same computation structure. The snippets might still differ in the variable or numeric constants they use. However, the way the snippets use these variables and constants is equivalent in both snippets. Thus, if we could assign the correct variables and constants to the code, we would get an identical computation in both snippets. We consider translations that reach this point as a successful template and attempt to fill the template as described in Section 4.5. A translation is determined fully successful only if filling the template (Section 4.5) is also successful.

This kind of evaluation allows us to overcome instruction reordering, variable renaming, minor translation errors and small modifications to the code (often due to optimizations).

4.4.5 Stopping Decompilation

Our framework terminates the decompilation iterations when 1 of 3 conditions is met:

1. Sufficient results: given a percentage threshold \( p \), after each iteration the framework checks the number of test samples that remain untranslated and stops when at least \( p\% \) of the initial test set was successfully decompiled.

2. No more progress: The framework keeps track of the amount of remaining test samples. When the framework detects that that number has not changed in \( x \) iterations, meaning no progress was made during these iterations, it terminates. Such cases highlight samples that are too difficult for our decompiler to handle.
3. Iteration limit: given some number \( n \), we can terminate the decompilation process after \( n \) iterations have finished. This criteria is optional and can be left empty, in which case only the first 2 conditions apply.

### 4.4.6 Extending the Language

An important feature of our framework is that we can focus the training done in the first phase to language features exhibited by the input. Essentially, we can start by “learning” to decompile a subset of the high-level language.

Learning to decompile some subset \( s \) of the high-level language takes time and resources. Therefore, given a new input dataset, utilizing another subset of the language \( s' \), we would like to reuse what we have learned from \( s \).

Because the vocabulary of \( s' \) is not necessarily contained in the vocabulary of \( s \), i.e. \( \text{vocab}(s') \not\subseteq \text{vocab}(s) \), we have implemented a dynamic vocabulary extension mechanism in our framework. When the framework detects that the current vocabulary is not the same as the vocabulary used for previous training sessions, it creates a new model and partially initializes it using value from a previously trained model. This allow us to add support for new tokens in the language without starting from scratch.

Note that all tokens are equivalent in the eyes of the NMT model. Specifically, the model does not know that a variable is different from a number or an operator. It only learns a difference between the tokens from the different contexts in which they appear. Therefore, using this mechanism, we can extend the language supported by the decompiler with new operators, features and constructs, as needed. For example, starting from a subset of the language containing only arithmetic expressions, we can easily add if statements to the subset without losing any previous progress we’ve made while training on arithmetic expressions.

The extension mechanism is also used during training on a specific language subset. At each iteration, our framework generates new training samples to extend the existing training set. These new samples can, for example, contain new variables/numbers that weren’t previously part of the vocabulary, thus requiring an extension of the vocabulary.

It is important to note that in a real-world use-case we don’t expect training sessions to be frequent. Additional training should only applied when dealing with new features, a new language or with relatively harder samples than previous samples. We expect the majority of decompilation problems to be solved using an existing model.

### 4.5 Filling the Template

In Section 4.4, we saw how the decompiler takes a low-level program and produces a high-level templated program, where some constant assignments require filling. We now describe how to fill the parameters in the templated program.
4.5.1 Motivation

From our experimentation with applying NMT models to code, we learned that NMT performs well at generating correct code structure. We also learned that NMT has difficulties with constants and generating/predicting the right ones. This is exhibited by many cases where the proposed translation differs from an exact translation by only a numerical constant or a variable.

The use of word embeddings in NMT is a major contributor to these translation errors. A word embedding is essentially a summary of the different contexts in which that word appears. It is very common in NLP for identifying synonyms and other interchangeable words. For example, assume we have an NMT model for NLP which trains on the sentence “The house is blue”. While training, the model will learn that different colors often appear in similar contexts. The model can then generalize what it has learned from “The house is blue” and apply that to the sentence “The house is green” which it has never encountered before. In practice, word embeddings are numerical vectors, and the distance between the embeddings of words that appear in similar contexts will be smaller than the distance between embeddings of words that do not appear in similar contexts. The model itself does not operate on the actual words provided by the user. It instead translates the input to embeddings and operates on those vectors.

Since we are dealing with code rather than natural languages, we have many more “interchangeable” words to handle. During training all numerical values appear in the same contexts, resulting in very similar (if not identical) embeddings. Thus, the model is often unable to distinguish between different numbers. Therefore, while word embeddings are still useful for generalizing from training examples, using embeddings in our case results in translation errors when constants are involved.

Due to the above we have decided to treat the output of the NMT model not as a final translation but as a template that needs filling. The first phase of our decompilation process verifies that the computation structure resulting from recompiling the translation matches that of the input. If that is the case, any differences are most likely the result of using incorrect constants. The second phase of our decompilation process deals with correcting any such false constants.

Given that the computation structure of our translation and the input is the same, errors in constants can be found in variable names and numeric values. In the first phase, as part of comparing the computation structure, we also verify that there are no cases where a variable should have been a numeric value or vice versa. That means we can treat these two cases in isolation.

We note that since we are dealing with low-level languages, in which there are often no variable names to begin with, using the correct name is inconsequential. In the case of variables, all that matter is that for each variable in the input there exists a variable in the translation that is used in exactly the same manner. This requirement is already
fulfilled by matching the computation structure (Section 4.4.4).

4.5.2 Finding assignments for constants

We focus on correcting errors resulting from using wrong numeric values. Denoting the input as \( i \), the translation as \( t \) and the result of recompiling the translation as \( r \), there are three questions that we need to address:

**Which numbers in \( r \) need to change? and to which other numbers?** Since the NMT model was trained on code containing numeric values and constants, the generated translation also contains such values (generated directly by the model) and constants (due to the numeric abstraction step we describe in Section 4.4.3), and replaced with their original values. We use these numbers as an initial suggestion as to which values should be used.

As explained in Section 4.4.4, we compare \( r \) and \( i \) by building their corresponding program dependence graphs and looking for an isomorphism between the graphs. If such an isomorphism is found, it essentially entails a mapping from nodes in one graph to nodes in the other. Using this mapping we can search for pairs of nodes \( n_r \) and \( n_i \) such that \( n_r \in r \) is mapped to \( n_i \in i \), both nodes are numeric values, but \( n_r \neq n_i \). Such nodes highlight which numbers need to be changed (\( n_r \)) and to which other numbers (\( n_i \)).

**Which numbers in \( t \) affect which numbers in \( r \)?** Note that although we know that \( n \in r \) is wrong and to be fixed, we cannot apply the fix directly. Instead we need to apply a fix to \( t \) that will result in the desired fix to \( r \). The first step towards achieving that is to create a mapping from numbers in \( t \) to numbers in \( r \) such that changing \( n_t \in t \) results in a change to \( n_r \in r \).

By making small controlled changes to \( t \) we can observe how \( r \) is changed. We find some number \( n_t \in t \), replace it with \( n_t' \) resulting in \( t' \) and recompile it to get \( r' \). We then compare \( r \) and \( r' \) to verify that the change we made maintains the same low-level computation structure. If that is the case, we identify all number \( n_r \in r \) that were changed and record those as affected by \( n_t \).

**How do we enact the right changes in \( t \)?** At this point we know which number \( n_t \in t \) we should change and we know the target value \( n_i \) we want to have instead of \( n_r \in r \). All we need to determine now is how to correctly modify \( n_t \) to end up with \( n_i \).

The simple case is such that \( n_t == n_r \), which means whatever number we put in \( t \) is copied directly to \( r \) and thus we simply need replace \( n_t \) with \( n_i \).

However, due to optimizations (some applied even when using \(-O0\)), numbers are not always copied as is. Following are three examples we encountered in our work with x86 assembly.

**Replacing numbers in conditions** Assuming \( x \) is a variable of type \( int \), given the code \( \text{if } (x >= 5) \), it is compiled to assembly equivalent to \( \text{if } (x > 4) \), which is semantically identical but is slightly more efficient.
Division/Multiplication by powers of 2 These operations are often replaced with semantically equivalent shift operations. For example, Division by 8 would be compiled as shift right by 3.

Implementing division using multiplication Since division is usually considered the most expensive operation to execute, when the divisor is known at compilation time, it is more efficient implement the division using a sequence of multiplication and shift operations. For example, calculating \( x/3 \) can be done as \( (x \times 1431655766) >> 32 \) because \( 1431655766 \approx 2^{32}/3 \).

We identified a set of common patterns used to make such optimizations in common compilers. Using these patterns, we generate candidate replacements for \( n_t \). We test each replacement by applying it to \( t \), recompiling and checking whether the affected values \( n_r \in r \) are now equal to their \( n_i \in i \) counterparts.

We declare a translation as successful only if an appropriate fix can be found for all incorrect numeric values and constants.

4.6 Experimental Evaluation

4.6.1 Implementation

We implemented our technique in a framework called TRAFIX\(^1\). Our framework takes as input an implementation of our compiler interface and uses it to build a decompiler. The resulting decompiler takes as input a set of sentences in a low-level language \( L_{low} \), translates the sentences and outputs a corresponding set of sentences in a high-level language \( L_{high} \), specifically \( C \) in our implementation. Each sentence represents a sequence of statements in the relevant language.

Our implementation uses the NMT implementation provided by DyNmt [DyN17] with slight modifications. DyNmt implements the standard encoder-decoder model for NMT using DyNet [NDG\(^+\)17], a dynamic neural network toolkit.

Compiler Interface The compiler interface consists of a set of methods encapsulating usage of the compiler and representation specific information (e.g. how does the compiler represent numbers in the assembly?). The core of the api consists of: (1) A compile method that takes a sequence of \( C \) statements and returns the sequence of statements in \( L_{low} \) resulting from compiling it (the returned code is “cleaned up” by removing parts of it that don’t contribute any useful information); and (2) An Instruction class that describes the effects of different instructions, which is used for building a PDG during translation evaluation (Section 4.4.4).

We implemented such compiler interfaces for compilation (1) from \( C \) to LLVM IR, and (2) from \( C \) to x86 assembly. Fig. 4.7 shows the result of compiling the simple \( C \)

\(^1\)Source code available at https://github.org/omerktz/trafix
\[ X_0 = X_1 + X_2; \]

(a) C code

\[
\begin{align*}
%1 &= \text{load i32, i32* } \text{@X1} \\
%2 &= \text{load i32, i32* } \text{@X2} \\
%3 &= \text{add i32 } %1, %2 \\
&\text{store i32 } %3, \text{i32* } \text{@X0} \\
\text{movl } X_1, %edx \\
\text{movl } X_2, %eax \\
\text{addl } %edx, %eax \\
\text{movl } %eax, X_0
\end{align*}
\]

(b) LLVM IR

(c) x86 assembly

Figure 4.7: Example of code structure alignment

\[
\begin{align*}
\text{Statements} & : = \text{Statement} \mid \text{Statements } \text{Statement} \\
\text{Statement} & : = \text{Assignment} \mid \text{Branch} \mid \text{Loop} \\
\text{Assignments} & : = \text{Assignment} \mid \text{Assignments Assignment} \\
\text{Assignment} & : = \text{Var } = \text{Expr;} \\
\text{Var} & : = \text{ID} \\
\text{Expr} & : = \text{Var} \mid \text{Number} \mid \text{BinaryExpr} \mid \text{UnaryExpr} \\
\text{UnaryExpr} & : = \text{UnaryOp Var} \mid \text{Var UnaryOp} \\
\text{UnaryOp} & : = \text{++} \mid - \\
\text{BinaryExpr} & : = \text{Expr BinaryOp Expr} \\
\text{BinaryOp} & : = \text{+} \mid - \mid * \mid / \mid \% \\
\text{Branch} & : = \text{if } (\text{Condition}) \{ \text{Statements} \} \mid \text{else } \{ \text{Statements} \} \\
\text{Loop} & : = \text{while } (\text{Condition}) \{ \text{Statements} \} \\
\text{Condition} & : = \text{Expr Relation Expr} \\
\text{Relation} & : = \geq \mid \geq \mid \leq \mid \leq \mid \equiv \mid \equiv
\end{align*}
\]

Table 4.1: Grammar for experiments. Terminals are underlined

statement of Fig. 4.7a using both compilers.

### 4.6.2 Benchmarks

We evaluate TRAFIX using random C snippets sampled from a subset of the C programming language. Each snippet is a sequence of statements, where each statement is either an assignment of an expression to a variable, an if condition (with or without an else branch), or a while loop. Expressions consist of numbers, variables, binary operator and unary operators. If and while statements are composed using a condition – a relational operator between two expression – and a sequence of statements which serves as the body. We limit each sequence of statements to at most 5. Table 4.1 provides the formal grammar from which the benchmarks are sampled.

All of our benchmarks were compiled using the compiler’s default optimizations. Working on optimized code introduces several challenges, as mentioned in Section 4.5.2, but is crucial for the practicality of our approach. Note that we didn’t strip the code after compilation. However, our "original" C code that we compile is already essentially stripped since our canonicalization step abstracts all names in the code.

During benchmark generation we make sure that there is no overlap between the Training dataset, Validation dataset and our Test dataset (used as input statements to the decompiler).

**Evaluating Benchmarks** Despite holding the ground-truth for our test set (the C used to generate the set), we decided not to compare the decompiled code to the
ground-truth. We observe that, in some cases, different C statements could be compiled to the same low-level code (e.g. the statements \(x = x + 1\) and \(x++\)). We decided to evaluate them in a manner that allows for such occurrences and is closer to what would be applied in a real use-case. We, thus, opted to evaluate our benchmarks by recompiling the decompiled code and comparing it against the input, as described in Section 4.4.4.

### 4.6.3 Experimental Design and Setup

We ran several experiments of TraFix. For each experiment we generated 2,000 random statements to be used as the test set. TraFix was configured to generate an initial set of 10,000 training samples and an additional 5,000 training samples at each iteration. An additional 1,000 random samples served as the validation set. There is no overlap between the test set and the training/validation sets. We decided, at each iteration, to drop half of the training samples from the previous iteration. This serves to limit the growth of the training set (and thus the training time), and assigns a higher weight to samples obtained through recent failures compared to older samples. Each iteration was limited to 2,000 epochs. In practice, our experiments never reached this limit. No iteration of our experiments with LLVM and x86 exceeded more than 140 epochs (and no more than 100 epochs when excluding the first iteration). For each test input we generated 5 possible translations using beam-search. We stopped each experiment when it has successfully translated over 95% of the test statements or when no progress was made for the last 10 iterations.

Recall that the validation set is periodically translated during training and used to evaluate training progress. TraFix is capable of stopping a training session early (before the epoch limit was reached) if no progress was observed in the last consecutive \(k\) validation sessions. Intuitively, this process detects when the model has reached a stable state close enough to the optimal state that can be reached on the current training set. In our experiments a validation session is triggered after processing 1000 batches of training samples (each batch containing 32 samples) and \(k\) was set to 10. All training sessions were stopped early, before reaching the epochs limit.

The NMT model consists of a single layer each for the encoder and decoder. Each layer consists of 100 nodes and the word embedding size was set to 300.

We ran our experiments on Amazon AWS instances. Each instance is of type r5a.2xlarge – a Linux machine with 8 Intel Xeon Platinum 8175M processors, each operating at 2.5GHz, and 64GiB of RAM, running Ubuntu 16.04 with GCC [gcc87] version 5.4.0 and Clang [cla07] version 3.8.0.

We executed our experiments as a single process using only a single CPU, without utilizing a GPU, in order to mimic the scenario of running the decompiler on an end-user’s machine. This configuration highlights the applicability of our approach such that it can be used by many users without requiring specialized hardware.
4.6.4 Results

Estimating Problem Hardness

As a measure of problem complexity, we first evaluated our decompiler on several different subsets of C using only a single iteration. The purpose of these measurements is to estimate how difficult a specific grammar is going to be for our decompiler.

We used 8 different grammars for these measurement. Each grammar is building upon the previous one, meaning that grammar \( i \) contains everything in grammar \( i - 1 \) and adds a new grammar feature (the only exception is grammar 4 which does not contain unary operators). The grammars are:

1. Only assignments of numbers to variables
2. Assignments of variables to variables
3. Computations involving unary operators
4. Computations involving binary operators
5. Computations involving both operators types
6. If branches
7. While loops
8. Nested branches and loops

Fig. 4.8 shows the success rate, i.e. percentage of successfully decompiled inputs, for the different grammars, of decompiling x86 assembly with and without compiler optimizations. Note that measured success rates are after only a single iteration of our decompilation algorithm (Section 4.4.1).

As can be expected, the success rate drop as the complexity of the grammar increases. That means that for more complicated grammar, our decompiler will require more iterations and/or more training data to reach the same performance level as on
simpler grammars.

As can also be expected, and as can be observed from the figure, decompiling optimized code is a slightly more difficult problem for our decompiler compared to unoptimized code. Although optimizations reduce our success rate by a few percents (at most 5% in our experiments), it seems that the decisive factor for the hardness of the decompilation problem is the grammar complexity, not optimizations.

Recall that, given a compiler, our framework learns the inverse of that compiler. That means that, in the eyes of the decompiler, optimizations are “transparent”. Optimizations only cause the decompiler to learn more complex patterns than it would have learned without optimizations, but don’t increase the number of patterns learned nor the vocabulary handled. Grammar complexity, on the other hand, increases both the number and complexity of the patterns the decompiler needs to learn and handle, and the vocabulary size, thus making the decompilation task much harder to learn.

We emphasize that enabling/disabling compiler optimizations in our framework required no changes to the framework. The only change necessary was adding the appropriate flags in the compiler interface.

### Iterative Decompilation

In our second set of experiments we allowed each experiment to execute iteratively to observe the effects of multiple iterations on our decompilation success rates.

We implemented and evaluated 2 instances of our framework: from LLVM IR to C, and from x86 assembly to C.

We ran each experiments 5 times using the configuration described in Section 4.6.3. We allowed each experiment to run until it reached either a success rate of 95% or 6 iterations. The results reported below are averaged over all 5 experiments.

#### Decompiling LLVM IR

Out of the 5 experiments we conducted using our LLVM IR instance, 3 reached the goal of 95% success rate after a single iteration. The other 2 experiments required one additional iteration to reach that goal. Table 4.2 reports average statistics for these two iterations. The columns epochs, train time and translate time report averages for each iteration (i.e. average of measurements from 5 experiments for the 1st iteration and from only 2 experiments for the 2nd iteration). The successful translations column reports the overall success rate, not just the successes in that specific iteration.

The statistics in the table demonstrate that our LLVM decompiler performed

<table>
<thead>
<tr>
<th>#</th>
<th>epochs</th>
<th>timings</th>
<th>successful translations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75.6</td>
<td>14:16</td>
<td>03:25</td>
</tr>
<tr>
<td>2</td>
<td>76.5</td>
<td>14:11</td>
<td>00:42</td>
</tr>
</tbody>
</table>

Table 4.2: Statistics of iterative experiments of LLVM IR
On average, our LLVM experiments successfully decompiled 97% of the benchmarks, before autonomously terminating. These include benchmarks consisting of up to 845 input tokens and 286 output tokens. We intentionally set the goal lower than 100%. Setting it higher than 95% and allowing our instances to run for further iterations would take longer but would also lead to a higher overall success rate.

The timing measurements reported in the table highlight that the majority of execution time is spent on training the NMT model. Translation is very fast, taking only a few seconds per input, as witnessed by the first iteration. The execution time of our translation evaluation (including parsing each translation into a PDG, comparing with the input PDG, and attempting to fill the templates correlating to the translations) is extremely low, taking only a couple of minutes for the entire set of benchmarks.

These observations are important due to the expected operating scenario of our decompiler. We expect the majority of inputs to be resolved using a previously trained model. Retraining an NMT model should be done only when the language grammar is extended or when significantly difficult inputs are provided. Thus, in normal operations, the execution time of the decompiler, consisting of only translation and evaluation, will be mere seconds.

**Decompiling x86 Assembly** Table 4.3 provides statistics of our x86 experiments. All of these experiments terminated when they reached the iterations limit which was set to 6.

Fig. 4.9 visualizes the successful translations column. The figure plots our average success rate as a function of the number of completed iterations. It is evident that with each iteration the success rate increases, eventually reaching over 88% after 6 iterations. Overall, our decompiler successfully handled samples of up 668 input tokens and 177 output tokens.

Our decompilation success rates on x86 were lower than that of LLVM, terminating at around 88%. This correlates with the nature of x86 assembly, which has smaller vocabulary than that of LLVM IR. The smaller vocabulary shortens overall training times, but also results in longer dependencies and meaningful patterns that are harder
Figure 4.9: Cumulative success rate of x86 decompiler as a function of how many iteration the decompiler performed to deduce and learn.

We note that, in case of a traditional decompiler, bridging the remaining gap of 13% failure rate would require a team of developers crafting additional rules and patterns. Using our technique this can be achieved by allowing the decompiler to train longer and on more training data.

4.7 Discussion

4.7.1 Limitations

Manual examination of our results from Section 4.6.4 revealed that currently our main limitation is input length. There was no threshold such that inputs longer than the threshold would definitely fail. We observed both successful and failed long inputs, often of the same length. We did however observe a correlation between input length and a reduced success rate. As the length of an input increases, it becomes more likely to fail.

We found no other outstanding distinguishing features, in the code structure or used vocabulary, that we could claim are a consistent cause of failures.

This limitation stems from the NMT model we used. long inputs are a known challenge for existing NMT systems [KK17]. NMT for natural languages is usually limited to roughly 60 words [KK17]. Due to nature of code (i.e. limited vocabulary, limited structure) we can handle inputs much longer than typical natural language sentences (668 words for x86 and 845 words for LLVM). Regardless, this challenge also applies to us, resulting in poorer results when handling longer inputs. As the field of
\[ X3 = 63 \times ( 5 \times X1 ) ; \]

(a) High-level code

\[
\begin{align*}
\text{movl } X1 , \%eax \\
\text{imull } 315 , \%eax , \%eax \\
\text{movl } \%eax , X3
\end{align*}
\]

(b) Low-level code

\[ X3 = ( X1 \times 43 ) \times 70 ; \]

(c) Suggested decompilation

Figure 4.10: Example of decompilation failure

NMT evolves to better handle long inputs, so would our results improve.

To verify that this limitation is not due to our specific implementation, we created another variant of our framework. This new variant is based on TensorFlow \[\text{ten15, ABC}^{+16}\] rather than DyNet. Experimenting with this variant, we got similar results as those reported in Section 4.6.4, and ultimately reached the same conclusion — the observed limitation on input length is inherent to using NMT.

Other Decompilation Failures

Though we do not consider this a limitation, another aspect that could be improved is our template filling phase. Our manual analysis identified some possibilities for improving our second phase – the template filling phase (Section 4.5).

The first type of failure we have observed is the result of constant folding — a compiler optimization that replaces computations involving only constants with their results. Fig. 4.10 demonstrates this kind of failure. Given the C code in Fig. 4.10a, the compiler determines that 63 \times 5 can be replaced with 315. Therefore, the x86 assembly in Fig. 4.10b contains the constant 315. Using the code of Fig. 4.10b as input, our decompiler suggests the C code in Fig. 4.10c.

Note that the decompiler suggested code that is identical in structure to the input. The first phase of our decompiler handled this example correctly, resulting in a matching code template. The failure occurred in the second phase, in which we were unable to find the appropriate numerical values. This failure occurs because our current implementation attempts to find a value for each number independently from other numbers in the code. Essentially, this resulted in floating-point numbers which were deemed unacceptable by the decompiler because our benchmarks use only integers.

This kind of failure can be mitigated by either (1) applying constant folding to the high-level decompiled code, (2) allowing the template to be filled with floating point numbers (which was disabled since the benchmarks contained only integers), or (3) encoding the code as constraints and using a theorem prover to find appropriate assignments to constants.

A similar example is found in Fig. 4.11. We left the suggested translation in
X2 = ((X0 \% 40) * 63) / ((98 - X1) - X0);

(a) High-level code

X2 = ((X0 \% N3) * N13) / (((N2 - X1) + N11) - X0);

(b) Suggested decompilation

Figure 4.11: Failure due to redundant number

X2 = 48 + (X5 * (X14 * 66));

(a) High-level code

X2 = ((N8 * X14) * X5) - N4;

(b) Suggested decompilation

Figure 4.12: Failure due to incorrect operator

this example as constants to simplify the example. One can see that the suggested translation in Fig. 4.11b is structurally identical to the expected output in Fig. 4.11a, up to the addition of N11. This example was not considered a matching code template by our implementation, because any value for N11 other than 0 results in a different computation structure. However, if N11 = 0, we get an exact match between the suggested translation and the expected output. Using a theorem prover based template filling algorithm could detect that and assign the appropriate values to the constants, including N11, resulting in equivalent code.

Fig. 4.12 shows another kind of failure. In this example the difference between the expected output and suggested translation is a + that was replaced with −. Currently only variable names and numeric constants are treated as template parameters. This kind of difference can be overcome by considering operators as template parameters as well. Since the number of options for each operator type (unary, binary) is extremely small, we could try all options for filling these template parameters.

4.7.2 Framework Tradeoffs

There are a few tradeoffs that should be taken into account when using our decompilation framework:

- Iterations limit – Applying an iterations limit allows to tradeoff decompilation success rates for a shortened decompilation time and would make sense in environments with limited resources (time, budget, etc.). On the other hand, setting the limit too low will prevent the decompiler from reaching its full potential and will result in low successful translations rate.

- Training set size – In our experiments we initialized the training set to 10,000 random samples and generated additional 5,000 new random samples each iteration. As we increase the training set size, so do the training time and memory consumption increase. Using too many initial training samples would be wasteful
in case of relatively simple test samples, in which a shorter training session, with fewer training samples, might suffice. On the other hand, using too few samples would result in many training sessions when dealing with harder test samples. This is also applicable when setting the number of random samples added at each iteration. Furthermore, rather than always generating a constant number of samples, one can dynamically decide the number of samples to generate based on some measure of progress (i.e. generate fewer samples when progressing at a higher rate).

- Patience – the patience parameter determines how many iterations to wait before terminating due to not observing any progress. Setting this parameter to high would result in wasted time. This is because any training performed since the last time we observed progress would essentially have been in vain. On the other hand, it is possible for the model to make no progress for a few iterations only to resume progressing once it generates the training samples it needed. Setting the patience parameter too low might cause the decompiler to stop before it can reach its full potential.

4.7.3 Extracting Rules

As mentioned in Section 4.1, traditional decompilers rely heavily on pattern matching. Development of such decompilers depends on hand-crafted rules and patterns, designed by experts to detect specific control-flow structures. Hand-crafting rules is slow, expensive and cumbersome. We observe that the successful decompilations produced by our decompiler can be re-templatized to form rules that can be used by traditional decompilers, thus simplifying traditional decompiler development. Appendix A provides examples of such rules.

4.7.4 Evaluating Readability

Measuring the readability of our translations requires a user study, which we did not perform. However, note that given some training set, a model trained on that set will generate code that is similar to what it was trained on. Thus, the readability of our translations stems from the readability of our training samples. Our translations are as readable as the training samples we generated. This was also verified by an empirical sampling of our results. Therefore, given readable code as training samples, we can surmise that any decompiled code we generate and output will also be readable.
Chapter 5

Related Work

There has been a lot of work on reverse engineering and analysis of binaries. In this chapter we briefly survey closely related work.

**Analysis of Executables.** Reps et al. [RLT+10, RBLT07] explored many aspects in the analysis of stripped binaries, and obtained impressive results during years of work on the problem. Their analyses (e.g., [RBL06, RB08, BR07]) can recover a lot of information from a stripped binary and statically verify challenging safety properties.

The tracelet extraction part of our work relies on points-to and aliasing analysis, which have been previously discussed and used on binaries and assembly code in several works, such as [ABTZ00, DMW98, RBLT07, BDEK99, BBC+06]. In [RB08], Reps et al. present a value-set analysis (VSA) to determine memory accesses. Our analysis borrows the notion of a set of possible values for each variable, but instead of the standard abstract values common in VSA, we use value expressions generated by our points-to analysis. The work in [RB08] puts an emphasis on identifying all variables and objects in the binary. In our work we identify objects of interest based on virtual calls and focus only on these objects. Therefore in our setting identifying the objects is simpler than in the general case. For similar reasons, the analysis in [RB08] is more complex than what we require and we use instead a simpler, more targeted analysis, as explained in Section 2.5.1.

Balakrishnan et al. [BR10] and Reps et al. [RBL06] presented tools that assist reverse engineering; however, neither tackled the problems we worked on.

Brumley et al. [BJAS11] presented a flexible platform for the static analysis of binaries, and investigated a wide range of interesting problems, including decompilation [SLWB13a] and identifying functions in binaries [BBW+14]. The technique presented in [BBW+14] uses machine learning to train a model of function start sequences and uses that to compute function boundaries. Similarly to our work, it recognizes that in certain settings a statistical approach is superior to standard approaches, both in usefulness and in accuracy.

Lee et al. [LAB11] presented a tool called TIE aimed at the problem of inferring primitive types for variables in a binary, which is related to the first problem we
tackled (Chapter 2). TIE uses inference rules that are based on the data transfers between variables and known functions. Under our setting, the technique of TIE would determine all objects as belonging to type `pointer` without being able to determine the actual type of the object. This result is correct, as all objects are in practice pointers to instances of a type, but it is not accurate enough to be useful in real-world setting.

Many previous works have tackled the problem of type reconstruction (see [CL16] for a recent survey and [HSB13] for a specific example). The problem of type reconstruction is conceptually similar to our problems. However, we do not try to recreate type structures. Instead, our goal is to identify types of objects (Chapter 2) and type hierarchies (Chapter 3) without referring to the type structure.

The problem of finding object boundaries in stripped binaries is generally a hard problem. Work by Gopan et al. [GDN+15] addresses this problem using a static analysis and heuristics, obtaining very good results in practice. In our work, we dealt with the closely related problem of determining the size of allocated memory for objects, and used an estimate as described in Section 2.5.6. Furthermore, we do not attempt to find all object boundaries and instead use virtual calls as our basis for identifying objects.

Jang et al. [JTL14] attempted to analyze source code to detect likely virtual call targets in order to impose restrictions on runtime targets. This approach could be used to prevent the first problem we faced (Chapter 2) by marking relevant targets during compilation. By analyzing the binary, we attempt to mark the targets post-compilation. We believe our approach is more feasible, as in many real-world scenarios it is not possible to intervene in the compilation process.

**Statistical Techniques.** Markov models have previously been used to classify behaviors. Jha et al. [JTM01] used Markov models for intrusion detection. They rely on identifying sequences of actions that would be improbable in other settings. We extend this notion and rely on differences between sequence probabilities across contexts to employ Markov models for type classification and class hierarchy reconstruction.

Schütze and Singer [SS94] used variable-order Markov Models to disambiguate words in a text depending on context. Given a word with several possible meanings, the proposed technique selects the correct one based on the surrounding text. Their scenario shares some characteristics with the scenario in which we operated in Chapter 3, supporting the suitability of VMMs to this problem.

**Modeling Source Code** Modeling source code using various statistical models has seen a lot of interest for various applications.

A common application of source code modeling is for predicting names for variable, methods and classes. Raychev et al. [RVK15] presented a statistical approach to predicting program properties using CRFs. Their technique leverages program structure to create dependencies and constraints used for probabilistic reasoning. This works well at the source code level as a lot of program structure is easy to recover. When working on stripped binaries, there is much less program structure to work with, and
this approach is therefore not applicable. The beauty of our approach is that it works without the need to recover a lot of structure (which is a hard problem). Moreover, in contrast to [RVK15], which, in practice, used large amounts of training data collected across several programs, we use less training data, obtained from the same program on which we perform the prediction.

Raychev et al. [RVK15] predicted variable names in obfuscated Javascript code. He et al. [HIT+18] also used CRFs but for the purpose of predicting debug information in stripped binaries, focusing on names and types of variables. Allamanis et al. [ABBS15] used neural language models to predict variable, method and class names. Allamanis et al. relied on word embeddings to determine semantically similar names. We consider this problem as orthogonal to our problem of decompilation and these works as supplementary solutions. Given a semantically equivalent source code produced by our decompiler, these techniques could be used to supplement it with variable names, etc.

Srinivasan et al. [IKCZ16] used LSTMs to generate natural language descriptions for C# source code snippets and SQL queries. Allamanis et al. [APS16] generated descriptions for Java source code using convolutional neural networks with attention. Hu et al. [HWLJ17] tackled the same problem by neural networks with a structured based traversal of the abstract syntax tree, aimed at better representing the structure of the code. Loyola et al. [LMM17] took a similar approach for generating descriptions of changes in source code, i.e. translates commits to source code repositories to commit messages. The success presented by these papers highlights that neural networks are useful for summarizing code, and supports the use of neural networks for decompilation.

Chen et al. [CLS18] used neural networks to translate code between high-level programming languages. This problem resembles that of decompilation, but translating low-level languages to high-level languages, as we do, is more challenging. The similarities between high-level languages are more prevalent than between high-level and low-level languages. Furthermore, translating source code to source code directly bypasses many challenges added by compilation and optimizations.

Levy et al. [LW17] used neural networks to predict alignment between source code and compiled object code. Their results can be useful in improving our second phase of TraFix, i.e. filling the template and correcting errors. Specifically, their alignment prediction can be utilized to pinpoint location in the source code that lead to errors.

Behavioral Patterns. Preda et al. [PCJD07], Warrender et al. [WFP99] and Mazeroff et al. [MGTF08] have all proposed using behavioral signatures to detect malware. They based their approaches on dynamically tracking events, mainly API and system calls. They show that behavioral patterns can identify intent in binaries. Fredrikson et al. [FCJ11] broaden the notion beyond malware classification and discuss matching binaries with behavior specifications. We extend this idea further and show that it can also be used to identify types, using the trained type VMMs as our specifications.

David et al. [DY14] used control-tracelets to find similar code fragments across
stripped binaries. Their control-tracelets provide a coarser abstraction of program execution, and are not suited to tracking the behavior of objects. In contrast, our object-tracelets track method calls and field access events for (abstract) objects. We borrow their notion of tracelets as a code identifier and adapt it to our setting, to identify object types.

Polino et al. [PSMZ15] have recognized the importance of behavioral patterns for reverse engineering. Their dynamic approach was aimed at providing the reverse engineer with useful patterns. We extract similar behavioral patterns using a static approach. Additionally, instead of providing them to the reverse engineer as is, we elevate these patterns to statistical models and use these models to match the patterns to other sequences in the binary.

Mishne et al. [MSY12] used static analysis to extract temporal specifications from a large corpus of code snippets. Their abstract histories are similar to our object tracelets. Raychev et al. [RVY14] construct a statistical language model for sequences of API calls in Java. The language model is based on object histories, similar to our object tracelets.

**Structural Similarity.** Madsen et al. [MLF13] suggested a technique based on “structural similarity” of types and objects. Their technique relies on object structure and the existence of names of fields and functions in the binary to find a matching type. While [MLF13] dealt with JavaScript programs, the technique presented in [Tu12] suggests a similar approach for Python programs implemented in a tool called *Mino*. Similarly, this approach also relies on variable/field names as the basis for the data. The technique of [Tu12] suggests using classifiers to determine types of variables. However, in contrast to our approach, the authors rely on dynamic execution to collect labeled data and only deal with a small set of generic primitive types. They do not deal with other user-defined objects types. Unfortunately, in a stripped binaries setting, the binaries contain no names that can be used as a basis for matching. Instead we use “behavioral similarity”, which takes into consideration how the object is used rather than which fields and functions it has. Without the explicit names utilized by *Mino* and similar works, classification of a single type/target or reconstruction of a single type hierarchy is not reliable enough, as we demonstrated.

**Reconstructing Class Hierarchies.** Srinivasan and Reps [SR14, SR13] present a tool called *Lego* that uses dynamic analysis for reconstructing the class hierarchy of a stripped binary. Lego dynamically records execution traces, which it uses to identify classes and reconstruct the hierarchy. Lego relies heavily on the existence of calls to methods of the parent-class, specifically destructors (finalizers). As we explained in Section 3.1, this kind of information is not guaranteed to exist in the binary. Unlike LEGO, *Rock* is able to reconstruct a hierarchy even when all destructors have been inlined. Like the authors of LEGO, we acknowledge the importance of tracking object behavior. However, our technique is purely static and does not require running the program or dealing with challenges of partial coverage typical in dynamic
techniques. Furthermore, interactive (GUI) applications (such as CGridListCtrlEx) pose a significant challenge to dynamic approaches, as actual user interactions are needed to drive the program. We were unable to obtain an instance of Lego for comparative testing.

Fokin et al. [FDCT11] discuss existing non-statistical techniques for obtaining the class hierarchy of a binary as part of their work on decompilation. Like us, they concluded that existing techniques, such as the use of Real Time Type Information (RTTI) records and others, are unreliable and prone to errors.

Pawlowski et al. [PCvdV+17] presented Marx, a state of the art tool for static hierarchy reconstruction. However, Marx arranges related types into type families rather than build an actual hierarchy. We go a (significant) step further by arranging the types in each family to form an hierarchy.

**Decompilation** The Hex-Rays decompiler [IDA00] was considered the state of the art in decompilation, and is still considered the de-facto industry standard. Schwartz et al. [SLWB13b] presented the Phoenix decompiler which improved upon Hex-Rays using new analysis techniques and iterative refinement, but was still unable to guarantee goto-free code (since goto instructions are rarely used in practice, they should not be part of the decompiler output). Yakdan et al. [YEGS15, YDGPS16] introduced Dream, and its predecessor Dream++, taking a significant step forward by guaranteeing goto-free code. RetDec [ret17], short for Retargetable Decompiler, is an open-source decompiler released in December 2017 by Avast, aiming to be the first “generic” decompiler capable of supporting many architectures, languages, ABIs, etc.

While previous work made significant improvements to decompilation, all previous work fall under the title of rule-based decompilers. Rule-based decompilers require manually written rules and patterns to detect known control-flow structures. These rules are very hard to develop, prone to errors and usually only capture part of the known control-flow structures. According to data published by Avast, it took a team of 24 developers 7 years to develop RetDec. This data emphasizes that traditional decompiler development is extremely difficult and time consuming, supporting our claim that the future of decompilers lies in approaches that can avoid this step. Our technique removes the burden of rule writing from the developer, replacing it with an automatic, neural network based approach that can autonomously extract relevant patterns from the data.

Schulte et al. [SRN+18] suggested addressing decompilation using big code. Their approach attempts to reconstruct high-level source code equivalent to a given input from snippets found in big code. They use an evolutionary search on their database – mutating and mixing snippets until they reach a satisfactory high-level source code. This is an interesting approach but it requires significantly more data than required by TRAFIX. Similarly to us, Schulte et al. also recognized the need to fix small errors in very similar code in order to reach equivalence. Ideas from their fixing techniques
could be incorporated to improve TraFix’s second phase.

Katz et al. [KRS18] suggested the first technique to use NMT for decompilation. While they set out to solve the same problem, in practice they provide a solution to a different and significantly easier problem - producing source-level code that is readable, without any guarantees for equivalence, not semantic or even syntactic. Further, the code they generate is not guaranteed to compile (and does not in practice). If you apply our evaluation criteria to their results, their accuracy is at most 3.8% (in contrast to our 70% precision using the same metric). Further, beyond the cardinal difference in the problem itself, they have the following limitations:

- They can only operate on code compiled with a special version of Clang which they modified for their purposes.
- All of their benchmarks are compiled without optimizations. We apply the compiler’s default optimizations to all of our benchmarks.
- They limit their input to 112 tokens and output to 88 tokens. This limits their input to single statements. We handle inputs of up to 950 input tokens and 200 output tokens. Each of our samples contains several statements.
- Their methodology is flawed as they do not control for overlaps between the training and test sets. We verify that there is no such overlap in our sets.
Chapter 6

Conclusion

The goal of this thesis was to suggest techniques that can assist reverse engineers in their work. To that end, we presented several statistical approaches that offer solutions to difficult reverse engineering problems.

In Chapter 2 we described an approach for identifying types of objects and targets of virtual function calls in an optimized stripped binary. This is a crucial step for understanding the control flow of a program. This chapter introduced the notion of *object tracelets* — sequences of operations applied to an object in the binary. These sequences capture how an object is used in practice in the binary. Additionally, Chapter 2 introduced the use of statistical language models as a means of summarizing and representing usage pattern for types. The notion of object tracelets aims to capture the subtle differences between types/objects in stripped optimized binaries. In such cases, all types/objects look similar and “clues” such as field/method names cannot be used for matching an object to its type. Chapter 2 demonstrated the usefulness of object tracelets and statistical language models as object and type representations. Using these tools, we successfully reduced the number of viable targets for virtual calls in our benchmarks from dozens to only a handful. This is the first step towards our goal, which highlights the potential value in the combination of programming languages techniques with machine learning tools.

Building on the foundations of Chapter 2, Chapter 3 dealt with reconstructing class hierarchies in optimized stripped binaries. This problem is crucial for program understanding. Furthermore, a solution to this problem has direct implications on the security of the program as it can be used the check *Control-Flow Integrity*. Control flow integrity is a security mechanism that guarantees only legal targets can be reached by indirect jumps. For virtual function calls, the set of legal targets depends on the class hierarchy. Using the statistical language models of Chapter 2 for representing types, Chapter 3 measured similarity between types. It uses Kullback-Leibler divergence for computing said similarity. The computed similarity scores served as a basis for describing the class hierarchy reconstruction as a weighted graph problems. The minimum-weight arborescence of said graph correlates to the most likely hierarchy in the binary. This
chapter demonstrated the impact our statistical solutions can have on security. Using the resulting hierarchy we are capable of generating stricter control flow integrity policies, reducing the risk of false-positives.

Lastly, Chapter 4 took a preliminary step into the realm of statistical neural de-compilation. Decompilation is the problem of “lifting” a low-level piece of code (i.e. binary code) to high-level human-readable source-code. While many previous works on decompilation focused solely on using the decompiled code as documentation (to better understand the low-level code), we added the additional requirement that the decompiled code be equivalent to the input and compilable (i.e. syntactically and grammatically correct). Chapter 4 frames the decompilation problem as a language translation problem. This allowed us at natural language translation for inspiration and apply insights from that field to our solution. The solution presented in Chapter 4 relies on neural machine translation as the basis for decompilation. The NMT model was wrapped programming languages methods that simplify the translation problem, verify and fix translation and drive the overall decompilation. Chapter 4 highlighted the viability of NMT based decompilation as an alternative to traditional decompilers. This new approach is capable of overcoming many pitfalls in traditional rule-based decompiler development.

Overall, the approaches described in this thesis demonstrate the statistical solutions are a viable and useful alternative whenever a precise solution is computationally outside our reach. Our statistical solution leverage the combination of programming languages and machine learning to produce solutions more powerful and capable than possible without it. This thesis opens the door to an exciting new realm of solutions, techniques and possibilities, that can provide practical solutions to yet unsolved problems.
Appendix A

Decompilation Rules extracted from TraFix

Table A.1 contains examples of decompilation rules extracted from our decompiler. Successful translations are generalized (i.e. numbers are replaced with abstract constants, variable names are normalized, etc.) to create a generic pattern. The resulting patterns can be embedded in the rule-set of a rule-based decompiler, to assist in the development of new decompilers. For brevity, we present mostly relatively simple rules, but longer and more complicated rules were also found by our decompiler (examples of such rules are found at the bottom of the table, below the separating line).
<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl X1, eax; addl N1, eax; movl eax, X2;</td>
<td>X2 = N1 + X1;</td>
</tr>
<tr>
<td>movl X1, eax; subl N1, eax; movl eax, X2;</td>
<td>X2 = X0 - N1;</td>
</tr>
<tr>
<td>movl X1, eax; imull N1, eax, eax; movl eax, X2;</td>
<td>X2 = X1 * N1;</td>
</tr>
<tr>
<td>movl X1, ecx; movl N1, eax; idivl ecx; movl eax, X2;</td>
<td>X2 = X1 / X2;</td>
</tr>
<tr>
<td>movl X1, eax; movl X2, ecx; idivl ecx; movl eax, X3;</td>
<td>X2 = X1 * 2X1;</td>
</tr>
<tr>
<td>movl X1, eax; sall N1, eax; movl eax, X2;</td>
<td>X2 = N1%X1;</td>
</tr>
<tr>
<td>movl X1, ecx; movl N1, eax; idivl ecx; movl edx, eax; movl eax, X2;</td>
<td>X2 = X1%X2;</td>
</tr>
<tr>
<td>movl X1, eax; movl X2, ecx; idivl ecx; movl eax, X3;</td>
<td>X2 = X1++;</td>
</tr>
<tr>
<td>movl X1, eax; sall N1, eax; movl eax, X2;</td>
<td>X2 = X1--;</td>
</tr>
<tr>
<td>movl X1, eax; imull N1, eax, eax; addl N2, eax; movl eax, X2;</td>
<td>X2 = ++X1;</td>
</tr>
<tr>
<td>movl X1, eax; addl N1, eax; sall N2, eax; movl eax, X2;</td>
<td>X2 = N2 + (N1 * X1);</td>
</tr>
<tr>
<td>movl X1, eax; imull N1, eax, ecx; movl N2, eax; idivl ecx; movl eax, X2;</td>
<td>X2 = (X1 + N1) * 2X2;</td>
</tr>
<tr>
<td>movl X1, eax; cmpl N2, eax; jg L0; movl N2, X2; L0:</td>
<td>X2 = N2/(X1 * N1);</td>
</tr>
</tbody>
</table>
| jmp L1; L0: movl N1, X1; L1: movl X2, eax; cmpl N2, eax; jg L0;       | if(X1 < (N1 + 1)){X2 = N2;}
| jmp L1; L0: movl N1, X1; L1: movl X2, eax; cmpl N2, eax; jne L0;      | while(X2 > N2){X1 = N1;}
| movl X1, eax; cmpl N1, eax; jne L0; movl N2, X2; movl X3, eax; movl eax, X4; movl eax, L0; | while(N2! = X) {X1 = N1;}
| movl X1, edx; movl X2, edx; cmpl eax, edx; jg L0; movl N1, X3; jmp L1; L0: movl N2, X4; L1: | if(N1 == X1){X2 = N2; X4 = X3;}
| jmp L1; L0: movl X1, eax; addl N1, eax; movl eax, X2; L1; movl X1, eax; cmpl N2, eax; jle L0; | if(X2 <= X3){X3 = N1; else{X4 = N2;}
| jmp L1; L0: movl X2, eax; addl 1, eax; movl eax, X2; movl X2, edx; movl X2, eax; addl edx, eax... | while(X3 < X2){X2 = N2 = N1 + X1;}
| movl X1, eax; addl 1, eax; movl eax, X1; movl X1, edx; movl X2, eax; movl N1, ecx; subl eax; ecx... | while((X1 - N1) > (X2%(X2 - N2))) {X3 = (++X2... |
| movl X1, edx; movl X4, eax; addl edx, eax; movl X4, ecx; movl X5, edx; addl edx, ecx; idivl ecx; ... | if(++X1 == (((X2 + (N1 - X2)) - N2) * (N3 - X3)... |
| movl N1, X1; movl X1, eax; movl eax, X2; movl X2, eax; movl X3, edx; addl N3, edx; subl edx, eax... | X2 = X2 * ((X3 + X1)/((X4 + X3))); X0 = (X7 + ... |
| jmp L1; L0: movl X1, ebx; movl N3, eax; idivl ebx; movl eax, X1; L1: movl X1, edx; movl X2 ... | X1 = N1; X2 = X1; if((N2 + (X2 - (X1 + N1))))... |
|                                                                       | while((X1 * X2) >= (N1%(X3 + N2))){X1 = X3... |

Table A.1: Decompilation rules extracted from TrafFix
Bibliography


[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani,


[MSY12] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the*


בהידור לאחור, במקומם עלnect על שאלת קדחתית, אנו שואפים להסביר את כל זה, לעьер
הבים. הפתרון שנamation מתייחס את הכור במקומם הביניות ברוח הנוגע לשפה. אנו שואבים
השראת הלקחים מצומצםombres עזר נמצאת בשתיו ומיישמים אתינו ל /^[ kod be específ עלולא. במקומם הפרותר שלנו מספרי תרגום שכמותו שמיןicator נסותר, ו. אנו עוסנים
את המודל של שילוב עכשויוט פועל החוזה. עטיה של האמאפשרות של הפרש אהת בעניין
תהליך,לאומת תרגום נוכיים, ולצאת תוקינו בתחרות חרדות קלום תרגום משכיל שמספר
התרשים, לאמות תרגום נוכיים, ולצאת תוקינו בתחרות חרדות קלום תרגום משכיל שמספר
התרשים (תקודית הניטות לינוק). 결וקת המשותה והשם הנעים בינויה האומרי מאפרת
וללתחנה על בעיות התמורות הרבוטית בפיתוח הכימי של תחכר כים. התחרות שלחרי כימי
מסתמכים על התוונות בתביון אנצף כולל גידי הצבעים ולאור찰 י/cards לכלג של עלי די מומחה
יידעון. הכנה תכונה, פיתוח מתריה לאחור או תהליך ארצק וOfStringל לאריך שניה עד הבת.
בגרות מתחרי כלשהו, השיטה שלינקה כלבוד באפור ואופוסמי מתריה להתחדס להתקדש
כך הנלי יתחנבו על זוגי הביקובה הנוכיים ( לפעבים להכדים) ויועדו את התהליך אפור
ממשקתיי.

השיטות המוצגות בתוライ במקומם בunicipט לוערכו באומראו השדרי בטכון מקוד ב广大市民י הקהל מקוד
פתוח ובאומראו מנוף רוח של קסוע מקוד. תחלת הערכה ו IList�ות את ישות השיטה.
אף תורמת ותא תונתו, להקרד בתוכנה ותא תוע蒨ית את.
היתר מאישרehler ינת למשה בדואים, כלומר מכל עולמה תוכן בנוי עלמו.

לאחר שאלים ובוחנה את בעיית הרווחה בלאחר, [KOGY19] בביעור של חומרים תכונה יקתלה כתוב בס붾ה עית, שפחה לעילופ תכונה המייתנה להיות שארIEnumerableים בארץ קדימה וברחבת תכונות התכונה. קיימהו יתור בכלוק רוח ושושמתומשה. הרווח למשה איה העד ושימש בה-popup לשנת הכנושה.

บทמרות נוספים

[KEY16] [KEY18] [KRY18] [KOGY19]

Technion - Computer Science Department - Ph.D. Thesis PHD-2019-03 - 2019
תקציר

תוכנה המשפיעה בחום על כל תחומי החיים. עם זאת, נ наличии תוכנות נוספות ומסמכים, לрактиות היום יום, לטבעה של התוכנה, נגש י Marijuana יי. הזנה, עם ומכים המגינים על התוכנה, יישר יי. הזנה, עם ומכים המגינים והמותר של התוכנה, יישר יי. הזנה, עם ומכים המגינים. הזנה, עם ומכים המגינים ובחינה על התוכנה, יישר יי. הזנה, עם ומכים המגינים. הזנה, עם ומכים המגינים. הזנה, עם ומכים המגינים. הזנה, עם ומכים המגינים.


גישות סטטיסטיות
להנדסה לאחור

حجر על מחקר

לשם مليו חלקי של הדרישות לתכנית החותר
דוקטור לפילוסופיה

עומר צי

רוגש לטכנולוגיה – מכון טכנולוגיה לישראלי
נוכם החשעלו חיפה אפריל 2019
גישה סטטיסטית
להנדסת ליאור
עומר צ"ז