Estimating Types in Binaries using Predictive Modeling

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

Ran El-Yaniv
Technion, Israel
rani@cs.technion.ac.il

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

Abstract

Reverse engineering is an important tool in mitigating vulnerabilities in binaries. As a lot of software is developed in object-oriented languages, reverse engineering of object-oriented code is of critical importance. One of the major hurdles in reverse engineering binaries compiled from object-oriented code is the use of dynamic dispatch. In the absence of debug information, any dynamic dispatch may seem to jump to many possible targets, posing a significant challenge to a reverse engineer trying to track the program flow.

We present a novel technique that allows us to statically determine the likely targets of virtual function calls. Our technique uses object tracelets – statically constructed sequences of operations performed on an object – to capture potential runtime behaviors of the object. Our analysis automatically pre-labels some of the object tracelets by relying on instances where the type of an object is known. The resulting type-labeled tracelets are then used to train a statistical language model (SLM) for each type. We then use the resulting ensemble of SLMs over unlabeled tracelets to generate a ranking of their most likely types, from which we deduce the likely targets of dynamic dispatches. We have implemented our technique and evaluated it over real-world C++ binaries. Our evaluation shows that when there are multiple alternative targets, our approach can drastically reduce the number of targets that have to be considered by a reverse engineer.

Categories and Subject Descriptors F.3.2(D.3.1)[Semantics of Programming Languages: Program analysis]; D.3.4 [Processors: compilers, code generation];

Keywords x86; static binary analysis; reverse engineering

1. Introduction

New software is released daily. Most software is delivered in binary form, without sources. To check that critical software is free from vulnerabilities and back-doors, it is often manually inspected by experts who try to understand how it works. Those experts gain an understanding of the binary via the difficult and tedious process of reverse engineering.

When reverse engineering a binary, the main goal is to understand the control and data flow of the program. This task is made more difficult when dealing with binaries originating from object-oriented code [49]. One significant challenge is indirect calls to dynamically computed targets which hide the target of a call that will be reached at runtime. Finding the right target typically requires the reverse engineer to manually examine each one of the possible targets, a process that is expensive and time consuming. The goal of this work is to assist the reverse engineering process 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 *statically* infer the likely targets for each indirect call site in the binary. Since the target of a virtual call is directly determined by the type of the object used in the indirect call, this is done by identifying the likely types of the objects.

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) [48], 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.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837674

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., [7, 45, 47]). A lot of past work has focused on identifying variables and aggregate type information [5] 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 4 for details). A similar problem was previously discussed in [4] but with poor results.

We draw some inspiration from classic work on reconstruction of aggregate types in COBOL [41]. 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 paper 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. Overview

In this section, we provide an informal overview of our approach. We use a simple example for illustrative purposes. More realistic examples can be found in Section 6.

***Type as a* statistical model *of* behaviors.** In this paper 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.

Such object and type descriptions can be used to answer difficult questions in a statistical manner, using classification based on our trained models. In this paper we discuss an application of our approach to determining 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 [14, 28], 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 paper refers to the low-level concept of a `vtable` created for each class by the compiler, and a 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. 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. We assume the function was compiled using Microsoft's Visual Studio compiler [2] and uses the application binary interface (ABI) set by the compiler for 32-bit x86 binaries (see Section 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, `[esp+4]` (corresponds to the `Socket` argument) and `[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, `[esp+4]` in this case.

```
1  class BasicSocket { // socket interface
2      virtual void connect() = 0;
3      virtual void receive() = 0;
4      virtual void close() = 0;
5      int errorCode, address;
6  };
7  class MySocket : public BasicSocket{
8      virtual void send(int n);
9      virtual void sendDefault();
10     int msgCounter;
11 };
12 class MinimalSocket : public BasicSocket {
13     virtual void send();
14 };
15 class OneWaySocket : public BasicSocket {
16     int msgCounter;
17
18 };
19 class MyFile {
20     virtual void close();
21     virtual void open();
22     virtual void seek(int n);
23     virtual void remove();
24     virtual char* read();
25     virtual void write(char* str);
26     char *name, current, *path;
27     int size;
28     MyFile* parent;
29 };
30
31 int sendInt(MySocket* s, int x)  {
32     s->msgCounter = 0;
33     s->connect();
34     if (x) {
35         s->send(x);
36     } else {
37         s->sendDefault();
38     }
39     return s->errorCode;
40 }
41
42 int readLast(char* data)  {
43     MyFile* f = new MyFile();
44     f->open();
45     f->seek(f->size);
46     return f->current;
47 }
```

Figure 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.

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 $Types(b)$ in b. For each $t \in Types(b)$, $instances(t)$ is a set of objects of type t (see Section 4.2 for details). From $instances(t)$ we extract $TT(t) = tr_1, tr_2, \ldots$, a set of tracelets for t, known as "type tracelets," such that $tr_i = \sigma_{i,1}, \sigma_{i,2}, \ldots$ and $\sigma_{i,j} \in \Sigma$, as defined in Section 3.2.
2. We build a set of statistical language models $\{M(t) | t \in Types(b)\}$ by training $M(t)$ on the corresponding $TT(t)$.
3. For each $o \in Objects(b)$ we extract $OT(o)$, a set of tracelets for object o, known as "object tracelets."

```
1       mov eax, [esp+4]
2       mov [eax+16], 0
3       mov eax, [esp+4]
4       mov edx, [eax]
5       mov ecx, [esp+4]
6       mov eax, [edx]
7       call eax            ; virtual call
8       cmp [esp+8], 0
9       jz branch
10      mov eax, [esp+8]
11      push eax
12      mov eax, [esp+4]
13      mov edx, [eax]
14      mov ecx, [esp+4]
15      mov eax, [edx+12]
16      call eax            ; virtual call
17      jmp merge
18  branch:
19      mov eax, [esp+4]
20      mov edx, [eax]
21      mov ecx, [esp+4]
22      mov eax, [edx+16]
23      call eax            ; virtual call
24  merge:
25      mov eax, [esp+4]
26      mov eax, [eax+4]
```

Figure 2: Assembly code generated for function `sendInt`. Lines 7,16,23 match the function invocations at lines 33,35,37 of Fig. 1.

4. For each $o \in Objects(b)$ we rank all $t \in Types(b)$ according to $P_t(o) = \prod_{s \in OT(o)} P_t(s)$ 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 there is a single object of interest, `[esp+4]`, referred to henceforth as o_1 (identification of objects is described in Section 5.2):

- The register `eax` in lines 3, 12 and 19 points to o_1.
- The register `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.
- The register `eax` in lines 6, 15 and 22 eventually points to an entry in the virtual table of o_1, and that entry is the call target.

Statically Tracking Events for Objects of Interest. We focus on object o_1. Because `eax` in line 1 points to o_1, we can determine that the "`mov [eax+16],0`" instruction in line 2 assigns the value 0 to a field of o_1. Similarly, we statically determine that, since `eax` in line 25 also points to o_1, the "`mov eax,[eax+4]`" instruction in line 26 reads a value from a field of o_1. 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 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. 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:

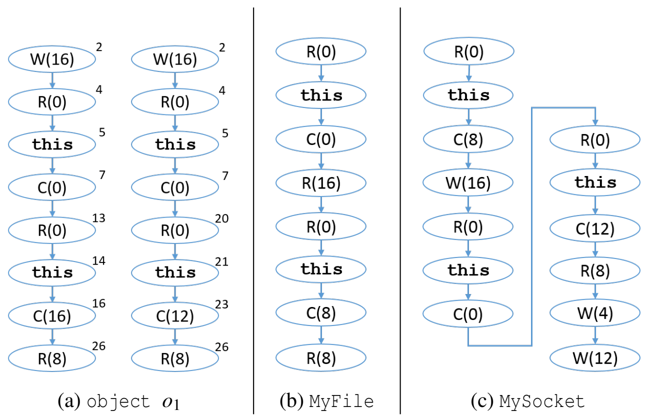


Figure 3: Example tracelets for object o_1 extracted from function `sendInt`, type `MyFile` from function `readLast`, and type `MySocket`.

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. *this* – object was used as *this* pointer

Additional kinds of events are explained in Section 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 `readLast` in Fig. 1 is part of the same binary as `sendInt`. This function yields the x86 assembly code of Fig. 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 `sendInt`, from this function we can extract the tracelet in Fig. 3b for the object represented by the value of `[f]`. Because `[f]` holds the return value of the constructor of type `MyFile`, we know that `[f]` is of type `MyFile` and we associate the tracelet in Fig. 3b with type `MyFile` (See Section 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. 3c shows an example tracelet collected for the type `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 4.3.

Since we are looking to find o_1's actual allocated type, we can immediately eliminate `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 o_1's type. Consider the set of tracelets we extracted for the object o_1. 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.

```
 1  push   16
 2  call   new
 3  mov    ecx, eax
 4  call   MyFile::MyFile
 5  mov    ecx, eax
 6  mov    [f], ecx
 7
 8  mov    eax, [f]
 9  mov    edx, [eax]
10  mov    ecx, [f]
11  mov    eax, [edx]
12  call   eax                    ; virtual call
13  mov    eax, [f]
14  mov    ecx, [eax+16]
15  push   ecx
16  mov    edx, [f]
17  mov    eax, [edx]
18  mov    ecx, [f]
19  mov    edx, [eax+8]
20  call   edx                    ; virtual call
21  mov    eax, [f]
22  mov    eax, [eax+8]
```

Figure 4: Assembly code generated for function `readLast`. Lines 12,20 match function invocations at lines 44,45 of Fig. 1.

We use this knowledge to determine that the types `MinimalSocket` and `OneWaySocket` cannot be candidates for o_1's type.

From now on, we focus only on the likely candidates of o_1'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 [15].

We compute the probability that each of o_1's tracelets from Fig. 3a originated from the resulting models (according to the formulas in Section 4.1) and multiply the results to get a score for the entire set (as explained in Section 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. 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 are the relevant implementations of `MySocket`'s functions.

3. Object Tracelets

In this section, we discuss the notion of object tracelets. 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.

This section assumes the technical aspects of analyzing the binary, locating objects and types, and determining events are known. In this section 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 5.

Table 1: Descriptions of the events tracked by our analysis

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. 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.
$Arg(i)$	Object passed as i-th argument to a function
ret	Object returned from called function
$call(f)$	A call to a concrete function f. 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 `this` pointer.

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.

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 [17, 43]) 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 1. These events are used as our alphabet Σ. 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.

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. In that way we use all the knowledge available to us from the binary. We evaluated whether including the address of the concrete function has an affect on our results. Experiments on a sample of our benchmarks showed the effects to be mostly marginal.

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 4.2).

Object Tracelets of `MySocket` instance. We examine the function `sendInt` from Fig. 1. This function deals with the object `s` of type `MySocket`. From our symbolic execution for object `s` we get two object 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. 3a.

4. Classification by SLMs

In Section 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 4.1. In Section 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 4.3, we use VMMs to build a ranking of possible types for each object.

4.1 Statistical Language Models

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

Let Σ 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, $P(x_1^T) = \Pi_{i=1}^T P(x_i|x_1 \cdots x_{i-1})$. 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.

A bottleneck when considering a fixed order model is that overestimating or underestimating the most 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 4.3.

In a *fixed-order Markov* model over some finite alphabet Σ, 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[1] [15].

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 the 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 $P_k(backoff|s)$ for all symbols that did not appear after context s in the training sequence, and would require applying the backoff mechanism. Thus, the backoff mechanism is constructed to satisfy the following recursive relation,

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

For the final model we define $P(\sigma|s) = P_D(\sigma|s)$ and $P(\sigma|\varepsilon) = 1/|\Sigma|$, where ε 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 [37], but quite a few other methods could be used instead, including Katz's well-known backoff model [14, 28]. Since our approach does not depend on a specific VMM algorithm, it could benefit from integrating more advanced algorithms (e.g., PAQ8 [34]).

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.

[1] In the context of PPM-related literature, "backoff" is termed "escape".

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 $functions(vt)$ to be the number of functions in the vtable. For each address $vt \in VT$, $size(vt)$ captures the allocation size of objects that use this vtable (determining the size is explained in Section 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 $ltypes\colon Objs \to \mathcal{P}(VT \times \mathbb{R})$, 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 $etypes\colon Objs \to VT$, 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 $TT\colon VT \to \mathcal{P}(\Sigma^*)$, 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, $TT(vt) = \bigcup_{etype(o)=vt} OT(o)$.

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 previous actions applied to the 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. 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 evidentally 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 [31] 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.

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 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 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:

$$\max(fields(ot)) \geq size(vt) \; or$$
$$\max(functions(ot)) \geq functions(vt).$$

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.

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 [51], proteomics [11], web query recommendations [24], music analysis [39], and behaviometric identification [38]. 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 [17].

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 [16, 19, 32, 50] and Fisher kernels [25].

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 [54] have been recently used successfully in a related problem of predicting program properties [42].

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.

5. Prototype Implementation

In this section we describe our implementation and the different steps it consists of.

5.1 Base Analysis

Our technique is based on the product of a static intra-procedural Steensgaard-style points-to analysis [53]. 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 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 [6, 22] 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 [2]. 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 [1] 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.

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 `[[OBJECT]+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.

Table 2: Identification of tracked events by our analysis

Event	Explanation
$C(i)$	Matching the call target to the virtual function call pattern described in Section 5.2
$R(i)$	A field of the object is used as the right operand of an operation
$W(i)$	A field of the object is used as the left operand of an operation (such as `mov`)
this	`ecx` pointing to the object at the location of a call operation.
$Arg(i)$	Finding the object on the stack as maintained by our analysis at the location of a call operation
ret	Matching return values in registers (`eax`) to an object
$call(f)$	A `call` command to a known address

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

$$Objs := Regs \mid \mathbb{Z} \mid [Objs] \mid (Objs\ op\ Objs)\ ,$$

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 4.2), we go over the states and search for field access patterns. The field access pattern is `[OBJECT+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.

5.3 Identifying Events

Table 2 explains for each of the events from Table 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 5.1.

5.4 Applying Analysis

Consider lines 19-23 from Fig. 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` $\rightarrow$ `[[ESP+4]]`
- `ecx` $\rightarrow$ `[ESP+4]`
- `eax` $\rightarrow$ `[[[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 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*.

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.

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. We identify the address of the `delete` operator (in stripped binaries as well), and by backtracking up the call hierarchy leading to it, we separate the constructors from the destructors.

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. 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 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.

6. Experimental Evaluation

In this section we describe the benchmarks we used to evaluate our technique and present our results.

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 [2] as release builds (which are optimized and stripped by default).

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

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 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 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.

6.3 Results

Table 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. 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

Table 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.

Benchmark	Statistics						Remaining Targets				
	binary size (Kb)	# types	# objects	# calls	total length of training data	area under curve	min	max	mean	median	variance
cppcheck.exe	97	12	1	1	2057	84.6	2	2	2	2	0
patl.exe	36.5	7	1	1	351	75.0	2	2	2	2	0
pop3.exe	24	5	1	1	12478	83.3	1	1	1	1	0
smtp.exe	26	5	1	1	12442	83.3	1	1	1	1	0
td_unittest.exe	101	7	1	1	21246	75.0	2	2	2	2	0
template_regtest.exe	69	4	1	1	3719	80.0	1	1	1	1	0
AntispyComplete.exe	247	6	4	5	13745	85.7	1	1	1	1	0
tinyserver.exe	46	12	4	6	6280	89.2	1	2	1.4	1	0.24
echoparams.exe	58	12	6	7	15840	89.0	1	2	1.43	1	0.24
ShowTraf.exe	137	38	6	7	171967	76.6	1	19	9.00	10	29.14
yafe.exe	68	26	5	7	33563	92.0	1	3	2.14	3	0.98
bafprp.exe	529	32	7	8	2285	67.0	1	21	10.88	11	59.11
tinyxmlSTL.dll	88	75	14	17	4960	93.1	1	19	5.18	4	24.26
gperf.exe	84	11	10	21	41085	90.8	1	3	1.10	1	0.18
MidiLib.dll	400	96	7	26	6301	98.0	1	9	1.88	1.5	2.56
tinyxml.dll	60	51	36	37	4449	82.7	1	42	8.89	4	128.91
libctemplate.dll	1233	229	39	60	17967	84.3	1	165	35.28	12	2247.22
Analyzer.exe	419	62	78	64	97278	86.8	1	34	8.13	9	34.48
Smoothing.exe	453	71	130	306	112027	97.5	1	25	1.71	1	5.76
CGridListCtrlEx.exe	151	35	220	406	163390	95.9	1	18	1.40	1	2.90

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 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. 5 give a visual representation of the number of functions that need to be examined for each call site. Given a ranking $t_1, t_2, ..., t_n$ of types generated by our technique, we select a value x for the x–axis, and examine the top x targets from the ranking, $t_1, ...t_x$. The graphs show the percentage out of all call sites in the benchmark for which $t_1, ...t_x$ 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 3 we can see the percentage of the graph that is under the coverage curve; higher percentages mean better results.

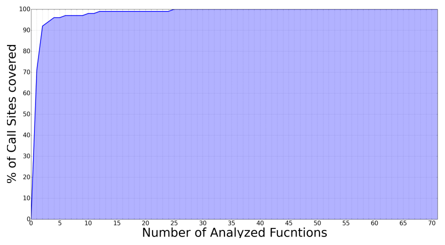
The values of the x–axis vary between different graphs. The top value of the x–axis is set according to the maximum number of functions that needed to be examined before our technique was used, meaning that the top value of the x-axis is determined by the case in which all the functions need to be examined.

`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.

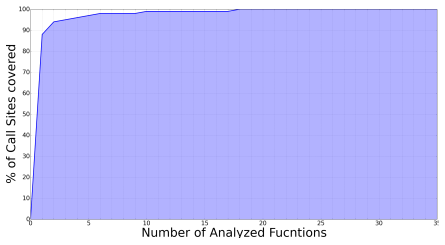
We find a correlation between the sizes of the training sets used to train the SLMs and the quality of our results. As can be expected, our results improve as the size of our training set increases. In Fig. 6 we see the distribution of types by their training set size for the benchmarks `Smoothing.exe` 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. 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 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 [10, 29]). Our edit-distance metric was based on the well-known Levenshtein-Damarau distance extended to sets of tracelets using min weight matching.

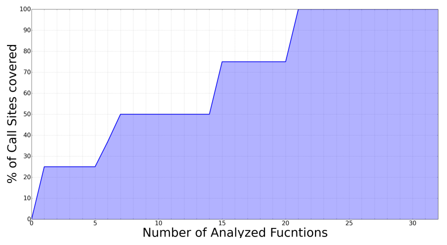
Fig. 7 shows the result of these methods on the `Smoothing.exe` benchmark. The figure is zoomed in to focus on the area of difference between the coverage curves. The caption includes the percentage of the full graph that is under each coverage curve. In this figure, we see that a reverse engineer using our VMM-based approach will only have to consider the 2 top ranked targets to find



(a) `Smoothing.exe`

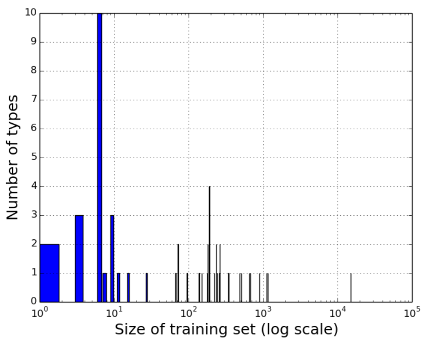


(b) `CGridListCtrlEx.exe`

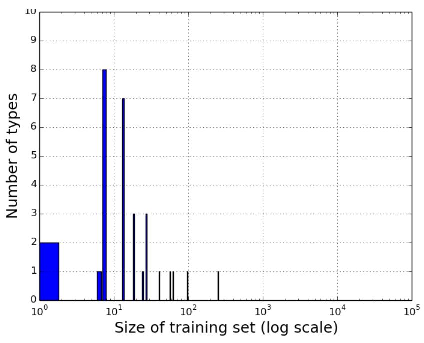


(c) `bafprp.exe`

Figure 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.



(a) `Smoothing.exe`



(b) `bafprp.exe`

Figure 6: Histogram of types according to training set size for 2 sample benchmarks

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).

We have discussed our technique with professional reverse engineers from the industry and the preliminary feedback was very enthusiastic.

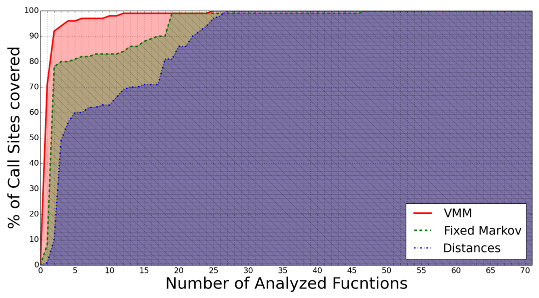


Figure 7: Comparison of coverage curves for benchmark `Smoothing.exe` using different ranking methods, zoomed in on area of difference; the VMM curve covers 95.5% of the figure, while the fixed-order model covers 93.1% and the edit-distance covers 87.2%

7. Related Work

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

Analysis of Executables. Reps et al. [46, 47] explored many aspects in the analysis of stripped binaries, and obtained impressive results during years of work on the problem. Their analyses (e.g.,[5, 44, 45]) can recover a lot of information from a stripped binary, and statically verify challenging safety properties. In [44], 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 variables, but instead of the standard abstract values common in VSA, we use value expressions generated by our points-to analysis. The work in [44] 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 [44] is more complex than what we require and we use instead a simpler, more targeted analysis, as explained in Section 5.1.

Balakrishnan et al. [7] and Reps et al. [45] presented tools that assist reverse engineering; however, neither tackled the problem of recovering C++ object types.

The problem of finding object boundaries in stripped binaries is generally a hard problem. Recent work by Gopan et al. [21] 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 5.6. Furthermore, we do not attempt to find all object boundaries and instead use virtual calls as our basis for identifying objects.

Brumley et al. [13] presented a flexible platform for the static analysis of binaries, and investigated a wide range of interesting problems, including decompilation [52] and identifying functions in binaries [9]. The technique presented in [9] 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.

Points-to and aliasing analyses have been previously discussed and used on binaries and assembly code in several works, such as [3, 8, 12, 18, 46]. This kind of analysis is also used as the basis for our work.

Lee et al. [30] presented a tool called TIE aimed at the related problem of inferring primitive types for variables in a binary. 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 our setting.

Preda et al. [40], Warrender et al. [56] and Mazeroff et al. [35] have all proposed using behavioral signatures to detect malware. They based their approaches on dynamically tracking events, mostly API and system calls. They show that behavioral patterns can identify intent in binaries. Bjorner et al. [20] 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.

Another related work, [17], 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.

We note that many previous works have tackled the problem of type reconstruction, for example [23]. The problem of type reconstruction is conceptually similar to our problem. However, we do not try to recreate type structures. Instead, our goal is to identify types of objects without referring to the type structure.

Statistical Techniques. Markov models have previously been used to classify behaviors. Jha et al. [27] used Markov models for intrusion detection. This work depends on identifying sequences of actions that, in other scenarios, would be improbable. We extend this notion and rely on differences between sequence probabilities across contexts to employ Markov models for type classification.

Raychev et al. [42] 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 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 [42], 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.

Mishne et al. [36] 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. [43] 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. In both works histories are easy to extract. In contrast, our approach requires an involved analysis to construct object tracelets.

Jang et al. [26] 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 problem we are facing 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.

Structural Similarity. Madsen et al. [33] suggested a technique based on "structural similarity" of types and objects. Their technique uses the structure of objects and the names of fields and functions to find a matching type. Unfortunately, these approaches do not work in a stripped binaries setting because the binaries don't contain any names that can be used to latch on to and eliminate types. Instead we use "behavioral similarity," which takes into consideration how the object is used rather than only what it can do (i.e, what fields and functions the object has). While [33] dealt with JavaScript programs, the technique presented in [55] 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 [55] 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 different kinds of objects. The authors assume that, given field/method names, deduction of object types is easy, but in our setting this kind of data simply doesn't exist. Without the explicit names utilized by *Mino*, classification of a single type/target isn't reliable enough, as demonstrated by our results.

8. Conclusion

We address the problem of *resolving targets* of dynamic dispatches in binaries. Given a standard stripped binary compiled from object-oriented code, we *statically* infer the likely targets for virtual function calls in the binary, by inferring the likely types for the objects used in the call.

We present a technique that uses *object tracelets*—statically constructed sequences of operations performed on an object—to approximately capture the runtime behaviors of the object. We rely on occurrences where the type of an object is known to automatically pre-label some of the tracelets. We then train SLMs for each type in the binary and use the trained SLMs to measure the likelihood that a tracelet originated from an SLM. We consider objects whose set of tracelets is highly likely to have originated from a type's model as belonging to that type. Using the likelihood probabilities calculated by our model, we rank the most likely types of an object, and deduce from that the likely targets of virtual calls.

We show that object tracelets can accurately determine the types of objects, and thus the targets of virtual function calls, even in real-world binaries where extensive optimization and stripping have been applied and no debug information is present. We implemented our approach and evaluated it over real-world stripped binaries. We show that in over 80% of virtual call sites we can drastically reduce the number of likely targets to fewer than 3 targets per call site.

Acknowledgements

The authors would like to thank Nimrod Partush and Yaniv David for commenting on an earlier version of this paper. The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 - ERC-COG-PRIME.

References

[1] Hex-rays interactive disassembler (ida) pro. https://www.hex-rays.com/products/ida/.

[2] Microsoft corporation. visual studio. https://www.visualstudio.com.

[3] AMME, W., BRAUN, P., THOMASSET, F., AND ZEHENDNER, E. Data dependence analysis of assembly code. *Int. J. Parallel Program. 28*, 5 (Oct. 2000), 431–467.

[4] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of c++ virtual function callsIn *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (1996), OOPSLA '96, ACM, pp. 324–341.

[5] BALAKRISHNAN, G., AND REPS, T. Divine: Discovering variables in executables. In *Verification, Model Checking, and Abstract Interpretation*, B. Cook and A. Podelski, Eds., vol. 4349 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 1–28.

[6] BALAKRISHNAN, G., AND REPS, T. Analyzing stripped device-driver executables. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 124–140.

[7] BALAKRISHNAN, G., AND REPS, T. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst. 32*, 6 (Aug. 2010), 23:1–23:84.

[8] BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. Thorough static analysis of device driversIn *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006* (2006), EuroSys '06, ACM, pp. 73–85.

[9] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary codeIn *23rd USENIX Security Symposium (USENIX Security 14)* (Aug. 2014), USENIX Association, pp. 845–860.

[10] BEGLEITER, R., AND EL-YANIV, R. Superior guarantees for sequential prediction and lossless compression via alphabet decomposition. *J. Mach. Learn. Res. 7* (Dec. 2006), 379–411.

[11] BEJERANO, G., AND YONA, G. Variations on probabilistic suffix trees: statistical modeling and prediction of protein families. *Bioinformatics 17*, 1 (2001), 23–43.

[12] BERGERON, J., DEBBABI, M., ERHIOUI, M. M., AND KTARI, B. Static analysis of binary code to isolate malicious behaviorsIn *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises* (1999), WETICE '99, IEEE Computer Society, pp. 184–189.

[13] BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. Bap: A binary analysis platform. In *Computer Aided Verification*, vol. 6806 of *Lecture Notes in Computer Science*. Springer, 2011, pp. 463–469.

[14] CHEN, S. F., AND GOODMAN, J. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics* (1996), Association for Computational Linguistics, pp. 310–318.

[15] CLEARY, J. G., AND WITTEN, I. H. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on 32*, 4 (1984), 396–402.

[16] CUTURI, M., AND VERT, J.-P. The context-tree kernel for strings. *Neural Networks 18*, 8 (2005), 1111–1123.

[17] DAVID, Y., AND YAHAV, E. Tracelet-based code search in executablesIn *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, ACM, pp. 349–360.

[18] DEBRAY, S., MUTH, R., AND WEIPPERT, M. Alias analysis of executable codeIn *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1998), POPL '98, ACM, pp. 12–24.

[19] ESKIN, E., WESTON, J., NOBLE, W. S., AND LESLIE, C. S. Mismatch string kernels for svm protein classification. In *Advances in neural information processing systems* (2002), pp. 1417–1424.

[20] FREDRIKSON, M., CHRISTODORESCU, M., AND JHA, S. Dynamic behavior matching: A complexity analysis and new approximation algorithms. In *Automated Deduction - CADE*, vol. 6803 of *Lecture Notes in Computer Science*. Springer, 2011, pp. 252–267.

[21] GOPAN, D., DRISCOLL, E., NGUYEN, D., NAYDICH, D., LOGINOV, A., AND MELSKI, D. Data-delineation in software binaries and its application to buffer-overrun discovery. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on* (May 2015), vol. 1, pp. 145–155.

[22] GUO, B., BRIDGES, M. J., TRIANTAFYLLIS, S., OTTONI, G., RAMAN, E., AND AUGUST, D. I. Practical and accurate low-level pointer analysisIn *Proceedings of the International Symposium on Code Generation and Optimization* (2005), CGO '05, IEEE Computer Society, pp. 291–302.

[23] HALLER, I., SLOWINSKA, A., AND BOS, H. Mempick: High-level data structure detection in c/c++ binaries. In *Reverse Engineering (WCRE), 2013 20th Working Conference on* (Oct 2013), pp. 32–41.

[24] HE, Q., JIANG, D., LIAO, Z., HOI, S. C., CHANG, K., LIM, E.-P., AND LI, H. Web query recommendation via sequential query prediction. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on* (2009), IEEE, pp. 1443–1454.

[25] JAAKKOLA, T., HAUSSLER, D., ET AL. Exploiting generative models in discriminative classifiers. *Advances in neural information processing systems* (1999), 487–493.

[26] JANG, D., TATLOCK, Z., AND LERNER, S. Safedispatch: Securing C++ virtual calls from memory corruption attacks. In *Network and Distributed System Security (NDSS) Symposium* (2014).

[27] JHA, S., TAN, K., AND MAXION, R. Markov chains, classifiers, and intrusion detection. *Computer Security Foundations Workshop, IEEE 0* (2001), 0206.

[28] KATZ, S. M. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *Acoustics, Speech and Signal Processing, IEEE Transactions on 35*, 3 (1987), 400–401.

[29] KRICHEVSKY, R., AND TROFIMOV, V. The performance of universal encoding. *IEEE Trans. Inf. Theor. 27*, 2 (Sept. 2006), 199–207.

[30] LEE, J., AVGERINOS, T., AND BRUMLEY, D. TIE: principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011* (2011).

[31] LIN, J. Divergence measures based on the shannon entropy. *IEEE Trans. Inf. Theor. 37*, 1 (Sept. 2006), 145–151.

[32] LODHI, H., SAUNDERS, C., SHAWE-TAYLOR, J., CRISTIANINI, N., AND WATKINS, C. Text classification using string kernels. *The Journal of Machine Learning Research 2* (2002), 419–444.

[33] MADSEN, M., LIVSHITS, B., AND FANNING, M. Practical static analysis of javascript applications in the presence of frameworks and librariesIn *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, ACM, pp. 499–509.

[34] MAHONEY, M. V. Adaptive weighing of context models for lossless data compression, 2005.

[35] MAZEROFF, G., GREGOR, J., THOMASON, M., AND FORD, R. Probabilistic suffix models for {API} sequence analysis of windows {XP} applications. *Pattern Recognition 41*, 1 (2008), 90 – 101.

[36] MISHNE, A., SHOHAM, S., AND YAHAV, E. Typestate-based semantic code search over partial programsIn *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2012), OOPSLA '12, ACM, pp. 997–1016.

[37] MOFFAT, A. Implementing the ppm data compression scheme. *Communications, IEEE Transactions on 38*, 11 (1990), 1917–1921.

[38] NISENSON, M., YARIV, I., EL-YANIV, R., AND MEIR, R. Towards behaviometric security systems: Learning to identify a typist. In

Knowledge Discovery in Databases: PKDD 2003. Springer, 2003, pp. 363–374.

[39] PAULUS, J., AND KLAPURI, A. Labelling the structural parts of a music piece with markov models. In *Computer Music Modeling and Retrieval. Genesis of Meaning in Sound and Music*. Springer, 2009, pp. 166–176.

[40] PREDA, M. D., CHRISTODORESCU, M., JHA, S., AND DEBRAY, S. A semantics-based approach to malware detectionIn *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2007), POPL '07, ACM, pp. 377–388.

[41] RAMALINGAM, G., FIELD, J., AND TIP, F. Aggregate structure identification and its application to program analysisIn *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999), POPL '99, ACM, pp. 119–132.

[42] RAYCHEV, V., VECHEV, M., AND KRAUSE, A. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2015), POPL '15, ACM, pp. 111–124.

[43] RAYCHEV, V., VECHEV, M., AND YAHAV, E. Code completion with statistical language modelsIn *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), PLDI '14, ACM, pp. 419–428.

[44] REPS, T., AND BALAKRISHNAN, G. Improved memory-access analysis for x86 executables. In *Compiler Construction*, L. Hendren, Ed., vol. 4959 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 16–35.

[45] REPS, T., BALAKRISHNAN, G., AND LIM, J. Intermediate-representation recovery from low-level codeIn *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (2006), PEPM '06, ACM, pp. 100–111.

[46] REPS, T., BALAKRISHNAN, G., LIM, J., AND TEITELBAUM, T. A next-generation platform for analyzing executables. In *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds., vol. 27 of *Advances in Information Security*. Springer US, 2007, pp. 43–61.

[47] REPS, T., LIM, J., THAKUR, A., BALAKRISHNAN, G., AND LAL, A. There's plenty of room at the bottom: Analyzing and verifying machine code. In *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., vol. 6174 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 41–56.

[48] ROSENFELD, R. Two decades of statistical language modeling: Where do we go from here? In *Proceedings of the IEEE* (2000), vol. 88, pp. 1270–1278.

[49] SABANAL, P. V., AND YASON, M. V. Reversing C++. `https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf`.

[50] SAIGO, H., VERT, J.-P., UEDA, N., AND AKUTSU, T. Protein homology detection using string alignment kernels. *Bioinformatics 20*, 11 (2004), 1682–1689.

[51] SCHÜTZE, H., AND SINGER, Y. Part-of-speech tagging using a variable memory markov modelIn *Proceedings of the 32Nd Annual Meeting on Association for Computational Linguistics* (1994), ACL '94, Association for Computational Linguistics, pp. 181–187.

[52] SCHWARTZ, E. J., LEE, J., WOO, M., AND BRUMLEY, D. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. *Proceedings of the USENIX Security Symposium* (2013), 16.

[53] STEENSGAARD, B. Points-to analysis in almost linear timeIn *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1996), POPL '96, ACM, pp. 32–41.

[54] SUTTON, C., AND MCCALLUM, A. An introduction to conditional random fields. *Machine Learning 4*, 4 (2011), 267–373.

[55] TU, S. MINO: Data-driven type inference for python. MIT 6.867 Fall 2012 Final Project, December 2012.

[56] WARRENDER, C., FORREST, S., AND PEARLMUTTER, B. Detecting intrusions using system calls: Alternative data models. In *In IEEE Symposium On Security And Privacy* (1999), IEEE Computer Society, pp. 133–145.