Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning

Sebastian Banescu, *Technische Universität München;* Christian Collberg, *University of Arizona;* Alexander Pretschner, *Technische Universität München*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/banescu

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning

Sebastian Banescu
Technische Universität München

Christian Collberg
University of Arizona

Alexander Pretschner
Technische Universität München

Abstract

Software obfuscation transforms code such that it is more difficult to reverse engineer. However, it is known that given enough resources, an attacker will successfully reverse engineer an obfuscated program. Therefore, an open challenge for software obfuscation is estimating the time an obfuscated program is able to withstand a given reverse engineering attack. This paper proposes a general framework for choosing the most relevant software features to estimate the effort of automated attacks. Our framework uses these software features to build regression models that can predict the resilience of different software protection transformations against automated attacks. To evaluate the effectiveness of our approach, we instantiate it in a case-study about predicting the time needed to deobfuscate a set of C programs, using an attack based on symbolic execution. To train regression models our system requires a large set of programs as input. We have therefore implemented a code generator that can generate large numbers of arbitrarily complex random C functions. Our results show that features such as the number of community structures in the graph-representation of symbolic path-constraints, are far more relevant for predicting deobfuscation time than other features generally used to measure the potency of control-flow obfuscation (e.g. cyclomatic complexity). Our best model is able to predict the number of seconds of symbolic execution-based deobfuscation attacks with over 90% accuracy for 80% of the programs in our dataset, which also includes several realistic hash functions.

1 Introduction

Software developers often protect premium features and content using cryptography, if secure key storage is possible. However, there are some risks regarding the use of cryptography in this context, i.e. code and data must be decrypted in order to be executable, respectively consumable by the end-user device. If end-users are malicious, then they can get access to the unencrypted code or data, e.g. by dumping the memory of the device on which the client software is running. Malicious end-users are called man-at-the-end (MATE) attackers and their capabilities include everything from static analysis to dynamic modification of the executable code and memory (e.g. debugging, tampering with code and data values, probing any hardware data bus, etc.).

In order to raise the bar against MATE attackers, obfuscation tools use code transformations to modify the original code such that it is harder to analyze and tamper with, while preserving the functionality of the program. Provably secure code obfuscation has been proposed in the literature, however, it is still impractical [3, 6, 7]. On the other hand, dozens of practical obfuscation transformations have been proposed since the early 1990s [14], however, their security guarantees are unclear.

Researchers and practitioners alike have struggled with evaluating the strength of different obfuscating code transformations. Many approaches have been proposed (see Section 2), however, despite the numerous efforts in this area, a recent survey on common obfuscating transformations and deobfuscation attacks indicates that after more than two decades of research, we are still lacking reliable concepts for evaluating the resilience of code obfuscation against attacks [42].

This paper makes the following contributions:

- A general framework for selecting program features which are relevant for predicting the resilience of software protection against automated attacks.
- A free C program generator, which was used to create a dataset of over 4500 different programs in order to benchmark our approach.
- A case study involving over 23000 obfuscated programs, where we build and test regression models to predict the resilience of the obfuscated programs against symbolic execution attacks.

- A set of highly relevant features for predicting the effort of attacks based on symbolic execution.
- A model that can predict the resilience of several code obfuscating techniques against an attack based on symbolic execution, with over 90% accuracy for 80% of the programs in our dataset.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 describes our framework and the C program generator. Section 4 describes the case-study. Section 5 presents conclusions and future work. Acknowledgements are expressed in Section 6. Details regarding the availability our dataset and software tools are given in Section 7.

2 Related Work

Collberg et al. [15] proposed a general taxonomy for evaluating the quality of obfuscating transformations. This taxonomy states that code obfuscation should be evaluated with respect to: *potency* against human-assisted attacks, *resilience* against automated attacks, *cost* (in terms of performance overhead) added by the obfuscating transformation and *stealth*, which measures the difficulty of identifying parts of obfuscated code in a given program. Collberg et al. [15] also proposed using several existing software features to evaluate potency, namely: program length, cyclomatic complexity, nesting complexity, data flow complexity, fan-in/-out complexity, data structure complexity and object oriented design metrics. However, in their empirical studies Ceccato et al. [11] have found that potency does not always correlate with the previous software metrics. Dalla Preda [17] proposes using abstract interpretation to model attackers, which can either break a certain obfuscation transformation or not. However, they do not propose any fine-grained features for measuring resilience. On the other hand, there have also been works that propose measures for resilience. Udupa et al. [46] propose using the edit distance between control flow graphs of the original code and deobfuscated code. Mohsen and Pinto [33] propose using Kolmogorov complexity. Banescu et al. [5] propose using the effort needed to run a deobfuscation attack. However, they do not attempt to predict the effort needed for deobfuscation, which has been identified as a gap in this field [45]. In this paper we focus on predicting the effort needed by an automated deobfuscation attack.

Our work is complementary to the *Obfuscation Executive* (OE) proposed by Heffner and Collberg [23]. The OE uses software complexity metrics and performance measurements to choose a sequence of obfuscating transformations, that should be applied to a program in order to increase its potency, while our paper is solely concerned with resilience. Moreover, the OE also proposes restrictions regarding which obfuscating transformations can follow each other. Our work focuses on prediction of resilience, which is something that the OE does not do. However, our approach could be integrated into the OE to improve the decision making process (see Section 4.4).

Karnick et al. [26] proposed to measure the quality of Java obfuscators by summing up potency and resilience and subtracting cost of memory consumption, file storage size and execution time from the sum. They measure potency with a subset of the features proposed by Collberg et al. [15]. They measure resilience by using concrete implementations of deobfuscators, measuring whether they were successful or if they encountered errors and averaging the measurements across the total number of deobfuscators. We acknowledge that using multiple concrete implementations of a deobfuscation attack (e.g. disassembly, CFG simplification) is important to weed out any issues specific to a particular implementation. However, in this work we aim to provide a more fine-grained measure of deobfuscation effort, instead of a categorical classification such as *succeeded* or *failed* for each deobfuscation attack implementation, as done in [26]. Moreover, we also predict this fine-grained effort.

Anckaert et al. [2] propose applying concrete software complexity metrics on four program properties (i.e. instructions, control flow, data flow and data), to measure resilience. Similarly to our work, Anckaert et al. measure resilience of different obfuscating transformations against concrete implementations of deobfuscation attacks. However, they apply deobfuscation attacks which are specific to different obfuscating transformations, while we use a general deobfuscation attack (based on symbolic execution) on all obfuscating transformations. Moreover, they disregard the effort needed for deobfuscation and measure the effect of different obfuscating transformations on software complexity metrics and the subsequent effect of deobfuscation on these metrics. In this paper we are chiefly concerned with predicting the effort needed to run a successful deobfuscation attack.

Wu et al. [48] propose using a linear regression model over a fixed set of features, for measuring the potency of obfuscating transformations. In contrast to our work, they do not provide any evaluation of their approach. They suggest obtaining the ground truth for training and testing a linear regression model, from security experts who manually deobfuscate the obfuscated programs and indicate the effort required for each program, which is far more expensive compared to our approach of using automated attacks. We obtain our ground truth by running an automated attack and recording the effort (measured in execution time), needed to deobfuscate programs. Moreover, we also propose a way of selecting which features to use for building a regression model.

In sum, Collberg's taxonomy [15] proposes evaluating obfuscation using four dimensions. Most of the related work focuses on simply *measuring* potency, resilience and cost. Wu et al. [48] discuss estimating potency. Zhuang and Freiling [50] propose using a naive Bayes algorithm to estimate the optimal sequence of obfuscating transformations, from a performance point of view. Kanzaki et al. [25] propose code artificiality as a measure to estimate stealth. However, there is a gap in estimating resilience, which we fill in this work.

3 Approach

Resilience is defined as a function of *deobfuscator effort*[1] and *programmer effort* (i.e. the time spent building the deobfuscator) [15]. However, in many cases we can consider the effort needed to build the deobfuscator to be negligible, because an attacker needs to invest the effort to build a deobfuscator only once and can then reuse it or share it with others. Our general approach is illustrated as a work-flow in Figure 1, where ovals depict inputs and outputs of the software tools, which are represented by rectangles. The work-flow requires a dataset of original (un-obfuscated) programs to be able to start (step 0 in Figure 1). To generate these programs we have developed a C code generator presented in Section 3.1. Afterwards, an obfuscation tool is then used to generate multiple obfuscated (protected) versions of each of the original programs (step 1 in Figure 1). Subsequently, an implementation of a deobfuscation attack (e.g. control-flow simplification [49], secret extraction [5], etc.) is executed on all of the obfuscated programs, and the time needed to successfully complete the attack for each of the obfuscated programs is recorded (step 2 in Figure 1). In parallel, feature values (e.g. source code metrics) are extracted from the obfuscated programs.

Once the *attack times* are recorded and software features are extracted from all programs, one could directly use this information to build a regression model for predicting the time needed for deobfuscation. However, some features could be irrelevant to the deobfuscation attack and/or they could be expensive to compute. Moreover, for most regression algorithms the resource usage during the training phase grows linearly or even exponentially with the number of different features used as predictors. Therefore, we add an extra step to our approach, namely a *Feature Selection Algorithm*, which selects only the subset of features which are most relevant to the attack (step 3 in Figure 1). Feature selection can be performed in many ways. Section 3.2 briefly describes how we approached feature selection. After the relevant features are selected, the framework uses this subset of features to build a regression model via a machine learning algorithm (step 4 in Figure 1).

[1]In this paper we quantify deobfuscator effort via the time it takes to run a successful attack on a certain hardware platform; however, note that we could easily map time to CPU cycles, to provide a hardware independent measure of attack effort.

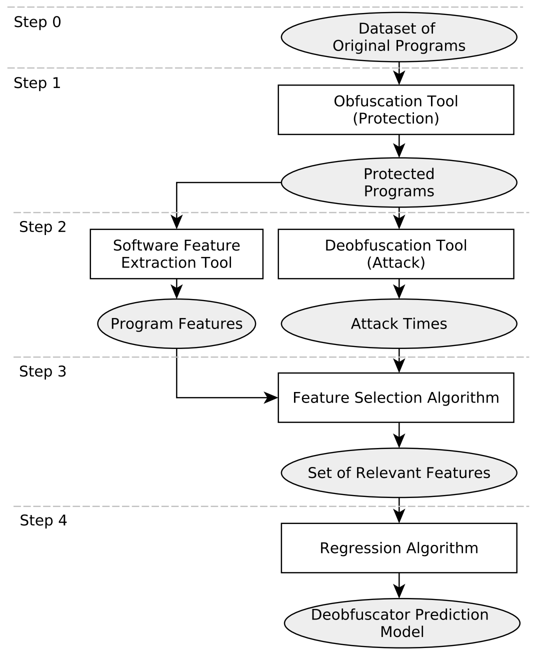

Step 0
Dataset of Original Programs
Step 1
Obfuscation Tool (Protection)
Protected Programs
Step 2
Software Feature Extraction Tool
Deobfuscation Tool (Attack)
Program Features
Attack Times
Step 3
Feature Selection Algorithm
Set of Relevant Features
Step 4
Regression Algorithm
Deobfuscator Prediction Model

Figure 1: General attack time prediction framework.

Note that the proposed approach is not limited to obfuscation and deobfuscation. One can substitute the obfuscation tool in Figure 1, with any kind of software protection mechanism (e.g. code layout randomization [38]) and the deobfuscation tool by any known attack implementation corresponding to that software protection mechanism (e.g. ROPeme [27]). This way the set of relevant features and the output prediction model will estimate the strength of the chosen protection mechanism against the chosen attack implementation.

3.1 C Program Generator

One important challenge of the proposed approach is obtaining a dataset of unobfuscated (original) programs for the input to the framework. This dataset should be large enough to serve as a training set for the regression model in the last step of the framework, because the quality of the model depends on the training set. Ideally, we would have access to a large corpus of open source programs that contain a security check (such as a license check) that needed to be protected against discovery and tampering, as presented in [4]. For example, we could select a collection of such programs from popular code sharing sites such as GitHub. Unfortunately, open source programs tend not to contain the sorts of security checks required by our study. To mitigate this we could manu-

```
 1  void f(int *in, int *out) {
 2    long s[2], local1 = 0;
 3    // Expansion phase
 4    s[0] = in[0] + 762;
 5    s[1] = in[0] | (9 << (s[0] % 16 | 1));
 6    // Mixing phase
 7    while (local1 < 2) {
 8      s[1] |= (s[0] & 15) << 3;
 9      s[(local1 + 1) % 2] = s[local1];
10      local1 += 1;
11    }
12    if (s[0] > s[1]) {
13      s[0] |= (s[1] & 31) << 3;
14    } else {
15      s[1] |= (s[0] & 15) << 3;
16    }
17    s[0] = s[1];
18    // Compression phase
19    out[0] = (s[0] << (s[1] % 8 | 1));
20  }
21  void main(int ac, char* av[]) {
22    int out;
23    f(av[1], &out);
24    if (out == 0xa199abd8)
25      printf("You win!");
26  }
```

Figure 2: Randomly generated program example.

ally insert a security check into a few carefully chosen open source programs. While this would have the advantage of using real code for the study, it does not scale for a large enough dataset. Moreover, we have noticed that in *capture the flag* (CTF) competitions, attackers always seem to locate the license checking code via pattern recognition or taint analysis [41], in order to reduce the part of the code which needs to be symbolically executed. Afterwards, they apply symbolic execution on the license checking code snippet, not on the whole executable code (e.g. built from a GitHub project), which removes the utility of using open source projects in the first place. Since we only want to focus on the second part of this attack (i.e. symbolically executing the license checking code snippet), our C program generator produces a large number of simple programs with diverse license checking algorithms, having a variety of control- and data-flows.

The code generator operates at the function level. Each generated function takes an array of primitive type (e.g. char, int) as input (i.e. `in`) and outputs another array of primitive type (i.e. `out`), as shown in Figure 2. Each function first expands the input array into a (typically larger) state array via a sequence of assignment statements containing operations (e.g. arithmetic, bitwise, etc.) involving the inputs (lines 3-5). After input expansion, the values in the state array are processed via control flow statements containing various operations on the state variables (lines 6-17). Finally, the state array is compressed into the (typically smaller) output array via assignment statements (lines 18-19). These three phases represent a generic way to map data from an input domain to an output domain, as a license check would do.

```
exp := (bb n)
     | (if exp exp)
     | (for exp)
```

Figure 3: `RandomFunsControlStructures` grammar

We implemented this approach as the `RandomFuns` transformation as part of the Tigress C Diversifier/Obfuscator [13]. This transformation offers multiple options[2] that can be tuned by the end user in order to control the set of generated programs. However, here we only provide a description of those options which have been used to generate the dataset of programs used in the experiments from Section 4, i.e.:

- `RandomFunsTypes` indicates the data type of the input, output and state arrays. The current implementation supports the following primitive types: char, short, int, long, float and double.
- `RandomFunsForBound` indicates the type of upper bound in a for loop. The possible types are: (1) a constant value, (2) a value from the input array and (3) a value from the input array modulo a constant.
- `RandomFunsOperators` indicates the allowable operators in the body of the function. Possible values include: arithmetic operators (addition `PlusA`, subtraction `MinusA`, multiplication `Mult`, division `Div` and modulo `Mod`), left shift `Shiftlt`, right shift `Shiftrt`, comparison operators (less than `Lt`, greater than `Gt`, less or equal `Le`, greater or equal `Ge`, equal `Eq`, different `Ne`) and bitwise operators (and `BAnd`, or `BOr` and xor `BXor`).
- `RandomFunsControlStructures` indicates the control structure of the function. If this option is not set, a random structure will be chosen. The value of this option is a string, which follows a simple grammar depicted in Figure 3, where (bb *n*) specifies that the structure should be a basic block with *n* statements, where *n* is an integer. Note that the branch conditions are implicit and randomly generated.
- `RandomFunsPointTest` adds an *if*-statement in the *main* function, immediately after the call to the random function (lines 24-25 in Figure 2). This if statement compares the output of the random function with a constant. If the two values are equal then "You win!" is printed on standard output, indicating that the random function was given an input which led it to execute the true branch of the *if*-statement. Few inputs of the random function take this path, hence, finding such an input is equivalent to finding a valid license key.

[2]For a full list of options and features visit the webpage of `RandomFuns` at `http://tigress.cs.arizona.edu/transformPage/docs/randomFuns`.

The reason why we chose to implement these features is that we suspect them to be relevant for the deobfuscation attack presented in [5], which is used in our case study presented in Section 4. An important limitation of the C code generator is that it does not add system calls inside the generated code. We plan to add this feature in future work.

3.2 Selecting Relevant Features

Given a set of several software features (e.g. complexity metrics), it is unclear which software features one should aim to change (by applying various obfuscating transformations), such that the resulting obfuscated program is more resilient against certain automated deobfuscation attacks. A conservative approach would be to simply use all available software features in order to build a prediction model. However, this approach does not scale for several regression algorithms, because of the large amount of hardware resources needed and also the time needed to train the model. There are several approaches for feature selection published in the literature, e.g. using genetic algorithms [8] or simulated annealing [29]. From our experiments we noticed that such feature extraction algorithms are time-consuming, i.e. even with datasets of the order of tens of thousands of entries and a few dozen features it takes weeks of computation time. We have experimented also with principal component analysis [40], however, this approach did not yield better results for our dataset. Therefore, in this section we describe a few light-weight approaches for selecting a subset of features, which are most relevant for a particular deobfuscation attack. The first approach is based on computing correlations and the second approach is based on variable importance in regression models. In Section 4 we compare these approaches by building regression models using the features selected by each approach.

3.2.1 First approach: Pearson Correlation

One intuitive way to select relevant features, first proposed by Hall [22], is by computing the Pearson correlation [39] between each of the software features and the attack time. The Pearson correlation is a value in the range $[-1,1]$. A positive value means that both the time needed for deobfuscation and the software feature tend to have the same increasing trend, while a negative value indicates that the deobfuscation time decreases as the software feature increases. If the absolute value of this correlation is in the range $[0.8,1]$ the variables are said to be *very strongly correlated*. Furthermore, the range $[0.6,0.8)$ corresponds to *strong correlation*, [0.4, 0.6) to *moderate correlation*, $[0.2,0.4)$ to *weak correlation*, and $(0,0.2)$ to *very weak correlation*. Finally, a value of 0 indicates the absence of correlation. After computing the correlation, we sort the features by their absolute correlation values in descending order and store them in a list L. The caveat in selecting the top ten features with the highest correlation is that several of those top ten features may contain couples which are highly correlated with each other. This means that we could discard one of them and still obtain about the same prediction accuracy. To avoid this issue, for each pair of highly correlated features in L, we remove the one with a lower correlation to the deobfuscation attack time. Afterwards, we select the remaining features with the highest correlations.

3.2.2 Second approach: Variable Importance

Another way of selecting relevant features from a large set of features is to first build a regression model (e.g. via random forest, support vector machines, neural networks, etc.), using all available features and record the prediction error. Concretely, we would:

1. Check the importance of each variable (i.e. feature) using the technique described in [9], i.e. add random noise by permuting values for the i-th variable and average the difference between the prediction error after randomization and before.
2. Repeat this for all $i = \{1,\dots,n\}$, where n is the total number of variables.
3. Rank the variables according to their average difference in prediction error, i.e. the higher the prediction error, the more important the variable is for the accuracy of the regression model.

Similarly, to the previous approach based on Pearson correlation, we select those features which have the highest importance. In order to reduce over-fitting the regression model to our specific dataset, we employ 10-fold-cross-validation, i.e. the dataset is partitioned into 10 equally sized subsets, training is performed on 9 subsets and testing is performed on the remaining subset, for each combination of 9 subsets. Variable importance is averaged over all of these 10 regression models. Then the features are ranked according to their average importance, i.e. difference in prediction error when the values of that variable are permuted. This procedure is called *recursive feature elimination* [21].

4 Case-Study

This section presents a case-study in which we evaluate the approach proposed in Section 3. We are interested in answering the following research questions:

RQ1 Which features are most relevant for predicting the time needed to successfully run the symbolic-execution attack presented in [5]?

RQ2 Which regression algorithms generate models that can predict the attack effort with the lowest error?

Due to space constraints, in this paper we will focus on the deobfuscation attack based on symbolic execution presented in [5], which is equivalent to extracting a secret license key hidden inside the code of the program via obfuscation. However, in future work we plan to apply the approach proposed in Section 3, to other types of automated attacks, such as control-flow simplification [49]. Note that even for other attacks the work-flow from Figure 1 remains unchanged. However, the attack implementation and the software features will change.

4.1 Experimental Setup

All steps of the experiment were executed on a physical machine with a 64-bit version of Ubuntu 14.04, an Intel Xeon CPU having 3.5GHz frequency and 64 GB of RAM. Subsequently we describe the tools that we have used and how we have used them. The following subsections correspond to the steps from 0 to 4 in Figure 1.

4.1.1 Dataset of Original Programs

We have used the code generator described in Section 3.1 to generate a dataset of 4608 unobfuscated C programs. The following is a list of parameters and their corresponding values we used to generate this dataset:

- The random seed value: `Seed` $\in \{1,2,4\}$ (3 values).
- The data type of variables: `RandomFunsTypes` $\in$ {char, short, int, long} (4 values).
- The bounds of *for*-loops: `RandomFunsForBound` $\in$ {constant, input, boundedInput} (3 values).
- The operators allowed in expressions: `RandomFunsOperators` presented in Table 1 (4 values), which also describes each parameter value.
- The control structures: `RandomFunsControlStructures` presented in Table 2 (16 values), which also shows the depth of the control flow.
- The number of statements per basic block was changed via the value of $n \in \{1,2\}$ from Table 2.

The total number of combinations is therefore: $3 \times 4 \times 3 \times 4 \times 16 \times 2 = 4608$. All other parameters were kept to their default values, except for the `RandomFunsPointTest`, which was set to true, meaning that the return value of the randomly generated function is checked against a constant value and if they are equal the program prints a distinctive message, i.e. "You win!" to standard output. We have set this constant value to be equal to the output of the randomly generated function when its input is equal to "12345". Therefore, all of the 4608 programs will print "You win!" on the standard output if their input argument is "12345". The reason for doing this will become clear when we explain the deobfuscation attack in Section 4.1.3.

`RandomFunsOperators` **Parameter Value**	**Description**
`PlusA, MinusA, Lt, Gt, Le, Ge, Eq, Ne`	Simple arithmetic and comparison operators
`PlusA, MinusA, Mult, Div, Mod, Lt, Gt, Le, Ge, Eq, Ne`	Harder arithmetic and comparison operators
`Shiftlt, Shiftrt, BAnd, BXor, BOr, Lt, Gt, Le, Ge, Eq, Ne`	Shift, bitwise and comparison operators
`PlusA, MinusA, Mult, Div, Mod, Lt, Gt, Le, Ge, Eq, Ne, Shiftlt, Shiftrt, BAnd, BXor, BOr`	Harder arithmetic, shift, bitwise and comparison operators

Table 1: *Operator* parameter values given to C code generator used for generating dataset.

`RandomFunsControlStructures` **Parameter Value (see grammar in Figure 3)**	**Ctrl-flow depth**	**Num. of *if*-stmts**	**Num. of Loops**
(if (bb *n*) (bb *n*))	1	1	0
(if (bb *n*))(if (bb *n*))	1	2	0
(if (bb *n*))(if (bb *n*))(if (bb *n*))	1	3	0
(if (if (bb *n*) (bb *n*)) (bb *n*))	2	2	0
(if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*)))	2	3	0
(if (if (if (bb *n*) (bb *n*)) (bb *n*)) (bb *n*))	3	3	0
(if (if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*))) (bb *n*))	3	4	0
(if (if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*))) (if (bb *n*) (bb *n*)))	3	5	0
(for (bb *n*))	1	0	1
(for (if (bb *n*) (bb *n*)))	2	1	1
(for (bb *n*))(for (bb *n*))	1	0	2
(for (for (bb *n*)))	2	0	2
(for (if (if (bb *n*) (bb *n*)) (bb *n*)))	3	2	1
(for (if (bb *n*) (bb *n*))(if (bb *n*) (bb *n*)))	2	2	1
(for (if (if (bb *n*) (bb *n*)) (if (bb *n*) (bb *n*))))	3	3	1
(for (for (if (bb *n*) (bb *n*))))	3	1	2

Table 2: *Control structure* parameter values given to C code generator used for generating dataset.

Since this set of 4608 programs might seem too homogeneous for building a regression model, we used another set of 11 non-cryptographic hash functions[3] in our experiments. Similarly to the randomly generated functions, these hash functions, process the input string passed as an argument to the program and it compares the result to a fixed value. In the case of the hash functions we print a distinctive message on standard output whenever the input argument is equal to "my_license_key". Table 3 shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) set of programs, as computed by the *Unified Code Counter* (UCC) tool [37] and the total number of lines of code (LOC). Each metric was computed on the entire C file of each program, which includes the randomly generated function and the main function. Note that by summing up the metrics on the first 6 rows we obtain the total number of lines of code in our C programs. The important thing to note from Table 3 is that

[3] `http://www.partow.net/programming/hashfunctions/`

Code Metric	Min	Med	Avg	Max
Calculations	10.00	27.00	34.64	152.00
Conditionals	7.00	10.00	10.02	16.00
Logical	4.00	9.00	12.17	69.00
Assignment	9.00	17.00	18.13	46.00
L1.Loops	2.00	3.00	2.85	4.00
L2.Loops	0.00	0.00	0.19	1.00
Total LOC	32.00	66.00	78.00	288.00
Average CC	2.67	3.33	3.21	4.00

Table 3: Overview of un-obfuscated randomly generated programs.

Code Metric	Min	Med	Avg	Max
Calculations	4.00	6.00	6.45	12.00
Conditionals	3.00	3.00	3.27	4.00
Logical	2.00	6.00	5.36	11.00
Assignment	8.00	9.00	9.91	16.00
L1.Loops	1.00	1.00	1.00	1.00
L2.Loops	0.00	0.00	0.00	0.00
Total LOC	18.00	25.00	25.99	44.00
Average CC	2.00	2.00	2.14	2.50

Table 4: Overview of un-obfuscated simple hash programs.

these 4608 programs vary in size and complexity, as was intended, in order to capture a representative range of license checking algorithms.

To increase the number of programs in this set, we generated 275 different variants for each of the non-cryptographic hashes using combinations of multiple obfuscation transformations. The point which we aim to show here is that even if we add a small heterogeneous subset to our larger homogeneous set of programs, the smaller subset is going to be predicted with the same accuracy as the programs from the larger set. Table 4 shows the minimum, median, average and maximum values of various code metrics of only the original (un-obfuscated) non-cryptographic hash functions, as computed by the UCC tool and the total number of lines of code (LOC). Each metric was computed on the entire C file of each program, which includes the hash function and the main function, but no comment lines or empty lines.

4.1.2 Obfuscation Tool

We have used five obfuscating transformations offered by Tigress [13], in order to generate five obfuscated versions of each of the 4608 programs generated by our code generator and the 11 non-cryptographic hash functions. The obfuscating transformations we have used are:

- *Opaque predicates*: introduce branch conditions in the original code, which are either always true or always false for any possible program input. However, their truth value is difficult to learn statically.
- *Literal encoding*: replaces integer/string constants by code that generates their value dynamically.
- *Arithmetic encoding*: replaces integer arithmetic with more complex expressions, equivalent to the original ones.
- *Flattening*: replaces the entire control-flow structure by a flat structure of basic blocks, such that it is unclear which basic block follows which.
- *Virtualization*: replaces the entire code with bytecode that has the same functional semantics and an emulator which is able to interpret the bytecode.

We obfuscated each of the generated programs using these transformations with all the default settings (except for opaque predicates where we set the number of inserted predicates to 16), we obtained $5 \times 4608 = 23040$ obfuscated programs[4]. We obfuscated each of the non-cryptographic hash functions with every possible pair of these 5 obfuscation transformations and obtained $25 \times 11 = 275$ obfuscated programs.

Table 5 and Table 6 show the minimum, median, average and maximum values of various code metrics of the obfuscated set of randomly generated programs, respectively the obfuscated programs involving simple hash functions, as computed by the UCC tool. Each metric was computed on the entire C file of each program, which includes the randomly generated function, the main function and other functions generated by the obfuscating transformation which is applied. For instance, the encode literals transformation generates another function which dynamically computes the values of constants in the code using a switch statement with a branch for each constant. Due to this reason we also notice that after applying the encode literals transformation to a program, its average cyclomatic complexity (CC) is slightly reduced because this function has CC=1 and it is averaged with two other functions with higher CCs. Comparing the numbers in these two table with those from Tables 3 and 4, it is important to note that the size and complexity of the obfuscated programs have increased by one order of magnitude on average, w.r.t. un-obfuscated programs.

4.1.3 Deobfuscation Tool

Since all of the original programs print a distinctive message (i.e. "You win!") when a particular input value is entered, we can define the deobfuscation attack goal as: *finding an input value that leads the obfuscated program to output "You win!", without tampering with the*

[4]Tigress transforms have multiple options that affect the generated code, and makes it more or less amenable to analysis. For this study, we avoid transformations and options that would generate obfuscated code not analyzable by KLEE, which we use as our deobfuscation tool.

Code Metric	Min	Med	Avg	Max
Calculations	22.00	98.00	183.36	870.00
Conditionals	4.00	21.00	105.41	504.00
Logical	4.00	14.00	63.75	458.00
Assignment	10.00	32.00	222.88	1078.00
L1.Loops	2.00	3.00	2.99	10.00
L2.Loops	0.00	0.00	0.25	12.00
Total LOC	42.00	168.00	578.64	2932.00
Average CC	1.80	5.25	15.73	66.75

Table 5: Overview of obfuscated randomly generated programs.

Code Metric	Min	Med	Avg	Max
Calculations	18.00	27.00	127.70	350.00
Conditionals	3.00	10.00	100.81	444.00
Logical	2.00	6.00	54.45	240.00
Assignment	11.00	17.00	217.36	963.00
L1.Loops	1.00	1.00	1.02	2.00
L2.Loops	0.00	0.00	0.36	3.00
Total LOC	35.00	61.00	501.70	2002.00
Average CC	1.50	3.33	18.80	76.00

Table 6: Overview of obfuscated simple hash programs.

program. As presented in [4], this deobfuscation goal is equivalent to finding a hidden secret key and can be achieved by employing an automated test case generator. A state of the art approach for test case generation is called *dynamic symbolic execution* (often it is simply called symbolic execution). Such an approach is implemented by several free and open source software tools such as KLEE [10], angr [44], etc. The first step of symbolic execution is to mark a subset of program data (e.g. variables) as symbolic, which means that they can take any value in the range of its type. Afterwards, the program is interpreted and whenever a symbolic value is involved in an instruction, its range is constrained accordingly. Whenever a branch based on a symbolic value is encountered, symbolic execution forks the state of the program into two different states corresponding to each of the two possible truth values of the branch. The ranges of the symbolic variable in these two forked states are disjoint. This leads to different constraints on symbolic variables for different program paths. The symbolic execution engine sends these constraints to an SMT solver, which tries to find a concrete set of values (for the symbolic variables), which satisfy the constraints. Giving the output of the SMT solver as input to the program will lead the execution to the path corresponding to that constraint.

Since we have the C source code for the obfuscated programs, we chose to use KLEE as a test case generator in this study. We ran KLEE with a symbolic argument length of 5 characters, on all of the un-obfuscated and obfuscated programs generated by our code generator, for 10 times each. All of the symbolic executions successfully generated a test case where the input was “12345”, which is the input needed to achieve the attacker goal. Similarly we ran KLEE with a symbolic argument length of 16 characters, on all of the un-obfuscated and obfuscated non-cryptographic hash functions, for 10 times each. Again the correct test cases were generated on all symbolic executions, but this time the input was “my_license_key”. Note that this is only one way to attack an obfuscated program, and that it does not produce a simplified version of the obfuscated code as in [49]. Rather, it extracts a hidden license key value from the obfuscated code. We computed the mean (M) and the standard deviation (SD) of the reported times across all the 10 runs of KLEE and obtained that 83% of the programs have a relative standard deviation ($RSD = SD/M$) under 0.25 and 94% have $RSD \leq 0.50$. This means that the difference between multiple runs of KLEE on the same program is small.

4.1.4 Software Feature Extraction Tools

Many papers [32, 43, 1, 4] suggest that the complexity of branch conditions is a program characteristic with high impact on symbolic execution. However, these papers do not clearly indicate how this complexity should be measured. One way to do this is by first converting the C program into a *boolean satisfiability problem* (SAT instance), and then extracting features from this SAT instance. There are several tools that can convert a C program into a SAT instance, e.g. the C bounded model checker (CBMC) [12] or the low-level bounded model checker (LLBMC) [31], etc. However, the drawback of these tools is that the generated SAT instances may be as large as 1GBs even for programs containing under 1000 lines of code, because they are not optimized. Hence, for our dataset, the generated SAT instances would require somewhere in the order of 10TBs of data and several weeks of computational power, which is prohibitively expensive.

Instead, we took a faster alternative approach for obtaining an optimized SAT instance from a C program, which we describe next. KLEE generates a *satisfiability modulo theories* (SMT) instance for each execution path of the C program. We selected the SMT instance corresponding to the difficult execution path that prints out the distinctive message on standard output[5]. These SMT

[5]Note that it is not necessary to execute KLEE to obtain the SMT instance corresponding to the difficult execution path. The developer knows the correct license key, therefore s/he can give the correct license key and record the instruction trace of the execution. Afterwards, the developer can substitute the constant input argument in the trace, with a symbolic input and then extract the path constraint by combining all expressions in the trace. This path constraint does not need to be solved by the SMT solver, but simply converted to SAT.

instances (corresponding to the difficult path), were the most time-consuming to solve by KLEE's SMT solver, STP [19]. Many SMT solvers, including Microsoft's Z3 [18], internally convert SMT instances to SAT instances in order to solve them faster. Therefore, we modified the source code of Z3 to output the internal SAT instance, which we saved in separate files for each of the programs in our dataset. For extracting features from these SAT instances we used SATGraf [36], which computes graph metrics for SAT instances, where each node represents a variable and there is an edge between variables if they appear in the same clause. SATGraf computes features such as the number of community structures in the graph, their modularity (Q value), and also the minimum, maximum, mean and standard deviation of nodes and edges, inside and between communities. Such features have been shown to be correlated with the difficulty of solving SAT instances [35]. Therefore, since symbolic execution includes many queries to an SMT/SAT solver, as shown in [4], these features are expected to be good predictors of the time needed for a symbolic execution based deobfuscation attack. In sum, we transform the path that corresponds to a successful deobfuscation attack into a SAT instance (via an SMT instance), and then compute characteristics of this formula, to be used as features for predicting the effort of deobfuscating the program.

For computing source code features often used in software engineering, on both the original and obfuscated programs, we used the *Unified Code Counter* (UCC) tool [37]. This tool outputs a variety of code metrics including three variations of the McCabe cyclomatic complexity, their average and the number of: calculations, conditional operations, assignments, logical operations, loops at three different nesting levels, pointer operations, mathematical operations, logarithmic operations and trigonometric operations. For the programs in our dataset the last four metrics are all zeros, therefore, in our experiments we only used the other eleven metrics. Additionally, we also propose using four other program features, namely: the execution time of the program, the maximum RAM usage of the program, the compiled program file size and the type of obfuscating transformation.

In total we have 64 features out of which 49 are SAT features which characterize the complexity of the constraints on symbolic variables and 15 are program features which characterize the structure and size of the code. In the following we show that not all of these features are needed for good prediction results.

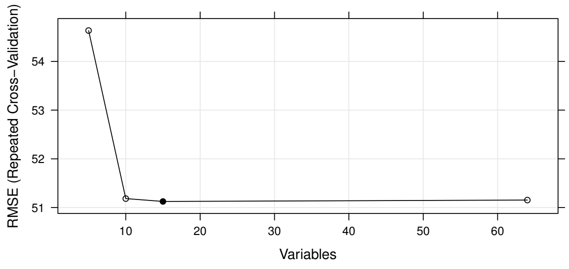


Figure 4: RF models with different feature subsets.

4.1.5 Regression Algorithms

For the purpose of regression we have used the R software environment[6] for statistical computing. R provides several software packages for regression algorithms out of which we used *e1071, randomForest, rgp* and *h2o*:

- The "e1071" package for regression via the *support vector machine* (SVM) algorithm.
- The "randomForest" package for regression via the *random forest* (RF) algorithm.
- The "rgp" package for regression via *genetic programming* (GP).
- The "h2o" package for regression via *neural networks* (NNs).

4.2 Feature Selection Results

This section presents the results for the *Feature Selection Algorithms* presented in Section 3.2. However, before selecting the most relevant features, we identify how many features (predictor variables) are needed to get good prediction results. For this purpose we performed a 10-fold-cross validation with linear and random forest (RF) regression models using all combinations of 5, 10 and 15 metrics, as well as a model with all metrics. The results in Figure 4 show that using 15 variables is enough to obtain an RF model with *root-mean-squared-error* (RMSE) values which are as good as those from RF models built using all variables. Similar results were obtained for linear models, except that the overall RMSE was higher w.r.t. that of the RF models. Therefore, in the experiments presented in the following sections, we will only select the top best 15 features in both of the two approaches described in Section 3.2.

4.2.1 First approach: Pearson Correlation

After employing the algorithm described in Section 3.2.1, we were left with a set of 25 features, with their Pearson correlation coefficients ranging from

[6]`https://www.r-project.org/`

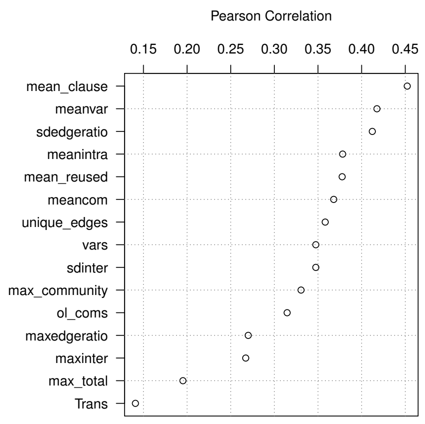


Figure 5: Top 15 features via first approach.

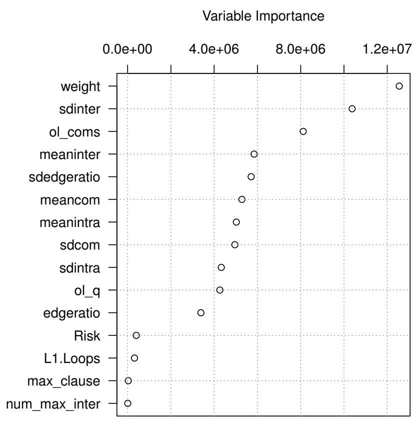


Figure 6: Top 15 features via second approach.

0.4523 to -0.0302. The top 15 metrics in this range are shown in Figure 5. The strongest Pearson correlation of the time needed for running the deobfuscation attack is with the average size of clauses in the SAT instance (*mean_clause*), followed by: the average number of times any one variable is used (*meanvar*), the standard deviation of the ratio of inter to intra community edges (*sdedgeratio*), the average number of intra community edges (*meanintra*), the average number of times a clause with the same variable (but different literals) is repeated (*mean_reused*), the average community size (*meancom*), the number of unique edges (*unique_edges*), the number of variables (*vars*), the standard deviation of the number of inter community edges (*sdinter*), the maximum number of distinct communities any one community links to (*max_community*), the number of communities detected with the online community detection algorithm (*ol_coms*), the maximum ratio of inter to intra community edges within any community (*maxedgeratio*), the maximum number of inter community edges (*maxinter*), the maximum number of edges in a community (*max_total*) and finally the type of obfuscation transformation employed.

None of the previous features are very strongly correlated to deobfuscation time. The first three features are moderately correlated, the following ten features are weakly correlated and finally the last two features are very weakly correlated. However, notice that the top fourteen features are all SAT features, and none are code metrics from the UCC tool or program features such as execution time, memory usage or file size.

4.2.2 Second approach: Variable Importance

To rank our features according to variable importance we performed recursive feature elimination via random forests, as indicated in Section 3.2.2. Figure 6 shows the top 15 features sorted by their variable importance.

The features selected using this approach are quite different from those selected in Section 4.2.1. The common features between these two approaches are: *sdinter, ol_coms, sdedgeratio, meancom* and *meanintra*. The first two common features are ranked 2nd and 3rd according to variable importance, however, the most important feature w.r.t. variable importance is the weight of the graph (*weight*), computed as the sum of positive literals minus the sum of negative literals. The 4th most important variable in Figure 6 is the average number of inter community edges (*meaninter*), followed by: *sdedgeratio, meancom, meanintra* (see descriptions of these 3 features in Section 4.2.1), the standard deviation of community sizes (*sdcom*), the standard deviation of intra community edges (*sdintra*), the modularity of the SAT graph structure (*ol_q*), the overall ratio of inter to intra community edges (*edgeratio*), the category of the McCabe cyclomatic complexity [30] (*Risk*), the number of outer-loops (*L1.Loops*), the size of the longest clause (*max_clause*) and the number of communities that have the maximum number of inter community edges (*num_max_inter*).

Similarly, to the first approach, the majority of selected features are SAT features. The only two features which are not SAT features are *Risk* and *L1.Loops* which are computed by the UCC tool. The number of loops was indeed indicated also in [4] as being an important feature. The *Risk* has four possible values depending on the value of the cyclomatic complexity (CC), i.e. low if CC $\in [1,10]$, moderate if CC $\in [11,20]$, high if CC $\in [21,50]$ and very high if CC is above 50. CC gives a measure of the complexity of the branching structure in programs (including if-statements, loops and jumps). However, it is remarkable that the CC value was ranked lower than the *Risk*.

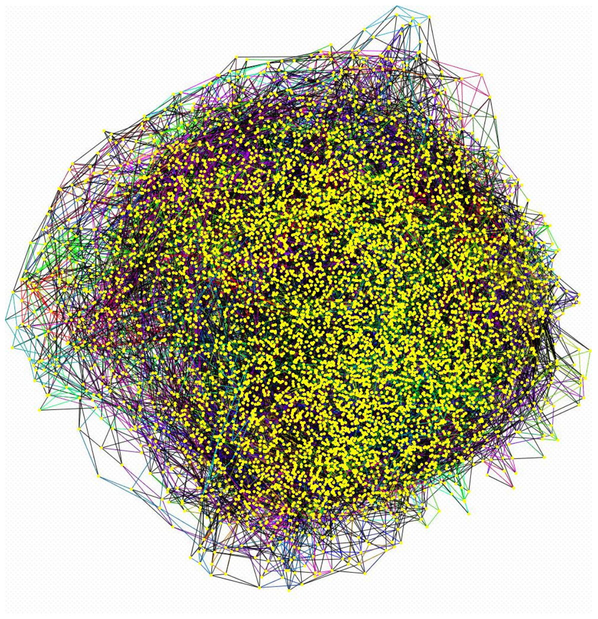


Figure 7: Graph representation of SAT instance corresponding to an MD5 hash with 27 rounds. Solving this instance takes approximately 25 seconds on our testbed.

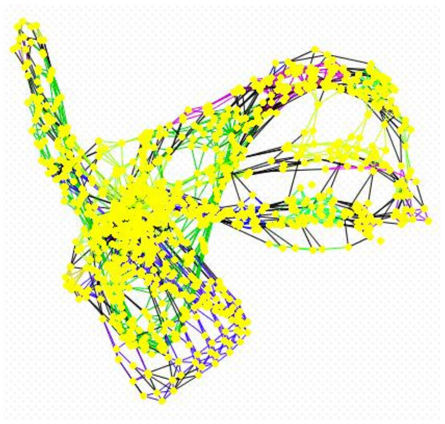


Figure 8: Graph representation of SAT instance corresponding to a program whose symbolic execution time is under 1 second.

4.2.3 Insights from Feature Selection Results

SAT features are important for symbolic execution, because most of the time of the attack is spent waiting for the SAT solver to find solutions for path constraints [4]. Taking a closer look at the common SAT features of both feature selection approaches, we can characterize those SAT instances, which are harder to solve. The graph representation of such an instance has a large number of *balanced* community structures, i.e. a similar number of intra- and inter-community edges. On the other hand, easy to solve instances tend to have *established* community structures, i.e. many more intra-community, than inter-community edges. To check this observation, we downloaded the Mironov-Zhang [32] and the Li-Ye [28] benchmark suites for SAT solvers, containing solvable versions of more realistic hash functions such as MD5 and SHA. All of these instances had *balanced* community structures. For example, Figure 7 illustrates the graph representation of the SAT instance[7] of the MD5-27-4 hash function of the Li-Ye benchmark suite[28] proposed during the 2014 SAT Competition. It is visible – from the number of yellow dots – that this graph has a high number of variables. More importantly it is also visible that one cannot easily distinguish graph community structures, because they are relatively small and well connected with other communities. This kind of structure is hard to solve, because each assignment of a variable has a large number of connections and therefore ramifications inside the graph at the time when *unit propagation* is performed by the SAT solver. However, note that if the graph is fully connected, then it is easy to solve. Therefore, there is a fine line between having too many connections and too few, where the difficulty of SAT instances increases dramatically. This last observation is similar to the *constrainedness of search* employed by Gent et al. [20], when analyzing the likelihood of finding solutions to different instances of the same search problem. This makes sense since a SAT solver is executing a search when it is trying to solve a SAT instance.

On the other hand, many of our randomly generated C programs which were fast to deobfuscate, had *established* community structures. For example, Figure 8 illustrates the graph representation of a program generated using our C code generator. This program was generated with the following parameter values:

- `RandomFunsTypes` was set to int.
- `RandomFunsForBound` was set to a constant value.
- `RandomFunsOperators` was set to `Shiftlt`, `Shiftrt`, `Lt`, `Gt`, `Le`, `Ge`, `Eq`, `Ne`, `BAnd`, `BOr` and `BXor`.
- `RandomFunsControlStructures` was set to (if (if (if (bb n) (bb n)) (if (bb n) (bb n))) (if (bb n) (bb n))).
- $n = 1$.
- `RandomFunsPointTest` was set to true.

Given these parameter values, this instance is expected to be fast to solve, because it does not involve any loops dependent on symbolic inputs and it only involves logical and bitwise operators.

[7]These graph representations were generated using the SATGraf tool [36].

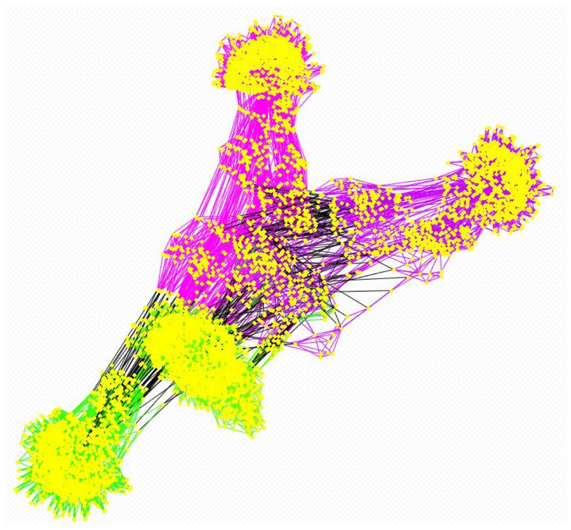


Figure 9: Graph representation of SAT instance corresponding to a non-cryptographic hash function which is solved in about 7.5 seconds.

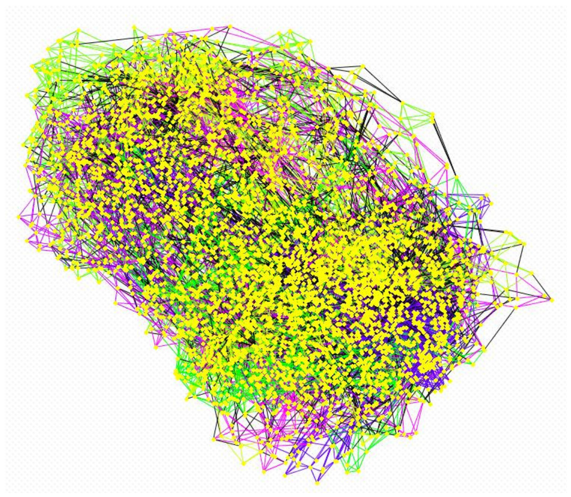


Figure 10: Graph representation of SAT instance corresponding to same non-cryptographic hash function from Figure 9, after being obfuscated with *virtualization* and subsequently control-flow *flattening*. This instance is solved in about 438 seconds.

In this context of representing SAT instances as graphs, it is interesting to note the effect of obfuscation transformations on SAT instances. For instance, Figure 9 illustrates the SAT instance of a non-obfuscated, non-cryptographic hash function from our dataset. The community structures of this hash function are *established*, hence, the instance can be solved in about 7.5 seconds. However, after applying two layers of obfuscation, first using the *virtualization* and then the *flattening*, transforms the SAT instance of this program into the one illustrated in Figure 10. This instance, has a *balanced* community structure, hence, slower to solve (438 seconds) and shares a resemblance to the MD5 instance from Figure 7. We have also noticed that the *arithmetic encoding* transformation has this effect on SAT instances. However, the *opaque predicate* and *literal encoding* alone do not have such an effect.

As a conclusion of this section we observe that balanced community structures translate to a high diffusion of the symbolic input to output bits, i.e. affecting any bit of the input license key will affect the result of the output. This is the case for collision-resistant hash functions, as well as the effect of obfuscation transformations like *virtualization*, *flattening* and *arithmetic encoding*.

4.3 Regression Results

For each of the regression algorithms presented next, we have used several different configuration parameters. Due to space limitations, we only present the configuration parameters which gave the best results. We randomly shuffled the programs in our 2 datasets of programs into one single dataset and performed 10-fold cross-validation for each experiment. To interpret the *root-mean-squared-error* (RMSE) we normalize it by the range between the fastest and slowest times needed to run the deobfuscation attack on any program from our dataset. Since our dataset contains outliers (i.e. either very high and very low deobfuscation times), the normalized RMSE (NRMSE) values are very low for all algorithms, regardless of the selected feature subsets, as shown in Table 7. This could be misinterpreted as extremely good prediction accuracy regardless of the regression algorithm and feature set. However, we provide a clearer picture of the accuracy of each regression model by computing the NRMSEs after removing 2% and 5% of outliers from both the highest and the lowest deobfuscation times in the dataset. This means that in total we remove 4%, respectively 10% of outliers. Instead of showing just the numeric values of the NRMSE for each these three cases (0%, 4% and 10% of outliers removed), we show cumulative distribution functions of the relative (normalized) error in the form of line plots, e.g. Figure 11. These line plots show the maximum and the median errors for all the three cases, where the x-axis represents the percentage of programs for which the relative error (indicated on the y-axis) is lower than the plotted value.

Note that in addition to the following regression algorithms we have also employed both linear models and generalized linear models [34]. However, the results of the models generated by these algorithms were either much worse compared to the models presented in the following, or the models did not converge after 24 hours.

	SVM	RF	GP	NN
UCC (11 features)	0.019	0.016	0.018	0.018
Pearson (15 features)	0.017	0.013	0.015	0.015
Var. Importance (15 features)	0.019	0.013	0.015	0.015

Table 7: The NRMSE between model prediction and ground truth (average over NRMSE of 10 models)

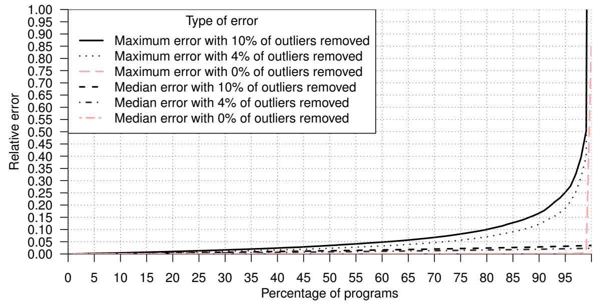


Figure 11: Relative prediction error of RF model.

4.3.1 Random Forests (RFs)

Random forests (RFs) were proposed by Breiman [9] as an extension of random feature extraction, by including the idea of "bagging", i.e. computing a mean of the prediction of all random decision trees. In our experiments we constructed a RF containing 500 decision trees.

Figure 11 shows the maximum and median relative errors for 0%, 4% and 10% of outliers removed. As more outliers are removed the relative error increases due to a decrease in the range of deobfuscation times in the dataset. However, even when 10% of outliers are removed, the maximum error is under 17% and the median error is less than 4% for 90% of the programs, which seems acceptable for most use cases.

Note that the model in Figure 11 was built using the 15 features selected via variable importance, presented in Section 4.2.2. We chose to show the results from the model built using these features because, they are better than those produced by models built using other subsets of features. As we can see from Figure 12, the relative error values when building models with UCC metrics only and with the Pearson correlation approach, give worse results in terms of both maximum and median error rates.

4.3.2 Support Vector Machines (SVMs)

Support vector machines (SVMs) were proposed by Cortes and Vapnik [16] to classify datasets having a high number of dimensions, which are not linearly separable.

Figure 13 shows the relative errors for the SVM model built using the features selected via the second approach (see Section 4.2.2). The accuracy of this model is lower than the RF model from Figure 11, i.e. the maximum relative error is just below 35% for 90% of the programs, when we remove 10% of the outliers. However, the median error is less than 7% in the same circumstances. The reason why SVM performs worse than RF is due to the bagging technique applied by RF, whereas SVM uses a single non-linear function.

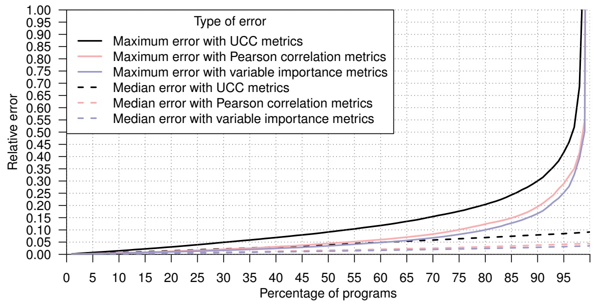


Figure 12: RF models with different feature sets.

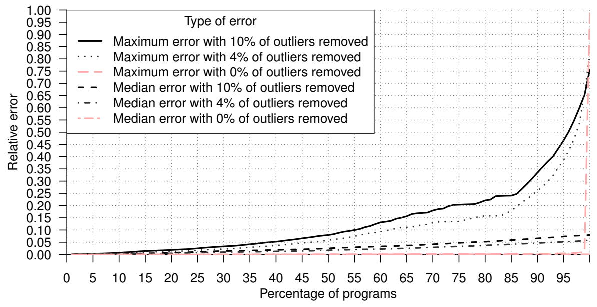


Figure 13: Relative prediction error of SVM model.

Again we chose to show the SVM model built using the features selected via variable importance in Figure 13, because, as we can see from Figure 14, the maximum and median error rates for this model are much lower than the SVM models built using only UCC metrics or the features selected via Pearson correlation. However, note that the maximum error of the model built using variable importance surpasses that of the other two models around the 90% mark on the horizontal axis. This means that for 10% of the programs the maximum error of the model built using the features selected by variable importance, is higher that the error of the other two models. However, note that the median error is around 10% lower in the same circumstances.

4.3.3 Genetic Programming (GP)

Given the set of all code features as a set of *input variables*, GP [24] searches for models that combine the input variables using a given *set of functions* used to process and combine these variables, i.e. addition, multiplication, subtraction, logarithm, sinus and tangent in our experiments. GP aims to optimize the models such that a

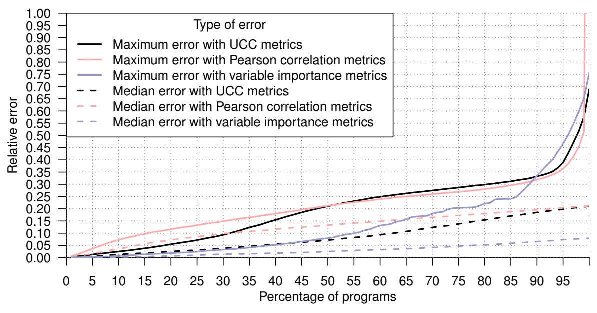


Figure 14: SVM models with different feature sets.

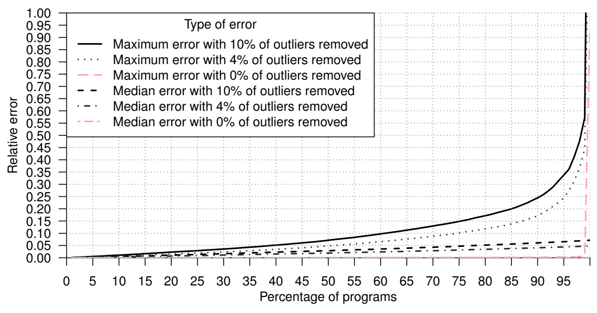


Figure 16: Relative prediction error of NN model.

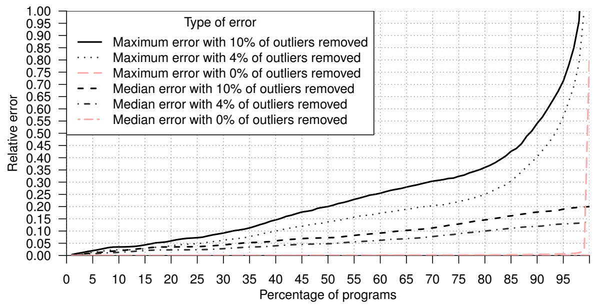


Figure 15: Relative prediction error of GP model.

given *fitness function* is minimized. For our experiments, we used the *root-mean-square error* (RMSE) between the actual time needed for deobfuscation and the time predicted by the model, as a fitness function. The output of GP is one of the generated models with the best fitness value. In our case this member is a function of the code features, which has the smallest error in predicting the time needed to execute the deobfuscation attack on every program. For instance, the best GP model built using the features selected via variable importance is presented in equation 1:

$$\begin{aligned} time = {} & (\text{edgeratio} + cos(\text{ol\_coms}) \\ & + cos(cos(\text{sdcom} + \text{num\_max\_inter}) + \text{L1.Loops})) \\ & * (\text{sdinter} * (\text{sdedgeratio} - sin(\text{meanintra} * -1.27))) \\ & * (\text{sdedgeratio} - sin(\text{meanintra} * -1.27)) \\ & * (1.03 - sin(0.04 * \text{sdinter})) \\ & * \text{sdedgeratio} + 10.2 \end{aligned} \tag{1}$$

Note that only seven distinct features were selected by the GP algorithm for this model, from the subset of 15 features. Figure 15 shows the maximum and median error values for the GP model from equation 1. Note that the maximum and median error levels for the dataset where 10% of outliers are removed, are 55%, respectively 19% for 90% of the programs. This error rate is much higher than both RFs and SVMs and is due to the fact that the GP model is a single equation.

4.3.4 Neural Networks (NNs)

Multi-layer neural networks (NNs) were introduced by Werbos [47] in the 1970s. Recently, the interest in NNs has been revived due to the increase in computational resources available in the cloud and in graphical processing units. A neural network has three characteristics. Firstly, the *architecture* which describes the number of neuron layers, the size of each layer and connection between the neuron layers. In our experiments we used a NN with five hidden layers each containing 200 neurons. The input layer consists of the set of code features and the output of the NN is a single value that predicts the time needed to run the deobfuscation attack on a program. Secondly, the *activation function* which is applied to the weighted inputs of each neuron. This function can be as simple as a binary function, however it can also be continuous such as a Sigmoid function or a hyperbolic tangent. In our experiments we use a ramp function. Thirdly, the *learning rule* which indicates how the weights of a neuron's input connections are updated. In our experiment we used the *Nesterov Accelerated Gradient* as a learning rule.

Figure 16 shows the maximum and median error of the NN model built using all metrics. Note that in the case of NNs it is feasible to use all metrics without incurring large memory usage penalties such as is the case for SVMs. The performance of this model is better than the SVM and GP models, but not better than the RF model.

4.4 Summary of Results

Based on the results presented above, we answer the research questions elicited in the beginning of Section 4. Firstly, in Figure 4 we have seen that given our large set of 64 program features, using only 15 is enough to obtain regression models with RMSEs as low as the regression models where all the features are used. From Figures 5 and 6 we have seen that both approaches to feature selection ranked SAT features above code metrics commonly used to measure resilience, namely cyclomatic complexity or the size of the program. This means that the most

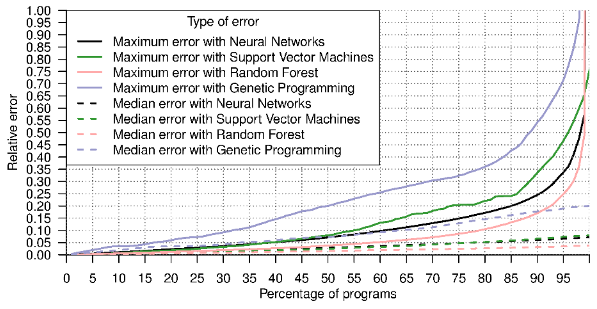


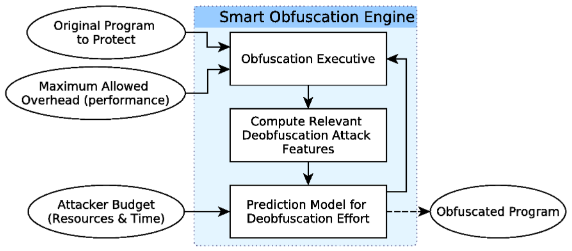
Figure 17: Comparison of regression algorithms.

important characteristics for symbolic execution based attacks is the complexity of the constraints on symbolic variables. The reason why SAT features have a higher impact on symbolic execution is that most of the time during symbolic execution is spent waiting for the SMT solver and these features indicate the time that is needed by the SMT solver to find a counter example for path constraints.

Secondly, Table 7 shows the RMSE for different regression models normalized by the fastest and slowest deobfuscation attacks in our dataset. Since our dataset contains outliers, the results from Table 7 are misleading. Therefore we removed 4% and 10% of the outliers from our dataset and plotted the cumulative distribution of the errors for each of the regression models. From Figures 12 and 14 we observe that the second approach to feature selection, based on variable importance, gives better results than the first approach, based on Pearson correlation. Therefore, in Figure 17 we plot the maximum and median errors of the models from the four different regression algorithms, where 10% of outliers are removed from the dataset. From this figure we observe that RF has the lowest overall maximum error rate, followed by NN, SVM and GP. However, the median error of the RF, NN and SVM models are all lower than 8% for all programs. This indicates that if the median error is the key performance indicator, it is much less important whether we pick RF, NN or SVM as the regression algorithm.

Another observation is that the size of the prediction models from RF models are generally smaller than those of SVM models. However, models obtained from GP and NN are one, respectively two orders of magnitude smaller than RF models. The size of SVM, RF and GP models grows proportionally to the number of features used. An advantage of NN models is their relatively small size of around 50 Kilobytes is constant for any number of features used. This is understandable because the number of weights and neurons is negligibly influenced by the number of features used to build the model.

In sum, the most relevant features for characterizing



Original Program to Protect
Maximum Allowed Overhead (performance)
Attacker Budget (Resources & Time)
Smart Obfuscation Engine
Obfuscation Executive
Compute Relevant Deobfuscation Attack Features
Prediction Model for Deobfuscation Effort
Obfuscated Program

Figure 18: Combining results with obfuscation tools.

the deobfuscation attack based on symbolic execution, are SAT features (RQ1). Moreover, the regression algorithm which yields the highest prediction accuracy is random forest (RQ2).

These results can be used to build the *Smart Obfuscation Engine* (SObE) shown in Figure 18, where the ovals represent inputs and outputs. SObE takes three inputs: (1) the original program source code, (2) the maximum allowed performance overhead of the resulting obfuscated program and (3) the resource and time available to the attacker (attacker budget). SObE first gives the original program to the *Obfuscation Executive* (OE) [23] (see Section 2), which applies a set of obfuscation transformations that satisfy the maximum allowed overhead. Afterwards, SObE computes the relevant features (determined in Section 4.2) on the obfuscated program and then uses the best prediction model from Section 4.3 to estimate the effort needed by the deobfuscation attack. If the effort is less than the attacker's budget, then this is signaled to the OE and the process restarts, otherwise the obfuscated program is output.

4.5 Threats to Validity

In our case study, we have generated a dataset of unobfuscated programs of up to a few 100s of lines of code (LOC). Obfuscating these programs generates programs having up to a few 1000s of LOC. Therefore, the regression models generated in the case study may not be accurate for all possible programs. However, in our ex-

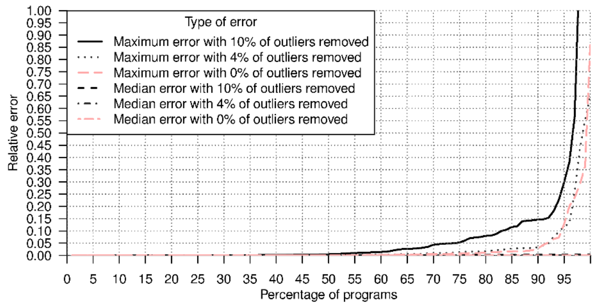


Figure 19: Relative error of hash functions only.

periments we have found that the size of the program is very weakly correlated with the time needed to run the deobfuscation attack based on symbolic execution. We show that the prediction accuracy of our best RF model (from Figure 11) is high even when including a small non-artificial dataset of programs containing non-cryptographic hash functions. Figure 19 shows the prediction error of our best RF model (trained using 10-fold-cross-validation on both datasets), for the samples in the smaller dataset alone, has similar levels to the prediction error of the entire dataset.

We also performed a reality check, i.e. we verified that the SAT features we identified are also relevant for the realistic hash functions from the Mironov-Zhang [32] and the Li-Ye [28] benchmark suites for SAT solvers. We selected the top 10 SAT metrics from Section 4.2 and trained a random forest (RF) model using the SAT instances corresponding to the C programs in our obfuscated dataset of randomly generated programs and non-cryptographic hash functions. Afterwards, we applied this RF model to a set of more realistic hash functions from the Mironov-Zhang [32] and the Li-Ye [28] benchmark suites for SAT solvers, containing solvable versions of more realistic hash functions such as MD5 and SHA. Table 8 shows the results obtained from applying the RF model to the hash functions, which were solvable by the *minisat* solver used by STP (KLEE's SMT solver), on our machine. Note that the Li-Ye [28], suite contains many other instances of MD5 with more rounds, however, those could not be solved within a 10 hour time limit on our test machine. The last column of Table 8 gives the ratio between the predicted and the actual time needed to solve each instance. Except for the *mizh-md5-47-4* and *mizh-md5-47-5* SAT instances, which are the most over- and respectively under-estimated, the rest of the predictions are quite encouraging, given that we have not trained the RF model with any such realistic SAT instances. Therefore, we obtained encouraging results with a median prediction error of 52%, which is quite remarkable given the fact that our model was not trained using these realistic instances.

Instance Name	Solver(s)	Predicted(s)	$\frac{Predicted}{Solver}$
MD5-27-4	25.37	71.56	2.82
mizh-md5-47-3	681.29	950.43	1.39
mizh-md5-47-4	235.53	1069.19	4.53
mizh-md5-47-5	1832.96	437.98	0.23
mizh-md5-48-2	445.19	523.70	1.17
mizh-md5-48-5	227.05	644.38	2.83
mizh-sha0-35-2	330.48	158.57	0.47
mizh-sha0-35-3	139.93	213.03	1.52
mizh-sha0-35-4	97.62	214.61	2.19
mizh-sha0-35-5	164.71	193.49	1.17
mizh-sha0-36-2	85.44	222.07	2.59

Table 8: Prediction results of realistic hash functions via RF model trained with SAT features from Section 4.2. The solver and predicted time are given in seconds.

5 Conclusions

This paper presents a general approach towards building prediction models that can estimate the effort needed by an automated deobfuscation attack. We evaluated our approach using a dataset of programs produced by our C code generator. For programs that our generated dataset is representative, features such as the complexity of path constraints (measured via SAT features), are more important than cyclomatic complexity, size of the program, number of conditional operations, etc. With a median error of 4% our best model can accurately predict the time it takes to deobfuscate a program using a symbolic execution based attack, for programs in our dataset. Moreover, we have also obtained encouraging results with realistic hash functions such as MD5 and SHA instances used in SAT competitions.

Note however, that our framework is not specific to symbolic execution and can be used for other attacks, other programs and other obfuscators. Finally, we compared different regression algorithms both in terms of prediction error and memory consumption and conclude that the choice of regression algorithm is less important than the choice of features when it comes to predicting the effort needed by the attack. However, we obtained the lowest maximum error using a random forest model, built with features selected using variable importance. In terms of memory usage genetic algorithms and neural networks have a lower memory footprint, however their training times may be much higher.

In future work we plan to use datasets consisting of real-world programs, additional obfuscation tools and deobfuscation attacks. We believe that obtaining representative datasets of programs would also be of paramount importance for benchmarking both new and existing obfuscation and deobfuscation techniques. Therefore, we believe this area of research needs much more work, since it could be a driving factor for the field of software protection.

Another avenue for future work is to employ other machine learning techniques in order to derive better prediction models for deobfuscation attacks. An interesting idea in this direction is deriving attack resilience features using deep neural networks. However, such a task would also require a set of representative un-obfuscated programs, which stresses the importance of future work in this direction.

6 Acknowledgments

The authors would like to thank the anonymous reviewers, Prof. Vijay Ganesh and Zack Newsham for the feedback regarding the SATGraf tool and tips for converting SMT instances into SAT instances. This work was partially supported by NSF grants 1525820 and 1318955.

7 Availability

Our code generator is part of the Tigress C Diversifier/Obfuscator tool. Binaries are freely available at:

`http://tigress.cs.arizona.edu/transformPage/docs/randomFuns`

Source code is available to researchers on request. Our dataset of original (unobfuscated) programs, as well as all scripts and auxiliary software used to run our experiments, are available at:

`https://github.com/tum-i22/obfuscation-benchmarks/`

References

[1] ANAND, S., BURKE, E. K., CHEN, T. Y., CLARK, J., COHEN, M. B., GRIESKAMP, W., HARMAN, M., HARROLD, M. J., MCMINN, P., ET AL. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software 86*, 8 (2013), 1978–2001.

[2] ANCKAERT, B., MADOU, M., DE SUTTER, B., DE BUS, B., DE BOSSCHERE, K., AND PRENEEL, B. Program obfuscation: a quantitative approach. In *Proc. of the ACM workshop on Quality of protection* (2007), ACM, pp. 15–20.

[3] APON, D., HUANG, Y., KATZ, J., AND MALOZEMOFF, A. J. Implementing cryptographic program obfuscation. *IACR Cryptology ePrint Archive 2014* (2014), 779.

[4] BANESCU, S., COLLBERG, C., GANESH, V., NEWSHAM, Z., AND PRETSCHNER, A. Code obfuscation against symbolic execution attacks. In *Proc. of 2016 Annual Computer Security Applications Conference* (2016), ACM.

[5] BANESCU, S., OCHOA, M., AND PRETSCHNER, A. A framework for measuring software obfuscation resilience against automated attacks. In *1st International Workshop on Software Protection* (2015), IEEE, pp. 45–51.

[6] BANESCU, S., OCHOA, M., PRETSCHNER, A., AND KUNZE, N. Benchmarking indistinguishability obfuscation - a candidate implementation. In *Proc. of 7th International Symposium on ESSoS* (2015), no. 8978 in LNCS.

[7] BARAK, B. Hopes, fears, and software obfuscation. *Communications of the ACM 59*, 3 (2016), 88–96.

[8] BARLAK, E. S. Feature selection using genetic algorithms.

[9] BREIMAN, L. Random forests. *Machine learning 45*, 1 (2001), 5–32.

[10] CADAR, C., DUNBAR, D., AND ENGLER, D. R. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI* (2008).

[11] CECCATO, M., CAPILUPPI, A., FALCARIN, P., AND BOLDYREFF, C. A large study on the effect of code obfuscation on the quality of java code. *Empirical Software Engineering 20*, 6 (2015), 1486–1524.

[12] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems* (2004), vol. 2988 of *LNCS*, Springer, pp. 168–176.

[13] COLLBERG, C., MARTIN, S., MYERS, J., AND NAGRA, J. Distributed application tamper detection via continuous software updates. In *Proc. of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACM, pp. 319–328.

[14] COLLBERG, C., AND NAGRA, J. Surreptitious software. *Upper Saddle River, NJ: Addision-Wesley Professional* (2010).

[15] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[16] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine learning 20*, 3 (1995), 273–297.

[17] DALLA PREDA, M. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, University of Verona, 2007.

[18] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[19] GANESH, V., AND DILL, D. L. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification* (2007), Springer, pp. 519–531.

[20] GENT, I. P., MACINTYRE, E., PROSSER, P., WALSH, T., ET AL. The constrainedness of search. In *AAAI/IAAI, Vol. 1* (1996), pp. 246–252.

[21] GRANITTO, P. M., FURLANELLO, C., BIASIOLI, F., AND GASPERI, F. Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products. *Chemometrics and Intelligent Laboratory Systems 83*, 2 (2006), 83–90.

[22] HALL, M. A. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.

[23] HEFFNER, K., AND COLLBERG, C. The obfuscation executive. In *International Conference on Information Security* (2004), Springer, pp. 428–440.

[24] HOLLAND, J. H. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[25] KANZAKI, Y., MONDEN, A., AND COLLBERG, C. Code artificiality: a metric for the code stealth based on an n-gram model. In *Proc. of the 1st International Workshop on Software Protection* (2015), IEEE Press, pp. 31–37.

[26] KARNICK, M., MACBRIDE, J., MCGINNIS, S., TANG, Y., AND RAMACHANDRAN, R. A qualitative analysis of java obfuscation. In *proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA* (2006).

[27] LE, L. Payload already inside: datafire-use for rop exploits. *Black Hat USA* (2010).

[28] LI, C.-M., AND YE, B. Sat-encoding of step-reduced md5. *SAT COMPETITION 2014*, 94–95.

[29] LIN, S.-W., LEE, Z.-J., CHEN, S.-C., AND TSENG, T.-Y. Parameter determination of support vector machine and feature selection using simulated annealing approach. *Applied soft computing 8*, 4 (2008), 1505–1512.

[30] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976), 308–320.

[31] Merz, F., Falke, S., and Sinz, C. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. In *International Conference on Verified Software: Tools, Theories, Experiments* (2012), Springer, pp. 146–161.

[32] Mironov, I., and Zhang, L. Applications of sat solvers to cryptanalysis of hash functions. In *International Conference on Theory and Applications of Satisfiability Testing* (2006), Springer, pp. 102–115.

[33] Mohsen, R., and Pinto, A. M. Evaluating obfuscation security: A quantitative approach. In *International Symposium on Foundations and Practice of Security* (2015), Springer, pp. 174–192.

[34] Nelder, J. A., and Baker, R. J. Generalized linear models. *Encyclopedia of statistical sciences* (1972).

[35] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., and Simon, L. Impact of community structure on sat solver performance. In *Theory and Applications of Satisfiability Testing–SAT 2014*. Springer, 2014, pp. 252–268.

[36] Newsham, Z., Lindsay, W., Ganesh, V., Liang, J. H., Fischmeister, S., and Czarnecki, K. Satgraf: Visualizing the evolution of sat formula structure in solvers. In *Theory and Applications of Satisfiability Testing–SAT 2015*. Springer, 2015, pp. 62–70.

[37] Nguyen, V. *Improved size and effort estimation models for software maintenance*. PhD thesis, University of Southern California, 2010.

[38] Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy* (2012), IEEE, pp. 601–615.

[39] Pearson, K. Note on regression and inheritance in the case of two parents. *Proc. of the Royal Society of London 58* (1895), 240–242.

[40] Pearson, K. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2*, 11 (1901), 559–572.

[41] Qiu, J., Yadegari, B., Johannesmeyer, B., Debray, S., and Su, X. Identifying and understanding self-checksumming defenses in software. In *Proc. of the 5th ACM Conference on Data and Application Security and Privacy* (2015), ACM, pp. 207–218.

[42] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., and Weippl, E. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR) 49*, 1 (2016), 4.

[43] Sharif, M. I., Lanzi, A., Giffin, J. T., and Lee, W. Impeding malware analysis using conditional code obfuscation. In *NDSS* (2008).

[44] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., and Vigna, G. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware.

[45] Sutherland, I., Kalb, G. E., Blyth, A., and Mulley, G. An empirical examination of the reverse engineering process for binary files. *Computers & Security 25*, 3 (2006), 221–228.

[46] Udupa, S., Debray, S., and Madou, M. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering* (2005).

[47] Werbos, P. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph.D. thesis, Harvard University.*.

[48] Wu, Y., Fang, H., Wang, S., and Qi, Z. A framework for measuring the security of obfuscated software. In *Proc. of 2010 International Conference on Test and Measurement* (2010).

[49] Yadegari, B., Johannesmeyer, B., Whitely, B., and Debray, S. A generic approach to automatic deobfuscation of executable code. In *Symposium on Security and Privacy* (2015), IEEE, pp. 674–691.

[50] Zhuang, Y., and Freiling, F. Approximating Optimal Software Obfuscation for Android Applications. In *Proc. 2nd Workshop on Security in Highly Connected IT Systems* (2015), pp. 46–50.