Statistical Reconstruction of Class Hierarchies in Binaries

Omer Katz
Technion, Israel
omerkatz@cs.technion.ac.il

Noam Rinetzky
Tel Aviv University, Israel
maon@cs.tau.ac.il

Eran Yahav
Technion, Israel
yahave@cs.technion.ac.il

Abstract

We address a fundamental problem in reverse engineering of object-oriented code: the reconstruction of a program's class hierarchy from its *stripped* binary. Existing approaches rely heavily on *structural* information that is not always available, e.g., calls to parent constructors. As a result, these approaches often leave gaps in the hierarchies they construct, or fail to construct them altogether. Our main insight is that *behavioral* information can be used to infer subclass/-superclass relations, supplementing any missing structural information. Thus, we propose the *first statistical approach* for static reconstruction of class hierarchies based on *behavioral similarity*. We capture the behavior of each type using a statistical language model (SLM), define a metric for *pairwise similarity* between types based on the *Kullback-Leibler divergence* between their SLMs, and lift it to determine the most likely class hierarchy.

We implemented our approach in a tool called Rock and used it to automatically reconstruct the class hierarchies of several real-world stripped C++ binaries. Our results demonstrate that Rock obtained *significantly* more accurate class hierarchies than those obtained using structural analysis alone.

ACM Reference Format:
Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. Statistical Reconstruction of Class Hierarchies in Binaries. In *Proceedings of 2018 Architectural Support for Programming Languages and Operating Systems (ASPLOS'18).* ACM, New York, NY, USA, 14 pages. https://doi.org/http://dx.doi.org/10.1145/3173162.3173202

1 Introduction

A 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 [40]. Additionally, knowing the class hierarchy helps to improve software security as it makes it possible to *harden* existing applications without breaking programs [30].

Interest in the analysis of executables has increased in recent years [3, 4, 13, 19, 21, 35–38, 44]. Most techniques have focused on the recovery of debug symbols and control-flow. There has been relatively little work on the reconstruction of class hierarchies, despite the importance of this problem to reverse engineering of modern software (as highlighted nicely in [30]). Furthermore, existing techniques, e.g., [17, 30, 40], 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., inlining calls to parent constructors).

The goal of this paper is to develop 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]

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

1. Identify the binary types in $bin(P)$, where binary types are represented as virtual function tables.
2. Construct a *hierarchy* over the binary types of $bin(P)$, such that binary type vt_p is an ancestor of type vt_c if the source class compiled into the binary type vt_c, denoted by $vtToSrc(vt_c)$, is a subclass derived from $vtToSrc(vt_p)$.

Class Hierarchies vs. Type Grouping Existing techniques [17, 30, 40] do not reconstruct an actual class hierarchy but 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

[1] To simplify presentation, we assume each class has at most one superclass. Our approach, however, does handle multiple inheritance (see Section 5.3).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS'18, March 24–28, 2018, Williamsburg, VA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/http://dx.doi.org/10.1145/3173162.3173202

RIGHTSLINK

```
1  std::string readInternal(DataSource ds) {
2          ds.connect();
3          std::string data = ds.read();
4          return data;
5  }
6
7  int readExternal(DataSource ds) {
8          ds.connect();
9          ds.verifiyCredentials();
10         std::string data = ds.read();
11         data = filterAndEscape(data);
12         return data;
13 }
```

Figure 1. Methods used to read data from data sources.

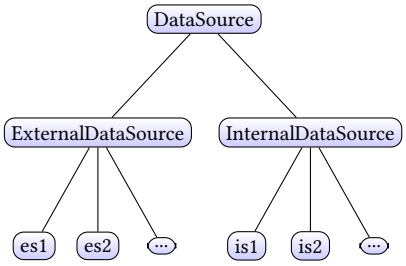

Figure 2. Class hierarchy of data sources.

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) [1], as done in [30], 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. 1. The program uses data from internal and external sources, each containing several types of data (see Fig. 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 6 demonstrate, our technique generates a more precise hierarchy than any existing technique.

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 subclass/superclass relations. The main idea is to capture the behaviors of each type using a *statistical language model* (SLM) [39] and define pairwise similarity based on the *Kullback-Leibler divergence* (DKL) [23]. 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., [30].
- *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.

Main Contributions The contributions of this paper are:

- A novel framework for statistical static reconstruction of a class hierarchy from stripped binaries. Our approach augments the technique of [21], which captures a *behavioral model* of each binary type using a statistical language model, with a notion of pairwise asymmetric distance between types based on the *Kullback-Leibler divergence*.

RIGHTSLINK

```
1  class Stream{ // socket interface
2      virtual void send(int n);
3  };
4  class ConfirmableStream : public Stream{
5
6      virtual void confirm();
7  };
8  class FlushableStream : public Stream{
9
10     virtual void flush();
11     virtual void close();
12 };
13
14 int useStream(Stream* stream){
15     stream->send(0);
16     stream->send(1);
17     stream->send(2);
18 }
19 int useConfirmableStream(ConfirmableStream* stream){
20     stream->send(0);
21     stream->confirm();
22     stream->send(1);
23     stream->confirm();
24     stream->send(2);
25     stream->confirm();
26 }
27 int useFlushableStream(FlushableStream* stream){
28     stream->send(0);
29     stream->send(1);
30     stream->send(2);
31     stream->flush();
32     stream->close();
33 }
```

Figure 3. Class definitions and example code.

- 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.

2 Overview

In this section, we provide an informal overview of our approach using a simple illustrative example. We provide a formal model and experimental results over realistic examples in later sections.

Motivating Example The code fragment shown in Fig. 3 defines the class hierarchy depicted in Fig. 4: (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`,

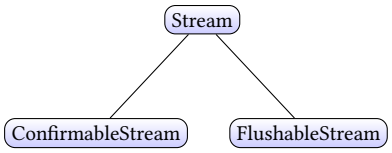

Figure 4. Class hierarchy as seen in source code.

```
1  class Class1{
2      virtual void f0(int n);
3  };
4  class Class2{
5      virtual void f0(int n);
6      virtual void f1();
7  };
8  class Class3{
9      virtual void f0(int n);
10     virtual void f1();
11     virtual void f2();
12 };
13
14 int useClass1(Class1* s){
15     s->f0(0);
16     s->f0(1);
17     s->f0(2);
18 }
19 int useClass2(Class2* s){
20     s->f0(0);
21     s->f1();
22     s->f0(1);
23     s->f1();
24     s->f0(2);
25     s->f1();
26 }
27 int useClass3(Class3* s){
28     s->f0(0);
29     s->f0(1);
30     s->f0(2);
31     s->f1();
32     s->f2();
33 }
```

Figure 5. Class definitions as observed in a stripped binary.

another extension of `Stream` that requires explicit flushing and closing of the stream. The `use`*X* 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. 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. 5 are generalized stripped names derived solely from the order of the functions in the code (such that `f0` is the 1st function 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. 4), using only information from the

RIGHTSLINK

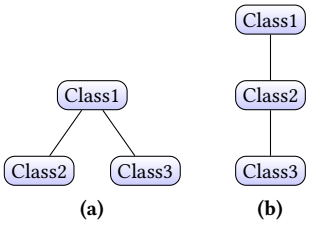

Figure 6. Possible class hierarchy structures.

stripped binary. We achieve our goal using a combination of structural analysis and behavioral analysis.

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. 6 are viable options.

Our structural analysis, described in Section 5, 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 [30].

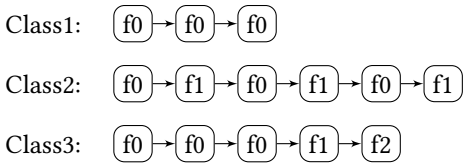

Figure 7. Usage sequences for instances of the different classes (extracted from the stripped binary).

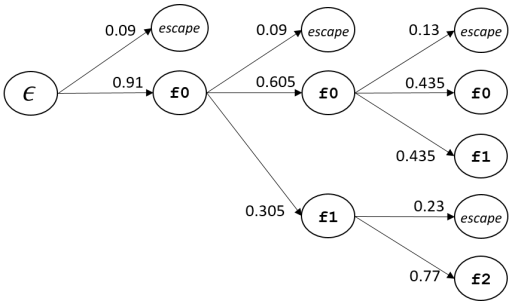

Figure 8. Trained statistical language model of `Class3`.

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 `use`*X* methods shown in Fig. 5 demonstrate the proper way to use each class *X*. Fig. 7 depicts the usage sequences extracted from the code of these methods.

A trained SLM (of depth 2) for `Class3` is depicted in Fig. 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 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. 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) [23].

[2]Object tracelets are formally defined in Section 3.2.
[3]The `escape` outcome is used as a *backoff* mechanism. See Section 3.1.

RIGHTSLINK

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 [41], evolution of languages [16], similarity of protein sequences [10], DNA sequences [48], 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 4.2.1). 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. 6a, which corresponds to the original hierarchy tree in Fig. 4, is indeed the most likely class hierarchy structure.

3 Background

In this section, we provide necessary background on statistical language models and how they can be used to represent potential runtime behaviors. Parts of this section summarize concepts presented by Katz et al. [21].

3.1 Statistical Language Models (SLM)

Given a finite alphabet Σ, a probabilistic model Pr trained on some training sequence $q_1^N = q_1 q_2 \cdots q_N$, such that $q_i \in \Sigma$, can be used to assign a probability to any single symbol $\sigma \in \Sigma$ given a past $s \in \Sigma^*$. This probability is denoted as $Pr(\sigma|s)$. Given a sequence $x_1^T = x_1 \cdots x_T$, the probability of the sequence can be computed using Pr as $Pr(x_1^T) = \Pi_{i=1}^{T} Pr(x_i|x_1 \cdots x_{i-1})$.

Given M 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.

Variable-order Markov Models In this work we use *variable-order Markov models* (VMMs). In a *fixed-order Markov model* of *order-k*, the conditional probability $Pr(\sigma|s)$ is assumed to equal $Pr(\sigma|s')$, where s' is a suffix of s of length k. VMM algorithms can adaptively determine the effective model order 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, referred to as variable-order n-gram models. We base these models on the *prediction by partial matching (PPM)* technique [12]. We use the variant PPM-C [29]; however, other methods, such as the well known Katz backoff model [11, 22], could also be applied.

Assuming a known upper bound D on the order of the model, a typical variable-order n-gram model is composed of several sub-models of orders $0 \leq k \leq D$. The model can revert to a lower-order model when the current model does not hold enough information. For each sequence s, to allow for symbols that did not appear after s in the training sequence to appear after s in test sequences, the model reserves a small probability mass known as *backoff*. Given a test sequence, its probability is computed using the following recursive relation:

$$Pr_k(\sigma|s = x_1^k) = \begin{cases} Pr_k(\sigma|s), & \text{if } s\sigma \text{ appeared in the training sequences;} \\ 1/|\Sigma|, & \text{if } |s| = 0 \\ Pr_k(backoff|s) \cdot Pr_{k-1}(\sigma|x_2^k), & \text{otherwise.} \end{cases}$$

Finally, we define $Pr(\sigma|s) = Pr_D(\sigma|s)$.

Training SLMs Given a set of sequences, training an SLM on these sequences consists of learning the probability of a symbol appearing in some context. The model learns this by reading all sequences and updating internal counters based on the currently read symbol and previous symbols (which determine the context). For example, given the sequences `aa` and `ab`, we can say that `a` is 100% likely to appear first in the sequence. We can also say that `a` is only 50% likely to appear after a previous `a`, meaning in the context set by `a`.

During training, the model essentially constructs a tree such that each level consists of all possible symbols and their associated probabilities. A context, or the sequence of previous symbols, is determined by the path in the tree used to reach the relevant node. Querying an SLM for the probability of a sequence becomes a traversal of the constructed tree. The model starts with an empty sequence, the root of the tree, and traverses the tree one symbol at a time, saving the relevant probability at each step. The probability of the sequence is the product of these probabilities.

3.2 Generating Type Usage Models

In this section, we briefly discuss the notion of *object tracelets*. Our discussion assumes that the technical aspects of analyzing a binary, locating objects and types, and determining events are known [21]. We only discuss the notion of *object tracelets* as used in our technique.

Tracelets and Events We represent the sequences of events applied to an object using *object tracelets* [21, 34], which capture the high-level usages of objects. Given a binary, we analyze its functions and maintain a set of explicit *object tracelets* for each (abstract) object. The events tracked are described in Table 1. Each object tracelet is a sequence of these events, which are used as our alphabet Σ.

Extracting Tracelets In the context of this work we are only interested in objects we can predetermine to belong to some type. We use the set of virtual tables (vtables) to represent the *explicit types* in the program, denoted as VT.

RIGHTSLINK

Event	Description
$C(i)$	Call to a virtual function at offset i in the object's virtual table
$R(i)$	Read from a field at offset i in the object
$W(i)$	Write to a field at offset i in the object
$this$	Object passed as `this` pointer to a function.
$Arg(i)$	Object passed as i-th argument to a function
ret	Object returned from called function
$call(f)$	A call to a concrete function f.

Table 1. Descriptions of the events tracked by our analysis

We use a static intra-procedural analysis to identify these objects. Our analysis relies on assignments of vtable addresses, as seen in object initialization/destruction, and on virtual functions, from which it can determine the object pointed to by the `this` pointer.

For each predetermined object o we create a symbolic value $\hat{o}$. An intra-procedural symbolic execution executes each function, tracking usages, accesses and actions applied to the symbolic values. The output for each symbolic value $\hat{o}$ is a set of sequences of events, each induced by a different symbolic execution path. These sequences, which were tracked on $\hat{o}$, are split into subsequences of limited length (up to length 7 in our experiments), which we refer to as the *object tracelets* of the object o, denoted as $OT(o)$.

We denote the set of tracelets for some type t as $TT(t)$ and define it as $TT(t) = \bigcup_{type(o)=t} OT(o)$.

Since our analysis and symbolic execution are entirely intra-procedural – we handle each procedure separately – they are inherently scalable. The number of procedures in a binary and the complexity of the call graph between procedures have no effect on our analysis.

The analysis time of our reported benchmarks (Section 6) was at most 2 hours per benchmark. Note that we can further scale our approach by parallelization, and, if necessary, trade off accuracy for scalability by extracting fewer and/or shorter traclets from each procedure.

Defining and building the SLMs. We define the SLMs' 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 SLM can parse and analyze. For each type t, we train an SLM instance using the tracelets from $TT(t)$. Training is conducted by inputting the tracelets to the model's learning algorithm one at a time, which generates a probabilistic model for this type, as explained in Section 3.1. The result is a trained SLM instance for each type that captures the behavior of objects of type t. We denote the SLM computed for binary type vt by $SLM(vt)$.

4 Behavioral Hierarchy Reconstruction

In this section, we formalize the problem of reconstructing class hierarchies from stripped binary code and present an approach to solve it using a static probabilistic technique.

4.1 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, 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.

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*.

RIGHTSLINK

4.2 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 4.1. A 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 3.2, 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.

4.2.1 Measuring Distance Between Types using Kullback-Leibler Divergence

Following [21], we model the behavior of binary types using statistical language models (See Section 3.2). 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 Σ, 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 4.1. 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.

4.2.2 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 4.1. 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.

RIGHTSLINK

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 $bin(P)$:

$$V = VT(P).$$

- 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:

$$E = \{vt_1 \to vt_2 \mid vt_1, vt_2 \in VT(P) \wedge vt_1 \neq vt_2\}.$$

- The weight of an edge $(vt_1 \to vt_2) \in E$ is defined as the distance between the behavioral models of vt_1 and vt_2:

$$w(vt_1 \to vt_2) = D_{KL}(SLM(vt_1) \parallel SLM(vt_2)).$$

A minimum-weight spanning arborescence for G_P constitutes a hierarchy tree that, according to Hypothesis 4.1, 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 5.

Finding a Minimum-Weight Spanning Arborescence To find a minimum-weight spanning arborescence we use the well-known Edmonds algorithm [15].

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 4.2. 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.

5 Pruning Infeasible Class Hierarchies via Structural Analysis

In this section, we describe the preprocessing phase of our analysis where we construct the *possible parent* relation, $\text{possibleParent}_P \subseteq VT(P) \times VT(P)$, over the binary types of $bin(P)$. Recall that $(vt_1, vt_2) \notin \text{possibleParent}_P$ 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 4) 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.

RIGHTSLINK

5.1 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_P^{DNA} 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_P^{DNA}.

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.

5.2 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. Th 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 ith 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.

5.3 Handling Multiple Inheritance

Under most ABIs, multiple inheritance is easy to detect from object layout. For example, in the MSVC [27] 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.

6 Evaluation

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

6.1 Benchmarks

We evaluated Rock over 19 stripped binaries built from open-source projects. We used 19 out of the 20 benchmarks previously used by Katz et al. [21]. 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 [26] (of size 21.6 Mb), but we do not report these results as we had no groundtruth to compare against. Furthermore, using these benchmarks also allowed us to reuse the framework and foundations already laid out by [21].

All benchmarks were built from source code as 32-bit binaries on a Windows machine using Microsoft's Visual Studio compiler [27], optimized and stripped, leaving no debug symbols in the binary.

RIGHTSLINK

Unlike [21], we filtered out compiler generated classes and hierarchies containing a single type. In these cases the hierarchy can be trivially reconstructed. Including these cases would only improve our results.

6.2 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 6.4.

6.3 Application Distance

Consider the example from Section 1 and the application presented in [21]: 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 [21] 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.

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 $successors_{GT}(t)$. 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, $successors_h(t)$. To evaluate the constructed hierarchy, we measure for each type t how many types from $successors_{GT}(t)$ are missing from $successors_h(t)$ 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).

Benchmark	size (Kb)	num of types	Without SLMs Missing	Without SLMs Added	With SLMs Missing	With SLMs Added
AntispyComplete	247	3	0.0	0.33	0.0	0.33
bafprp	529	23	0.3	0.0	0.3	0.0
cppcheck	97	6	0.0	0.0	0.0	0.0
MidiLib	400	20	0.0	0.0	0.0	0.0
patl	36.5	4	0.0	0.0	0.0	0.0
pop3	24	2	0.0	0.0	0.0	0.0
smtp	26	2	0.0	0.0	0.0	0.0
tinyxml	60	9	0.89	0.0	0.89	0.0
tinyxmlSTL	88	15	0.6	0.27	0.6	0.27
yafe	68	15	0.0	0.2	0.0	0.2
Analyzer	419	24	0.21	6.79	0.25	1.38
CGridListCtrlEx	151	28	0.0	0.46	0.07	0.07
echoparams	58	4	0.0	2.25	0.0	0.0
gperf	84	10	0.0	3.8	0.0	0.5
libctemplate	1233	36	0.25	0.33	0.25	0.11
ShowTraf	137	25	0.04	0.4	0.04	0.08
Smoothing	453	31	0.19	7.9	0.23	1.1
td_unittest	101	2	0.0	1.0	0.0	0.5
tinyserver	46	4	0.0	2.25	0.0	0.25

Table 2. Application Distance from H_P. Structurally resolvable benchmarks are above the line and below it are unresolvable benchmarks.

6.4 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 2. For the remainder of this section we will focus on the 9 benchmarks that were not structurally resolved.

Table 2 presents the results of our evaluation using the application distance from Section 6.3. 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

RIGHTSLINK

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. 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

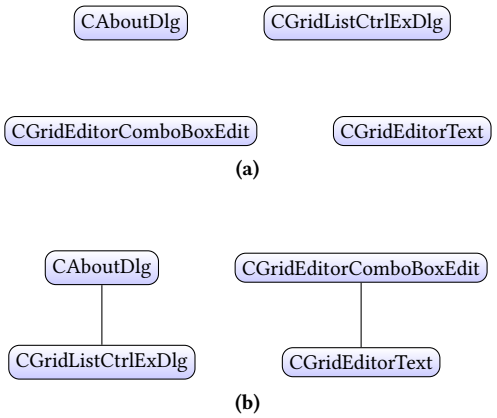

Figure 9. Class hierarchies for CGridListCtrlEx.exe: (a) ground truth, and (b) reconstructed hierarchy.

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

RIGHTSLINK

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).

7 Related Work

Much work has been devoted to reverse engineering and binary analysis. In this section we briefly survey closely related work.

Reconstructing Class Hierarchies. Srinivasan and Reps [44, 45] 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, 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. [17] 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. [30] 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.

Analysis of Executables. Reps et al. [37, 38] explored many aspects of the analysis of stripped binaries, and obtained impressive results during years of work on the problem. Their analyses (e.g.,[3, 35, 36]) 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 analyses, which have been previously discussed and used on binaries and assembly code in several works, such as [2, 5, 7, 14, 21, 37].

Balakrishnan et al. [4] and Reps et al. [36] presented tools that assist reverse engineering; however, neither tackled the problem of recovering C++ class hierarchies.

Brumley et al. [8] presented a flexible platform for the static analysis of binaries and investigated a wide range of interesting problems, including decompilation [43] and identifying functions in binaries [6].

Bao et al. [6] use machine learning to identify function boundaries. The authors recognize as we do that a statistical approach is more useful and accurate than standard approaches in certain settings.

Many past works have tackled the problem of type reconstruction (see [9] for a recent survey). This problem is conceptually similar to our problem. However, we do not try to recreate type structures. Instead our goal is to identify type hierarchies without referring to the type structure.

Behavioral Patterns. Preda et al. [32], Warrender et al. [47] and Mazeroff et al. [25] 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. [18] broaden this notion beyond malware classification and discuss matching binaries with behavior specifications. In our work, we use statistical language models that capture type behaviors as a basis for type similarity and class hierarchy reconstruction.

David et al. [13] use control tracelets to find similar code fragments across stripped binaries. Their tracelets track instructions and are not suited to tracking the behavior of types. Katz et al. [21] present the notion of object tracelets that track method calls and field access events for objects

RIGHTSLINK

and types. We borrow their notion of object tracelets and the representation of types using statistical language models.

Polino et al. [31] 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. [28] 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. [34] 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. Several techniques have used structural similarity to estimate types. Madsen et al. [24] 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 Madsen et al. [24] dealt with JavaScript programs, Tu [46] suggests a similar technique for Python programs in a tool called *Mino*. 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 tools like *Mino*, reconstruction of a single type hierarchy is not reliable enough, as we demonstrated.

Statistical Techniques. Jha et al. [20] use 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 class hierarchy reconstruction.

Raychev et al. [33] presented a statistical approach to predicting program properties using CRFs. Their technique leverages program structure to create dependencies and constraints used for probabilistic reasoning. Unfortunately, when dealing with stripped binaries, this approach is not applicable since there is much less program structure to work with. Our approach works without the need to recover a lot of structure.

Schütze and Singer [42] 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 ours, potentially supporting the suitability of VMMs to our problems.

8 Conclusion

We address the problem of *reconstructing type hierarchies* of stripped binaries, a fundamental problem of reverse engineering. Given a standard stripped binary b, compiled from object oriented code, we find the most likely type hierarchy for b.

We present a technique that relies on *behavioral similarity* to measure the likelihood that type t_1 is derived from type t_2 and derive a type hierarchy based on that likelihood. We use SLMs that summarize type usages and we use the Kullback-Leibler divergence to measure distance between these SLMs. We base our technique on the hypothesis that, since related types are often used in similar ways, the distance between SLMs of related types will be smaller than that of unrelated types.

We show that our technique is capable of recreating the likely type hierarchy of a real-world stripped binary even when existing techniques cannot do so. We implemented our approach in a tool called Rock. We show that the hierarchies created by our tool can be used to provide practical, useful answers to key questions of reverse engineers.

Acknowledgments

The research leading to the results presented in this paper 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.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the Conference on Computer and Communications Security.*

[2] Wolfram Amme, Peter Braun, François Thomasset, and Eberhard Zehendner. 2000. Data Dependence Analysis of Assembly Code. In *International Journal Parallel Programming* (2000).

[3] Gogul Balakrishnan and Thomas Reps. 2007. DIVINE: Discovering Variables in Executables. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation.*

[4] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. In *ACM Transactions on Programming Languages and Systems* (2010).

[5] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems.*

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security Symposium.*

[7] J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. 1999. Static Analysis of Binary Code to Isolate Malicious Behaviors. In *Proceedings of the Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises.*

[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the International Conference on Computer Aided Verification.*

[9] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. In *ACM Computing Surveys* (2016).

RIGHTSLINK

[10] John A Capra and Mona Singh. 2007. Predicting functionally important residues from sequence conservation. In *Bioinformatics* (2007).
[11] Stanley F Chen and Joshua Goodman. 1996. An empirical study of smoothing techniques for language modeling. In *Proceedings of the Annual Meeting on Association for Computational Linguistics*.
[12] John G Cleary and Ian H Witten. 1984. Data compression using adaptive coding and partial string matching. In *IEEE Transactions on Communications* (1984).
[13] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[14] Saumya Debray, Robert Muth, and Matthew Weippert. 1998. Alias Analysis of Executable Code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[15] Jack Edmonds. 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards B* (1967).
[16] Ran El-Yaniv, Shai Fine, and Naftali Tishby. 1998. Agnostic Classification of Markovian Sequences. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*.
[17] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *Proceedings of the Working Conference on Reverse Engineering*.
[18] Matthew Fredrikson, Mihai Christodorescu, and Somesh Jha. 2011. Dynamic Behavior Matching: A Complexity Analysis and New Approximation Algorithms. In *Proceedings of the International Conference on Automated Deduction*.
[19] Denis Gopan, Evan Driscoll, Ducson Nguyen, Dimitri Naydich, Alexey Loginov, and David Melski. 2015. Data-delineation in Software Binaries and Its Application to Buffer-overrun Discovery. In *Proceedings of the International Conference on Software Engineering*.
[20] S. Jha, K. Tan, and R.A. Maxion. 2001. Markov Chains, Classifiers, and Intrusion Detection. In *Proceedings of the IEEE workshop on Computer Security Foundations*.
[21] Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Binaries Using Predictive Modeling. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[22] Slava M Katz. 1987. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech, and Signal Processing* (1987).
[23] S. Kullback and R. A. Leibler. 1951. On Information and Sufficiency. In *The Annals of Mathematical Statistics* (1951).
[24] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*.
[25] Geoffrey Mazeroff, Jens Gregor, Michael Thomason, and Richard Ford. 2008. Probabilistic suffix models for API sequence analysis of Windows XP applications. In *Pattern Recognition* (2008).
[26] Microsoft Corporation. [n. d.]. Skype. https://www.skype.com/en/. ([n. d.]).
[27] Microsoft Corporation. [n. d.]. Visual Studio. https://www.visualstudio.com. ([n. d.]).
[28] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*.
[29] Alistair Moffat. 1990. Implementing the PPM data compression scheme. In *IEEE Transactions on Communications* (1990).
[30] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs. In *Network and Distributed System Security Symposium*.
[31] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. 2015. Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.
[32] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. 2007. A Semantics-based Approach to Malware Detection. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[33] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.
[34] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[35] Thomas Reps and Gogul Balakrishnan. 2008. Improved Memory-Access Analysis for x86 Executables. In *Proceedings of the International Conference on Compiler construction*.
[36] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. 2006. Intermediate-representation Recovery from Low-level Code. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*.
[37] Thomas Reps, Gogul Balakrishnan, Junghee Lim, and Tim Teitelbaum. 2005. A Next-Generation Platform for Analyzing Executables. In *Proceedings of the Third Asian conference on Programming Languages and Systems*.
[38] Thomas Reps, Junghee Lim, Aditya Thakur, Gogul Balakrishnan, and Akash Lal. 2010. There's plenty of room at the bottom: analyzing and verifying machine code. In *Proceedings of the International Conference on Computer Aided Verification*.
[39] Ronald Rosenfeld. 2000. Two decades of statistical language modeling: Where do we go from here? In *Proceedings of the IEEE* (2000).
[40] Paul Vincent Sabanal and Mark Vincent Yason. 2007. Reversing C++. In *BlackHat USA*.
[41] Dominik Schnitzer. [n. d.]. Musly: Audio Music Similarity. http://www.musly.org. ([n. d.]).
[42] Hinrich Schütze and Yoram Singer. 1994. Part-of-speech Tagging Using a Variable Memory Markov Model. In *Proceedings of the Annual Meeting on Association for Computational Linguistics*.
[43] Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium* (2013).
[44] Venkatesh Srinivasan and Thomas Reps. 2014. Recovery of Class Hierarchies and Composition Relationships from Machine Code. In *In Proceedings of the International Conference on Compiler construction*.
[45] Venkatesh Karthik Srinivasan and Thomas Reps. 2013. Software-Architecture Recovery from Machine Code*. (2013). https://minds.wisconsin.edu/handle/1793/65091.
[46] Stephen Tu. 2012. MINO: Data-driven type inference for Python. (2012). http://people.csail.mit.edu/stephentu/papers/mino.pdf.
[47] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. 1999. Detecting Intrusions Using System Calls: Alternative Data Models. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[48] Golan Yona and Michael Levitt. 2002. Within the twilight zone: a sensitive profile-profile comparison tool based on information theory. In *Journal of Molecular Biology* (2002).

RIGHTSLINK