Metadata Recovery From Obfuscated Programs Using Machine Learning

Aleieldin Salem
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
salem@cs.tum.edu

Sebastian Banescu
Technische Universität München
Boltzmannstr. 3
85748 Garching bei München, Germany
banescu@cs.tum.edu

ABSTRACT

Obfuscation is a mechanism used to hinder reverse engineering of programs. To cope with the large number of obfuscated programs, especially malware, reverse engineers automate the process of deobfuscation i.e. extracting information from obfuscated programs. Deobfuscation techniques target specific obfuscation transformations, which requires reverse engineers to manually identify the transformations used by a program, in what is known as *metadata recovery attack*. In this paper, we present *Oedipus*, a Python framework that uses machine learning classifiers viz., decision trees and naive Bayes, to automate metadata recovery attacks against obfuscated programs. We evaluated Oedipus' performance using two datasets totaling 1960 unobfuscated C programs, which were used to generate 11.075 programs obfuscated using 30 configurations of 6 different obfuscation transformations. Our results empirically show the feasibility of using machine learning to implement the metadata recovery attacks with classification accuracies of 100% in some cases.

CCS Concepts

•**Security and privacy** → **Software security engineering;** *Software reverse engineering;*

Keywords

Obfuscation, Machine Learning, Reverse Engineering

1. INTRODUCTION

Obfuscation is the process of transforming a program P into another program P', which has the same functionality as P (e.g. input-output behavior) and which conceals the code and/or data of program P from reverse engineers. Courtesy of its simplicity and cost-effectiveness, obfuscation is the *de facto* mechanism to hinder reverse engineering of legitimate software to the end of limiting software piracy. Nevertheless, malware authors are also leveraging obfuscation in writing their malware instances [22]. Unlike software vendors, malware authors do not use obfuscation to protect the intellectual property of their instances; they rather attempt to evade detection by conventional mechanisms that continue to hinge on signatures.

Many non-cryptographically secure obfuscation transformations, can protect an application against a reverse engineer for a limited amount of time. In other words, given enough time, an experienced reverse engineer can unravel the concealed functionality of a program, and retrieve the secrets it may withhold. Reverse engineering a program usually includes retrieving the original program code amidst the obfuscated code. This process is referred to as deobfuscation, and can be performed manually, automatically, or semi-automatically. Despite being the most reliable option, manual deobfuscation is a time-consuming process, especially if the obfuscated program is complex in terms of structure or functionality. Furthermore, malware analysts who reverse engineer malware are faced with millions of variants of obfuscated malware instances per day. Thus, manual deobfuscation cannot cope with such a release rate.

Automatic deobfuscation techniques are meant to automatically extract some information from an obfuscated program (e.g. a hidden key, an algorithm, etc.) [11, 25, 28]. Effectively, they enable reverse engineers to cope with a large number of programs. However, deobfuscation attacks are often specific to some type of obfuscation technique (e.g. [25] is specific to control-flow flattening, [23] is specific to variable splitting, etc.). Therefore, reverse engineers first have to determine which obfuscation transformation has been applied to a program and only afterward they can apply the automated deobfuscation attack corresponding to that transformation. Retrieving information about the obfuscation transformations utilized by a program, also known as *metadata recovery attack*, is typically a manual task, and is, therefore, a potential bottleneck of the reverse engineering process.

In order to recognize the obfuscation transformations used by a program, a reverse engineer needs knowledge of different transformations and how to distinguish between them. In fact, different obfuscation transformations are expected to exhibit distinguishable complexity and side effects on the program [6]. So, by studying these side effects, one can associate patterns with different transformations. Searching for those patterns within the obfuscated program's code or behavior allows recognizing the transformations employed by

SSPREW '16, December 05 - 06, 2016, Los Angeles, CA, USA

ISBN 978-1-4503-4841-6/16/09...$15.00

DOI: http://dx.doi.org/10.1145/3015135.3015136

an obfuscated program. This makes machine learning algorithms a suitable candidate for automating the metadata recovery attack, assuming that the obfuscated programs under test share common patterns. This is usually made possible if similar tools have been used to obfuscate programs.

Fortunately, the utilization of the same or similar obfuscation tools is common practice. Malware authors and software vendors alike tend to reuse similar tools and techniques in obfuscating their code [10]. Needless to say, programs obfuscated using the same tool are expected to share common characteristics and patterns e.g. similar system calls, data structures, and code patterns. Consequently, given a set of different programs obfuscated using the same tool, we can train a machine learning algorithm to learn the common patterns imposed by the tool on those programs, and successfully recognize them upon examining obfuscated programs in the validation set, effectively implementing the metadata recovery attack. Technically, we can model this attack as a supervised learning problem viz., classification, in which the classes correspond to obfuscation transformations, and the data points are programs using the same tool which offers several obfuscation transformations.

In this paper, we present and evaluate a machine learning-based approach to implement the metadata recovery attack. We implemented a framework called *Oedipus*, that uses *Decision trees* and *Naive Bayes* to classify obfuscated C programs generated by the *Tigress* obfuscator [4] according to the transformations they employ. Our evaluation results show that machine learning algorithms can be successfully used to infer the obfuscation transformations used by a program, which can aid reverse engineers by significantly reducing the amount of time needed to reverse engineer a program.

This paper makes the following contributions:

1. Presents a machine learning approach to implement the metadata recovery attack against obfuscated programs.

2. An open source Python framework called *Oedipus*[1], which generates obfuscated programs using Tigress[2], extracts various types of features from them, and uses machine learning algorithms to classify obfuscated programs according to the obfuscation transformations they employ.

3. An evaluation using two sets of obfuscated programs containing thousands of instances, in order to verify the accuracy of Oedipus.

The rest of the paper is organized as follows. Section 2, provides background information about the obfuscation transformations considered in this work. Section 3 presents Oedipus, its architecture, design and implementation. We evaluate Oedipus in section 4 and discuss the results. Section 5 presents works related to Oedipus. Lastly, section 6 draws conclusions from the conducted experiments and presents ideas for future works.

2. BACKGROUND

An obfuscation transformation can be thought of as a function F_τ that applies some operations on a program P and returns a functionally-equivalent program[3] P' obfuscated using the technique τ. Formally, this relationship can be represented as $P' = F_\tau(P)$. The functional-equivalence property ensures that P' carries out the same functionality as P i.e. the transformation τ does not affect the functionality of the original program P.

The effect of an obfuscation transformation is usually measured in terms of four dimensions viz., potency, resilience, stealth, and cost [1] [6]. *Potency* is concerned with the obscurity a transformation adds to a program. Usually, this measure is associated with human cognition i.e. how difficult to understand does a human find a given obfuscated program. The second dimension, *resilience*, measures the resilience of an obfuscation transformation against automated deobfuscation mechanisms. *Stealth* measures how well does an obfuscated segment of code blend in with the rest of the program [1]. In other words, how easy is it to spot an obfuscated segment of code. Finally, *cost* quantifies the performance penalty and resource consumption overhead that an obfuscation transformation adds to the program.

There are many obfuscation transformations that target different aspects of a program. Those transformations can be categorized into layout transformations, code transformations, and data transformations [6]. In this paper, we consider six obfuscation transformations that span both control and data obfuscation categories. They are: *Virtualization*, *Just-in-time Compilation (Jitting)*, *Opaque Predicates*, *Control-flow Flattening*, *Encoding Literals* and *Encoding Arithmetic*.

2.1 Layout Transformations

Layout transformations aim at scrambling the appearance and layout of the source code, rendering it unintelligible. Layout transformations include: renaming variables to random names, removing white spaces between lines of code, etc. [6]. Consequently, this category has high potency. Nonetheless, automated deobfuscation programs are less affected by such transformations. Layout transformations are cheap in terms of performance cost and therefore very popular for languages such as JavaScript. We do not use layout transformations, because these kinds of transformations are not applicable to x86 binary programs, which we focus on in this work.

2.2 Code Transformations

Code transformations affect the *aggregation*, *ordering* or *control flow* of programs. Hence, they are usually categorized into three categories corresponding to their effect on the code. Aggregation transformations break down computations that belong together and aggregate irrelevant ones. Ordering transformations randomize the order in which instructions are executed. Transformations that belong to this category are relatively rare, especially since shuffling program computations risks altering the original program semantics [2]. Hence, reordering is usually limited to independent code blocks. Lastly, control-flow transformations manipulate the computational structure of the program by either inserting new branch instructions or introducing algorithmic changes to it [6].

2.2.1 Virtualization Obfuscation

[1]https://github.com/tum-i22/Oedipus

[2]http://tigress.cs.arizona.edu/

[3]We define functional equivalence as follows. Two programs P and Q are said to be functionally-equivalent if and only if for every input i in the set of all possible inputs $\{0,1\}^*$, both P and Q yield the same output $o \in \{0,1\}^*$

Virtualization obfuscation is an example of control flow transformations. It creates a virtual environment within the program by implementing an interpreter that executes a set of custom instructions generated from the original code [20]. The interpreter is usually implemented as a switch statement whose cases handle various types of instructions. The instructions themselves tend to be modelled after RISC instructions e.g. *add*, *mov*, *jmp*, etc.. In order to thwart reverse engineering attempts, the custom language is chosen at random during obfuscation time, and the original code is permanently destroyed [20].

2.2.2 Just-in-Time Compilation (Jitting)

Just-in-Time Compilation (Jitting) is another example control-flow transformation. Jitting translates the original program into a set of statements each of which issues a call to a specific function. Upon executing a statement, the function hooked to it will compile the statement on-the-fly and load it into memory for execution. Effectively, the original function is dynamically compiled into machine code during runtime [5].

2.2.3 Flattening

Flattening is a control flow transformation that introduces algorithmic changes to the program by converting branching statements to procedural ones. Effectively, the control flow appears to be flattened. Nevertheless, the original control flow of the program needs to be maintained. In flattening, blocks of code are wrapped into an infinite loop. The infinite loop is controlled by a controlling block that determines the next block to be executed, in a process known as *dispatch*. There are different ways to implement the dispatch; they primarily differ in the mechanism used to transfer control to different call blocks. The common dispatch methods are *switch*, *goto*, *indirect*, and *call* dispatches.

2.2.4 Opaque Predicates

Opaque predicates are examples of control-flow transformation often coupled with inserting bogus code which is an aggregation transformation. A predicate P is opaque if its boolean outcome is known to the obfuscator during obfuscation, but difficult to statically deduce by the deobfuscator [1] [6]. For an obfuscator, the challenge is to design such a resilient predicate. Predicates can be designed to always evaluate to *True*, always evaluate to *False*, or evaluate to both values. The branches that never execute are usually called *bogus* branches that are meant to confuse static analysis based deobfuscation techniques [5]. This is the primary objective of adding opaque predicates i.e. to confuse static analysis e.g. disassembly.

2.2.5 Encoding Arithmetic

Encoding arithmetic expressions is another example of an aggregation transformation. It substitutes simple arithmetic expressions by more complex ones. For instance, the expression $z = x + y + w$ can be substituted by $z = (((x \wedge y) + ((x\&y) << 1))|w) + (((x \wedge y) + ((x\&y) << 1))\&w)$. A list of possible substitutions is presented in [27].

2.3 Data Transformations

Data transformations affect the *storage*, *encoding*, *aggregation*, or the *ordering* of the program's data structures [6]. The first category, *storage*, changes the container within which the data is stored in an attempt to conceal the data's original form. *Encoding* attempts to alter the representation of the data. This often requires changing the data type of the data or using functions to store and produce its value. *Aggregation* combines different variables into larger structures. This technique prevents a reverse engineer from drawing a clear cut between different data objects and identifying their functionality/usage within the program. Finally, *ordering* data transformations randomize the order of methods and instance variables within classes and formal parameters within methods.

2.3.1 Encoding Literals

Encoding Literals is an example of the encoding data transformation technique. In this work, we only consider string and integer literals. For string literals, the value of a string is usually encoded by computing its value via a function. To encode integer literals, opaque expressions are used [4].

3. OEDIPUS

This section presents the design and implementation of Oedipus[4], our Python framework for metadata recovery attacks.

3.1 Design

Figure 1 shows the four phases of the classification process in Oedipus. In the first phase, we use Tigress to generate obfuscated versions of a given dataset of C programs, which is an input for Oedipus. Phase two extracts different types of static and dynamic features from the obfuscated programs' executables. In phase three, a classifier is trained using one type of the generated feature vectors i.e. a training set. The fourth and final phase is concerned with classifying the remaining feature vectors, or the test dataset, using the trained classifier. The output of the final phase is the classification accuracy rate scored by the trained classifier. The accuracy denotes the percentage of obfuscated programs for which Oedipus recognized the transformations they employ correctly.

Prior to discussing the implementation of Oedipus, we go over the design decisions we made during each of the aforementioned phases. We categorize such decisions as generation of raw data, feature extraction from raw data, and choice of classification approaches.

3.1.1 Generation of Raw Data

In the machine learning terminology, raw data usually refers to a representation of data samples that contain noisy features and, hence, need to be processed in order to extract informative features from the data samples prior to training a model. Within our context, the raw data comprises different representations of the obfuscated programs. Although we possess the source code of the obfuscated programs, we assume that, as adversaries, we only have access to the programs' executables. Those executables depict our raw data, and they have two formats. The first format of the executables is compiled using GCC and instructed to strip

[4]Oedipus was a mythical Greek king of Thebes. During one of his journeys, he defeated the Sphinx, who would devour all travellers that fail to solve its riddles, by solving what came to be known as the riddle of the Sphinx. We consider the question of whether machine learning can implement the metadata recovery attack a riddle that our framework attempts to solve.

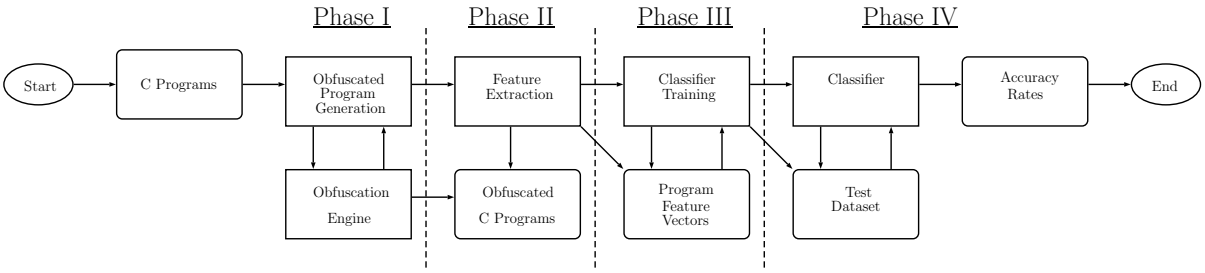

Figure 1: An overview of classification process.

the symbols table and relocation information from the executable, whereas the second format compiles the obfuscated programs using GCC with default settings, i.e. with symbols and relocation information. Therefore, we have two formats of raw data: *stripped* and *non-stripped*. Using these two formats, we wish to examine the effect of stripping executables (of names of functions and other symbols), on classification accuracy.

We further split these two formats of raw data i.e. stripped and non-stripped, into two categories that examine the two states that every program has i.e. *static* and *dynamic*. Given an executable of an obfuscated program, we retrieve two types of disassemblies from it. Firstly, we disassemble the executable before running it, which gives us the *static disassembly*. Then we run the executable, and record the instructions it executes during runtime and call that *dynamic disassembly*. Consequently, we wind up with four breeds of raw data i.e. ones that embody the combinations of stripping symbol table information and disassembling the executable at different states.

3.1.2 Feature Extraction

As mentioned in section 1, different obfuscation transformations leave distinguishable side effects on the obfuscated programs. For instance, the encode arithmetic transformation is expected to have a long series of complex arithmetic operations, whilst virtualization adds a large number of repetitive memory load and store operations to the program. These side effects can be identified by examining the instructions of the obfuscated program before or during runtime. In our case, those instructions are x86 assembly instructions that we extract from the obfuscated program executable. Following the same example, the version of a program obfuscated using the encode arithmetic transformation should contain a number of arithmetic instructions e.g. add, mul, sub, etc., greater than the version of the same program obfuscated using virtualization, which should contain a larger number of memory instructions e.g. mov, push and pop. We can generalize this notion and argue that every obfuscation transformation may add a particular instruction pattern, as a side effect, to the programs it obfuscates.

If we consider the set of disassembly files of our obfuscated programs as a set of text documents D each of which comprising a set of keywords or terms separated by spaces $t_1, ..., t_n$, then we can seamlessly match the previously discussed idea of x86 instruction patterns in disassemblies to the *Term Frequency Inverse Document Frequency (TF-IDF)* features as follows. The TF-IDF features are renownedly used within the context of clustering text documents according to the topics they address e.g. politics, sports, arts, etc. The TF-IDF method determines the relative frequency of terms in a certain document, and compares it to the inverse proportion of that term across all other documents in the corpus [18]. Effectively, it calculates the relevance of a term t^* to a certain document. Mathematically, the higher the TF-IDF value of a term, the more relevant it is to a document, which makes that particular term a more informative feature of the document. For example, the term *president* is more relevant and, hence, more recurring in politics documents; consequently, it is expected to be higher in politics documents than in their sports counterparts.

The disassembly files of our obfuscated programs can be considered as text documents whose topics are the obfuscation transformation employed by the programs. Note that our disassemblies are a list of x86 instructions, each comprising of one opcode (e.g. *add, mul, sub*) and 0 or more operands separated by spaces (e.g. a register value *eax*, a constant *0x5*, a memory reference *[0x12345678]*. Therefore, every opcode and every operand is a different term for the TF-IDF algorithm. Similar to the previous example, the term *push* is expected to have higher TF-IDF values in disassembly files of programs obfuscated using virtualization. Effectively, TF-IDF features can help emboss the side effects imposed by different obfuscation transformations on the obfuscated programs, which facilitates classifying them.

Calculating the TF-IDF value t^*_d of a term t^* in a given document $d \in D$ is carried out as follows:

$$t^*_d = f_{t^*,d} \times \log\left(\frac{|D|}{f_{t^*,D}}\right),$$

where $f_{t^*,d}$ is the number of times the term t^* appears in document d, $|D|$ is the number of documents in the dataset, and $f_{t^*,D}$ is the number of documents in which the term t^* has occurred.

If a term t^* never appears in the corpus, the expression $f_{t^*,D}$ yields a zero leading to a division-by-zero problem. Therefore, the formula is altered to have a denominator of $f_{t^*,D} + 1$. Some implementations add a hypothetical document in which all terms occur. This adds one to both $|D|$ and $f_{t^*,D}$. Another amendment adds one to the last multiplication term, so as not to ignore terms that have zero IDF values i.e. terms that occur in all documents in the corpus. The TF-IDF equation is altered to:

$$t^*_d = f_{t^*,d} \times \left(\log\left(\frac{|D|+1}{f_{t^*,D}+1}\right) + 1\right)$$

We use TF-IDF to extract features from disassembly files of obfuscated programs. The process yields a feature vector per program that comprises the TF-IDF values of the top 128 terms encountered in all the disassembly files.

3.1.3 Choice of Classifiers

The objective of the classifiers in Oedipus is to use the TF-IDF feature vectors representing the obfuscated programs to train a model capable of classifying the programs according to the obfuscation transformations they employ. Nevertheless, there is a plethora of classification algorithms to choose from. In fact, deciding upon classification algorithms that suit the metadata recovery problem is also an objective of our research effort. In this paper, we chose to test classification algorithms that, we believe, emulate the processes adopted by reverse engineers while examining obfuscated programs viz., we use *Decision Trees* and *Naive Bayes* classifiers to carry out metadata recovery attacks against obfuscated programs. We also experimented with the prominent *Support Vector Machine* classifier. However, it required an infeasible amount of resources upon being trained with a large number of obfuscated programs.

Decision trees emulate a decision-making process in which data samples are split into two, or more, segments based on querying the values of quantitative or qualitative data features. The querying and splitting process continues until a confident decision about the class of a data sample is made. This process can be matched to that of a reverse engineer examining an obfuscated program. That is, a reverse engineer continuously examines different segments of obfuscated programs looking for clues to consider or exclude some transformations until s/he is confident that the program under test has been obfuscated using a specific transformation.

We have argued that different transformations exhibit distinguishable patterns. Hence, decision trees should be able to segregate different transformations by examining the feature vectors representing the obfuscated programs. In this case, the features used to train the decision tree are TF-IDF features, which still segregate different transformations–as discussed earlier–in terms of parts of assembly instructions they use more frequently. Therefore, the splitting decisions in the tree would be made according to the TF-IDF values of different terms in assembly instructions.

Reverse engineers can sometimes adopt a probabilistic approach to reverse engineering. In other words, a reverse engineer can combine various evidence they gathered during program inspection to reach a conclusion that the program has probably been obfuscated using a particular transformation. For example, assume that a reverse engineer gathered information about an obfuscated program that (a) it contains what seems to be an infinite loop, (b) it computes a value that is used as a predicate in a conditional jump statement, and (c) it uses a noticeable amount of unconditional jumps. The reverse engineer can make a decision about the transformation used by the program by combining those three conditions i.e. since (a) and (b) and (c) do hold, therefore the program is probably obfuscated using control flow flattening. Optionally, the engineer can assign a value to the decision's probability.

Needless to say, reverse engineers usually do not follow such a formal process. Nonetheless, they implicitly carry out this logical, probabilistic process during program inspection. Naive Bayes is a probabilistic classifier that–assuming their occurrence is independent–combines different events to calculate the probability of a data sample belonging to a specific class. Within our context, the events are TF-IDF features depicting the assembly instructions within the obfuscated programs. So, the probabilistic decision of a program being obfuscated using a particular transformation is made based on combining the probabilities of encountering assembly instructions within the disassemblies of the obfuscated program.

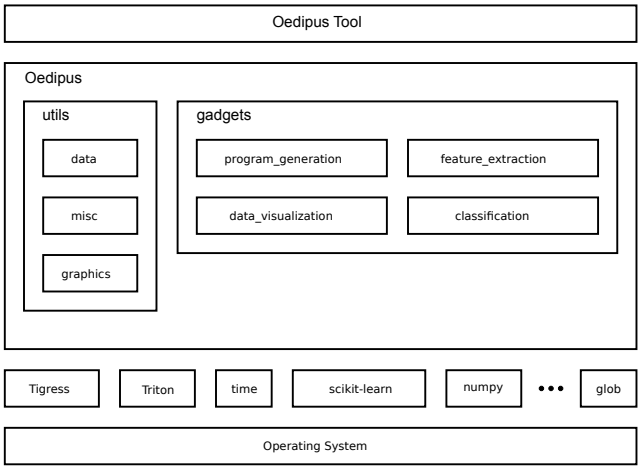

Figure 2: The structure of the Oedipus framework, and the tools that interact with it. The layered architecture is meant to depict functional dependency. That is to say, *oedipus_tool* depends on the functionalities exposed by the framework, whereas the framework hinges on a group of tools and libraries, etc..

3.2 Implementation

For extensibility purposes, Oedipus is implemented in the form of modules. The framework comprises two main modules viz., *utils* and *gadgets* as shown in the middle of figure 2. The two modules comprise sub-modules exposing functionalities that can be used separately. In other words, the process depicted in figure 1 need not be implemented with every run. In contrast, any Python tool can be implemented to leverage the functionalities of a sub-module separately.

The framework, Oedipus, is designed to support the four phases mentioned in the overview from figure 1. We combine the phases of model training and classification into one viz., classification. Thus, the phases of program generation, feature extraction, and classification are implemented by the *program_generation*, *feature_extraction*, and *classification* sub-modules, respectively as seen in figure 2. The layered layout in the figure implies dependency in a top-to-bottom manner. That is to say, *oedipus_tool* depends on the functionalities exposed by Oedipus; in turn, Oedipus utilizes various tools and libraries to deliver its functionalities such as the Tigress obfuscation tool [5], the Triton dynamic binary analysis framework [21] and various Python libraries.

3.2.1 Utilities

The *utils* module of Oedipus contains all the utility functions that are regularly needed by other modules and tools built on top of the framework. We grouped functions that deliver similar functionalities together. Consequently, the *utils* module comprises of three sub-modules, namely *data*, *graphics*, and *misc*. The *data* submodule contains three types of functions. The first type of functions is responsible for loading feature vectors from text files into Python data structure i.e. lists and dictionaries. The functions were written to

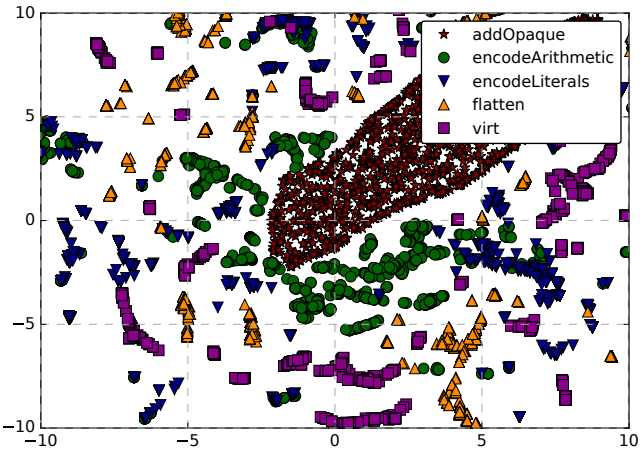

Figure 3: A 2-dimensional visualization of obfuscated programs using the TF-IDF features and the t-SNE visualization algorithm.

handle different types of features, which comprise numerical/nominal values. The *graphics* submodule contains message display functions.The *misc* submodule contains functions that perform miscellaneous tasks including generating random numbers and strings, calculating the average of values in lists and tuples, removing temporary files, and so forth.

3.2.2 Gadgets

The *gadgets* module contains four sub-modules viz., *program_generation*, *feature_extraction*, *data_visualization*, and *classification*.

The *program_generation* module is responsible for implementing phase one of the classification process, which includes generating obfuscated programs. Currently, the module is tailored to use Tigress. However, we plan on extending it to support different obfuscation tools. As mentioned in section 2, we obfuscate the input C programs using six obfuscation transformations i.e. virtualization, jitting, flattening, opaque predicates, encoding arithmetic, and encoding literals. The module supports obfuscating programs multiple times.

The *feature_extraction* sub-module supports extracting various types of static and dynamic features from the source code, disassembly, and executable files of the obfuscated programs. In this paper, we focus on the extraction of TF-IDF features from static and dynamic disassembly files.

The *data_visualization* sub-module is can accomplish two tasks. Firstly, given numerical representations of the obfuscated programs, it uses either *Principal Component Analysis (PCA)* or *t-distributed stochastic neighbor embedding (t-SNE)* to plot the programs as data points in a 2- or 3-dimensons. Figure 3, for instance, depicts a 2-dimensional visualization of TF-IDF representations obfuscated programs using t-SNE. The sub-module also generates plots of classification accuracies achieved by the trained classifiers.

Lastly, as the name suggests, the *classification* sub-module is responsible for training a classification algorithm, and testing it against our input dataset of obfuscated programs. The sub-module uses the Scikit-learn implementations of the decision tree and naive Bayes classifiers that support K-Fold cross-validation,and exposes their functionalities to tools like oedipus_tool.

3.2.3 Utilized Tools

Oedipus was implemented exclusively in Python 2.7 for interoperability. The framework depends on various tools and libraries, including:

- **Tigress**: An obfuscation tool for C programs developed and maintained by the University of Arizona [4].
- **Scikit-learn**: A machine learning library for the Python programming language [7]. We use Scikit-learn's implementations of decision trees, naive Bayes, K-Fold cross-validation, and classification accuracy calculation.
- **Gensim**: Gensim is a Python vector space modeling tool [19], that we use to implement a memory-friendly version of extracting the TF-IDF features from our obfuscated programs. The Scikit-learn implementation of such functionality attempts to load all documents in memory prior to extracting the TF-IDF features from them, which requires large amounts of memory in the case of large document corpora. Gensim implements this functionality in an incremental manner by loading one document at a time in memory and updating the TF-IDF values of different terms accordingly.
- **GCC**: Oedipus uses GCC [16] to compile the obfuscated programs in order to extract static and dynamic raw data from the obfuscated programs' executables.
- **objdump**: We use GNU Binutils' objdump tool [15] to extract the static disassembly from the obfuscated programs' executables. We simply disassemble the executables of obfuscated programs and store the results in text files.
- **GDB**: Lastly, we use GDB [17] to generate dynamic disassembly from the obfuscated programs' executables. We wrote a GDB script that steps into every instruction executed by a program during runtime, and logs it to a file in the form of an assembly instruction.

4. EXPERIMENTS

This section first presents two datasets of C programs and how we have used them in our experiments. Afterward, it presents experiments we have performed to verify our primary hypothesis of whether machine learning algorithms can implement metadata recovery attacks against obfuscated programs. To run these experiments we developed the *Oedipus tool* on top of the Oedipus framework. This tool implements two classification approaches using *Decision Trees* and *Naive Bayes*, which we also describe in this section. To this end, we are interested in answering the following research questions:

1. With what accuracy can *Decision Trees* and *Naive Bayes* classify the obfuscation transformations which were employed to protect a binary program?
2. Does accuracy differ if we use the static disassembly or dynamic disassembly of an obfuscated program as raw data for the machine learning classifiers?
3. Does accuracy improve if we filter the raw data such that we replace constant values (including addresses) with generic values?

4. Does accuracy improve if we use binaries which are not stripped off debugging symbols?

5. What is the accuracy of classifiers for completely new programs, i.e. programs whose original and obfuscated versions have never been seen by the classifier in the training phase?

6. Does the heterogeneity of the programs in the dataset used for training and testing the machine learning classifiers have an influence on accuracy?

4.1 Datasets of C Programs

Tigress only supports obfuscating programs written in the C language. Therefore, we constructed two sets of C programs, one heterogeneous (containing 40 programs) and one homogeneous (containing 1920 programs). Each of these programs in these datasets is obfuscated as described in section 4.2.1, which results in a total of 11,075 obfuscated programs used in our experiments.

The former dataset comprises of 40 programs that implement basic functionalities, such as factorial, sorting, searching, string operations, file I/O operations, etc. [8]. In order to guarantee some degree of diversity in the programs' structures, we collected programs that utilize most, if not all, features of the C language i.e. pointers, structs, recursion, and so forth.

In reality, more sophisticated programs are expected to share language constructs and features. For example, a web browser and a word processor written in the same language are expected to use similar language features in order to implement the module responsible for receiving user input from the keyboard. Given that the 40 programs implement primitive, yet different, functionalities, we do not expect them to share the same features. For instance, the factorial and Fibonacci programs are focused on arithmetic operations, whereas sorting algorithms depend more on array-based operations. This phenomenon may result into noisy segments of the obfuscated programs that confuse the classification algorithms, because the different functionalities of the 40 programs add different patterns to programs that are obfuscated using the same transformation.

The second dataset is homogeneous and contains 1920 C programs generated by the Tigress *RandomFuns* transformation. These programs share a common template, i.e. they have a main function which calls a randomly generated function on the input arguments. This randomly generated function contains various control flow instructions (e.g. *if*-statements, *for*-loops, etc.), arithmetic, logic and bitwise operations (e.g. +, <, &, etc.), involving the input arguments and constants. Finally, it returns a value dependent on the input arguments. The main function prints this return value and also compares it with a constant. If they are equal it prints a distinctive message on the standard output. Note that the control flow structure, the data types and the operations are different for each of the 1920 C programs in this dataset.

4.2 Oedipus Tool

As discussed in section 3, the Oedipus framework exposes different APIs in an independent manner to allow users to start the classification process from whichever phase they prefer. The tool, conveniently called *oedipus_tool.py*, supports different modes of operation that interface with the gadgets of the Oedipus Framework. The modes are:

- **generate**: Generate obfuscated versions of C programs.
- **extract**: Compile obfuscated programs, then generate static and dynamic disassemblies.
- **filter-traces**: Generates filtered versions of disassemblies, as discussed later in section 4.2.3.
- **extract-from-traces**: A shorter version of *extract*, in which the first phase is ignored and TF-IDF features are extracted directly from the disassembly files.
- **classify-exp1**: Uses either decision trees or naive Bayes to classify obfuscated programs according to the transformations they employ using K-Fold cross validation.
- **classify-exp2**: Uses either decision trees or naive Bayes to classify obfuscated programs according to the transformations they employ using a custom type of cross-validation, as discussed later in section 4.2.5.
- **visualize**: Visualizes the feature vectors of obfuscated programs using either the PCA or the t-SNE algorithms.

The mode of operation and other necessary parameters e.g. source directory of the input files, the number of folds for cross-validation, the verbosity of debugging messages, etc., are specified using command-line arguments.

4.2.1 Generate Mode

In **generate** mode, Oedipus requires the command line options for the Tigress obfuscation engine in order to obfuscate a dataset of programs. For the first dataset of 40 programs we have used the five obfuscation transformations presented in Section 2, and provided various other options for transformations, which led to 39 different parameter configurations of the Tigress obfuscation engine. This yields $40 \times 39 = 1560$ obfuscated programs. Out of the resulting obfuscated programs, 90 could not be compiled due to bugs in Tigress. Therefore, we removed these programs and were left with 1470 obfuscated programs from the first dataset.

For the second dataset we used the default settings for the same five obfuscation transformations presented in Section 2. This yields $1921 \times 5 = 9605$ obfuscated programs, which all were compiled successfully.

4.2.2 Extract Mode

The executables of obfuscated programs are further processed by the Oedipus tool in **extract** mode to compile the obfuscated programs–once with GCC's stripping flag *-s*, and once without it. Hence, for every single program in the two datasets, we have two types of executables. From each type of executable the Oedipus tool uses the framework to generate two types of disassemblies viz., static and dynamic. The static disassembly files are saved to text files with the *.objdump* extension. To generate the dynamic disassemblies the Oedipus tool uses gdb to record the assembly instructions they issue at runtime. For this purpose, the Oedipus tool requires test cases for the executables as an input. In the following paragraphs we describe how we obtained the inputs for each of the two datasets.

For programs in the first dataset we used a combination of test cases generated by the KLEE symbolic execution engine [3] and manually-elicited test cases. For the manually-elicited inputs, we attempted to include test cases for all input types, for each program. For example, some of the manual inputs for the factorial program were 0, 1, a random-negative number, a random character, a random string, etc..

This process resulted into generating 17,831 dynamic disassembly files, that we gave the extension *.dyndis*.

On the other hand, programs in the second dataset did not require specific user inputs, especially since they did not implement any particular functionality. The only constraint was to forward the programs a five-digit integer. Furthermore, the value of program input did not have a significant effect on program behavior. Hence, for every single executable, we generated exactly one random input. We removed programs which did not terminate correctly in a time interval of 5 minutes after being passed a random input. This yielded 8156 dynamic disassembly files.

The exact same process (Oedipus tool in **extraction** mode) is carried out for the stripped version of the program executables using the same inputs that were used to run their non-stripped counterparts. To distinguish between the disassemblies of both types, we add an "s" to the file extensions e.g. *.dyndiss* and *.objdumps*.

4.2.3 Filter-traces Mode

The generated assembly files contain a detailed information about the executed instructions; this includes variables values, memory locations, and some initialization code from *libc*. We believe that the majority of such information is in some cases, noisy and confusing for classifiers, especially since they are neither relevant to the program functionality nor to the obfuscation transformation. Instead, they are very specific to the underlying operating system and architecture. So, we believe that these irrelevant values can be safely removed. We implemented a filtration mechanism–supported by the *filter-traces* mode of the Oedipus tool to remove irrelevant information. The mechanism focuses on the instructions issued by the function implementing the functionality of the program, and replaces either memory locations with the string *mem*, immediate values with the string *imm*, or both. We filtered out both types of values, and generated a filtered version of every single disassembly file. The filtered versions had *_both* appended to the extension e.g. the filtered version of *.dyndis* is *.dyndis_both*.

4.2.4 Extract-from-traces Mode

At this point, we have 8 types of disassembly files for each file in the input dataset of C programs. We use the Oedipus tool in *extract-from-traces* mode to extract TF-IDF features from them according to the description from section 3.1.2. This results in 8 files with the following extensions:

- **tfidf**: TF-IDF features generated from non-stripped, non-filtered dynamic disassembly files.
- **tfidf_both**: TF-IDF features generated from non-stripped, filtered dynamic disassembly files.
- **tfidfobj**: TF-IDF features generated from non-stripped, non-filtered static disassembly files.
- **tfidfobj_both**: TF-IDF features generated from non-stripped, filtered static disassembly files.
- **tfidfs**: TF-IDF features generated from stripped, non-filtered dynamic disassembly files.
- **tfidfs_both**: TF-IDF features generated from stripped, filtered dynamic disassembly files.
- **tfidfobjs**: TF-IDF features generated from stripped, non-filtered static disassembly files.
- **tfidfobjs_both**: TF-IDF features generated from stripped, filtered static disassembly files.

4.2.5 Classify-exp1 and Classify-exp2 Modes

We used the previously listed datatypes to train a decision tree classifier and a naive Bayes classifier. For the former classifier, we varied two attributes; we varied the splitting criterion used by the tree during training to be *Gini* index and *entropy*, and varied the maximum allowed depth of the tree to the values $\{2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16\}$. As for the latter classifier, we used two techniques to reduce the dimensionality of the feature vectors viz., *SelectKBest* features and *PCA*. We varied the target dimensionality to the values $\{8, 16, 32, 64, 128\}$. The two classifiers were used in two experiments (corresponding to the **classify-exp1** and **classify-exp2** modes of the Oedipus tool) we conducted for each of the two datasets.

The primary difference between the two experiments is how cross-validation is implemented. In experiment one, we adopt the conventional K-Fold cross validation with a value of $K = 10$. That is to say, the data comprising TF-IDF feature vectors is divided into 10 segments; 9 segments are used to train a classifier i.e. decision tree or naive Bayes, and the remaining segment is used to test the classifier's accuracy. This process is repeated 10 times varying the training and test datasets incrementally. The overall accuracy of a classifier is calculated by averaging the accuracy achieved at each iteration. The objective of experiment one is to get an objective estimate of the accuracy of the classifier by varying the training and test datasets.

Experiment two implements another cross-validation method. In this case, the training and test datasets are based on program functionality. That is to say, the test dataset is designed to exclude any programs that have been used in the training based on the functionality they implement. For instance, in order to use the factorial program in the test dataset, we need to make sure that it, or any of its obfuscated versions, has never been used in training. This process is repeated for 10 times and the overall classification accuracy is calculated as the average of achieved accuracies. The objective of this type of experiment is to study the effect of functionality on the classification accuracy. In other words, could the classifier recognize the obfuscation transformation employed by a program even it has never seen this type of program before?

4.3 Results

In this subsection, we present the results of running the Oedipus tool presented in section 4.2 on each of the datasets presented in section 4.1. Afterward, we discuss the results and threats to validity.

4.3.1 Dataset 1: 40 Programs

Table 1 includes the classification accuracies achieved using TF-IDF vectors from the first dataset. The results are tabulated for both experiments using accuracies achieved at a depth of 8 for the decision tree and reduced dimensionality of 64 for the naive Bayes classifier. Each row in the table depicts a combination of a certain: (1) type of raw data, i.e. static or dynamic, (2) disassembly filtration, i.e. raw or filtered and (3) symbols stripping during compilation. For example, the first row depicts the classification accuracies achieved using TF-IDF features extracted from non-filtered, stripped, static disassemblies i.e. *.objdumps* files. Each col-

Table 1: Classification accuracies for experiments 1 and 2 (in red) using 40 self-gathered C programs.

			Naive Bayes		Decision Tree	
			SelectKBest	PCA	Gini	Entropy
Static	raw	stripped	0.37 / 0.40	0.38 / 0.39	0.99 / 0.38	0.98 / 0.40
		non-stripped	0.38 / 0.40	0.96 / 0.40	0.96 / 0.25	0.96 / 0.38
	filtered	stripped	0.61 / 0.44	0.38 / 0.44	0.98 / 0.39	0.99 / 0.46
		non-stripped	0.86 / 0.40	0.40 / 0.40	0.99 / 0.44	0.99 / 0.61
Dynamic	raw	stripped	0.35 / 0.45	0.36 / 0.54	0.99 / 0.61	0.99 / 0.39
		non-stripped	0.60 / 0.40	0.35 / 0.42	0.99 / 0.56	0.99 / 0.40
	filtered	stripped	0.86 / 0.48	0.65 / 0.34	0.96 / 0.48	0.96 / 0.52
		non-stripped	0.92 / 0.55	0.62 / 0.35	0.99 / 0.57	0.99 / 0.41

umn in the table depicts a combination of a certain: (1) classification algorithm, i.e. naive Bayes or decision tree and (2) the corresponding feature selection method or splitting criterion Each cell of the table lists 2 values separated by a slash ("/"). The first value is the accuracy of the Oedipus tool in **Classify-exp1** mode (experiment 1) and the second value is the accuracy of **Classify-exp2** mode (experiment 2).

Examining the accuracies, one can observe the following vis-á-vis to the research questions at the beginning of section 4. Firstly, with a few exceptions, decision tree classifiers achieved higher classification accuracies than their naive Bayes counterparts (RQ1). Secondly, TF-IDF features extracted from dynamic disassemblies achieved higher classification accuracies than those extracted from static disassemblies (RQ2). Thirdly, filtering the disassemblies appeared to have helped, again with a few exceptions, achieve higher classification accuracies across different experiments, data types, and classification algorithms (RQ3). Moreover, we could not observe a specific pattern with regard to the effect of stripping symbols off obfuscated programs during compilation (RQ4). Lastly, the accuracies achieved in experiment 1 are, in general, much higher than those achieved in experiment 2 (RQ5).

4.3.2 *Dataset 2: Random Programs*

The classification accuracies achieved using TF-IDF vectors from the second dataset is tabulated in the same manner as seen in table 2. Some of the observations we made for the first dataset persist for its second counterpart. For instance, decision tree classifiers continued to achieve higher classification accuracies than naive Bayes classifiers, as well (RQ1). Filtration also continued to contribute to achieving higher classification accuracies for the majority of data types (RQ3). The classification accuracies achieved in experiment one continue to be higher than those achieved in experiment two (RQ5).

Unlike the first dataset, features extracted from dynamic disassemblies did not always outperform those extracted from static disassemblies (RQ2); dynamic disassemblies could not help naive Bayes classifiers to achieve high accuracies, whereas decision trees achieve 10% higher accuracies upon the utilization of features extracted from dynamic disassemblies. Another difference is concerned with using features extracted from stripped executables (RQ4). In contrast to the first dataset, stripping maintained a steady pattern across different experiments, data types, and classifications. Stripping did not significantly affect the classification accuracy.

Lastly, we made two observations by comparing experiments conducted using the two datasets (RQ6). The difference between the accuracies achieved in experiment two and experiment one is significantly larger upon using the second dataset in comparison to using the first dataset. Consequently, the classification accuracies in experiment two, using the second dataset, is much lower than those achieved using the first dataset. In other words, the second dataset yielded significantly worse classification accuracies in experiment two than its first counterpart.

4.3.3 *Discussion*

Reiterating over the results from the two datasets, we answer each of the research questions posted in the beginning of section 4. *Research question 1:* Decision tree classifiers achieved accuracies ranging from 88% to 100% in experiment 1 and ranging from 20% to 61% in experiment 2, which is better than randomly guessing one out of six transformation classes i.e. $\frac{1}{6} \approx 17\%$. Naive Bayes classifiers were slightly worse achieving classification accuracies ranging from 37% to 96% in experiment 1 and from 19% to 55% in experiment 2.

Research question 2: Despite a couple of exceptions, we can conclude that using dynamic disassemblies to extract features resulted in better classification accuracies, especially if a variety of inputs have been utilized to explore various runtime behaviors in a program. That is why TF-IDF features extracted from dynamic disassemblies always achieved better accuracies upon using the first dataset, which included programs each of which was run with several inputs.

Research question 3: The results we achieved imply that filtering out the immediate values and memory locations in disassembly files boosted classification accuracies. We believe that such filteration, combined with the TF-IDF features helped the classifiers identify obfuscation transformations that heavily rely on memory and arithmetic operations, especially since the memory locations and immediate values were replaced with particular strings as discussed in earlier.

Research question 4: Our results did not indicate a significant change in classification accuracies inflicted by stripping the program off debugging symbols during compilation. We believe, though, that stripping would have an effect on accuracy if the obfuscation transformation repeatedly uses a particular set of symbols that share a common naming scheme. For instance, if the jittin transformation names some functions as jit_add, jit_mul, jit_sub, etc., using the TF-IDF features, a classifier could easily segregate programs obfuscated using such transformations from programs obfuscated using other transformations.

Research question 5: The classification accuracy of a classifier used to retrieve the obfuscation transformation employed by a program largely depends on whether that program–or another program with similar functionalities–has been used

Table 2: Classification accuracies for experiments 1 and 2 (in red) using 1920 random C programs.

			Naive Bayes		Decision Tree	
			SelectKBest	PCA	Gini	Entropy
Static	raw	stripped	0.80 / 0.19	0.68 / 0.19	0.89 / 0.20	0.88 / 0.20
		non-stripped	0.80 / 0.19	0.69 / 0.19	0.89 / 0.20	0.88 / 0.19
	filtered	stripped	0.84 / 0.26	0.70 / 0.22	0.89 / 0.40	0.89 / 0.42
		non-stripped	0.87 / 0.23	0.78 / 0.20	0.89 / 0.35	0.88 / 0.34
Dynamic	raw	stripped	0.57 / 0.02	0.64 / 0.44	0.96 / 0.45	0.97 / 0.45
		non-stripped	0.57 / 0.02	0.63 / 0.44	0.96 / 0.45	0.96 / 0.45
	filtered	stripped	0.89 / 0.42	0.84 / 0.52	0.93 / 0.59	0.93 / 0.61
		non-stripped	0.96 / 0.36	0.91 / 0.14	1.00 / 0.61	1.00 / 0.61

during the training phase of a classifier i.e. experiment one versus experiment two.

Research question 6: Finally, as per the results achieved from experiment two, we conclude that there is an inverse relationship between the heterogeneity of programs used in training and test datasets and the achieved classification accuracies. That is to say, the less the relationship, in terms of functionality and structure, between the programs in the training dataset and test dataset, the lower the classification accuracy. In other words, if the classifier has never encountered a particular type of programs during training e.g. sorting programs, it is less likely that it can recognize the obfuscation transformations they employ during the test phase.

4.3.4 *Threats to validity*

In the experiments presented here, we only used one obfuscation engine, namely Tigress. Increasing the number of obfuscation transformation implementations by adding multiple tools may affect the accuracy. However, we are confident that if a certain obfuscation transformation has a certain pattern then it will be possible to recognize it using machine learning algorithms. As indicated in section 5, Sun et al. [24] have performed a similar study to ours where they used multiple packing tools and they achieved similar accuracy levels.

Another threat to the validity of our results is that we only used 2 datasets of relatively small C programs. We did not use more complex applications consisting of thousands of functions such as web browsers for the following reasons. Firstly, we consider that metadata recovery attacks should be applied at the level of granularity of single functions, because different functions could be obfuscated using different transformations. Secondly, we consider our dataset to be representative for a large number of programs, because they use all common programming language constructs. Thirdly, devising test cases that cover all functions of a complex application, can be an extremely difficult and resource consuming process and is not the focus of our work.

5. RELATED WORK

The authors are not aware of any works which aim to perform *metadata recovery*, i.e. recovery of information about which obfuscation transformation was applied to an obfuscated binary. However, there are several works from the area of identification and classification of packed binaries, which we consider related to our work.

Lyda and Hamrock [13] use binary entropy analysis to identify encrypted and packed code inside binaries. They propose a tool called *Bintropy* which is able to compute the entropy of blocks of a binary executable. Using a rule-based methodology it estimates whether or not a binary contains encrypted or compressed bytes.

The *Fast Library Identification and Recognition Technology*[5] (FLIRT) is a feature of a state of the art tool for reverse engineering called *IDA Pro*, which uses signature-based pattern matching to identify library functions in static binary code. FLIRT stores signatures of common libraries in a database and uses it while IDA Pro is disassembling a binary executable. The information provided by FLIRT is aimed at facilitating the reverse engineering process by a human analyst.

Perdisci et al. [14] use pattern matching on static binaries in order to detect whether a binary is packed or unpacked. Similarly to our work, the authors use machine learning techniques such as Naive Bayes and decision trees for the purpose of classification. Differently to our work where the number of classes is equal to the number of obfuscation transformations implemented by a certain obfuscation engine, the classification problem in the work of Perdisci et al., is limited to 2 classes, i.e. packed or unpacked. Ugarte-Pedrero et al. [26] propose a semi-supervised learning approach called *Learning with Local and Global Consistency* (LLGC) in order to classify packed and unpacked binaries. Their empirical evaluation shows that using only 10% of the dataset of malware instances from Perdisci et al. [14], LLGC achieves an accuracy lower by only 9% of that reported by Perdisci et al.

Kanzaki et al. [12] propose *code artificiality* as a metric for the stealth of obfuscated code, i.e. a measure of indistinguishability between obfuscated and unobfuscated code. Similarly to our work the use transformations from the Tigress obfuscator in their case study. Their results show that static obfuscation transformations such as control flow flattening are more stealthy than dynamic obfuscation transformation such as jitting. Unlike our work, they do not aim to recover the obfuscation transformation applied to a binary, but to measure its stealth instead.

The goal of the work by Sun et al. [24] is most similar to our work since they not only try to distinguish between packed and unpacked binaries, but also the name of the packer which was used. Similarly to our work they also used Naive Bayes and decision trees plus other classification algorithms. However, while we use TF-IDF for feature extraction, they use a refined version of the sliding window randomness test with trunk pruning method by Ebringer et al. [9]. Unlike Sun et al. [24], we not only extract static features from the obfuscated programs; we also execute the obfuscated programs, record their instruction traces, and extract TF-IDF features from them. This is a significant difference, because we show that classification accuracy using dynamic instruction traces

[5] https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml

as features, is in sum better than static code as features.

6. CONCLUSIONS

Our primary hypothesis is that machine learning is capable of implementing metadata recovery attacks by classifying obfuscated programs according to the transformations they employ. In this paper, we presented Oedipus, a Python framework is capable of generating obfuscated versions of C programs, extracting various types of features from them, visualizing them, and using decision trees and naive Bayes classifiers to classify them according to the obfuscation transformations they employ.

Using two sets of programs, a variety of representations of obfuscated program executables, the TF-IDF features, the aforementioned classifiers, and two types of experiments, we managed to empirically prove the feasibility of using machine learning to implement the metadata recovery attacks with classification accuracies as high as 100%. We also studied the effects of varying the heterogeneity of obfuscated programs, the format of the raw data, and the classification algorithms on the achieved accuracy.

We plan on extending Oedipus as follows. Firstly, we wish to consider more layers of obfuscation, and examine whether Oedipus is capable of implementing the metadata recovery attacks against them. Secondly, we aspire to gather more real-world applications, such as text editors, web browsers, messengers, etc., and conduct our experiments on them. Lastly, since this work is partly motivated by helping reverse engineers study obfuscated malware samples, we plan on acquiring malicious samples, whose source code is available, and attempt to reproduce the same results we achieved with benign programs.

7. REFERENCES

[1] A. Balakrishnan and C. Schulze. Code obfuscation literature survey. *CS701 Construction of Compilers*, 19, 2005.

[2] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO)*, pages v–vi, May 2015.

[3] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[4] C. Collberg. *Tigress: Transformations Index*. University of Arizona, 2015.

[5] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM.

[6] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[7] D. Cournapeau. Scikit-learn.

[8] Cquestions.com. C programming interview questions and answers, 2015.

[9] T. Ebringer, L. Sun, and S. Boztas. A fast randomness test that preserves local detail. In *Virus Bulletin 2008*, pages 34–42. Virus Bulletin Ltd, 2008.

[10] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[11] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[12] Y. Kanzaki, A. Monden, and C. Collberg. Code artificiality: a metric for the code stealth based on an n-gram model. In *Proceedings of the 1st International Workshop on Software Protection*, pages 31–37. IEEE Press, 2015.

[13] R. Lyda and J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[14] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14):1941–1946, 2008.

[15] G. Project. Gnu binutils.

[16] G. Project. Gnu compiler collection.

[17] G. Project. Gnu debugger.

[18] J. Ramos. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, 2003.

[19] R. Rehurek. Gensim.

[20] R. Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[21] J. Salwan and F. Saudel. Triton: A concolic execution framework for x86-64 binaries. In *Symposium sur la securite des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.

[22] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 1st edition, 2012.

[23] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*. Citeseer, 2011.

[24] L. Sun, S. Versteeg, S. Boztaş, and T. Yann. Pattern recognition techniques for the classification of malware packers. In *Australasian Conference on Information Security and Privacy*, pages 370–390. Springer, 2010.

[25] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[26] X. Ugarte-Pedrero, I. Santos, P. G. Bringas, M. Gastesi, and J. M. Esparza. Semi-supervised learning for packed executable detection. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 342–346. IEEE, 2011.

[27] H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

[28] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. Technical report, Technical report, Department of Computer Science, The University of Arizona, 2014.